NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

SARA SÁ ALMEIDA

Bachelor in Computer Science and Engineering

TYPE-DRIVEN SYNTHESIS OF EVOLVING DATA MODELS

COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
October, 2021

DEPARTMENT OF
COMPUTER SCIENCE

TYPE-DRIVEN SYNTHESIS OF EVOLVING DATA MODELS

SARA SÁ ALMEIDA

Bachelor in Computer Science and Engineering

Adviser: João Costa Seco
Associate Professor, NOVA University Lisbon

Co-adviser: Bernardo Toninho
Assistant Professor, NOVA University Lisbon

COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
October, 2021

Type-driven Synthesis of Evolving Data Models

Acknowledgements

I would like to thank my thesis advisors, João Costa Seco, and Bernardo Toninho, for all the guidance and invaluable input throughout the research, development and writing of my thesis.

I also want to express my gratitude towards everyone involved in the GOLEM project[1] for the advice I got during the project's weekly meetings, the opportunity to be involved in such a project and for providing me with a grant[2] which allowed me to dedicate all my time to research.

Special thanks to professor João Lourenço who spent countless hours building this template[3] which was an invaluable help to me and all the students who have used it throughout the years. I am sure that not only did a lot of effort go into creating the template but also maintaining it, clarifying doubts, and improving it even up to now.

I want to thank Teresa, for being my core support at university, the best project partner and a great friend. I also thank my friends, and my Erasmus friends for being with me during my academic studies and for still being in my life.

At last, I express my gratitude and appreciation to my family. Paula, Luísa, Conceição, and Nelson, thank you. Without your presence, this challenging year would have not been the same, to endure through writing a dissertation while dealing with a pandemic was certainly not an easy task. I am grateful for all the love and support I got from you. I cannot forget to thank my favorite dogs, Oliver and Ric. Oliver, you are the best dog I could ever have and I appreciate all the companionship. Ric, the newest arrival, I am so happy you joined the family, you bring so much joy.

[1] Golem Project, Ref. LISBOA-01-0247-FEDER-045917

[2] Grant Reference, BM/GOLEM/2020/005

[3] This document was created using the (pdf/Xe/Lua)LaTeX processor, based on the NOVAthesis template, developed at the Dep. Informatics of Nova University Lisbon by João M. Lourenço. [26]

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.

The aim of this dissertation, which is integrated in GOLEM, is to develop a synthesis framework that, based on high-level specifications, correctly defines and evolves a rich data layer component by means of high-level operations. The construction of the framework was approached by defining a specification language to express richly-typed specifications, a target language which is the goal of synthesis and a type-directed synthesis procedure based on proof-search concepts.

The range of real database operations the framework is able to synthesize is demonstrated through a case study. In a component-based synthesis style, with an extensible library of base operations on database tables (specified using the target language) in context, the case study shows that the synthesis framework is capable of expressing and solving a wide variety of data schema creation and evolution problems.

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

Resumo

Os sistemas modernos de *software* comercial são frequentemente caracterizados como aplicações centradas em dados. Estas aplicações definem os dados como o seu principal e persistente ativo, e utilizam um único modelo de dados para as suas funcionalidades, gestão de dados, e atividades analíticas.

Além disso, uma vez que as aplicações são efémeras, contrariamente aos dados, existe a necessidade de continuamente evoluir o esquema de dados para introduzir novas funcionalidades. Neste sentido, o conjunto rico de características e em constante evolução que é esperado das aplicações modernas encontra-se restricto, não só pela quantidade de dados disponíveis, mas também pela sua estrutura, dependências internas, e a capacidade de crescer e evoluir a representação dos dados de uma forma uniforme e rápida.

O projeto GOLEM tem como objetivo a produção de novos métodos de automação de programas integrado no desenvolvimento de aplicações centradas nos dados em sistemas *low-code*. Neste contexto, um dos objetivos principais de automação é a camada de dados, compreendendo a estrutura dos dados e as respectivas condições de integridade, como também as regras de validação e controlo de acessos.

O objetivo desta dissertação, integrada no projeto GOLEM, é o desenvolvimento de um sistema de síntese que, baseado em especificações de alto nível, define e evolui corretamente uma camada de dados rica com recurso a operações de alto nível. A construção deste sistema baseia-se na definição de uma linguagem de especificação que permite definir especificações com tipos ricos, uma linguagem de expressões que é considerada o objetivo da síntese e um procedimento de síntese orientada pelos tipos.

O espectro de operações reais de bases de dados que o sistema consegue sintetizar é demonstrado através de um caso de estudo. Com uma biblioteca extensível de operações sobre tabelas no contexto, o caso de estudo demonstra que o sistema de síntese é capaz de expressar e resolver uma grande variedade de problemas de criação e evolução de esquemas de dados.

Palavras-chave: Síntese de Programas, Síntese Orientada pelos Tipos, Tipos Refinados, Modelos de Dados, Interfaces, Evolução de Esquemas de Dados

Contents

List of Figures

List of Tables

List of Listings

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 [44]. 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 [44]. 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 [11, 44, 39, 19] 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 [44]. 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 from the production of hardware. The focus was 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 [11, 44, 39, 19, 8]. This crisis was characterized [11, 44, 39, 19, 8] 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 the production/cost of software and reducing 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 [44, 11, 39] 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 a risk to human lives. On this matter, there were two lines of reasoning [44, 39]: 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 [39]. 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 [11, 44], 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 afterward. Which revealed to be foundational work to the field of automatic programming/program synthesis.

Nowadays, a widespread population has access to computers. A reality that before did not seem possible, when just a few computers existed. Personal computer 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 Motivation

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.

1.2 GOLEM

The GOLEM project [1] aims to produce new methods of program automation integrated into the development of data-centric applications in low-code frameworks.

The overall architecture of the project is represented in Figure 1.1. The architecture is based on layers where the pipeline combines user experience, models and synthesis engines which interact with the current OutSystems development Model.

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.

1.3 Objectives

This work, which is integrated in GOLEM, will focus on the challenge of defining and evolving a rich data layer component using high-level operations, and investigate the ability to synthesize both database scripts and/or code for a data layer access component that

[1] GOLEM project, Ref. LISBOA-01-0247-FEDER-045917

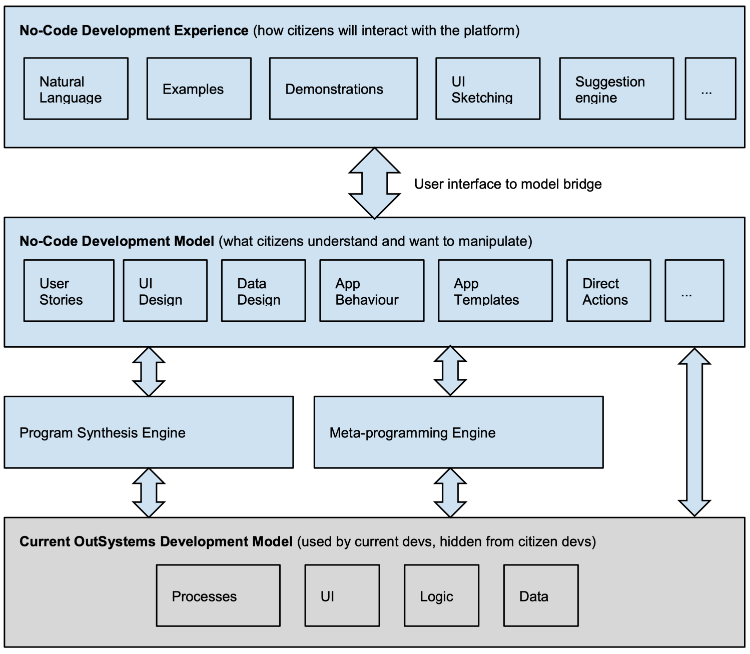
No-Code Development Experience (how citizens will interact with the platform)
Natural Language
Examples
Demonstrations
UI Sketching
Suggestion engine
...
User interface to model bridge
No-Code Development Model (what citizens understand and want to manipulate)
User Stories
UI Design
Data Design
App Behaviour
App Templates
Direct Actions
...
Program Synthesis Engine
Meta-programming Engine
Current OutSystems Development Model (used by current devs, hidden from citizen devs)
Processes
UI
Logic
Data

Figure 1.1: GOLEM Architecture

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 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. In the overall GOLEM architecture (Figure 1.1), our work comprises the meta-programming engine. The type-based synthesis specification is built and given to the synthesis mechanism by the layers above (user experience and model). The effects of the synthesized operations will manifest at the lower level on the current OutSystems development model within the data and logic components.

1.4 Contributions

Our work presents the following contributions:

- Definition of a core language for the specification of data schemas. The language is used to specify the synthesis problem.
- Definition of a core language for data schema modifications, which is the target of our synthesis procedure.
- Definition of a library of base operation on tables, expressed in the target language.
- Implementation of a type-directed synthesis procedure for the creation/modification of data schemas.

1.5 Document Structure

The next sections are organized as follows:

- Chapter 2 - Literature Review covers the state of the art in the area of program synthesis.
- Chapter 3 - Summary of Key Related Work identifies the most relevant pieces of work in the context of our solution.
- Chapter 4 - Synthesis framework specifies the components of our approach and some implementation details.
- Chapter 5 - Library of Operations presents in detail each base operation on tables.
- Chapter 6 - Case Study demonstrates the range of problems the framework can express and solve, by focusing on some specific situations, followed by an analysis of the synthesis examples presented.
- Chapter 7 - Conclusion expresses the concluding remarks and some pointers to interesting future work.

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" [22]. 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 [3, 36, 22]. Early approaches to the derivation of programs used theorem-proving as a constructive procedure to build proofs and extract programs from those proofs [27]. These approaches were followed by techniques that perform synthesis by transforming specifications into correct-by-construction programs without the need for proofs [27]. 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 [3] stimulated further improvements of synthesizers, allowing the specification of synthesis constraints that can be passed to these 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 [3].

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) [3], which is a community effort to formalize the core ideas behind these approaches as the SyGuS problem. In [3], 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 generalize from incomplete examples or synthesis that applies 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 [3, 36, 41]. 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 [41], or generating a program from a library of components [22, 15]. 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 [22] 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 [41], 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 [3]. 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 that employ heuristics or learned ranking functions.

A simpler ranking technique is to enforce preferences over candidate programs. For instance, in [45], 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 [15] uses around 15,000 code snippets from Stackoverflow to train a 2-gram model that allows ordering program candidates based on the score from this model. FlashFill [40] 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 [40, 16], there is the need to involve users in the synthesis process, to increase users' 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 [29], 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 [32], where the program space is defined by the grammar in (2.1).

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

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 number 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 [45, 32]. 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 equivalence 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) [32]. 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 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 [32].

There are alternative forms of pruning the search space that uses deductive techniques, which mix enumeration and deduction. In [1], 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 [15] 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 [41], 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 result 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 [35, 1].

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 [3].

SAT solvers determine the satisfiability of a formula but they cannot take into consideration background theories (that give an interpretation of predicates and functions) [7]. SMT solvers, are able to deal with the satisfiability of formulas given a background theory [7]. Advances in SAT solving [3, 7] allowed to build better SMT solvers.

As in program verification, program synthesis also takes advantage of SMT solvers. The use of such solvers allows delegating 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 [22]. The program space and specification are encoded into SMT formulas that are leveraged to the solver.

The SKETCH system [41], 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 [22] 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 [22], 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) [3] 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 [22] that encode the constraints directly, which is a complex task and then they have to map the solution back to a program [43]. Solver-aided languages [41, 43] 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 [41] 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 [43] 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 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 [18, 3, 38]. The use of a Markov Chain Monte Carlo (MCMC) sampler together with a cost function over the program space is an example [38] 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 [38], 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 [38]. The MCMC sampler in the case of [38] uses the cost function and returns programs with a low cost. In the formulation in [3], 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 [23] 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 [1] and guiding the search with learned probabilistic models [24]. When we discussed the enumerative technique and the combination with deduction, we mentioned the divide and conquer strategy. The divide and conquer strategy [1] 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 [1] 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 [24] is to formulate the problem as a syntax-guided synthesis problem that 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 in 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 [24, 9] . That may complicate the task of using machine learning and neural networks if such an 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 [22]. 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 [41]. Another example is to provide a set of input-output examples of the expected behavior of the program and the main bit manipulation operators [22]. Within the work of formalizing the syntax-guided problem [3], 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 [15]. Data wrangling concerns activities of data manipulation, such as data extraction and transformation. FlashFill [40] 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 [5] 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 [15]. 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 program that can be synthesized is queries [45, 47, 14]. 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 [47, 45].

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 [22]. Since it is hard to understand deobfuscated code and many approaches deobfuscate the program manually, the work in [22] 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 [38]. Performance constraints enable the synthesizer to return an optimized program. The repair of programs [23] 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 [28, 27]. 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 derivation from specifications, extracting programs from proofs, and type-directed approaches. Initial approaches to program synthesis applied deductive theorem-proving techniques to obtain programs [27]. A program could be extracted from the deductive proof of a theorem. Later, newer techniques involving specification transformation or rewriting appeared [27]. More recently, specific type-directed [35, 31] reasoning has been used when the specification defines the types of input and output. In addition, using solvers has permitted to automate previously intricate tasks [42].

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 [10] was foundational in this area. A non-automated way of thinking about the derivation of programs from specifications. In the work, Dijkstra [10] 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 [10, 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 precondition 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 [10] from Dijkstra's work that requires the establishment of an 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 [10], 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 $wp(\text{“}x := x-y\text{''}, P) = (gcd(X,Y) = gcd(x-y,y) \textbf{ and } x-y > 0 \textbf{ and } y > 0)$, 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

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

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 [42] 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 functions are not specified. The ranking function here has the same goal as 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 [42] 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 [10] 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 [28, 27]. 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 [27]. In the early '70s, Manna et al. [28] 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 [28], the authors mention that the use of both in a system could be useful. Even though newer techniques appeared to involve transforming the program's specification instead of proving a theorem, there was still work [27] 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 [28] 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 that 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 [35]. In this way, candidates that do not typecheck are rejected, but still a lot of combinations are considered. Current type-directed techniques [35, 31] 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 [31] or added predicates to types (refinement types) [35], 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 [35] 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 [35].

Without looking at the details yet, let us 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 [35] works 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 [35]. 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 adds an unknown predicate to the set of components and searches 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. In 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).

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

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 [31]. 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 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 in the number 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 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 [28, 27]. In Dijkstra's work [10], to build an iterative program, there was the need to define a variant function to prove termination and an 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 [35].

3 Summary of Key Related Work

We now summarize the most significant pieces of work from the literature that relates to data-centric and program evolution synthesis.

3.1 Program Synthesis and Data-centric Synthesis

As mentioned above, we aim to use type-directed synthesis techniques. Current type-directed approaches often use type information jointly with additional data, such as examples [31] or predicates added to the types [35], to further restrict the search space.

In the context of functional programming, in [35] 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 [35] 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 [31] also is inserted in the context of functional synthesis. The work [31] 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 [35] 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 [31, 35] allow to greatly prune candidates dividing the problem of finding a solution in subproblems propagating type and other information top-down, this allows solving each subproblem only considering this restricted information and finding/evaluating 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.

3.2 Synthesis and Program Evolution

Some work focus on synthesizing evolutions/changes [46, 37]. In the context of data-driven applications. The work [46] 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 [46] 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 [37] 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 backward 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 [41] 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).

4

Synthesis Framework

4.1 Overview

Our technical approach and respective implementation embody the three conceptual ideas presented in the literature review (Section 2). A synthesis problem may be characterized by three key dimensions: user intent; search space and search technique.

Our main goal is to be able to synthesize a program from a type-based specification using type-directed techniques. To that purpose, we designed two languages consisting of a λ-calculus (where the types are the specification language and the terms are the target language) and a synthesis procedure. We shall characterize our technical approach in the same three dimensions as follows:

1. User Intent: **Specification Language**;
2. Search Space: **Target Language**;
3. Search Technique: **Type-directed Synthesis Procedure**.

The flow of interaction between the three dimensions in our approach is the following: the specification language allows to specify rich type-based specifications with refinement and dependent types; the target language defines the terms that can be used to construct the target program; and, finally, the type-directed synthesis procedure is guided by the types present in the specification (based on the structure of the type) to gradually build a program, using the terms in the target language, such that it satisfies the complete specification.

We can visualize the three dimensions in the overall architecture of the synthesis framework present in 4.1. The tool we developed is defined by the core type-based synthesis procedure. It receives a specification expressed in the specification language which is built by the top layers of the GOLEM framework. The procedure may resort to an SMT Solver to verify the satisfiability of formulas and at the end outputs a combination of terms expressed in the target language (which is typified by the specification language). At last, the output terms will manifest their effects at the level of a data management layer.

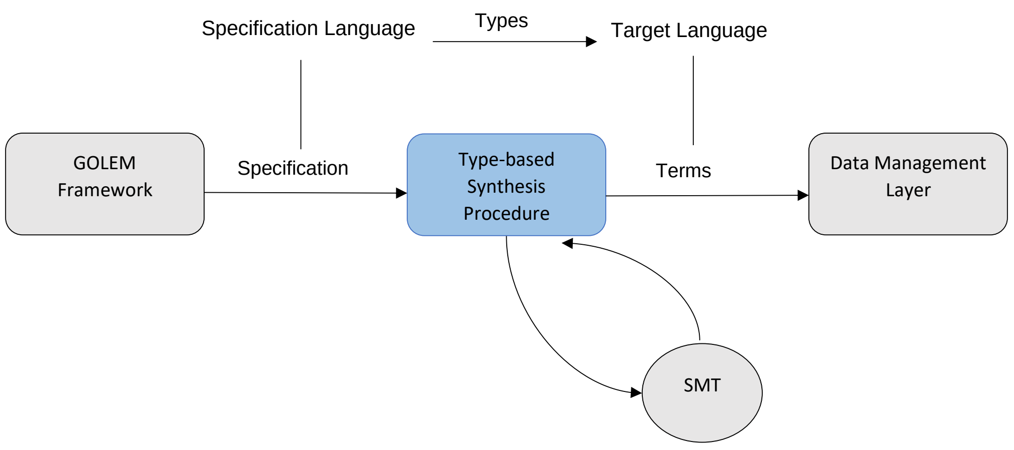

Figure 4.1: Synthesis Framework Architecture

Throughout the following sections, we will analyze in depth the technical details regarding our approach to each of the three dimensions. We will start with the specification language (Section 4.2), with the definition of the syntax of types and relations on types, such as subtyping and well-formedness. Next, we move on to the target language (Section 4.3) to understand the syntax of terms, the type checking procedure, and the semantics. Proceeding to the third dimension, the core of the approach, we have a look at the synthesis procedure (Section 4.4) and rules. To finish the presentation of the technical approach, we present some implementation (Section 4.5) specific topics.

4.1.1 Example: Employee Table Scenario

We now introduce a simple example to illustrate the framework concepts presented next. To see a more complete example, from specification to solution, we refer the reader to the case study in Section 6.

Let us consider a scenario where a domain expert, in a company, that knows the area quite well but is only slightly familiar with databases and programming. The expert knows that the company keeps records on the identifiers of each employee and now wishes to keep their salary information as well (with integer values). Why would the expert learn some programming language in a hurry for this task, like SQL for example, when he could only specify what he wants and obtain a script that performs the operation?

The challenge here is to specify the expert's intention and obtain a script that changes the database from its initial state to the desired one. If this user intent is specified properly, our framework could provide such a script. Which in this case, would be an operation that changes the employee's table schema (and inherent data) to include a new salary column with the desired data. Figure 4.2 demonstrates the desired change on the database schema (id stands for the employee's identifier and PK for primary key).

Over the three next sections, we will develop this example. In Section 4.2 we will see how to specify the user intent using our syntax. Next, in Section 4.3, we define the

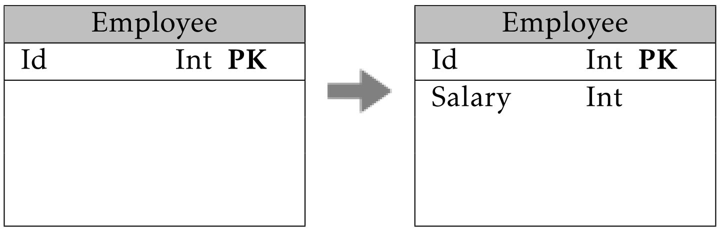

Figure 4.2: Employee Table Schema Change

target program and solution to this problem. Finally, in Section 4.4 the reader will understand how does it work to synthesize a solution, namely how to go from the specification information to the target program.

4.2 Specification Language

The specification language features rich types: dependent and refinement types that allow us to define dependent function types where the output type may mention the input and types refined with formulas, and parametric polymorphism where we may define generic types which can then be instantiated.

We also define and present relations on types such as well-formedness (Section 4.5) and a subtyping relation (Section 4.2.3) between types. We express substitutions both on types τ and formulas φ as $[x/v]\tau$, standing for replacing v by x in τ (extends to formulas).

4.2.1 Syntax

The syntax of types is given by the grammar in Figure 4.3. We have organized our syntax of types in three levels, distinguishing between Schemes, Types, and Base Types.

Variables range over x and v. Rows r define the table base type and formulas, φ and ψ, are used in refinement types. The labels l of records and tables may be identifiers or a label polymorphic variable L. Table names n may also be identifiers or polymorphic variables N.

Moreover, we have a finite number of polymorphic variables, namely X, R, N and L, for types, rows, names, and labels, respectively. Each polymorphic variable may be tagged. We have defined two types of tags: Skolem and Universal. The Universal tag denotes a variable associated with a universal quantifier (e.g. X_{U_n}). The Skolem tag denotes a variable associated with an existential quantifier (e.g. X_{S_k}). This enables us to when we encounter a polymorphic variable, know with which type of quantifier it is associated with. Note that we do not have existential quantifiers in the syntax but, as we will see in the Synthesis (Section 4.4), we will use a technique in which we use universal quantifiers as if they were existential ones. Now we shall analyze in more detail each of the constructs.

Row	$r ::= \varepsilon$	Empty Row
	$\mid l : \tau, r$	Cons
	$\mid R$	Row Variable
Base Types	$B ::= int \mid bool$	Basic Types
	$\mid \{\overline{l : \tau}\}$	Record
	$\mid \text{Table}[n]\{r \mid r\}$	Table
Formulas	$\varphi, \psi ::= e$	Predicates
	$\mid \varphi$ and ψ	
	$\mid \varphi$ or $\psi \mid$ not φ	Boolean Logic
	$\mid l$ **in** r	Row Membership
Types	$\tau ::= B$	Base Type
	$\mid \tau_1 \rightarrow \tau_2$	Function
	$\mid (x : \tau_1) \rightarrow \tau_2$	Dependent Function
	$\mid (\tau_1, \tau_2)$	Pair
	$\mid \{v : B \mid \varphi\}$	Refinement Type
	$\mid X$	Type Variable
Schemes	$S ::= \forall X.S$	Forall Type
	$\mid \forall_R R.S$	Forall Rows
	$\mid \forall_N N.S$	Forall Names
	$\mid \forall_L L.S$	Forall Labels
	$\mid \tau$	Type Scheme

Figure 4.3: Syntax of Schemes, Types and Base Types

Base Types and Rows. Basic types such as int and bool are examples of base types. A record $\{\overline{l : \tau}\}$ is a collection of pairs label l and type τ. A table $\text{Table}[n]\{r_1 \mid r_2\}$ is uniquely identified by a name n (if this one is fully instantiated) and is formed by two rows, where r_1 identifies the table's primary keys and r_2 all the other columns.

The row type r defines the structure of tables, both the collection of keys and other columns. Rows can be empty (ε), which means that there are no columns, or they can be a collection of columns (pairs label l and type τ). They can also be formed by only one row variable R or a collection of columns ending with R. For intuition, let us see a few examples on rows, to get a better understanding of the construct. The examples naturally extend to tables:

1. $\{\,\}$ - empty row;

2. $\{ l_1 : \tau_1, l_2 : \tau_2 \}$ - collection of columns (l_1, τ_1) and (l_2, τ_2);

3. $\{ R \}$ - an unknown collection of columns denoted by row variable R;

4. $\{ L : X, R \}$ - row with a label L and type X, both polymorphic, and other unspecified columns denoted by R.

Types and Formulas. Besides base types, our type language comprises $\tau_1 \rightarrow \tau_2$ that describes a function from τ_1 to τ_2 while $(x : \tau_1) \rightarrow \tau_2$ describes a dependent function type, with x identifying the argument of the function. Dependent function types are very useful, especially in the kind of synthesis specifications we will want to have since they allow the output type of a function to mention its input. We also have pairs (τ_1, τ_2) with τ_1 in the first component and τ_2 in the second, and a type variable X. Finally, refinement types, $\{v : B | \varphi\}$, are types refined by a formula φ. We have imposed the restriction that only base types B can be refined. The type being refined is identified by a variable v which may appear in the refinement φ.

The formulas are: predicates formed by boolean expressions in the language of terms; boolean operators - conjunction (and), disjunction (or), negation (not); or, a membership formula that asserts that a label l belongs to row r. We can express more properties over base types by introducing new formulas which allows for more expressive synthesis specifications.

Schemes. Polymorphic types or type schemes define types that have a polymorphic variable bound by a universal quantifier, we call schemes to these generic types. We have defined four kinds of universal quantifiers for each of the polymorphic variables. A scheme $\forall X . S$ binds a type polymorphic variable X in scheme S. Schemes are built recursively with the bound variable always on the outer part of the scheme. When we reach a type τ, we no longer add universal quantifiers, which means that we have defined that universal quantifiers are always on the outer part of types.

4.2.2 Well-formedness

We define a unary relation on types given by the judgment $\Gamma; \Sigma \vdash \tau$ which states that a type τ is well-formed under context $\Gamma; \Sigma$ (see below for an explanation of the context).

For a type to be well-formed, all of its subcomponents must be well-formed and any free polymorphic variable must appear in the context. This is a particularly important relation since we do not want to define procedures or relations over poorly formed types. The relation naturally extends to the other constructs defined in the syntax (Figure 4.3). Let us first present the context and then we will see the rule system.

4.2.2.1 Context

The context (illustrated in Figure 4.4) is composed of two environments $(\Gamma;\Sigma)$ and denotes the available information on variables. The information is useful to define relations and procedures, such as well-formedness but also subtyping, type checking, etc...

$$
\begin{array}{ll}
\Gamma ::= \emptyset \,|\, \Gamma, x : \tau \,|\, \Gamma, x : S & \text{term variable binding} \\
\Sigma ::= \emptyset \,|\, \Sigma, X \,|\, \Sigma, R \,|\, \Sigma, N \,|\, \Sigma, L & \text{polymorphic variable binding}
\end{array}
$$

Figure 4.4: Variable Binding Context

The first environment Γ keeps the information on variable identifiers x and corresponding types τ or schemes S. Environment Σ retains information on polymorphic variables.

4.2.2.2 Well-formedness Rules

The rule system is given in Figure 4.5. Most rules are standard, denoting that a type is well-formed if its subcomponents are well-formed.

We draw the reader's attention to the rules WF-TAB, WF-CONS, WF$_R$, WF$_X$, WF-REC, WF-PRED, WF-REFT and WF-$\forall$. According to WF-TAB, a table type Table$[n]\{k|r\}$ is well-formed if its name n and both rows k and r are well-formed. The rule for names is not present in the rule system for presentation purposes but, we remind the reader that a name is either a specific name defined by an identifier or a name variable N. This means that a name is well-formed if the identifier is well-formed or if N is in Σ. We also want to guarantee that there are no repeated labels among rows k and r. $L(k)$ and $L(r)$ stand for the set of labels present in rows k and r, respectively. We want to guarantee that the intersection of these sets $(L(k) \cap L(r))$ is empty $(\emptyset)$ and therefore there are no duplicate labels.

The well-formedness of rows is defined by three rules. There is not much to say about an empty row so let us look at the cons case. WF-CONS states that a cons with structure $(l : \tau, r)$ is well-formed if label l, type τ and row r are well-formed. Labels follow the same reasoning as names, just varying in the polymorphic variable that is L. We also want to ensure the no duplicate labels property within a row. That is why we state that the label l in cons $(l : \tau, r)$ cannot belong to the following row r. We want to guarantee that we do not have free polymorphic variables, rules WF$_R$ and WF$_X$ state that a row variable and a type variable are well-formed if they belong to Σ.

Now, for records $\{\overline{l : \tau}\}$ which denote a collection (possibly empty and with size up to n) of labels and types, every label l_i and corresponding type τ_i must be well-formed. Note that we also do not allow repeated labels in records.

When it comes to refinement types, formulas, and schemes, we highlight that in rule WF-PRED, when a formula is a predicate boolean expression from the language of terms we check that expression e has a boolean type ($\Gamma;\Sigma \vdash e \Leftarrow bool \rightsquigarrow C$ denotes the checking judgment that we will see in Section 4.3.2). Formulas are a part of refinement types. In

Base Types $\boxed{\Gamma;\Sigma \vdash B}$

$$\text{WF-I}\,\frac{}{\Gamma;\Sigma \vdash \mathit{int}} \qquad \text{WF-B}\,\frac{}{\Gamma;\Sigma \vdash \mathit{bool}}$$

$$\text{WF-TAB}\,\frac{\begin{array}{c}\Gamma;\Sigma \vdash n \quad L(k) \cap L(r) = \emptyset \\ \Gamma;\Sigma \vdash k \quad \Gamma;\Sigma \vdash r\end{array}}{\Gamma;\Sigma \vdash \text{Table}[n]\{k|r\}} \qquad \text{WF-REC}\,\frac{\begin{array}{c}i = 0 \dots n \\ \Gamma;\Sigma \vdash l_i \quad \Gamma;\Sigma \vdash \tau_i\end{array}}{\Gamma;\Sigma \vdash \{\overline{l : \tau}\}}$$

Row $\boxed{\Gamma;\Sigma \vdash r}$

$$\text{WF-EMPTY}\,\frac{}{\Gamma;\Sigma \vdash \varepsilon} \qquad \text{WF-CONS}\,\frac{\begin{array}{c}\Gamma;\Sigma \vdash l \quad \Gamma;\Sigma \vdash \tau \\ l \notin r \quad \Gamma;\Sigma \vdash r\end{array}}{\Gamma;\Sigma \vdash l : \tau, r}$$

$$\text{WF}_R\,\frac{R \in \Sigma}{\Gamma;\Sigma \vdash R}$$

Types $\boxed{\Gamma;\Sigma \vdash \tau}$

$$\text{WF-FUNT}\,\frac{\Gamma;\Sigma \vdash \tau_1 \quad \Gamma;\Sigma \vdash \tau_2}{\Gamma;\Sigma \vdash \tau_1 \to \tau_2} \qquad \text{WF-PAIR}\,\frac{\Gamma;\Sigma \vdash \tau_1 \quad \Gamma;\Sigma \vdash \tau_2}{\Gamma;\Sigma \vdash (\tau_1, \tau_2)}$$

$$\text{WF-REFT}\,\frac{\Gamma;\Sigma \vdash B \quad \Gamma, v : B;\Sigma \vdash \varphi}{\Gamma;\Sigma \vdash \{v : B | \varphi\}} \qquad \text{WF-PI}\,\frac{\Gamma;\Sigma \vdash \tau_1 \quad \Gamma, x : \tau_1;\Sigma \vdash \tau_2}{\Gamma;\Sigma \vdash (x : \tau_1) \to \tau_2}$$

$$\text{WF}_X\,\frac{X \in \Sigma}{\Gamma;\Sigma \vdash X}$$

Schemes $\boxed{\Gamma;\Sigma \vdash S}$

$$\text{WF-}\forall\,\frac{\Gamma;\Sigma, X \vdash S}{\Gamma;\Sigma \vdash \forall X.S} \qquad \text{WF-}\forall_R\,\frac{\Gamma;\Sigma, R \vdash S}{\Gamma;\Sigma \vdash \forall_R R.S}$$

$$\text{WF-}\forall_N\,\frac{\Gamma;\Sigma, N \vdash S}{\Gamma;\Sigma \vdash \forall_N N.S} \qquad \text{WF-}\forall_L\,\frac{\Gamma;\Sigma, L \vdash S}{\Gamma;\Sigma \vdash \forall_L L.S}$$

Formulas $\boxed{\Gamma;\Sigma \vdash \varphi}$

$$\text{WF-PRED}\,\frac{\Gamma;\Sigma \vdash e \Leftarrow \mathit{bool} \rightsquigarrow C}{\Gamma;\Sigma \vdash e} \qquad \text{WF-AND}\,\frac{\Gamma;\Sigma \vdash \varphi \quad \Gamma;\Sigma \vdash \psi}{\Gamma;\Sigma \vdash \varphi \text{ and } \psi}$$

$$\text{WF-OR}\,\frac{\Gamma;\Sigma \vdash \varphi \quad \Gamma;\Sigma \vdash \psi}{\Gamma;\Sigma \vdash \varphi \text{ or } \psi} \qquad \text{WF-NOT}\,\frac{\Gamma;\Sigma \vdash \varphi}{\Gamma;\Sigma \vdash \text{ not } \varphi}$$

$$\text{WF-IN}\,\frac{\Gamma;\Sigma \vdash l \quad \Gamma;\Sigma \vdash r}{\Gamma;\Sigma \vdash l \textbf{ in } r}$$

Figure 4.5: Well-formedness Rules

WF-REFT, for refinement types to be well-formed, both the base type B and formula φ must be well-formed. And finally, in WF-$\forall$ (as for the other schemes) we add the type polymorphic variable X to Σ.

4.2.3 Subtyping

We define a subtyping relation on types. Our subtyping relation, given by the judgement $\Gamma;\Sigma \vdash \tau_1 <: \tau_2 \rightsquigarrow C$, determines that type τ_1 is subtype of type τ_2 if the constraints C hold. Intuitively, when type τ_1 is subtype of τ_2, it means that we can use values of type τ_1 as if they had type τ_2. The relation extends to base types.

This relation proves to be very useful, for example when we use the type of a problem to restrict the types of subproblems and solve each one individually. Combining each subproblem does not guarantee that it will match the original type exactly. If the combined solution's type is a subtype of the initial type, that means we can use it as having the original type. The same goes for situations where we infer types of terms and want to check that the inferred type is compatible with other types.

Now, that the reader has an intuition on this relation, we will introduce a constraint language that will be necessary for the subtyping rules we will introduce shortly after.

4.2.3.1 Constraint Language

We now introduce the notion of constraints into our language. The specification language has regular and refinement types. To define procedures and relations on refinement types, we cannot solely rely on the type's shape but we have to deal with the inherent formula. We introduce a constraint language as a way of producing constraints during the execution of procedures. These constraints are called Verification Conditions (VC). We eventually discharge the collected constraints into an SMT Solver to confirm their satisfiability.
To express these constraints, we extend our syntax as follows:

$$\begin{aligned} c, c' ::= &\ \varphi \\ &|c \wedge c' \\ &|\forall v : B.c \\ &|\varphi \implies c \end{aligned}$$

Our constraints may be predicates formed by a formula φ, a conjunction of constraints, a bounded universal quantification on variable v with base type B, or an implication. As an example of a constraint, we have $n = 1 \implies n > 0$ that stands for a constraint to verify that, if n has value 1, then it is positive.

Above we defined the subtyping judgement as 'type τ_1 is subtype of type τ_2 if the constraints C hold'. This shows that in the subtyping rules we will collect constraints according to this syntax and that we can only conclude that τ_1 is indeed a subtype of τ_2

if the constraints are satisfiable. To make such a conclusion, we have to encode the constraints and discharge them to an SMT Solver. For an explanation of our SMT encoding, we refer the reader to the implementation details in Section 4.5.

4.2.3.2 Subtyping Rules

Let us go through the subtyping rules present in figs. 4.6 and 4.7.

Base Types. We will start with the rules for base types in Figure 4.6. Rule <:-REFL is valid for every base type, which is the reflexive rule stating that a base type is equal only to itself. In this language we use the following notion of equality: equality up to the renaming of variables or alpha-equality.

A record is the only base type that has a special subtyping rule. Rule <:-REC is a width and depth subtyping rule, which means that we are doing subtyping both in the fields of a record and in the types of each common field. Width subtyping in records allows us to refer to a record by abstracting the number of fields. For example, with records $\{a : bool, b : int\}$ and $\{a : bool\}$ we may say that $\{a : bool, b : int\}$ is a subtype of $\{a : bool\}$. A supertype record is a more general record, which in this example, we are abstracting over all other fields and only saying that a record needs to have a label a with type bool. The more specific type $\{a : bool, b : int\}$ only has to verify the restriction of having label a with type bool. Depth subtyping verifies that for the type of each label present in the supertype, the type is also a supertype of the corresponding label's type in the subtype. As in the record example, label a is associated with type bool in the supertype, which means that the type of label a in the subtype must also be a subtype of bool. The type in the subtype is bool, which means that it is a subtype due to rule <:-REFL.

To summarize, for a record R_1 to be a subtype of R_2, every label of R_2 must be in R_1 and the types of each label in R_2 must be supertypes of the types of the corresponding labels in R_1.

$$\text{<:-REFL}\ \frac{}{\Gamma;\Sigma \vdash B <: B \rightsquigarrow true} \qquad \text{<:-REC}\ \frac{\begin{array}{c} i = 0\ldots n \quad j = 0\ldots m \quad m \leq n \\ \text{foreach j} .\ l_j \in \overline{l_i} \wedge \Gamma;\Sigma \vdash \tau_i <: \tau_j \rightsquigarrow C_j \end{array}}{\Gamma;\Sigma \vdash \{\overline{l_i : \tau_i}\} <: \{\overline{l_j : \tau_j}\} \rightsquigarrow \overline{C_j}}$$

Figure 4.6: Base Types Subtyping Rules

Types. Now we can have a look at the subtyping rules for types in Figure 4.7. We have three rules (<:-RB, <:-BR and <:-Reft) that explain the subtyping between a refinement type and a base type, the reverse and between two refinement types, respectively. Two rules for the subtyping between dependent function types (<:-Pi and <:-Pi-R), depending on whether the argument's type is refined or not.

$$\text{<:-RB}\ \frac{\Gamma;\Sigma \vdash B_1 <: B_2 \rightsquigarrow C}{\Gamma;\Sigma \vdash \{v : B_1 \,|\, \varphi\} <: B_2 \rightsquigarrow C} \qquad \text{<:-BR}\ \frac{\Gamma;\Sigma \vdash B_1 <: B_2 \rightsquigarrow C \qquad \Gamma, v : B_2;\Sigma \vdash \varphi}{\Gamma;\Sigma \vdash B_1 <: \{v : B_2 \,|\, \varphi\} \rightsquigarrow C \wedge \forall v : B_2 . \varphi}$$

$$\text{<:-Reft}\ \frac{\begin{array}{c}\Gamma;\Sigma \vdash B_1 <: B_2 \rightsquigarrow C \quad \Gamma, v_1 : B_1;\Sigma \vdash \varphi_1 \\ \varphi' = [v_1/v_2]\varphi_2 \quad \Gamma, v_1 : B_1;\Sigma \vdash \varphi'\end{array}}{\Gamma;\Sigma \vdash \{v_1 : B_1 \,|\, \varphi_1\} <: \{v_2 : B_2 \,|\, \varphi_2\} \rightsquigarrow C \wedge \forall v_1 : B_1 . \varphi_1 \implies \varphi'}$$

$$\text{<:-Pi}\ \frac{\begin{array}{c}\Gamma;\Sigma \vdash s_2 <: s_1 \rightsquigarrow C_1 \\ \Gamma, x_2 : s_2;\Sigma \vdash [x_2/x_1]\tau_1 <: \tau_2 \rightsquigarrow C_0\end{array}}{\Gamma;\Sigma \vdash (x_1 : s_1) \to \tau_1 <: (x_2 : s_2) \to \tau_2 \rightsquigarrow C_1 \wedge C_0}$$

$$\text{<:-Pi-R}\ \frac{\begin{array}{c}\Gamma;\Sigma \vdash s_2 <: s_1 \rightsquigarrow C_1 \quad \Gamma, x_2 : s_2;\Sigma \vdash [x_2/x_1]\tau_1 <: \tau_2 \rightsquigarrow C_0 \\ s_2 = \{v : B | \varphi\} \quad \varphi' = [x_2/v]\varphi \quad \Gamma, x_2 : s_2;\Sigma \vdash \varphi'\end{array}}{\Gamma;\Sigma \vdash (x_1 : s_1) \to \tau_1 <: (x_2 : s_2) \to \tau_2 \rightsquigarrow C_1 \wedge \forall x_2 : B . \varphi' \implies C_0}$$

Figure 4.7: Types Subtyping Rules

Let us exemplify the first three rules with combinations from the three types: int, $\{v : int | v > 0\}$ and $\{v : int | v = 1\}$. Rule <:-RB refers to subtyping between a refinement type and a base type, so we can exemplify it as: $\{v : int | v > 0\}$ <: int. How can $\{v : int | v > 0\}$ be a subtype of int? All it takes is that int is a subtype of int, which we know it is. As in rule <:-RB, all we need is that the refined base type B_1 is a subtype of B_2. Because $\{v : int | v > 0\}$ is more specific, it can be abstracted to having only type int. The constraints for the subtyping rule are the ones needed for B_1 to be a subtype of B_2.

Rule <:-BR defines the reverse of <:-RB. Now we want the subtyping relation int <: $\{v : int | v > 0\}$. A basic intuition is that this rule is not as straightforward as the previous one: we know that *int* cannot be a subtype of $\{v : int | v > 0\}$ because we cannot use any integer value as having a positive type. This is why in rule <:-BR we maintain the subtyping verification between base types B_1 and B_2 with constraints C, but we verify the well-formedness of the formula φ and add a new constraint to verify this refinement. The final constraint is a conjunction between C and a universal quantifier on the refinement type's identifier v with base type to guarantee that φ always holds. In the current example, the generated constraint is: true $\wedge \forall v : int . v > 0$ which does not hold and thus this rule allows a correct conclusion.

The third rule is a subtyping relation between two refinement types. We will use example $\{v : int | v = 1\} <: \{v : int | v > 0\}$ to exemplify rule <:-Reft. Intuitively, it makes sense that one is a positive value. In rule <:-Reft we maintain the restriction that the base type of the subtype must be a subtype of the base type of the supertype (with constraint C). To verify the relation between the refinements of each of the refinement types, we add a constraint to verify that if the subtype's refinement holds then the supertype's refinement also holds.

This amounts to a constraint that is a universal quantifier bound on a variable (in the rule is v_1) that identifies the type B being refined and an implication between the subtype's and supertype's refinements. To use an implication between both refinements we first unify the variables, that is why in the rule the identifier v_2 is replaced by v_1 in φ_2, so both φ_1 and φ_2 are bound to v_1 (and we can bind the constraint to v_1 with base type B_1). We also need to ensure that both refinements are well-formed. Applying the rule to the example we introduced, we would obtain the constraint $true \land \forall v : int . v = 1 \implies v > 0$ which states that one base type must be subtype of the other (true because int <: int) and when $v = 1$ always implies $v > 0$, and enables us to conclude the subtyping relation in the example.

Now we draw the reader's attention to the rules on subtyping between dependent function types (<:-Pi and <:- Pi-R). Subtyping between functions is covariant in the body's type and contra-variant in the argument's type. Which means that, for example, for the functions in the rules, for $(x_1 : s_1) \to \tau_1 <: (x_2 : s_2) \to \tau_2$ we check that for the function bodies, τ_1 must be a subtype of τ_2, but for the arguments we reverse this (s_2 must be a subtype of s_1). The final constraint is a conjunction between the two subtyping checks. For example, imagine that we would want to check the subtyping relation between functions $(x : \{a : bool\}) \to \{a : bool, c : bool\}$ and $(x : \{a : bool, b : int\}) \to \{a : bool\}$. All we need is to verify $\{a : bool, b : int\} <: \{a : bool\}$, the function's argument can have more information (in this case more fields in the record) as long as it has a label a with type bool, and $\{a : bool, c : bool\} <: \{a : bool\}$ stating that we can return more values as long as we ensure that we return the label a with type bool.

Rule <:Pi-R differs from <:-Pi only in that fact that considers the case of the argument's type s_2 being a refinement type. In that case, we want to ensure that the refinement implies the constraints on the function's bodies.

4.2.3.3 Algorithmic Subtyping

The theoretical rules for subtyping given in figs. 4.6 and 4.7 do not give direct guidelines for implementation. The rules are only presented as a relation between a context $(\Gamma;\Sigma)$, two types τ_1 and τ_2, and a constraint C.

Here we define the algorithmic subtyping procedure. Operationally we implement subtyping as an algorithm that takes as input types τ_1 and τ_2 and yields as output a constraint C that must be checked to certify the subtyping result (`C ==>` $\tau_1 <: \tau_2$). To check if constraint C holds, C is encoded (see Section 4.5) and discharged to an SMT Solver which returns a satisfiability result.

4.2.4 Example: Employee Table Specification

Now that we have seen the syntax of the specification language, we are equipped with enough information to specify the running synthesis problem. Let us remember the desired change defined in Figure 4.2, we have an expert that needs a way to specify the following change: add a column 'Salary' with integer type (Int) to table 'Employee'. How do we write the type of this change (and therefore, the specification of the problem)?

We want to be able to express the current state of the table and the desired one. We can start by describing the current structure of the 'Employee' table. The table has the name 'Employee' and a primary key 'Id' with type Int.
According to the syntax of types, we express it as (Type 4.1):

$$Table[Employee]\{Id : Int| \quad \} \tag{4.1}$$

Id is the only column, there are no others. What about the type of the desired table? The desired table is similar to the previous (Type 4.1) with an added column with the label 'Salary' and type Int.
We can describe it as follows:

$$Table[Employee]\{Id : Int|Salary : Int\} \tag{4.2}$$

Type 4.1 refers to the current structure of the table and type 4.2 refers to the desired structure. The program that we want should receive the current structure of 'Employee' and modify it so that we obtain the new structure. This means that this program receives a value of type 4.1 and somehow produces a value of type 4.2. The specification function must also provide a value for the new column to be inserted.

It is straightforward to see now that the synthesis specification and type signature of this program must be a function. Specifically, a dependent function, whose inputs are the type 4.1 and a value 'v' of type int, and an output type 4.2. The syntax of functions only defines one argument for a function, but we can define multiple argument functions through currying.
The complete synthesis specification for this example is:

(told: $Table[Employee]\{Id : Int| \quad \}) \rightarrow (v : Int) \rightarrow Table[Employee]\{Id : Int|Salary : Int\}$

The Identifier told (for 'old table') in the specification 4.2.4 refers to the input table of the function so that the program with this type signature can use the input and modify it accordingly to have the output type. The input 'v' is used to define the default value for the salary column.

4.3 Target Language

The target language comprises language terms such as records, tables, and target operations. Polymorphism is a feature of the language and is present through polymorphic abstraction and application. We will present the syntax (Section 4.3.1), the type checking mechanism (Section 4.3.2) and the language's operational semantics (Section 4.3.3). We will express substitutions on terms e as $[x/v]e$, standing for replacing v by x in e.

4.3.1 Syntax

The syntax of terms is given in Figure 4.8. The terms in the language follow a standard pattern of basic terms (numbers and booleans) and variables (x). Identifiers are represented by id. To which we add the terms described below.

Terms	$e ::= \mathrm{num} \mid \mathrm{bool}$	Basic Terms
	$\mid x$	Variable
	$\mid op(\overline{e})$	Native Operations
	$\mid \mathbf{let}\, x = e \;\mathbf{in}\; e'$	Let
	$\mid e : \tau$	Type Annotation
	$\mid (e, e')$	Pair
	$\mid e.1$	Pair First Projection
	$\mid e.2$	Pair Second Projection
	$\mid \{\overline{l = e}\}$	Record
	$\mid \pi_l\, e$	Record Projection
	$\mid \mathbf{new}\, \tau$	Table
	$\mid \mathrm{Table}[\alpha]$	Table Identifier
	$\mid \lambda x . e$	Function
	$\mid \lambda x : \tau . e$	Typed Function
	$\mid e\, e'$	Function Application
	$\mid \Lambda X . e$	Type Abstraction
	$\mid \Lambda_R R . e$	Row Abstraction
	$\mid \Lambda_N N . e$	Name Abstraction
	$\mid \Lambda_L L . e$	Label Abstraction
	$\mid e[\tau]$	Type Application
	$\mid e[r]_R$	Row Application
	$\mid e[n]_N$	Name Application
	$\mid e[l]_L$	Label Application

Figure 4.8: Syntax of Terms

Functions, Applications, Let Binders and Type Annotation. Functions may have a typed argument with type τ and are therefore defined by $\lambda x : \tau . e$, or not, defined by $\lambda x . e$. Standing for function application, we have $e\, e'$, where e is the function and e' its argument. Let binders follow the usual construct of $\mathbf{let}\, x = e \;\mathbf{in}\; e'$ where expression e denoted by x may appear in e'. And, we annotate the type τ of an expression by $e : \tau$.

Pairs, Records, Projections and Tables. The notation (e, e') denotes pairs with first component e and second component e'. The terms $e.1$ and $e.2$ are the pair's first and second projections, respectively, wherein the pair (e, e') projecting the first component gives us e and projecting the second gives us e'. Records denoted by $\{\overline{l = e}\}$ represent a collection of pairs label l and term e. To project any expression from a record, given a label l, we have the projection operator $\pi_l\, e$, where l stands for the label we wish to project from and this label must belong to record e.

Finally, our tables are defined by the constructor **new** and the type τ of a table (the table defined in the syntax of types in Figure 4.3). To refer to already existing tables, Table[α] is a reference for a table with name α (α stands for a fully instantiated table name and is unique, when n may be a name variable).

Polymorphic Abstraction and Application. To abstract over the types (which come from the specification language) present in the language terms, we provide a polymorphic abstraction for each of the polymorphic variables (type/row/label and name). Consider the type abstraction, $\Lambda X.\, e$ represents an abstraction on a type variable X on term e.

To replace the abstracted polymorphic variables by types, we define polymorphic applications for each variable as well. The corresponding type application is $e[\tau]$, where τ replaces the bound variable in abstraction term e.

Operations We define native operations abstracted by the construct $op(\overline{e})$, where $\overline{e}$ stands for the operation's arguments. Our native operations comprise unary and binary operations for integer and boolean values such as plus, minus but also boolean logical operators.

They also comprise native operations on tables, so that the target language offers a way of manipulating a table's schema and inherent data. The key idea is to define these operations on tables through an identifier x and an operation type τ $(x : \tau)$. Identifier x stands for the operation's name. We will establish type τ as a polymorphic function, that matches the current state of a table on the input and the output is the desired state.

Note that we define these operations as symbolic, the idea is that they can be used by the synthesis procedure but they do not have a defined operational semantics (they are 'axioms'). This means that the system is flexible because we can model any operation that we can define by a type without worrying about implementing the functionality directly.

For the reader to get a grasp on these table operations, we show here the example of an operation that inserts a column into a table. For an extensive view of our library of operations, we refer the reader to Section 5. For example, imagine that we have a table **new** Table$[n_1]\{l_1 : \tau_1 \,|\, \}$ and we want to add a general column $l_2 : \tau_2$. The function type has to match the current table's structure and produce as output a table with that extra column.

We first define a generic operation. We identify, for example, the operation as 'insertColumn'. Regarding the type, according to the syntax of types given in Figure 4.3, we need polymorphic variables for names, label, types, and row (since we do not want to define an operation for every specific operation type), to be able to abstract over the current table's structure. We then define polymorphic name N, rows *keys* and *rest*, type

T and label L. Our current table structure's type will then be:

$$\forall_N N.\forall_R keys.\forall_R rest.\text{Table}[N]\{keys|rest\}$$

Which represents a table with any name N, any collection of keys ('keys') and any collection of columns ('rest'). Our desired output adds a label L with type T to collection 'rest', and is thus represented by:

$$\forall T.\forall_N N.\forall_R keys.\forall_R rest.\forall_L L.\text{Table}[N]\{keys|L:T,rest\}$$

Besides both tables, we need an input v representing the default value to fill the newly introduced column L and must have the same type T. We need one final detail before assembling the operation's type. As we mentioned, refinement types allow us to express properties about the values belonging to the type. Note that we want to add label L to the input table, but we need to ensure that a label L is not already in the table since we do not allow for duplicate labels. That is why we have the row membership formula, we will use it to refine the input table so that we have the input condition that the entry table cannot have label L. The operation, defined by a polymorphic function, to insert a column into a table obtains the final structure:

$$\begin{aligned}&\forall T.\forall_N N.\forall_R keys.\forall_R rest.\forall_L L.\\&told:\{v:\text{Table}[N]\{keys|rest\}|\ \textbf{not}\ L\ \textbf{in}\ rest\}\to v:T\to\text{Table}[N]\{keys|L:T,rest\}\end{aligned}$$

Representing an operation that receives a table with any name N, any collection of keys 'keys' and columns 'rest' and adds label 'L' with type 'T' to it. The newly introduced column is filled with value v. Note that the input table is refined, as mentioned, to prevent it from having a general column (not a key) with label L.

Finally, we can obtain the specific operation to add column $l_2:\tau_2$ into **new** $\text{Table}[n_1]\{l_1:\tau_1|\}$ by simply instanciating the polymorphic variables of the insertColumn function. N is replaced by n_1, keys by $l_1:\tau_1$, rest by ε and L and T by l_2 and t_2. The polymorphic application term $\text{insertColumn}[t_2][n_1][l_1:\tau_1][\varepsilon][l_2]$ results in type $\text{Table}[n_1]\{l_1:\tau_1|\}\to v:\tau_2\to\text{Table}[n_1]\{l_1:\tau_1|l_2:\tau_2\}$, whose output type is the type of table we want.

4.3.2 Type Checking

Standard type checking rules follow the typing judgment $\Gamma;\Sigma\vdash e:\tau$, which states that a term e is well-formed with type τ in context $\Gamma;\Sigma$.

Many recent approaches [35, 31] in program synthesis do not follow this standard approach but have incorporated Bidirectional Type checking in their frameworks. We will also present a set of bidirectional type checking rules. Bidirectional type checking [34] allows for, as the name explicits, bidirectional exchange of type information (i.e. upwards and downwards). It comprises two modes: checking and synthesis. When checking, type information is passed downwards (from the roots of a derivation to the leaves) and when synthesizing, type information is passed upwards (vice versa). Depending on the structure of the term, the term may be checked against a type or a type may be synthesized for it. Normally, constructor terms are the ones that are checked since there is enough type information to be passed down and restrict the type of the subcomponents. Destructors are usually on the synth side, since there is no type information to pass downwards, its type is synthesized and passed upwards.

Bidirectional typing allows incorporating more type inference into a type system (through the synthesis phase) so that type annotations are less often required and also permit local error detection [21, 13] since type information is exchanged between AST nodes that are close to each other (both in checking and synthesis modes, type information, either synthesized or not, is passed to the next closest subcomponent).

The traditional typing rules are modified by splitting them into synthesis and checking rules. Two new judgments arise. The checking judgment of $\Gamma \vdash e \Rightarrow \tau$, where e type checks against τ in context Γ and the synthesis judgment $\Gamma \vdash e \Leftarrow \tau$ where type τ is synthesized for term e in context Γ. To these new judgments, we add a relation with a set of constraints C, to account for refinements and refinement type checking. And, of course, our judgments will be within the already defined typing environments in Figure 4.4. Thus, the new type checking judgements are the following:

- $\boxed{\Gamma;\Sigma \vdash e : \tau}$ Term e is well typed and has type τ
- $\boxed{\Gamma;\Sigma \vdash e \Rightarrow \tau \rightsquigarrow C}$ Term e synthesizes type τ with constraints C
- $\boxed{\Gamma;\Sigma \vdash e \Leftarrow \tau \rightsquigarrow C}$ Term e checks at type τ with constraints C

Now we will see our bidirectional typing rules. Starting with the checking rules according to the checking judgment and then the synthesis rules. We conclude this section, where we presented declarative typing rules, by explaining how the rules translate into an algorithm.

4.3.2.1 Checking Rules

The checking judgment for types τ is $\Gamma;\Sigma \vdash e \Leftarrow \tau \rightsquigarrow C$, which naturally extends for schemes as well.

We begin by seeing the checking rules for schemes, we define declarative typing rules to check polymorphic terms against type schemes. The rules are given in Figure 4.9.

Type checking a polymorphic abstraction, for example, the type abstraction $\Lambda X.e$, against a universal quantifier scheme (rules CHECK-TABS to CHECK-LABS), namely $\forall X.S$, amounts to checking that term e has type S and by adding the bound type variable X to Σ. The constraint of that check is the overall constraint. We add X into context so that when we are type checking e against S, we know that X is not free in both the term and type.

The last rule is the classical subsumption rule in which a term e is checked against a type S which is also a scheme, we infer a type for it and check that is indeed a subtype

$$\text{CHECK-TABS}\,\frac{\Gamma;\Sigma,X \vdash e \Leftarrow S \rightsquigarrow C}{\Gamma;\Sigma \vdash \Lambda X.e \Leftarrow \forall X.S \rightsquigarrow C} \qquad \text{CHECK-RABS}\,\frac{\Gamma;\Sigma,R \vdash e \Leftarrow S \rightsquigarrow C}{\Gamma;\Sigma \vdash \Lambda_R R.e \Leftarrow \forall_R R.S \rightsquigarrow C}$$

$$\text{CHECK-NABS}\,\frac{\Gamma;\Sigma,N \vdash e \Leftarrow S \rightsquigarrow C}{\Gamma;\Sigma \vdash \Lambda_N N.e \Leftarrow \forall_N N.S \rightsquigarrow C} \qquad \text{CHECK-LABS}\,\frac{\Gamma;\Sigma,L \vdash e \Leftarrow S \rightsquigarrow C}{\Gamma;\Sigma \vdash \Lambda_L L.e \Leftarrow \forall_L L.S \rightsquigarrow C}$$

$$\text{CHECK-SUB-SCH}\,\frac{\Gamma;\Sigma \vdash e \Rightarrow S' \rightsquigarrow C_1 \quad \Gamma;\Sigma \vdash S' <: S \rightsquigarrow C_2}{\Gamma;\Sigma \vdash e \Leftarrow S \rightsquigarrow C_1 \wedge C_2}$$

Figure 4.9: Scheme Checking Rules

of the type we were originally checking against. The conjunction of the constraints (synthesizing a type and subtyping) is the final constraint.
Now we go into the set of checking rules in Figure 4.10.

Abstractions, Pairs and Let Binders. Abstractions with untyped arguments appear on two rules, CHECK-ABS and CHECK-ABSREF. When checking a lambda abstraction ($\lambda x.e$) against a dependent function type ($(x : \tau_1) \to \tau_2$), we add the dependent's function argument's identifier x and type τ_1 to context. So that we can check that the abstraction's body e checks against the function's return type τ_2 (CHECK-ABS). What differs in ABS-REF is the fact that the dependent function's argument type is refined $\{v : B|\varphi\}$, so we need to take the refinement φ into consideration as well. We check that φ is well-formed (with the refined type identifier v replaced by the argument's identifier x) and produce a constraint with a universal quantifier bound on x with type B, so that if the refinement holds then the constraint C holds. We want to ensure that for any refined argument x, whenever the refinement holds then C holds.

Pairs checking rule (WF-PAIR) is standard. Each of the pair's (e, e') components needs to check against the corresponding component in the pair type (τ_1, τ_2). Both checking constraints should hold as well.

Rules for let binders also follow the standard checking rules. In CHECK-LET, for the term $\textbf{let}\, x = e \textbf{ in } e'$ we synthesize the type of the bound term e. With the bound x and corresponding type τ_1 in context, we can now check e' against type τ_2. CHECK-LETREF takes into consideration the case where the bound term has a refined type and again, as in CHECK-ABSREF, we must take the refinement into consideration in the produced formula to guarantee that whenever the refinement holds then the general constraint C holds.

Records, Tables and Subsumption. The checking rule for records (CHECK-REC) mimics the subtyping rule for records (Section 4.2.3), if we are considering width subtyping. In the fact that we can type check a record term against a record type with fewer fields. What we need to guarantee is that the labels present in the record type are present in the record term with a term e that type checks against the corresponding type τ. Every

$$\text{CHECK-ABS}\;\frac{\Gamma, x:\tau_1;\Sigma \vdash e \Leftarrow \tau_2 \leadsto C}{\Gamma;\Sigma \vdash \lambda x.e \Leftarrow (x:\tau_1) \to \tau_2 \leadsto C}$$

$$\text{CHECK-ABSREF}\;\frac{\Gamma, x:\{v:B|\varphi\};\Sigma \vdash e \Leftarrow \tau_2 \leadsto C \quad \varphi' = [x/v]\varphi \quad \Gamma, x:B;\Sigma \vdash \varphi'}{\Gamma;\Sigma \vdash \lambda x.e \Leftarrow (x:\{v:B|\varphi\}) \to \tau_2 \leadsto \forall x:B.\varphi' \implies C}$$

$$\text{CHECK-PAIR}\;\frac{\Gamma;\Sigma \vdash e \Leftarrow \tau_1 \leadsto C_1 \quad \Gamma;\Sigma \vdash e' \Leftarrow \tau_2 \leadsto C_2}{\Gamma;\Sigma \vdash (e,e') \Leftarrow (\tau_1,\tau_2) \leadsto C_1 \wedge C_2}$$

$$\text{CHECK-REC}\;\frac{\begin{array}{c} i = 0 \ldots n \quad j = 0 \ldots m \quad m \leq n \\ \text{foreach j .} \quad l_j \in \overline{l_i} \wedge \Gamma;\Sigma \vdash e_j \Leftarrow t_j \leadsto C_j \end{array}}{\Gamma;\Sigma \vdash \{\overline{l_i = e_i}\} \Leftarrow \{\overline{l_j : \tau_j}\} \leadsto \overline{C_j}}$$

$$\text{CHECK-RECREF}\;\frac{\Gamma;\Sigma \vdash \{\overline{l = e}\} \Leftarrow \{\overline{l:\tau}\} \leadsto C \quad \varphi' = [\{\overline{l = e}\}/v]\varphi \quad \Gamma;\Sigma \vdash \varphi'}{\Gamma;\Sigma \vdash \{\overline{l = e}\} \Leftarrow \{v:\{\overline{l:\tau}\}|\varphi\} \leadsto \varphi' \wedge C}$$

$$\text{CHECK-LET}\;\frac{\Gamma;\Sigma \vdash e \Rightarrow \tau_1 \leadsto C_1 \quad \Gamma, x:\tau_1;\Sigma \vdash e' \Leftarrow \tau_2 \leadsto C_2}{\Gamma;\Sigma \vdash \mathbf{let}\, x = e \;\mathbf{in}\; e' \Leftarrow \tau_2 \leadsto C_1 \wedge C_2}$$

$$\text{CHECK-LETREF}\;\frac{\begin{array}{c} \Gamma;\Sigma \vdash e \Rightarrow \tau_1 \leadsto C_1 \quad \tau_1 = \{v:B|\varphi\} \\ \Gamma, x:\tau_1;\Sigma \vdash e' \Leftarrow \tau_2 \leadsto C_2 \quad \varphi' = [x/v]\varphi \quad \Gamma, x:B;\Sigma \vdash \varphi' \end{array}}{\Gamma;\Sigma \vdash \mathbf{let}\, x = e \;\mathbf{in}\; e' \Leftarrow \tau_2 \leadsto C_1 \wedge \forall x:B.\varphi' \implies C_2}$$

$$\text{CHECK-TABLE}\;\frac{n = n' \quad k = k' \quad r = r'}{\Gamma;\Sigma \vdash \mathbf{new}\; Table[n]\{k|r\} \Leftarrow Table[n']\{k'|r'\} \leadsto true}$$

$$\text{CHECK-TABLEREF}\;\frac{\Gamma;\Sigma \vdash \mathbf{new}\;\tau \Leftarrow B \leadsto C \quad \varphi' = [\mathbf{new}\;\tau/v]\varphi \quad \Gamma;\Sigma \vdash \varphi' \quad pf = P(\Gamma)}{\Gamma;\Sigma \vdash \mathbf{new}\;\tau \Leftarrow \{v:B|\varphi\} \leadsto C \wedge pf \implies \varphi'}$$

$$\text{CHECK-SUB}\;\frac{\Gamma;\Sigma \vdash e \Rightarrow \tau' \leadsto C_1 \quad \Gamma;\Sigma \vdash \tau' <: \tau \leadsto C_2}{\Gamma;\Sigma \vdash e \Leftarrow \tau \leadsto C_1 \wedge C_2}$$

Figure 4.10: Checking Rules

checking constraint between the label's terms and types must hold. When checking a record against a refined record type (CHECK-RECREF), the record is checked against the record type and we take the refinement φ into consideration by adding it to the final constraint by a conjunction.

A table expression is simply the wrapping of the table type with the **new** construct. The checking rule, CHECK-TABLE, needs to guarantee that both table types are equal, by ensuring that both names n and n', and the pairs of rows k/k' and r/r' are equal (and that all these constructs are well-formed). For two rows to be equal, they do not have to have the same order in the collection of labels and types, as long as they have the same collection of labels with the same types. Any name, type, row, or type variable present in the table cannot be free (must be in context).

When type checking a table against a refinement type (CHECK-TABLEREF), we type check the table against the base type and add the refinement to the final constraint. Note that we add the implication $pf \implies \varphi'$ to the constraint and not only φ. We try to gather as much information as we can from the context to verify the refinement, pf = $P(\Gamma)$ denotes a conjuntion of the refinements present in context (with the refinement identifier's replaced by the variable's identifier). For example, if we are trying to verify the refinement $x > 0$ and we have the variable x in context with type $\{v : Int | v = 1\}$, we can produce the contraint $x = 1$ so that we can verify $x > 0$, i.e. $x = 1 \implies x > 0$.

The last rule is the connection between the check and synth modes. The subsumption rule (CHECK-SUB) deals with terms that are not checkable in our system by synthesizing a type for the term and verifying that the synthesized type is a subtype of the initial type. This means that we can use a term of this type as having the initial one.

4.3.2.2 Synthesis Rules

Now that we have seen the checking rules, we will see the synthesis rules corresponding to judgement $\Gamma;\Sigma \vdash e \Rightarrow \tau \rightsquigarrow C$. As before, we start with schemes and then types.

$$\text{SYN-TAPP}\frac{\Gamma;\Sigma \vdash e \Rightarrow \forall X.S \rightsquigarrow C}{\Gamma;\Sigma \vdash e[\tau] \Leftarrow [\tau/\mathrm{X}]S \rightsquigarrow C} \qquad \text{SYN-RAPP}\frac{\Gamma;\Sigma \vdash e \Rightarrow \forall_R R.S \rightsquigarrow C}{\Gamma;\Sigma \vdash e[r]_R \Leftarrow [r/\mathrm{R}]S \rightsquigarrow C}$$

$$\text{SYN-NAPP}\frac{\Gamma;\Sigma \vdash e \Rightarrow \forall_N N.S \rightsquigarrow C}{\Gamma;\Sigma \vdash e[n]_N \Leftarrow [n/\mathrm{N}]S \rightsquigarrow C} \qquad \text{SYN-LAPP}\frac{\Gamma;\Sigma \vdash e \Rightarrow \forall_L L.S \rightsquigarrow C}{\Gamma;\Sigma \vdash e[l]_L \Leftarrow [l/\mathrm{L}]S \rightsquigarrow C}$$

Figure 4.11: Scheme Synthesis Rules

The rules for synthesis in schemes are present in Figure 4.11. Let us explain the rule SYNTH-TAPP for type application since the other polymorphic applications naturally follow the same reasoning. For type applications, we want to synthesize the type of $e[\tau]$. The term to which we apply τ must be a type abstraction term ($\Lambda X.e$), so we synthesize the type of e and obtain type $\forall X.S$, which is a universal quantifier on type variables bound on X, and a constraint C. To obtain the final application type we replace the quantifier's bound X by τ and the final constraint is C.

The synthesis rules to synthesize a type τ for regular terms (not polymorphic) are given in Figure 4.12. Note that we said that normally we synthesize types for destructors but we will also show rules for pairs, records, and tables, since our language is small and we have enough information to synthesize the type.

Variables, Operations, Booleans and Integers. We start with variables and operations (SYN-VAR and SYN-OPS). In the environment Γ we keep both variable's/operation's identifiers id and the corresponding types τ. This means that we synthesize the type τ by doing a lookup in Γ with the variable/operation id. Type τ always holds (constraint true).

$$\text{SYN-VAR}\ \frac{\Gamma(x) = \tau}{\Gamma;\Sigma \vdash x \Rightarrow \tau \rightsquigarrow \mathit{true}} \qquad \text{SYN-OPS}\ \frac{\Gamma(op) = \tau}{\Gamma;\Sigma \vdash op(\overline{e}) \Rightarrow \tau \rightsquigarrow \mathit{true}}$$

$$\text{SYN-BOOL}\ \frac{\Gamma;\Sigma \vdash b : \mathit{bool}}{\Gamma;\Sigma \vdash b \Rightarrow \{v : \mathit{Bool} \mid v = b\} \rightsquigarrow \mathit{true}} \qquad \text{SYN-INT}\ \frac{\Gamma;\Sigma \vdash i : \mathit{int}}{\Gamma;\Sigma \vdash i \Rightarrow \{v : \mathit{Int} \mid v = i\} \rightsquigarrow \mathit{true}}$$

$$\text{SYN-ABS}\ \frac{\Gamma, x : \tau_1;\Sigma \vdash e \Rightarrow \tau_2 \rightsquigarrow C}{\Gamma;\Sigma \vdash \lambda x : \tau_1 . e \Rightarrow (x : \tau_1) \rightarrow \tau_2 \rightsquigarrow C}$$

$$\text{SYN-APP}\ \frac{\Gamma;\Sigma \vdash e \Rightarrow (x : \tau_1) \rightarrow \tau_2 \rightsquigarrow C_1 \quad \Gamma;\Sigma \vdash e' \Leftarrow \tau_1 \rightsquigarrow C_2}{\Gamma;\Sigma \vdash e\,e' \Rightarrow [e'/x]\tau_2 \rightsquigarrow C_1 \wedge C_2}$$

$$\text{SYN-ANNOT}\ \frac{\Gamma;\Sigma \vdash e \Leftarrow \tau \rightsquigarrow C}{\Gamma;\Sigma \vdash e : \tau \Rightarrow \tau \rightsquigarrow C} \qquad \text{SYN-PAIR}\ \frac{\Gamma;\Sigma \vdash e_1 \Rightarrow \tau_1 \rightsquigarrow C_1 \quad \Gamma;\Sigma \vdash e_2 \Rightarrow \tau_2 \rightsquigarrow C_2}{\Gamma;\Sigma \vdash (e_1, e_2) \Rightarrow (\tau_1, \tau_2) \rightsquigarrow C_1 \wedge C_2}$$

$$\text{SYN-PROJ1}\ \frac{\Gamma;\Sigma \vdash e \Rightarrow (\tau_1, \tau_2) \rightsquigarrow C}{\Gamma;\Sigma \vdash e.1 \Rightarrow \tau_1 \rightsquigarrow C} \qquad \text{SYN-PROJ2}\ \frac{\Gamma;\Sigma \vdash e \Rightarrow (\tau_1, \tau_2) \rightsquigarrow C}{\Gamma;\Sigma \vdash e.2 \Rightarrow \tau_2 \rightsquigarrow C}$$

$$\text{SYN-REC}\ \frac{i = 0 \ldots n \quad \Gamma;\Sigma \vdash e_i \Rightarrow \tau_i \rightsquigarrow C_i}{\Gamma;\Sigma \vdash \{\overline{l = e}\} \Rightarrow \{v : \{\overline{l : \tau}\} \mid \mathit{ref}(v)\} \rightsquigarrow \overline{C_i}}$$

$$\text{SYN-RPROJ}\ \frac{\Gamma;\Sigma \vdash e \Rightarrow \{\overline{l : \tau}, l_i : \tau_i, \overline{l : \tau}\} \rightsquigarrow C \quad l = l_i}{\Gamma;\Sigma \vdash \pi_l\, e \Rightarrow \tau_i \rightsquigarrow C}$$

$$\text{SYN-TABLE}\ \frac{}{\Gamma;\Sigma \vdash \mathbf{new}\,\tau \Rightarrow \tau \rightsquigarrow \mathit{true}}$$

Figure 4.12: Synthesis Rules

For boolean and integer terms (SYN-BOOL and SYN-INT), we produce a more specific type other than the standard bool/int types. To allow us to have more information in the type we synthesize, we produce a refinement type where the base type is the standard type (e.g. int) and then we add an equality refinement that tells us the exact value of the term (e.g. v = 3). For example, for term 3 the type would be $\{v : \mathit{int} \mid v = 3\}$. Stating that the term is not only an integer but also that it has the value 3 (and the same for boolean values).

Abstraction, Application and Annotation. SYN-ABS defines how to synthesize a type for an annotated abstraction $\lambda x : \tau . e$. With the argument's identifier and type in context, we synthesize the type of the function's body (τ_2) to produce a dependent function type.

For applications $e\,e'$, SYN-APP synthesizes the type of e which must be a function so that it can type check the application's argument against the function's argument type. A conjunction of the constraints from the checking and synthesis is the final constraint. Thus, the final application type is the function's body type with the function's argument identifier replaced by the application's argument.

Finally, the type of an annotation (SYN-ANNOT) is the type τ annotating the term, after checking that the term does have type τ.

Pairs, Records, Projections, Tables. Let us begin with the rules of pairs and pair projection. SYN-PAIR synthesizes the types of pairs (e_1, e_2) by synthesizing the types of each pair component thus producing a pair type. To project either the first (SYN-PROJ1) or second component (SYN-PROJ2), we synthesize the type of the term being projected from and obtain a pair type. Then, from the pair type, project either the first or second component's type, respectively.

Now, for records and record projection. The straightforward way of synthesizing the type of a record $\{\overline{l = e}\}$, is by traversing the collection of labels l and terms e, synthesizing a type for each e. To obtain the record type $\{\overline{l : \tau}\}$, with the same labels tied to the respective term's type. We took this a step forward by producing a refined record type in which we transpose the refinements of each label's type to the refinement of the overall record (identified by ref(v)). So that we can have more information available in the overall record type.

Let us see an example. Consider the record type $\{salary : \{v : int | v > 0\}, active : \{v : bool | v = true\}\}$, we want to transpose the available information that we have on each type to the overall type by producing the refined record type $\{v : \{salary : \{v : int | v > 0\}, active : \{v : bool | v = true\}\} | v.salary > 0$ and $v.active = true\}$. This is extra information that we can use, for example, on the SMT encoding. Record projection $\pi_l\, e$, SYN-RPROJ, synthesizes the record type of e and takes the projection label l to obtain the corresponding type τ. For tables with $\mathbf{new}\,\tau$, the type is the one already in the term (τ).

4.3.2.3 Algorithmic Type Checking

In this section, we have presented the declarative rules for type checking, both checking and synthesis rules. We now explain how we translate it into an algorithm.

The type checking judgments were defined as a relation between a context $(\Gamma;\Sigma)$, a term e, type τ and constraint C. Operationally, we implement the checking algorithm as receiving as inputs the term e and type τ, and having the output C. For the synthesis rules, only e is an input and both type τ and constraint C are outputs. Constraint C is drawn from the contraint language and relates to the refinements. Constraint C must be checked to confirm the type checking result (`C ==> e` $\Leftarrow \tau$ `and C ==> e` $\Rightarrow \tau$), if C holds then term e type checks against type τ or synthesizes type τ. To check the validity of the constraint, we again encode it (Section 4.5.1) and discharge it into an SMT Solver (like we do for the subtyping constraints in Section 4.2.3).

4.3.3 Operational Semantics

The semantics of a language defines how terms are evaluated into values. For this language, the values are given in Figure 4.13. As values, we have the standard ones of functions, numbers, and booleans. Pairs and records are also a part of the set of values, plus all four polymorphic abstractions. Table$[\alpha]$ defines a table identifier, a reference for table with name α, where name α is a fully instantiated name (it is not a name variable) and uniquely identifies the table (we cannot have more than one table with the same name).

$v, u ::= num \mid bool$	Basic Values
$\mid \lambda x : \tau . e \mid \lambda id . e$	Functions
$\mid (v, u)$	Pair
$\mid \{\overline{l = v}\}$	Record
$\mid \text{Table}[\alpha]$	Table Identifier
$\mid \Lambda X . e$	Type Abstraction
$\mid \Lambda_R R . e$	Row Abstraction
$\mid \Lambda_N N . e$	Name Abstraction
$\mid \Lambda_L L . e$	Label Abstraction

Figure 4.13: Language Values

4.3.3.1 Operational Semantics Rules

As for the rules of how a term is evaluated into a value, we show in Figure 4.14 the operational semantics given in a style of SOS rules (where the behavior of a term is defined by the behavior of its subterms). We have a small-step, call-by-value operational semantics, which means that we show every step from the initial term to obtaining a value and that on function application we evaluate the argument even if it is not used in the body. We also adopt a left-to-right evaluation strategy.

We define a new evaluation environment DB ::= DB, α_τ | $\emptyset$. DB keeps track of all fully instantiated table names α and their respective type τ. This environment gives us a store for the existing tables for the evaluation of **new** τ and the native operations. The relation DB, e $\longrightarrow$ DB, e' denotes the evaluation relation where there is a possible one-step evaluation from term e to e' in context DB. Let us now analyze the rules.

Table and Operations. For tables, we have the evaluation rule EVAL-TABLE. We can only create a new table with type τ, where τ will have the form Table$[\alpha]\{k|r\}$, if the fully instantiated name α does not already exist in DB. In the case it does not exist, we add α_τ to our context and return a table identifier Table$[\alpha]$ for the table.

Native operations op($\overline{e}$), as we explained, contain boolean and integer operations, plus operations on tables. EVAL-OP1 and EVAL-OP2 evaluate these operations by first evaluating each operation argument $\overline{e}$ and keeping the corresponding values $\overline{v}$. Then, when all the arguments are evaluated, return the corresponding evaluation of the operation in context DB (i.e. $[[op(\overline{v})]]_{DB}$), which is a value v and the new context DB' which may have been changed by operation op.

$$\text{EVAL-OP1}\ \frac{DB, e \longrightarrow DB, e'}{DB, op(\overline{v}, e, \overline{e}) \longrightarrow DB, op(\overline{v}, e', \overline{e})} \qquad \text{EVAL-OP2}\ \frac{[[\, op(\overline{v}) \,]]_{DB} = DB', v}{DB, op(\overline{v}) \longrightarrow DB', v}$$

$$\text{EVAL-TABLE}\ \frac{\alpha \notin DB}{DB, \mathbf{new}\,\tau \longrightarrow (DB, \alpha_\tau), \text{Table}[\alpha]} \qquad \text{EVAL-ANNOT}\ \frac{DB, e \longrightarrow DB, e'}{DB, e : \tau \longrightarrow DB, e' : \tau}$$

$$\text{EVAL-LET1}\ \frac{DB, e_1 \longrightarrow DB, e_1'}{DB,\ \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \longrightarrow DB,\ \mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2}$$

$$\text{EVAL-LET2}\ \frac{DB, e_2 \longrightarrow DB, e_2'}{DB,\ \mathbf{let}\ x = v_1\ \mathbf{in}\ e_2 \longrightarrow DB, [v_1/x]e_2'}$$

$$\text{EVAL-RPROJ1}\ \frac{DB, e \longrightarrow DB, e'}{DB, \pi_l\, e \longrightarrow DB, \pi_l\, e'}$$

$$\text{EVAL-RPROJ2}\ \frac{DB, e_i \longrightarrow DB, e_i'}{DB, \pi_l\{\overline{l = v}, l_i = e_i, \overline{l = e}\} \longrightarrow DB, \pi_l\{\overline{l = v}, l_i = e_i', \overline{l = e}\}}$$

$$\text{EVAL-RPROJ3}\ \frac{l \in \overline{l_i} \quad i = 1 \ldots n}{DB, \pi_l\{\overline{l_i = v_i}\} \longrightarrow DB, v_l}$$

$$\text{EVAL-PAIR1}\ \frac{DB, e \longrightarrow DB, e'}{DB, e.1 \longrightarrow DB, e'.1} \qquad \text{EVAL-PAIR2}\ \frac{DB, e \longrightarrow DB, e'}{DB, e.2 \longrightarrow DB, e'.2}$$

$$\text{EVAL-PAIR'}\ \frac{DB, e_1 \longrightarrow DB, e_1'}{DB, (e_1, e_2) \longrightarrow DB, (e_1', e_2)} \qquad \text{EVAL-PAIR''}\ \frac{DB, e_2 \longrightarrow DB, e_2'}{DB, (v_1, e_2) \longrightarrow DB, (v_1, e_2')}$$

$$\text{EVAL-PAIR1'}\ \frac{}{DB, (v_1, v_2).1 \longrightarrow DB, v_1} \qquad \text{EVAL-PAIR2'}\ \frac{}{DB, (v_1, v_2).2 \longrightarrow DB, v_2}$$

$$\text{EVAL-APP1}\ \frac{DB, e_1 \longrightarrow DB, e_1'}{DB, e_1\, e_2 \longrightarrow DB, e_1'\, e_2}$$

$$\text{EVAL-APP2}\ \frac{DB, e_2 \longrightarrow DB, e_2'}{DB, v_1\, e_2 \longrightarrow DB, v_1\, e_2'} \qquad \text{EVAL-APP3}\ \frac{DB, e_3 \longrightarrow DB, e_3'}{DB, (\lambda id.e_3)\, v_2 \longrightarrow DB, [v_2/id]e_3'}$$

$$\text{EVAL-TAPP1}\ \frac{DB, e \longrightarrow DB, e'}{DB, e[\tau] \longrightarrow DB, e'[\tau]} \qquad \text{EVAL-TAPP2}\ \frac{DB, e \longrightarrow DB, e'}{DB, (\Lambda X \,.\, e)[\tau] \longrightarrow DB, [\tau/X]e'}$$

$$\text{EVAL-RAPP1}\ \frac{DB, e \longrightarrow DB, e'}{DB, e[r]_R \longrightarrow DB, e'[r]_R} \qquad \text{EVAL-RAPP2}\ \frac{DB, e \longrightarrow DB, e'}{DB, (\Lambda_R R \,.\, e)[r]_R \longrightarrow DB, [r/R]e'}$$

$$\text{EVAL-NAPP1}\ \frac{DB, e \longrightarrow DB, e'}{DB, e[n]_N \longrightarrow DB, e'[n]_N} \qquad \text{EVAL-NAPP2}\ \frac{DB, e \longrightarrow DB, e'}{DB, (\Lambda_N N \,.\, e)[n]_N \longrightarrow DB, [n/N]e'}$$

$$\text{EVAL-LAPP1}\ \frac{DB, e \longrightarrow DB, e'}{DB, e[l]_L \longrightarrow DB, e'[l]_L} \qquad \text{EVAL-NAPP2}\ \frac{DB, e \longrightarrow DB, e'}{DB, (\Lambda_L L \,.\, e)[l]_L \longrightarrow DB, [l/L]e'}$$

Figure 4.14: Operational Semantics

LET and Annotation. EVAL-LET1 and EVAL-LET2 denote the standard evaluation of a let construct. First, we evaluate the term e_1 bound to x and then evaluate e_2 (with any occurrence of x replaced by v_1). The rule of the annotation of a term with a type τ, EVAL-ANNOT, is simply evaluating the annotated term e.

Pairs and Records. Regarding pairs, rules EVAL-PAIR1 and EVAL-PAIR2 evaluate the term being projected from. After we obtain a pair expression, in which we must evaluate both components (rules EVAL-PAIR' and EVAL-PAIR"). Once we have a pair of values, we either project the first value or the second, for the first or second component's projection respectively (EVAL-PAIR1' and EVAL-PAIR2').

Rules EVAL-RPROJ1 and EVAL-RPROJ2 evaluate a record projection $\pi_l\, e$, by evaluating e, originating record $\{\overline{l = e}\}$. We then proceed by evaluating every term in the record's collection. Finally, we project the value v_l associated with label l in the record.

Application and Polymorphic Application. The last evaluation rules are for application, both polymorphic and non-polymorphic. For function application, EVAL-APP1 to EVAL-APP3, starts by evaluating the left term to obtain a function value and then evaluates the argument (since we are doing call-by-value). With the argument value, we evaluate the function's body e_2 with any occurrence of id replaced by v_2. The application rules for functions with an untyped argument naturally extend to the ones with a typed one.

Polymorphic applications apply types/rows/names/labels to the corresponding polymorphic abstractions. Let us just see the type application rules, EVAL-TAPP1 and EVAL-TAPP2, since the other polymorphic evaluations follow the same reasoning. To evaluate a type application $e[\tau]$, we first evaluate the left term to obtain the type abstraction $\Lambda X.e$. Once we have the type abstraction, we replace the bound type variable X with the application type τ.

4.3.4 Example: Employee Table Target Program

Remember the formulation of our running example, where an expert wants to specify the insertion of a column with label 'Salary' and integer type into 'Employee' table. In Section 4.2.4 we defined the specification of this synthesis problem as:

(told: $Table[Employee]\{Id : Int|\quad\}) \rightarrow (v : Int) \rightarrow Table[Employee]\{Id : Int|Salary : Int\}$

Where we specify the table's current structure, a value for the column we introduce and the desired table. In this section we have seen the target language, that will be used to gradually build the goal program. In this case, the script that inserts the column salary into 'Employee' table and also deals with the inherent data.

Let us try to imagine what the goal program can be. Note that often there are multiple forms of building programs with the same goal type. We want a program that satisfies the synthesis specification we defined. We will look at the specification's type structure and specify the program.

The specification type is a dependent function type which means that our program term will also be a function with two typed arguments, namely

$$\lambda told : Table[Employee]\{Id : Int| \quad \}.\lambda v : Int.e$$

The program's arguments keep the same identifiers 'told' and 'v', and also the argument types $Table[Employee]\{Id : Int| \quad \}$ and int. The expression e stands for an unknown term, which is the subproblem we will see now.

To solve the subproblem of finding a term to fill in for e, we use the fact that the shape of this term is restricted since it must have type $Table[Employee]\{Id : Int|Salary : Int\}$ (the output of the specification function). In Section 4.3.1 (target language's syntax) we mentioned that the target language offers native operations to modify tables, abstracted as ops. We purposely exemplified these operations with a polymorphic operation called 'insertColumn' (although we have many others - Section 5) that inserts a column into a table. That operation is exactly the one we need to fill in for e, but not before instantiating it because it is polymorphic.
The type of 'insertColumn' is the following:

$$\begin{aligned}&\forall T.\forall_N N.\forall_R keys.\forall_R rest.\forall_L L.\\&told : \{v : \text{Table}[N]\{keys|rest\}|\,\textbf{not}\ L\ \textbf{in}\ rest\} \rightarrow v : T \rightarrow \text{Table}[N]\{keys|L : T, rest\}\end{aligned}$$

To use 'insertColumn' in our function's body, we need to find suitable instantiations for the polymorphic variables. A sequence of polymorphic applications to 'insertColumn' will be the term we want to build.

Looking at the current table we can already identify a few of the instantiations. N representing the name is replaced by Employee, the column id:Int is the only column in the table and it is a key column. This means that 'keys' is instantiated to $id : int$ and 'rest' to the empty row. Finally, we want to insert a column with label salary and type int, replacing L and T, respectively. Applying these types to 'insertColumn' results in the term insertColumn[int][Employee][id:Int][ε][salary] with type:

$$\begin{aligned}&told : \{v : \text{Table}[Employee]\{Id : Int|\}|\,\textbf{not}\ salary\ \textbf{in}\ \varepsilon\} \rightarrow v : Int \rightarrow\\&\text{Table}[Employee]\{Id : Int|salary : Int\}\end{aligned} \tag{4.3}$$

To conclude the synthesis solution, we highlight that we need the function's body to give us the type of the desired table and not a function type. We apply the input table 'told' and value 'v' to the function 4.3 and obtain term (insertColumn[int][Employee][id:Int] [ε][salary])(told)(v) with type $Table[Employee]\{Id : Int|Salary : Int\}$, which is the desired output type. The final program that receives a table and a value and inserts a column into it can look like:

$$\begin{aligned}&\lambda told : Table[Employee]\{Id : Int| \quad \}.\lambda v : Int.\\&(\text{insertColumn}[int][Employee][id : Int][\varepsilon][salary])(told)(v)\end{aligned}$$

The term (insertColumn$[int][Employee][id : Int][\varepsilon][salary])(told)(v)$ replaces the expression e and we have reached the final solution. To verify that this program satisfies the specification, we can simply type check it.

4.4 Synthesis

Program synthesis defines a procedure that, based on a specification, searches the space of possible programs to build a program that satisfies the specification. In Section 4.2 we have seen the specification language and in Section 4.3 the target language. In this section, we present the synthesis procedure, which based on a richly-typed specification gradually builds a program with the terms in the target language to satisfy the initial specification.

We begin with the introduction of preliminary concepts in Section 4.4.1. Next, we present the declarative synthesis rules (Section 4.4.2) and algorithmic synthesis. We conclude with a set of unification rules (Section 4.4.3).

4.4.1 Preliminary Concepts

Type-directed synthesis approaches generally follow a proof-theoretic approach or, the so-called, proofs-as-programs approach. Where the initial problem is divided into subproblems by applying inference rules that match on the conclusion and whose premisses represent the subproblems (restricted on their type), until there is no subproblem left to solve.

Proof-search techniques can be applied since there is a close relation between types and propositions, and between programs and proofs, through the Curry-Howard correspondence. The Curry-Howard correspondence [33] states that logic propositions may be viewed as types and their proofs as well-typed programs, the proof of a proposition corresponds to a program with the corresponding type. This guarantees that if the synthesis procedure finds a program, then it satisfies the initial type (since it was guided by it).

Natural deduction systems allow us to reason about logic propositions and their connectives using a set of inference rules. We can define the rules into two sets. Introduction rules comprising rules that introduce connectives, and elimination rules that apply knowledge within the connectives. Doing proof-search within such a system requires back and forward reasoning since introduction rules are applied bottom-up and elimination rules top-down.

Natural deduction is not directly well-suited for proof search because of the back and forward reasoning. Sequent calculus embodies an evolution from natural deduction in the sense that every rule is ‘turned’ into a bottom-up rule.

4.4.1.1 Sequent Calculus

Sequent Calculus [25] introduces the sequent judgment $A_1,\ldots,A_n \vdash B_1,\ldots,B_k$, where the left part is called the antecedent and the right part the consequent, A and B are propositions. The propositions $B_1,\ldots,B_k$ are the ones to prove and assumptions $A_1,\ldots,A_n$ can be used to prove a goal (to the right of the turnstile $\vdash$).

The introduction and elimination inference rules (from natural deduction) become right and left rules, respectively. Right rules work on the consequent of the sequent (or right side). Left rules work on the assumptions of the antecedent (left side). Elimination rules are formulated into the application of assumptions which now can be used bottom-up and proof search can now only work bottom-up.

We will not go into detail on the inference rules of the sequent calculus since it is not the focus of our work. Let us just see the right and left rule for implication (corresponding to the function type $\tau_1 \rightarrow \tau_2$). A, B, and P are propositions and Γ stands for a collection of assumptions.

The right rule →R works on the consequent, to prove implication $A \rightarrow B$ we can decompose it into proving B based on assumption A.

$$\rightarrow\!\mathrm{R}\,\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}$$

The left rule →L works on the antecedent assumptions to prove goal proposition P. We can prove P with assumption $A \rightarrow B$ by proving A and then proving P with assumption B.

$$\rightarrow\!\mathrm{L}\,\frac{\Gamma \vdash A \quad \Gamma, B \vdash P}{\Gamma, A \rightarrow B \vdash P}$$

As we can see, this calculus is more amenable to proof search procedures. Even though there still exists a lot of non-determinism in the choice of inference rules to use during a proof. Let us discuss it next.

4.4.1.2 Non-determinism in Proof Search

The non-determinism regarding the choices of inference rules may be categorized according to the type of choice. We highlight a few choice types from the categorization in [25] and relate to helpful techniques, as follows:

- Conjunctive Choices. When multiple subgoals have to be proved, it does not matter which ones are proved first, as long as we proved all of them. Note that for a few cases, some subgoals may interfere with others, but we are stating this for independent subgoals.

- Disjunctive Choices.

  - When there are multiple left and right rules that can be applied and we do not know which one to apply, we can use the focusing technique which states that a few rules may be applied first.
  - With the multiple possibilities of choice, it is a case of choosing the right path that enables us to conclude the proof. Some paths may not help the proof and backtracking may help with returning to a previous point of choice.

- Universal Choices. For universal quantifiers, its right rule is ∀R:

$$\forall R\,\frac{\Gamma \vdash [a/x]A}{\Gamma \vdash \forall x.A}$$

  The bound x in the quantifier is replaced by a new parameter a which is universally quantified in the formula. In this case, the choice of parameter does not matter as long as it is new in the formula.

- Existential Choices. Note that as we said previously we do not have existential quantifiers in our syntax, but when viewing a universal quantifier through its left rule, we view it as an existential choice. For universal quantifiers, its left rule is ∀L:

$$\forall R\,\frac{\Gamma, [a/x]A \vdash P}{\Gamma, \forall x.A \vdash P}$$

  To apply the assumption $\forall x.A$ to prove the goal P, we need to instantiate the bound x with a new parameter in the formula which is existentially quantified in it. This represents a point of non-determinism, if we viewed the formula as a universal quantifier type, this would be equal to instantiating it with a type and there are many possibilities. The choice of existential parameters may be solved by a unification technique.

We will now present some context into the techniques we mentioned (focusing, backtracking, and unification).

Focusing. Focused proof search or focusing was introduced by Andreoli [4]. This technique permits the removal of non-essential non-determinism in proof search by characterizing rules into invertible and non-invertible and imposing an order on their execution. In Invertible rules, we can prove the conclusion from its premises and also prove the premises from the conclusion, hence the term invertible.

Andreoli [4] noted that the connectives of linear logic may be categorized as synchronous or asynchronous. Asynchronous connectives have invertible right or left rules and synchronous is dual. Asynchronous connectives may be decomposed eagerly in proof-search since the order in which they are applied does not matter and does not change the proof. After there are no more invertible rules to apply, one may apply the non-invertible rules.

Thus, focusing divides the process of proof search into inversion and focusing. In the inversion phase ($\Uparrow$), all invertible rules may be applied eagerly in any order. When there are no more invertible rules to apply, the focusing phase ($\Downarrow$) begins. All non-invertible rules are applied until the proof is complete, fails, or a proposition with an invertible rule is found, thus the inversion phase re-starts.

Focusing introduces four judgments. For a context Γ with assumptions and A, B and P as propositions. Left and right inversion, $\Gamma \Uparrow \vdash A$ and $\Gamma \vdash \Uparrow A$, respectively. The symbol $\Uparrow$ indicates the context or proposition being inverted. Left and right focusing: $\Gamma, \Downarrow A \vdash P$ and $\Gamma \vdash \Downarrow P$. The symbol $\Downarrow$ shows the proposition under focus.

Focusing will be our main synthesis technique, in which we will express the declarative synthesis rules in Section 4.4.2.

Backtracking. When faced with disjunctive choices, e.g. $A \vee B$, we have to choose one path to continue the proof. Imagine that we choose A and at the end, we did not prove our goal yet and there are no more rules to apply, in that case, we would like to go back to $A \vee B$ and choose B instead. To be able to achieve that, we can use a backtracking mechanism that when the proof fails is able to return to a previous point of choice, in the proof, and continue on with the option that was not chosen previously (B in our example). We will implement a mechanism of backtracking which is explained Section 4.5.2.

Unification. In the presence of universal quantifiers, when focusing we need to choose an instantiation for the bound variable, since we are applying the given quantifier. There are multiple choices for instantiation and trying every choice may not be the way to go.

Unification allows removing some of the existential non-determinism by delaying the instantiation of variables. When faced with a universal quantifier in the focusing phase, one may simply instantiate the bound variable with a new existential variable. During proof-search, constraints in the form $A \sim B$ (A is unified with B), are collected between propositions with existential variables and other propositions as to later decide on a suitable instantiation given the collection of constraints.

The unification constraints $A \sim B$ will be present in the synthesis rules in Section 4.4.2 and in Section 4.4.3 we present our unification declarative rules.

4.4.2 Synthesis Rules

We now present the synthesis rules. Our synthesis mechanism approaches synthesis in a proof-theoretic way of proofs-as-programs, where we explore the type's shape to gradually build a program. The synthesis judgments are inversion and focusing judgments, with the exception that we do not have right non-invertible or left invertible rules, so we will only invert on the right and focus on the left of the sequent.
The synthesis judgements are:

- $\boxed{\Gamma;\Sigma;C \vdash \tau \rightsquigarrow e;C'}$ Invert type τ to obtain term e with constraint C' based on context $\Gamma;\Sigma$ and constraint C
- $\boxed{\Gamma,[x:\tau];\Sigma;C \vdash \sigma \rightsquigarrow e;C'}$ Focus on type τ (in Γ) to obtain term e with type σ and constraints C' based on context $\Gamma;\Sigma$ and constraint C

In these judgments, C represents a collection of constraints of two natures. The first is the constraints that follow the syntax in Section 4.2.3.1 and define constraints related to refinements. The second is constraints of the form $\tau_1 \sim \tau_2$ that represent a unification constraint between two types τ_1 and τ_2. We abstract these two kinds of constraints as C, since in the synthesis rules there is rarely the need to consider both separately. We will see the unification rules next in Section 4.4.3.

4.4.2.1 Inversion Rules

Inversion rules are defined by the judgment $\Gamma;\Sigma;C \vdash \tau \rightsquigarrow e;C'$. Let us first look at scheme inversion rules.

The rules for scheme inversion are given in Figure 4.15. Variable 'fv' denotes a fresh variable. When tagged as $fv_{_{Un}}$ denotes a fresh variable standing for a universal variable (bound in a universal quantifier).

$$\text{I-}\forall\ \frac{\Gamma;\Sigma,fv_{_{Un}};C \vdash [fv_{_{Un}}/X]S \rightsquigarrow e;C'}{\Gamma;\Sigma;C \vdash \forall X.S \rightsquigarrow \Lambda fv_{_{Un}}.e;C'} \qquad \text{I-}\forall_R\ \frac{\Gamma;\Sigma,fv_{_{Un}};C \vdash [fv_{_{Un}}/R]S \rightsquigarrow e;C'}{\Gamma;\Sigma;C \vdash \forall_R R.S \rightsquigarrow \Lambda_R fv_{_{Un}}.e;C'}$$

$$\text{I-}\forall_N\ \frac{\Gamma;\Sigma,fv_{_{Un}};C \vdash [fv_{_{Un}}/N]S \rightsquigarrow e;C'}{\Gamma;\Sigma;C \vdash \forall_N N.S \rightsquigarrow \Lambda_N fv_{_{Un}}.e;C'} \qquad \text{I-}\forall_L\ \frac{\Gamma;\Sigma,fv_{_{Un}};C \vdash [fv_{_{Un}}/L]S \rightsquigarrow e;C'}{\Gamma;\Sigma;C \vdash \forall_L L.S \rightsquigarrow \Lambda_L fv_{_{Un}}.e;C'}$$

Figure 4.15: Scheme Inversion Rules

We invert a scheme (rules I-$\forall$ to I-$\forall_L$) by removing the quantifier through instantiation. We replace the bound variable with $fv_{_{Un}}$ bound universally in the scheme and invert scheme S. This allows us to know, during the synthesis process, that the variable is universally quantified. The resulting expression is a polymorphic abstraction bound on the polymorphic variable, whose body is the expression e resulting from inverting the instantiated scheme.

The inversion rules for monomorphic types are given in Figure 4.16. Let us begin with tables (I-new). Independently from the structure of the table, regarding the collection of keys and other columns, we can only create a new table if in context (Γ) there is no other table with the same fully-instantiated name α. If there is not, we create a new table with type Table$[\alpha]\{k|r\}$.

For the record type $\{\overline{l_i:\tau_i}\}$, synthesizing a record (I-record) means that we create a record term with every label in the type and synthesize a term e_i corresponding to the inversion of each τ_i. The collection $\overline{C_i}$ denotes the constraints resulting from the inversion of each τ_i.

$$\text{I-new}\,\frac{\text{forallRows keys, rest. } \nexists x : \tau \in \Gamma \quad \text{such that } \tau = \text{Table}[\alpha]\{keys|rest\}}{\Gamma; \Sigma; C \vdash \text{Table}[\alpha]\{k|r\} \rightsquigarrow \textbf{new } \text{Table}[\alpha]\{k|r\}; C}$$

$$\text{I-record}\,\frac{i = 0 \ldots n \quad \text{for each i . } \Gamma; \Sigma; C \vdash \tau_i \rightsquigarrow e_i; C_i}{\Gamma; \Sigma; C \vdash \{\overline{l_i : \tau_i}\} \rightsquigarrow \{\overline{l_i : e_i}\}; \overline{C_i}}$$

$$\text{I-Pi}\,\frac{\Gamma, x : \tau_1; \Sigma; C \vdash \tau_2 \rightsquigarrow e; C'}{\Gamma; \Sigma; C \vdash (x : \tau_1) \rightarrow \tau_2 \rightsquigarrow \lambda x : \tau_1 . e; C'}$$

$$\text{I-Pair}\,\frac{\Gamma; \Sigma; C \vdash \tau_1 \rightsquigarrow e_1; C_1 \quad \Gamma; \Sigma; C \vdash \tau_2 \rightsquigarrow e_2; C_2}{\Gamma; \Sigma; C \vdash (\tau_1, \tau_2) \rightsquigarrow (e_1, e_2); C_1 \wedge C_2}$$

$$\text{I-reft}\,\frac{\Gamma; \Sigma; C \vdash B \rightsquigarrow e; (C_u, C_r) \quad \varphi' = [e/v]\varphi \quad \Gamma; \Sigma \vdash \varphi'}{\Gamma; \Sigma; C \vdash \{v : B|\varphi\} \rightsquigarrow e; (C_u, C_r \implies \varphi')}$$

Figure 4.16: Inversion Rules

To invert a dependent function type $(x : \tau_1) \rightarrow \tau_2$, we invert the return type τ_2 with the argument's identifier x and type τ_1 in context. The synthesized term is an abstraction with x and τ_1 plus the synthesized body term e. For non-dependent functions, we generate an id for the argument and also invert τ_2.

For pairs (I-Pair), we simply invert each of the component's types and then assemble a pair term with the synthesized terms. Finally, for refined types $\{v : B|\varphi\}$ (I-reft), we proceed by inverting the base type B. Only in this rule do we 'break' the abstraction of C and separate the constraints regarding unification (C_u) and the ones for refinements (C_r), that comes from inverting B. We check the well-formedness of the refinement φ' (where we replace the refinement identifier v by term e) and produce a constraint that ensures that every constraint produced while synthesizing term e with type B implies that the formula φ' holds ($C_r \implies \varphi'$). Which in practice means that not only e has type B but also the refined type $\{v : B|\varphi\}$ because it ensures that the refinement holds.

4.4.2.2 Focusing Rules

The focusing rules are defined by the judgment $\Gamma, [x : \tau]; \Sigma; C \vdash \sigma \rightsquigarrow e; C'$.

The rules for scheme focusing are given in Figure 4.17. Variable fv denotes a fresh variable and fv_{Sk} denotes an existentially quantified fresh variable. As we mentioned in Section 4.4.1, when we discussed the non-deterministic choices in proof-search, focusing on a scheme (rules FOC-$\forall$ to FOC-$\forall_L$) implies applying it, which we do by instantiating the bound variable. Since there are many options for the instantiation, we choose to delay it by replacing the bound variable with fv_{Sk} for later unification.

To prove the type τ we continue by inverting the previously bound scheme S (where the polymorphic variable is replaced by fv_{Sk}) and we give the scheme a new identifier x'.

To obtain term e, we focused on x′ with scheme type S, which we do not originally have in our context, but we have x with the universally quantified scheme S. That is why for the final expression e we replace any use of x' for the polymorphic application $x[fv_{Sk}]$ so that we obtain the type in which we focused on.

$$\text{FOC-}\forall\ \frac{\Gamma,[x':[fv_{Sk}/X]S];\Sigma;C\vdash\tau\rightsquigarrow e;C'}{\Gamma,[x:\forall X.S];\Sigma;C\vdash\tau\rightsquigarrow[x[fv_{Sk}]/x']e;C'}$$

$$\text{FOC-}\forall_R\ \frac{\Gamma,[x':[fv_{Sk}/R]S];\Sigma;C\vdash\tau\rightsquigarrow e;C'}{\Gamma,[x:\forall_R R.S];\Sigma;C\vdash\tau\rightsquigarrow[x[fv_{Sk}]_R/x']e;C'}$$

$$\text{FOC-}\forall_N\ \frac{\Gamma,[x':[fv_{Sk}/N]S];\Sigma;C\vdash\tau\rightsquigarrow e;C'}{\Gamma,[x:\forall_N N.S];\Sigma;C\vdash\tau\rightsquigarrow[x[fv_{Sk}]_N/x']e;C'}$$

$$\text{FOC-}\forall_L\ \frac{\Gamma,[x':[fv_{Sk}/L]S];\Sigma;C\vdash\tau\rightsquigarrow e;C'}{\Gamma,[x:\forall_L L.S];\Sigma;C\vdash\tau\rightsquigarrow[x[fv_{Sk}]_L/x']e;C'}$$

Figure 4.17: Scheme Focusing Rules

Focusing on types is represented in Figure 4.18. Rule I-Focus is the phase changing rule, in which type σ is atomic and we can no longer apply any inversion rule so we start the focusing phase. We focus on any type τ identified by x present in context to synthesize an expression with type σ.

$$\text{I-Focus}\ \frac{\Gamma,[x:\tau];\Sigma;C\vdash\sigma\rightsquigarrow C'}{\Gamma,x:\tau;\Sigma;C\vdash\sigma\rightsquigarrow e;C'}\qquad\text{FOC-}\tau\ \frac{}{\Gamma,[x:\tau];\Sigma;C\vdash\sigma\rightsquigarrow x;C\wedge\tau\sim\sigma}$$

$$\text{FOC-Pi}\ \frac{\Gamma,[x':\tau_2];\Sigma;C\vdash\sigma\rightsquigarrow e_2;C_2\quad\Gamma;\Sigma;C_2\vdash\tau_1\rightsquigarrow e_1;C_1}{\Gamma,[x:((x_1:\tau_1)\rightarrow\tau_2)];\Sigma;C\vdash\sigma\rightsquigarrow[(x\,e_1)/x']e_2;C_1}$$

$$\text{FOC-Reft}\ \frac{\Gamma,[x':B];\Sigma;C\vdash\sigma\rightsquigarrow C'\quad\varphi'=[x/v]\varphi\quad\Gamma;\Sigma\vdash\varphi'\quad pf=P(\Gamma)}{\Gamma,[x:\{v:B|\varphi\}];\Sigma;C\vdash\sigma\rightsquigarrow e;C\wedge pf\wedge\varphi'}$$

$$\text{FOC-Pair1}\ \frac{\Gamma,[x':\tau_1];\Sigma;C\vdash\sigma\rightsquigarrow e;C'}{\Gamma,[x:(\tau_1,\tau_2)];\Sigma;C\vdash\sigma\rightsquigarrow[x.1/x']e;C'}$$

$$\text{FOC-Pair2}\ \frac{\Gamma,[x':\tau_2];\Sigma;C\vdash\sigma\rightsquigarrow e;C'}{\Gamma,[x:(\tau_1,\tau_2)];\Sigma;C\vdash\sigma\rightsquigarrow[x.2/x']e;C'}$$

$$\text{FOC-Record}\ \frac{\Gamma,[x':\tau_i];\Sigma;C\vdash\sigma\rightsquigarrow e;C'}{\Gamma,[x:\{\overline{l:\tau},l_i:\tau_i,\overline{l:\tau}\}];\Sigma;C\vdash\sigma\rightsquigarrow[(\pi_{l_i}x)/x']e;C'}$$

Figure 4.18: Focusing Rules

The most general focusing rule is FOC-τ, where τ is already an atomic type, so all we can do is to add a unification constraint $\tau\sim\sigma$ to check whether we can unify type τ with type σ. This means that we can apply a term of type τ to obtain a term of type σ.

Rule FOC-Pi denotes that when we are focusing on a dependent function type $(x : \tau_1) \to \tau_2$, we first focus on the return type τ_2 with fresh identifier x', to check whether applying the function enables us to obtain a term of type σ. We then can find an expression with the argument's type τ_1 based on the necessary constraints C_2 when we focused on the return type. In the final term e_2 we replace any use of x' by the application $x\, e_1$ so that we obtain a term of type τ_2 and where e_1 represents an argument for the function with type τ_1.

When focusing on a refinement type $\{v : B|\varphi\}$, rule FOC-Reft continues to focus on the base type B and checks that the refinement φ' is well-formed in order to add it to the final constraint $(C \wedge \varphi')$. As we did in the type checking rules (Section 4.3.2) for checking a table against a refinement type (CHECK-TABLEREF), to obtain as much information as we can from the context Γ, we add a conjunction of all refinements $(pf = P(\Gamma))$ in context Γ to the final constraint $(C \wedge pf \wedge \varphi')$.

For pairs (τ_1, τ_2), rules FOC-Pair1 and FOC-Pair2 focus either on the first component or the second to achieve a term of type σ, any use of a component's type identified by x' is replaced (in term e) by a projection on the pair's component.

At last, to focus on a record $\{\overline{l : \tau}\}$ (rule FOC-Record), we non-deterministically choose a record type τ_i with label l_i to focus on. In the resulting term e, we replace any use of x' by a record projection on the record with label l_i $(\pi_{l_i}\, x)$.

4.4.2.3 Algorithmic Synthesis

The declarative synthesis rules present synthesis as a relation between a context $(\Gamma; \Sigma)$, constraints C/C' and types τ/σ.

In an algorithm, we implement inversion as receiving as inputs the constraints C and type τ to return expression e and constraint C' as output. When we change from the inversion phase to the focusing one, we select an identifier x and corresponding type τ to focus on. If the type we selected does not work, we continue by selecting another type from the context until we focus on a type that enables us to obtain an expression of the desired type or there is no type left to explore in context. The focusing phase receives a type τ with identifier x selected from the context, constraints C and type σ as input, to produce the output expression e and constraints C'. The backtracking mechanism, that will be explained in Section 4.5.2, enables us to return to a previous viable point when a synthesis path fails.

We can distinguish the constraints, as we mentioned, in two categories: unification and refinements constraints. The refinements constraints are drawn from the constraint language in Section 4.2.3.1. They will be encoded (Section 4.5.1) and discharged to an SMT Solver to confirm any results regarding refinements. The unification constraints $\tau_1 \sim \tau_2$, between types, are relayed to a unification algorithm (whose declarative rules are explained next in Section 4.4.3) to obtain a substitution for the polymorphic variables that are existentially quantified (which means that we want to instantiate it with a specific type). We go through the collection of unification constraints to produce a collection of substitutions, that we apply on the final expression e. So that every existentially quantified variable in e is instantiated with a type.

The output collection of constraints C' must be checked for satisfiability, both the refinements constraints hold and there is a possible substitution given the unification constraints, to confirm the synthesis result (`C' ==>` $\tau \rightsquigarrow e$). Meaning that if C' holds then expression e is synthesizable from type τ.

4.4.3 Unification

Unification [25] is a technique that finds suitable instantiations for existential variables such that two types match or unify. The unification constraint $\tau_1 \sim \tau_2$ states that type τ_1 unifies with type τ_2.

We define the unification judgement as $\vDash \tau_1 \sim \tau_2 : \theta$ denoting that τ_1 unifies with τ_2 given the unifier θ. A unifier θ is a collection of substitutions on existential variables. We will want to choose the most general unifier so that we are doing the choice of least commitment.

Given a collection of unification constraints $\overline{\tau_1 \sim \tau_2}$, we unify each type pair $\tau_1 \sim \tau_2$ and propagate the unifier to the next type pair to unify (rule unif), as to try to find a general unifier for a collection of constraints or find that it does not exist.

$$\text{unif}\,\frac{\vDash [\theta_{i-1}]\tau_i \sim [\theta_{i-1}]\tau_j : \theta_i}{\vDash \overline{\tau_i \sim \tau_j} : \overline{\theta_i}}$$

$[\theta_{i-1}]\tau_i$ denotes the application of the substitutions in the previous unifier θ_{i-1} to the to type τ_i being unified at the moment.

In the unification rules we will see next, $\theta_1 \circ \theta_2$ stands for unifier composition between unifiers θ_1 and θ_2. $[Y/X]\theta$ is the addition of a new substitution to unifier θ, where X is replaced by Y. [] is an empty substitution. X and Y are type variables. When tagged with 'Un' or 'Sk', they are universal or existential variables, respectively. When we are doing the unification $X_{_{Sk}} \sim X_{_{Sk}}$, we are trying to unify existential variable X with itself. Moreover, when unifying $X_{_{Sk}} \sim Y_{_{Sk}}$, we are trying to unify two different variables X and Y that are existentially quantified. The same goes for row variables R and K.

Let us now explain the rules for type, base type, and row unification. We do not show the unification rules for names and labels, since these are pretty straightforward, either two names unify or not, the same goes for labels.

Types. The rules for type unification are given in Figure 4.19. For types τ and σ, we unify two dependent function types by unifying the argument types with each other and the return types. Equally simple, unifying two pairs (rule U-Pair) amounts to unifying both first components and both second components.

Type variables always unify with themselves (rules U-Un and U-Sk). We defined that different existential type variables, e.g. X_{Sk} and Y_{Sk}, do not unify with each other to prevent unification cycles. For an existential type variable and a universal type variable, the existential one is instantiated with the universal one (rules U-Un/Sk and U-Sk/Un).

Finally, we can unify an existential type variable X_{Sk} with any type τ (rules U-Sk/τ and U-τ/Sk) if the variable is not free in τ ($fv(\tau)$ denotes all free type variables in τ). The goal is to prevent entering a unification cycle, e.g. $x \sim f(x)$, if we unify x with something that contains x, we are always introducing new existential variables.

$$\text{U-Pi}\ \frac{\vDash \tau_1 \sim \tau_2 : \theta_1 \quad \vDash \sigma_1 \sim \sigma_2 : \theta_2}{\vDash (x : \tau_1) \to \sigma_1 \sim (x : \tau_2) \to \sigma_2 : \theta_1 \circ \theta_2} \qquad \text{U-Pair}\ \frac{\vDash \tau_1 \sim \sigma_1 : \theta_1 \quad \vDash \tau_2 \sim \sigma_2 : \theta_2}{\vDash (\tau_1, \tau_2) \sim (\sigma_1, \sigma_2) : \theta_1 \circ \theta_2}$$

$$\text{U-Un}\ \frac{}{\vDash X_{Un} \sim X_{Un} : []} \qquad \text{U-Sk}\ \frac{}{\vDash X_{Sk} \sim X_{Sk} : []}$$

$$\text{U-Un/Sk}\ \frac{}{\vDash X_{Un} \sim Y_{Sk} : [X/Y]\theta} \qquad \text{U-Sk/Un}\ \frac{}{\vDash X_{Sk} \sim Y_{Un} : [Y/X]\theta}$$

$$\text{U-Sk/}\tau\ \frac{X \notin fv(\tau)}{\vDash X_{Sk} \sim \tau : [\tau/X]\theta} \qquad \text{U-}\tau\text{/Sk}\ \frac{Y \notin fv(\tau)}{\vDash \tau \sim Y_{Sk} : [\tau/Y]\theta}$$

Figure 4.19: Type Unification Rules

Base Types. Base types are unifiable through the rules in Figure 4.20. We draw the reader's attention to rules U-TAB and U-REC. For tables (U-TAB), to unify two tables, we unify each pair of names $n \sim n'$, keys $k_1 \sim k_2$ and columns $r_1 \sim r_2$. Records (U-REC) are unifiable by unifying every label l_i with a label l_j and unifying the corresponding types τ_i and τ_j. Note that there is no pre-defined order, a label l_i may unify with the first label in the other record or some other. As long as we find a suitable unification for the record types.

Row. Last, but not least, we have row unification given in Figure 4.21. There is not much to explain about empty row (rule U-Empty) and row variable's (rules U-Un/UnR to U-r/Sk) unification rules. Rule U-Empty is straightforward and for row variables, the rules follow the same pattern as for type variables (rules in Figure 4.19) but for row variables R and K and $fr(r)$ denoting all free row variables in r. One exception is that, for rows, we allow the unification between different existentially quantified row variables, which is essential for our synthesis mechanism so that we can do multiple applications of table operations that abstract (using row variables) over different table structures. Which makes it necessary to unify two existential rows.

The case for row's cons (U-Cons1 to U-Cons3) is slightly more challenging to express but not intellectually difficult to understand. We want to model the following idea, with three declarative rules, that the unification between two rows does not need to have a

$$\text{U-I}\ \frac{}{\vDash int \sim int : []} \qquad \text{U-B}\ \frac{}{\vDash bool \sim bool : []}$$

$$\text{U-TAB}\ \frac{\vDash n \sim n' : \theta_1 \quad \vDash k_1 \sim k_2 : \theta_2 \quad \vDash r_1 \sim r_2 : \theta_3}{\vDash Table[n]\{k_1 | r_1\} \sim Table[n']\{k_2 | r_2\} : \theta_1 \circ \theta_2 \circ \theta_3}$$

$$\text{U-REC}\ \frac{i = 0 \ldots n \quad j = 0 \ldots n \quad \vDash l_i \sim l_j : \theta_i \quad \vDash \tau_i \sim \tau_j : \theta_j}{\vDash \{\overline{l_i : \tau_i}\} \sim \{\overline{l_j : \tau_j}\} : \overline{\theta_i} \circ \overline{\theta_j}}$$

Figure 4.20: Base Type Unification Rules

pre-defined order. For example, for two rows $\{L_{Sk} : int, rest_{Sk}\}$ and $\{l_1 : int, l_2 : int\}$, we may want to unify L_{Sk} with l_1 or with l_2, then the unification for $rest_{Sk}$ would be dual.

We capture this reasoning by stating, in rule U-Cons1, that the labels that are explicit in both cons unify with each other and also both types and rows. The other alternative rule U-Cons2 is to state that to unify rows $\{l : \tau, r_1\}$ and $\{r_2\}$, there is a possible division of row r_2 defined by $r_2' \uplus r_2''$. Such that there is a part of r_2 (r_2'') that unifies with l, idependently of the order, and then we continue by unifying r_1 with the rest of r_2 (r_2') that was not unified with l. Rule U-Cons3 unifies the same rows in reverse order.

$$\text{U-Cons1}\ \frac{\vDash l_1 \sim l_2 : \theta_1 \quad \vDash \tau_1 \sim \tau_2 : \theta_2 \quad \vDash r_1 \sim r_2 : \theta_3}{\vDash l_1 : \tau_1, r_1 \sim l_2 : \tau_2, r_2 : \theta_1 \circ \theta_2 \circ \theta_3}$$

$$\text{U-Cons2}\ \frac{r_2 = r_2' \uplus r_2'' \quad \vDash r_1 \sim r_2' : \theta_1 \quad \vDash l : \tau \sim r_2'' : \theta_2}{\vDash l : \tau, r_1 \sim r_2 : \theta_1 \circ \theta_2}$$

$$\text{U-Cons3}\ \frac{r1 = r_1' \uplus r_1'' \quad \vDash r_2 \sim r_1' : \theta_1 \quad \vDash l : \tau \sim r_1'' : \theta_2}{\vDash r_1 \sim l : \tau, r_2 : \theta_1 \circ \theta_2}$$

$$\text{U-Empty}\ \frac{}{\vDash \varepsilon \sim \varepsilon : []} \qquad \text{U-Un/UnR}\ \frac{}{\vDash R_{Un} \sim R_{Un} : []}$$

$$\text{U-SkX/SkXR}\ \frac{}{\vDash R_{Sk} \sim R_{Sk} : []} \qquad \text{U-SkX/SkYR}\ \frac{}{\vDash R_{Sk} \sim K_{Sk} : [K/R]\theta}$$

$$\text{U-Un/SkR}\ \frac{}{\vDash R_{Un} \sim K_{Sk} : [R/K]\theta} \qquad \text{U-Sk/UnvR}\ \frac{}{\vDash K_{Sk} \sim R_{Un} : [R/K]\theta}$$

$$\text{U-Sk/r}\ \frac{R_{Sk} \notin fr(r)}{\vDash R_{Sk} \sim r : [r/R]\theta} \qquad \text{U-r/Sk}\ \frac{R_{Sk} \notin fr(r)}{\vDash r \sim R_{Sk} : [r/R]\theta}$$

Figure 4.21: Row Unification Rules

4.4.4 Example: Employee Table Synthesis

At last, we conclude our running example. With the goal in mind, which is to synthesize a program to insert an integer column 'salary' into the table of employees, let us remember the key elements we have formalized throughout the chapter.

After analyzing the specification language in Section 4.2, we have defined the synthesis specification as:

$$(\text{told: } Table[Employee]\{Id : Int|\quad\}) \rightarrow (v : Int) \rightarrow Table[Employee]\{Id : Int|Salary : Int\}$$

Which defines the current table, the desired table and the default value for the new column. Then, understanding the target language allowed us to write a potential program satisfying the specification. The program is:

$$\lambda told : Table[Employee]\{Id : Int|\quad\}.\lambda v : Int.$$
$$(\text{insertColumn}[int][Employee][id : Int][\varepsilon][salary])(told)(v)$$

Which receives the current table, default value v and applies the function's inputs to a built-in operation 'insertColumn', after being properly instantiated.

Now we show the reader how to synthesize this program, starting with the type specification. If we find a derivation, by applying the synthesis rules to the specification, that allows us to reach this final program, then it satisfies the specification and thus is our goal program. Note that applying rules in a different order or even different rules may lead to different programs equally satisfying the specification. Our derivation will be biased towards the program we defined.

Let us begin the inversion phase. We will consider the previous judgments without constraints and term e in it, so that the presentation is clearer. We will still refer to the term being constructed and collected constraints. The context contains a pre-defined library of operations. We want to invert the specification within contexts $\Gamma;\Sigma$.

$$\Gamma;\Sigma \vdash (\text{told: } Table[Employee]\{Id : Int|\quad\}) \rightarrow (v : Int) \rightarrow Table[Employee]\{Id : Int|Salary : Int\}$$

The only rule we can apply is I-Pi and we do so twice. We add both arguments to context, the term up to now is $\lambda told : Table[Employee]\{Id : Int|\quad\}.\lambda v : Int.?$ and we need to invert the function's return type to obtain the body term.

$$\Gamma, \text{told: } Table[Employee]\{Id : Int|\quad\}, v : Int; \Sigma \vdash Table[Employee]\{Id : Int|Salary : Int\}$$

We cannot apply rule I-new since we already have one employee table in context, so all we can do now is to switch into the focusing phase (rule I-Focus).

$$\Gamma, [insertColumn : IC]; \Sigma \vdash Table[Employee]\{Id : Int|Salary : Int\}$$

We focus on the polymorphic function 'insertColumn' with type IC that is present in the context. Note that this is a point of choice, we could choose anything from the context, but we know that we will need this function in particular.

$$\begin{aligned} IC = \quad & \forall T.\forall_N N.\forall_R keys.\forall_R rest.\forall_L L. \\ & told : \{v : \text{Table}[N]\{keys|rest\} \mid \textbf{not } L \textbf{ in } rest\} \rightarrow v : T \rightarrow \text{Table}[N]\{keys|L : T, rest\} \end{aligned}$$

The term up to now and using 'insertColumn' is $\lambda told : Table[Employee]\{Id : Int| \quad \}.\lambda v : Int.insertColumn$. We still need to find suitable instantiations and arguments for insertColumn. Focusing on a polymorphic function (rules FOC-∀ to FOC-$\forall_L$) means that we will perform an instantiation per quantifier, obtaining new existential variables. Function 'insertColumn' now has the new type IC'.

$$\begin{aligned} IC' = &\text{ told: } \{v : \text{Table}[N'_{Sk}]\{keys'_{Sk}|rest'_{Sk}\}| \textbf{ not } L'_{Sk} \textbf{ in } rest'_{Sk}\} \\ &\rightarrow v : T'_{Sk} \rightarrow \text{Table}[N'_{Sk}]\{keys'_{Sk}|L'_{Sk} : T'_{Sk}, rest'_{Sk}\} \end{aligned}$$

We will express the polymorphic instantiations using the existential variables now, which later, by unification, will be replaced by actual types ($\lambda told : Table[Employee]\{Id : Int| \quad \}.\lambda v : Int.insertColumn[T'_{Sk}][N'_{Sk}][keys'_{Sk}][rest'_{Sk}][L'_{Sk}]$).
Function insertColumn' with type IC', is not polymorphic anymore, so we focus on it.

$$\Gamma, [insertColumn' : IC']; \Sigma \vdash Table[Employee]\{Id : Int|Salary : Int\}$$

Focusing on insertColumn' (rule FOC-Pi) means that we will have two steps to perform. First, we focus on the return type to check if by applying the function we can obtain an expression of the type we need. Second, we invert the argument type to obtain an argument expression. We separate the two (numbers 1. and 2.) and start with focusing.

1. By focusing on the function's body we focus again on a function because the function with two arguments was defined by currying. So again we divide this in two steps (numbers 1.1. and 1.2.), focusing on the return type and inverting the argument (FOC-Pi).

   $$\Gamma, [x' : v : T'_{Sk} \rightarrow \text{Table}[N'_{Sk}]\{keys'_{Sk}|L'_{Sk} : T'_{Sk}, rest'_{Sk}\}]; \Sigma \vdash Table[Employee]\{Id : Int|Salary : Int\}$$

   1.1. When focusing on the return type, we cannot apply any other rule other than FOC-τ since we are focusing on an atomic type. Note that both the focus type and the goal type are tables so maybe we can prove our goal.

   $$\Gamma, [x'' : \text{Table}[N'_{Sk}]\{keys'_{Sk}|L'_{Sk} : T'_{Sk}, rest'_{Sk}\}]; \Sigma \vdash Table[Employee]\{Id : Int|Salary : Int\}$$

   We add the unification constraint below and proceed with inverting the argument.
   $\text{Table}[N'_{Sk}]\{keys'_{Sk}|L'_{Sk} : T'_{Sk}, rest'_{Sk}\} \sim Table[Employee]\{Id : Int|Salary : Int\}$

   1.2. We invert the argument's type to see if we can find an argument expression for the function application. So we invert the type T'_{Sk} for the new column.

   $$\Gamma; \Sigma \vdash T'_{Sk}$$

   Since the type is already atomic we go immediately to focusing (I-Focus).

   $$\Gamma, [v : Int]; \Sigma \vdash T'_{Sk}$$

We focus on the specification's argument v which is the value for the new column and add the unification constraint $Int \sim T'_{Sk}$ to check whether we can use v as our argument. With v as an argument, the program looks like:

$$\lambda told : Table[Employee]\{Id : Int| \quad \}.\lambda v : Int.$$
$$(insertColumn[T'_{Sk}][N'_{Sk}][keys'_{Sk}][rest'_{Sk}][L'_{Sk}])(v)$$

2. Finally, we invert the first argument of the specification.

$$\Gamma;\Sigma \vdash \{v : \text{Table}[N'_{Sk}]\{keys'_{Sk}|rest'_{Sk}\}| \textbf{ not } L'_{Sk} \textbf{ in } rest'_{Sk}\}$$

We apply I-reft to the refined type and invert the table base type. We know that the refinement is well-formed due to WF-IN and the refinement constraint $\text{C} \implies$ **not** L'_{Sk} **in** $rest'_{Sk}$ is produced, where C stands for a constraint that may result from inverting the table base type.

$$\Gamma;\Sigma \vdash \text{Table}[N'_{Sk}]\{keys'_{Sk}|rest'_{Sk}\}$$

Again, the type is atomic, so we continue by focusing (I-Focus):

$$\Gamma,[told : Table[Employee]\{Id : Int| \quad \}];\Sigma \vdash \text{Table}[N'_{Sk}]\{keys'_{Sk}|rest'_{Sk}\}$$

And add a unification constraint between the argument table in context and the table we are inverting: $Table[Employee]\{Id : Int| \quad \} \sim \text{Table}[N'_{Sk}]\{keys'_{Sk}|rest'_{Sk}\}$

Using this table as another argument the program now is:

$$\lambda told : Table[Employee]\{Id : Int| \quad \}.\lambda v : Int.$$
$$(insertColumn[T'_{Sk}][N'_{Sk}][keys'_{Sk}][rest'_{Sk}][L'_{Sk}])(told)(v)$$

To conclude the example, we solve the following collection of unification constraints:

$$Table[Employee]\{Id : Int| \quad \} \sim \text{Table}[N'_{Sk}]\{keys'_{Sk}|rest'_{Sk}\}$$
$$Int \sim T'_{Sk}$$
$$\text{Table}[N'_{Sk}]\{keys'_{Sk}|L'_{Sk} : T'_{Sk}, rest'_{Sk}\} \sim Table[Employee]\{Id : Int|Salary : Int\}$$

And obtain a consistent unifier substitution which is: $N'_{Sk} \sim Employee$; $L'_{Sk} \sim salary$; $T'_{Sk} \sim Int$; $keys'_{Sk} \sim Id : Int$; $rest'_{Sk} \sim \varepsilon$

We also apply the substitutions to the refinement constraint and obtain: **not** salary **in** ε. Since we did not find any other constraint (C = true) and we instantiated the existential variables. This constraint is trivially true.

We apply the unifier to the final program and get:

$$\lambda told : Table[Employee]\{Id : Int| \quad \}.\lambda v : Int.$$
$$(\text{insertColumn}[int][Employee][id : Int][\varepsilon][salary])(told)(v)$$

Which is exactly the program we specified initially and thus satisfies the specification.

4.5 Implementation Challenges

In this section, we present a few implementation challenges related to the technical approach that we have been explaining in this chapter. Namely, how to encode the constraints from the constraint language (Section 4.2.3.1) to discharge to an SMT Solver (Section 4.5.1), how do we implement a backtracking mechanism (Section 4.5.2) and restrictions/optimizations to the synthesis implementation. We have developed a prototype implementation of the synthesis framework which is written in OCaml.

4.5.1 SMT Encoding

Following the procedures of synthesis, subtyping, and type checking, we are faced with a collection of constraints that were gathered throughout their execution. If this collection of constraints is non-empty, to conclude that a type synthesizes a term, that is a subtype of another type or even that a term type checks against it, we need to check the satisfiability of the constraints.

4.5.1.1 SMT-LIB

We use an SMT Solver to check the satisfiability of first-order formulas regarding some background theories. Current SMT Solvers need formulas to be encoded in a special format defined by SMT-LIB. The SMT-LIB is a standard with the goal to define general guidelines on the languages of SMT Solvers and its background theories plus to gather a set of useful benchmarks, to aim research and development in SMT. Also to provide a form of comparing tools.

The general idea of the SMT-LIB syntax [6] is to find a common way of describing decision problems. The syntax is expressed in the form of common LISP's S-expressions, where we have the operation and then its arguments (e.g. (+ 2 3), for the sum of 2 and 3). There needs to be a manner of encoding variables, logical connectives, and boolean sorted formulas. As well as declare sorted constants with regard to some known background theories (e.g. integer arithmetic, real arithmetic, quantifier-free formulas, arrays with or without extensionality, etc...). Plus uninterpreted functions. Formulas are inserted by the command 'assert' and when all formulas have been expressed, one can check their satisfiability with some command such as 'check-sat'. A satisfiability result is returned: SAT for satisfiable, UNSAT for unsatisfiable and UNKNOWN for an unknown satisfiability.

These are very general guidelines, each SMT solver tool differs but must satisfy the standard. A recent release of the current SMT-LIB standard (version 2.6) is present in [6].

4.5.1.2 Approach

We have chosen to use Z3 [30], an SMT Solver, which provides a string-based interface. Thus, we will encode our constraints into a string formula in the SMT-LIB format and send it to Z3, in order to check its satisfiability.

```
1   (declare-datatypes D)
2   (declare-fun FUN)
3   (declare-const C)
4   (assert F)
5   (check-sat)
```

Listing 4.1: Z3 input

The general structure of our input to Z3 is given in Listing 4.1. Let us analyze each line of the input. Line 4 is the main one, here we assert the satisfiability of the constraint, defined by a formula F succeding the keyword 'assert'. We may express formulas in several assertions, which are then considered together.

Lines 1 to 3 are auxiliary definitions. In Line 3, we declare constants used in the formula and their sort (or type) (e.g. (declare-const a (Int))). SMT-LIB's background theories provide some useful types such as integers and booleans, but we will need some types for which a definition is non-existent. To declare new types (Line 1), we encode a datatype as D and declare it through (declare-datatypes D). We can then use the datatype as a constant's sort. All functions FUN (their input and output sort) are defined in Line 2 (e.g. (declare-fun f (Int) Int), for a function that receives and returns integer values). And, finally, the command to require an answer (SAT/UNSAT/UNKNOWN) from Z3 is given in the last line by check-sat (Line 5).

It is straightforward to declare functions, constants, and check for satisfiability (through the commands mentioned above). It is, of course, very specific to each formula we want to encode. Notwithstanding, next we highlight the encoding of types through the encoding of datatypes and a few details on the encoding of formulas. Keep in mind that even though the structure in Listing 4.1 is our general input structure to Z3, we only declare the constants, datatypes and functions needed by the formula being asserted.

Encoding types in Z3. For a few of our types, we do not already have a predefined manner of expressing them in SMT-LIB. To encode the type of pairs, records, and tables, we have defined a datatype (for each) that must be declared previous to its use in constants or formulas. The keyword 'declare-datatypes' is followed by the datatype definition.

Pair. First, we shall see the pair datatype. A pair is composed of two components, the first and the second, which may have several different types. There must be a form of projecting its first or second component. Thus, the encoding of the pair datatype is:

```
(declare-datatypes (T1 T2) ((Pair (mk-pair (first T1) (second T2)))))
```

For the pair (T1, T2), T1 and T2 are the pair's first and second components, respectively. Both components may be instantiated with any desired type. To project the pair's first component, in a formula, we use the 'first' label followed by a pair constant (for the second it is the same, using 'second'). Let us see an example:

```
1   (declare-datatypes (T1 T2) ((Pair (mk-pair (first T1) (second T2)))))
2   (declare-const p1 (Pair Int Int))
3   (assert (> (second p1) 20))
```

In Line 1, we declare the pair datatype to use in the pair constant p1 declaration, Line 2. The pair constant's sort is an instantiation of the pair datatype, which now is a pair of integers. We assert a formula that checks whether the pair's second component is bigger than 20, Line 3, where we project the second component of p1 by using the label 'second' (i.e. second p1).

Record. To encode records, we recall the syntax of this type. Type $\{\overline{l:\tau}\}$ stands for a record that represents a collection of pairs label l and type τ. We need to encode the collection of pairs label/type into the datatype. In addition, we must be able to project a type from a record based on a specific label. This information amounts to the encoding:

```
(declare-datatypes () ((r_l1_t1_..._ln_tn
                          (mk-l1_t1_..._ln_tn (l1 t1) ... (ln tn) ))))
```

In the datatype above, $l1\_t1\_...\_ln\_tn$ defines the datatype's name and $mk{-}l1\_t1\_...\_ln\_tn$ the constructor, identifying the record type $\{l_1 : \tau_1, ..., l_n : \tau_n\}$.

The items of the collection that define a record do not have a predefined order, which means that, we want to guarantee that there is only one datatype per type of record, independently of the order of its labels and types. To guarantee that, before defining the constructor of a record datatype, we order the content of the record by the labels in alphabetical order. This way we ensure that two records with the same content where their collections are ordered differently, will match against the same datatype (e.g. record types $\{a : Int, b : Int\}$ and $\{b : Int, a : Int\}$ both correspond to datatype's constructor $mk - a\_Int\_b\_Int$). Following the constructor, we define each label l and type t by a collection of (l t) definitions. Let us see an example.

```
1   (declare-datatypes () ((r_a_Int_b_Bool_
2                             (mk-r_a_Int_b_Bool_ (a Int) (b Bool) ))))
3   (declare-const r11 r_a_Int_b_Bool_)
4   (assert (= (a r11) (b r11)))
```

In Lines 2 and 3 we declare a datatype for the record type $\{a : Int, b : Bool\}$ and declare a constant $r11$ with that type. We assert a formula, in Line 4, to check that a projection on the record with label a (a $r11$) equals the projection on the same record with label b (b $r11$).

Table. Now we see the encoding of tables. Even though tables are formed by a collection of labels and types, also without a pre-defined order, we encode tables in a different manner from records. When comparing record types $\{a : Int, b : Bool\}$ and $\{a : Int, c : Int\}$, it is clear that these originate different datatypes and constants. If we compare table rows $\{a : Int, b : Bool\}$ and $\{a : Int, c : Int\}$, again we know that these represent different rows.

When we consider table rows with row variables, namely $\{a : Int, R_1\}$ and $\{a : Int, R_2\}$, we cannot say with total certainty that these rows are equal, if $R_1 = R_2$ we can, but row variables stand for an unknown collection. This means that we have to have a manner of encoding the concept of row variables and row membership. Let us see the datatype definition.

```
1   (define-sort Bag () (Array String Int))
2   (define-fun bag-union ((x Bag) (y Bag)) Bag
3               ((_ map (+ (String Int) Int)) x y))
4   (define-fun member ((b Bag) (e String)) Bool
5               (ite (> (select b e) 0) true false))
6
7   (declare-datatypes () ((Fields (mk-l1_t1 (l1 t1))
8                                  (        ...         )
9                                  (mk-l2_t2 (l2 t2)))))
10  (declare-datatypes ()
11      ((Table (mk-table (all (Bag)) (rest (Bag)) (fields Fields)))))
12
13  (declare-fun eq ((Table) (Table)) Bool)
14  (assert (forall ((t1 Table) (t2 Table))
15          (= (= (bag-union (all t1) (rest t1))
16          (bag-union (all t2) (rest t2))) (eq t1 t2))))
```

Tables are defined by the unique Table datatype in Line 11. The table datatype allows us to define table types where we keep the information on all labels (except the ones in the row variable) in 'all', the ones belonging to a row variable (if the table has any row variable) and we name it 'rest' in the datatype, plus all table constructors in 'fields' with datatype Fields.

We define an auxiliary sort named Bag (Line 1), that stores values in an unordered manner. The bag will map labels (defined as strings) to their multiplicity (Int) in the collection. The union of two bags is represented by the function bag-union (Lines 2 and 3). To know if a specific label belongs to the bag, we define a membership function called member (Lines 4 and 5) , which defines that a label belongs to a bag if its multiplicity is bigger than 0. The collection of all labels and labels belonging to a row variable have a Bag sort since we are not interested in their order, only in the content.

The Fields datatype (Lines 7 to 9), allows us to keep track of the constructors of different tables in case we need to know the type of a label. For each collection of labels/types, we add a new Fields constructor.

To conclude the definition of the table datatype, we define an equality notion between tables (Function eq in Line 13). We want to define the equality of tables based on what we know on the whole content of labels. Two tables are equal if they agree on the labels present both in the 'all' and 'rest' parts, which amounts to equality between bag unions (Lines 14 to 16).

Let us see an example on tables. Consider the tables: Table$[a]\{|l_1 : \tau_1, R_1|\}$ and

Table[a]$\{|l_2 : \tau_2, R_2\}$, which can be made equal if $R_1 = l_2 : \tau_2$ and $R_2 = l_1 : \tau_1$. Let us check their equality purely on the information we have now.

```
1   (declare-const t1 (Table))
2   (declare-const t2 (Table))
3   (assert (= (member (all t1) "l1") true))
4   (assert (= (member (all t2) "l2") true))
5   (assert (= (member (rest t1) "l1") false))
6   (assert (= (member (rest t2) "l2") false))
7   (assert (eq t1 t2))
8   (check-sat)
```

Assuming we declared the table datatype and all of its auxiliary definitions previously, we declare a constant for each table, namely t1 and t2 (Lines 1 and 2), respectively. We use the membership function to assert the current knowledge, we know that t1 has label l1 (Line 3) and so it cannot have label l1 in R_1 (Line 5). The same goes for label $l2$ in t2 (Lines 4 and 6).

We then check their equality (eq t1 t2 in Lines 7 and 8). The satisfiability result is SAT since there is a chance of the tables being equal if we consider having more labels in row variables R_1 and R_2. The tables are only considered different if we know for sure that the content belonging to both tables differs.

```
1   (declare-const rest_ (Bag))
2   (assert (= (rest t1) rest_))
3   (assert (= (rest t2) rest_))
4   (assert (= (member (rest t2) "l1") true))
5   (check-sat)
```

Imagine now that we come across more information and discover that both R_1 and R_2 are equal and that R_2 has label l1.

To add to the previous encoding, we create a bag constant named '*rest_*' (Line 1) and define that both table's 'rest' parts equal to this constant (Lines 2 and 3), to define the equality between both row variables. Then, we assert that l1 belongs to R_2 (Line 4).

At last, when we again check for satisfiability (Line 5), the result comes as unsatisfiable because we asserted that l1 could not belong to R_1, now we know that both rows are equal and l1 belongs to R_2. Which reveals that equality is not possible.

Encoding constraints in Z3. To encode constraints, we encode from the constraints to the expressions present in those constraints, by encoding a part through the encoding of its subparts in SMT-LIB syntax. There is not much to say about most of it due to its simplicity. We gather constant's, function's, datatype's information to declare and assemble the formula to assert gradually. We will only highlight here a few cases where the encoding deserves some further explanation.

Universal Quantifier. Constraint $\forall s : bty.c$ is encoded by encoding constraint c (decl), which we join to declaration (forall ((s bty)) decl).

By encoding c, we gather the identifiers and types of constants to declare, from which we remove s, since s is bound and it is not declared as a constant. We declare the datatype of the bound's variable base type bty, if it was not already declared.

Row Membership. To encode the formula l **in** r, we consider the possible types of rows. If we are considering the empty row, we simply encode it as 'false', since l cannot belong to an empty row ε. When considering the cons $\{l_1 : \tau_1, r_1\}$ case, either l is equal to l_1 or belongs to r_1 (e.g. (or (= l l_1) r'), where r' denotes the encoding of row r). In the case that r is a row variable, we resort to the member function of the table datatype, stating that l is a member of row variable r (e.g. (= (member r l) true)).

Pair. The first time we encounter a pair, we declare the pair datatype (as seen above), only once since the datatype does not change according to different pairs. We memorize each pair type as to only declare a constant per pair. For example, if we have two pairs of integers, we only declare one constant. We assign a new identifier to the pair if this one is not already a variable identifier, in which case we use the already defined identifier.

Formulas in the constraints are formed by expressions, which means that we can have pair expressions such as (2, 3) and (10, 7). These pairs are different from each other, and as we said, we only define one constant per pair type, since both have the type of pair of integers, they would be identified by the same constant, and thus deemed equal. We differentiate between pairs, when we have a pair expression (i.e. not identified by a variable), with the direct use of the pair datatype's constructor (mk-pair) followed by the pair's components. For example, the pairs (2, 3) and (10, 7) are encoded as (mk-pair 2 3) and (mk-pair 10 7), so Z3 can tell them apart.

As we mentioned above, we project the first or second component of the pair by the label 'first' or 'second', followed by the pair's identifier constant, or datatype constructor and component values (e.g. (first (mk-pair 2 3))).

Record. For each record type, we declare a record datatype. We keep each record type alphabetically ordered by label, as to know when a type is new and needs to be declared.

Again, we can have record expressions, which means that we can refer to the type by an identifier or by directly using the corresponding datatype constructor. For a record with label a and integer type, we have the constructor $mk - a\_Int$ and we could encode the record $\{a = 4\}$ as $(mk - a\_Int\, 4)$.

A label projection on a record, as we exemplified when we defined the datatype, is the label plus the constant identifier or record constructor and values (e.g. (a $(mk - a\_Int\, 4)$)).

Table. For tables, we declare the datatype once, but after going through the whole constraint since we collect the types of every table to declare as constructors in fields. We save every different alphabetically ordered table type to declare a constant per type. We join the key and other columns' information.

We also declare a constant with Bag sort per each different row variable and whenever we find a row variable in a table we connect with the general constant by asserting '(assert (= (rest t1)) *rest_*)', for a table t1 with label rest for the variable and a *rest_* constant. For example, for the table Table$[a]\{\,|R_1\,\}$, we would need to create a constant for R_1 and assert their equality.

At last, we can assert the membership of any label to the bag of all labels or the bag of labels in the row variable by using the member function (e.g. (= (member (rest t1) "l2") true)), for table t1 and label l2.

4.5.2 Backtracking

In Section 4.4.1 we understood that in proof search we are faced with multiple types of choices. We defined that disjunctive choice are the ones where we have multiple paths we can go to and if the one we choose does not succeed, we would like to return to the previous point of choice and choose differently.

To achieve that, we implemented a mechanism of backtracking, which allows us to start from a previous point of the synthesis when we get an error. Our backtracking mechanism consists of using a style of programming called continuation-passing style (CPS) together with a stack.

CPS is a style of programming in which functions receive what is called a continuation, a function, defining the next action to perform. The control is passed through the continuation. Continuations are functions that define the next step to perform, when the current action finishes, it returns the results by calling the continuation function with the results as arguments and performs the next step.

We also use a stack in which we can push and pop things from it. The stack together with CPS gives us a manner of expressing many possible programs. In CPS style we always define the next step as a continuation function. When we have multiple possibilities of next steps, the so-called disjunctive choices, we have different continuations. The stack is crucial for this in the sense that we keep a stack of continuations, along with the synthesis procedure, when we are faced with disjunctive choices we can follow any and push the others to the stack. Not only in the event of an error, we can terminate the current action and pop a continuation from the stack resuming from that point. As with the stack, we can discover multiple programs satisfying the specification, if the stack holds all the next possible continuations, if we explore all the content in the stack, we can find more than one program.

We can find many disjunctive choices during the synthesis procedure. For example, on focusing, when we choose a type to focus on, we could also continue searching in the context. Focusing on a pair or record consists of focusing on one specific component or label, having many other possibilities. Also in label unification, when unifying two collections that do not need to follow a specific order, the combination of possible unifications is wide. There are many other points of choice. Synthesis is an exploratory process that brings a lot of combinatory into the equation.

4.5.3 Restrictions and Optimizations

Synthesis is a combinatory problem that is prone to infinite cycles. When implementing the declarative rules into an algorithm, we can try to make some restrictions and optimizations so that we break a few of those cycles or make synthesis faster by optimizing in certain places. Let us see some restrictions and optimizations that we set in our synthesis implementation.

Skolem Unification. As to avoid infinite cycles, we avoid unifying different skolem variables ($X_{sk} \sim Y_{sk}$) with each other since we can get stuck in a cycle of infinite unifications. The only exceptions that we allow are between row variables and also between name variables, since we need to be able to unify these skolem variables with each other so that we can use the polymorphic table operations in a row (e.g. two insertColumn operations).

Goal Repetition. In the synthesis inversion phase, our goal is a type τ, which we decompose by applying invertible rules and proving subgoals. In between the switching between inversion and focusing phases, we may be left with goals to prove which we proved previously and that will not lead us to a different solution. We want to avoid proving the same goal multiple times, to achieve that we keep a store on already proved goals to prevent the synthesis procedure from the inversion of repeated goals.

Constraint Solving. We interleave the synthesis process with constant constraint solving so that we can find wrong solutions early. In the example in Section 4.4.4, we only solved the constraints in the end, when we had the final program already. That was only for presentation purposes, in reality, we solve the constraints regularly throughout the synthesis process to detect an inconsistent set of constraints and determine that we need to build another solution immediately.

5

Library of Operations

In this chapter, we present the library of built-in operations on tables. When we presented the syntax of terms, and target of the synthesis procedure, in Section 4.3.1, we explained that we defined a set of operations, both boolean and arithmetic, but also on tables.

We now focus on table operations. The goal is that, in a component-based synthesis style, we define a library of basic operations on tables that is expressive enough and so the synthesis process becomes a composition of a subset of the components present in this library.

Let us start by explaining the thought process behind the set of operations (or ideation phase) in Section 5.1, so that after we can see in detail each operation in Section 5.2.

5.1 Ideation Phase

We want to define a set of operations that spans real-world examples, that is why the process of designing the library started with an ideation phase to come up with the proper syntax and a concise set of useful operations.

The core task of the ideation phase was to develop a gradual example starting with the definition of a data schema and making small changes so that we could discuss the effects of each change. In the beginning, there was no pre-defined set of operations or syntax, the research process consisted of exploration and discussion which culminated in the syntax of both the specification and target language, but also the set of operations we now focus on.

Our key point of discussion was to find the smallest and most expressive set of data schema operations. We also focused on understanding how to deal with the inherent data and how to adapt any function that depends on the current schema. At last, we worked on defining a proper syntax plus exploring what might be an interesting set of refinements.

Example Analysis. An illustrative example resulting from the ideation phase is given in Listing 5.1. Consider that the syntax does not exactly reflect the syntax we defined previously and that create/update represent commands defining an idea of the action (i.e. create for creating a new table, update for altering the table) only for demonstration purposes. ‘Employee{name:string}’ stands for a table with the name ‘Employee’ and a column with label ‘name’ and string type.

```
 1 create Employee{name:string}
 2 update Employee{name:string, salary:int}
 3 update Employee{id:int, name:string, salary:int}
 4 update Employee{id:{v:int | key(v)}, name:string, salary: int}
 5 update Employee{id:{v:int | key(v)}, name:string, salary: {v:int|v > 0}}
 6 update Employee{id:{v:int | key(v)},firstName:string,salary:{v:int|v > 0}}
 7 update Employee{id:{v:int | key(v)}, firstName:string}
 8 update Employee{id:{v:int | key(v)}, firstName:string, startDate: Date}
 9 update Employee{id:{v:int | key(v)}, firstName:string, startDate: Date,
10                 endDate:{v:Date | v > startDate}}
11 update Employee{id:int, firstName:string,
12                 endDate:{v:Date | v > startDate}}
```

Listing 5.1: Illustrative example from the Ideation Phase

We first started by creating the Employee table with label name and type string (Line 1). The creation of a table does not need much more. Now we want to focus on alterations. We update the table by inserting a column with label salary and integer type (Line 2). If the table has any records, we want to ensure that we also provide a default value to fill the rows on this column.

Imagine that then we understand that we need to add an identifier *id* to the table (Line 3) but we did not ensure that the identifier values are unique, we need that if we want to use the *id* as the table’s primary key. So the next step is to mark the column *id* as a key by, for example, refining the type with a ‘key’ predicate (Line 4). This action can probably cause problems, depending on the data present in the column’s rows and the functions that depend on it. We need to adapt any function depending on the current data schema and define a manner of adapting the data. We may provide a default value or an update function to adapt the data in the column’s rows. The same goes for updating the type of salary to only allow for positive values in Line 5, we need to adapt the data and any dependencies.

Consider a renaming operation, where we want to change the label of a column, as in the example in Line 6, we want to change ‘name’ to ‘firstName’, in the case we want to add another name, e.g. last name. This is an operation that requires changing any functions that depend on it but since the data remains the same, all we need to do is to obtain the data from column ‘firstName’ instead of ‘name’.

Dropping a column is slightly more challenging. For example, in Line 7, to drop the salary column, there is the chance that this operation may not even be allowed, since there may be functions depending on this column or also foreign keys. It is crucial that we again understand the dependencies and if it is possible to adapt or the drop operation is simply not possible. In this line of thought consider Lines 8 to 12, we insert a startDate column and then an endDate column whose type is refined so that we cannot have end dates smaller than the start dates. In the last operation, where we want to delete the startDate column, this is not possible since there is a dependency on it. Unless, for example, we remove the dependency by updating the type.

We now conclude this example, but not without a final remark. We understood that we need to have an insertion operation but also to provide a value for the column's rows, that we need to be able to update a column's type and provide a manner of adapting the inherent data. We may also rename columns, in which case we can update the dependencies. At last, that dropping operations are not straightforward, it may be the case that the result of specifying a drop operation is that the operation is not possible.

5.2 Operations

The set of operations, based on the real-world example we explored, is given in Table 5.1. Due to the similarity to real database operations, e.g. SQL, we can easily compile these operations to real code. The synthesis process of orchestrating a combination of operations may result in a collection of real database operations. We not only found an expressive set of operations but also an extensible one, the way we implemented these operations as symbolic which are defined by their type and put into context, facilitates the process of adding new operations further down the line.

Each operation is expressed using the target language's syntax and also making use of the polymorphic feature of the language, we express polymorphic type signatures so that the operations can be applied to any table simply by instantiation. Applying the instantiated operation's with the right arguments allows us to obtain the desired table type. Let us see each type of operation present in Table 5.1.

Inserting. To insert a column into a table we defined the operation 'insertColumn'. The operation's arguments are the target table and a default value for the column's rows. We want to insert a column L with type T, so we ensure that the column is not already present in the input table by refining the table type with formula '**not** L **in** Rest'. The output table type represents the argument table with the extra column.

Consider the example where we want to insert a column salary with type int to the table Table[$Employee$]{$id : Int |$ }, we would instantiate 'insertColumn' as:

$$\text{insertColumn[Employee][salary]}[\varepsilon]\text{[id:Int][int]}$$

To obtain a function with type:

$$told : \{v : \text{Table}[Employee]\{id : Int|\,\}|\ \textbf{not}\ salary\ \textbf{in}\ \varepsilon\} \to v : int \to \text{Table}[Employee]\{id : Int|salary : Int\}$$

The application of the instantiated function with a table and integer arguments gives us the result type Table$[Employee]\{id : Int|salary : Int\}$.

Dropping. To drop a column from a table we have the operation 'dropColumn'. The operation's argument is the current table where we guarantee that the table has the column L that we want to remove by defining the type Table$[N]\{keys|L : T, rest\}$ that will only match on tables with label L. The output table type keeps the 'rest' without label L.

Consider the scenario where we want to remove the salary label with integer type from the table Table$[Employee]\{id : Int|salary : Int\}$, we instantiate 'dropColumn' as:

$$\text{dropColumn[Employee][salary]}[\varepsilon]\text{[id:Int][int]}$$

To obtain a function with type:

$$told : \text{Table}[Employee]\{id : Int|salary : Int\} \to \text{Table}[Employee]\{id : Int|\,\}$$

The application of the instantiated function with a table gives us the result table type Table$[Employee]\{id : Int|\,\}$, without the salary column.

Updating. For updating the type of a column we offer two possibilities: updating the type with a default value for the column's data ('updateColumn') or with an update function ('UpdateColumnF'). Both update functions receive the current table where we ensure, through the type's shape, that it has a label L with type T_1. We want to update type T_1 to T_2. On updateColumn, we receive another argument v with the default value of type T_2 for the column's data. Moreover, on updateColumnF we receive a second argument f with an update function that will change the data from type T_1 to T_2. Finally, the result type shows that the type of label L was updated to T_2.

We can exemplify UpdateColumnF by updating the integer type of column salary to positive ($\{v : Int|v > 0\}$) in table Table$[Employee]\{id : Int|salary : Int\}$, as follows :

$$\text{updateColumnF[Employee][salary]}[\varepsilon]\text{[id:Int][int]}[\{v : Int|v > 0\}]$$

To obtain a function with type:

$$told : \text{Table}[Employee]\{id : Int|salary : Int\} \to f : (int \to \{v : Int|v > 0\}) \to \text{Table}[Employee]\{id : Int|salary : \{v : Int|v > 0\}\}$$

If we apply the instantiated function with the current table as argument and a function with type $int \to \{v : Int|v > 0\}$, we obtain a table of type Table$[Employee]\{id : Int|salary : \{v : Int|v > 0\}\}$. The same goes for UpdateColumn but with a default value instead of a function.

Renaming. The renaming operation is 'Rename'. To rename label L_1 to L_2, first of all, we ensure that the input table type has a column with label L_1 (Table$[N]\{keys|L_1 : T, rest\}$). We need to impose an input condition that the input table does not already have a column L_2 since we do not allow for duplicate labels (**not** L_2 **in** rest). The output table type is simply the table with L_1 renamed to L_2.

Let us consider that we want to rename the salary column to firstSalary in the table type Table$[Employee]\{id : Int|salary : Int\}$, so we instantiate the operation:

$$\text{rename[Employee][salary][firstSalary]}[\varepsilon]\text{[id:Int][int]}$$

The instantiated function has type:

$$told : \{v : \text{Table}[Employee]\{id : Int|salary : Int\}|\ \textbf{not}\ firstSalary\ \textbf{in}\ \varepsilon\} \rightarrow \text{Table}[Employee]\{id : Int|firstSalary : Int\}$$

Applying the function with an input table we obtain Table$[Employee]\{id : Int|firstSalary : Int\}$.

Projecting. The last operation is the projection of a column from a table, operation 'TableProj.' The operation receives a table with column L and proceeds to project the column by producing a table with only this column.

Consider the table Table$[Employee]\{id : Int|salary : Int\}$, if we want to project the salary column, we can instantiate TableProj as follows:

$$\text{tableProj[Employee][salary]}[\varepsilon]\text{[id:Int][int]}$$

Then instantiated operation has type:

$$told : \text{Table}[Employee]\{id : Int|salary : Int\} \rightarrow \text{Table}[Employee]\{\,|salary : Int\}$$

Applying this function with argument table Table$[Employee]\{id : Int|salary : Int\}$ gives us a table of type Table$[Employee]\{\,|salary : Int\}$, where salary is the only column.

Denomination	Type Signature	Description
InsertColumn	$\forall_N N.$ $\forall_L L.$ $\forall_R Rest, Keys.$ $\forall T.$ $told : \{v : Table[N]\{\ldots Keys \mid \ldots Rest\} \mid$ **not** L **in** $Rest\} \rightarrow$ $v : T \rightarrow$ $Table[N]\{Keys \mid L : T, Rest\}$	Insert column L with type T and value v in table with name N
DropColumn	$\forall_N N.$ $\forall_L L.$ $\forall_R Rest, Keys.$ $\forall T.$ $told : Table[N]\{\ldots Keys \mid L : T, \ldots Rest\} \rightarrow$ $Table[N]\{Keys \mid Rest\}$	Drop column L with type T from a table with name N
UpdateColumn	$\forall_N N.$ $\forall_L L.$ $\forall_R Rest, Keys.$ $\forall T_1, T_2.$ $told : Table[N]\{\ldots Keys \mid L : T_1, \ldots Rest\} \rightarrow$ $v : T_2 \rightarrow$ $Table[N]\{Keys \mid L : T_2, Rest\}$	Update column L in table with name N to type T_2 with value v
UpdateColumnF	$\forall_N N.$ $\forall_L L.$ $\forall_R Rest, Keys.$ $\forall T_1, T_2.$ $told : Table[N]\{\ldots Keys \mid L : T_1, \ldots Rest\} \rightarrow$ $f : (T_1 \rightarrow T_2) \rightarrow$ $Table[N]\{Keys \mid L : T_2, Rest\}$	Update column L in table with name N to type T_2 whose values are updated by the application of function f
Rename	$\forall_N N.$ $\forall_L L_1, L_2.$ $\forall_R Rest, Keys.$ $\forall T.$ $told : \{v : Table[N]\{\ldots Keys \mid L_1 : T, \ldots Rest\}$ $\mid$ **not** L_2 **in** $Rest\} \rightarrow$ $Table[N]\{Keys \mid L_2 : T, Rest\}$	In a table with name N, change the label of column L_1 to L_2
TableProj	$\forall_N N.$ $\forall_L L.$ $\forall_R Rest, Keys.$ $\forall T.$ $told : Table[N]\{\ldots Keys \mid L : T, \ldots Rest\} \rightarrow$ $Table[N]\{\mid L : T\}$	Obtain a table with name N but only one column L with type T

Table 5.1: Library of Operations

6 Case Study

We now present a case study where the goals are: to define a data schema, through synthesized operations and using some of the operations described in Chapter 5, and demonstrate the range of problems we can specify and solve.

6.1 Scenario

Consider the scenario where we want to gradually define a data schema using a synthesis tool. We will start by specifying the creation and then the changes, interleaving with an analysis, until we have gathered a script of operations that defines the desired data schema. Our research question is: How do we use type-based specifications to synthesize the creation of a data schema and the inherent changes?

We now proceed with the case study and explain in detail the synthesis process. Our example will consist of employees and addresses. The creation of a data schema starts with the creation of one or multiple data tables. Note that we have all the operations described in Chapter 5 available in the synthesis context.

6.1.1 Data Schema Creation

We start by creating the table of employees, which we specify by $\text{Table}[Employee]\{id : Int | salary : Int, addressNumber : Int\}$, where the column's labels are self-explanatory. Since there is no table in context, by rule I-new, we obtain the program term:

$$\text{new Table}[Employee]\{id : Int | salary : Int, addressNumber : Int\}$$

6.1.2 Single Operation

We can update the table we just created by using a single operation. Consider that we want to rename column salary to firstSalary, we specify the change by:

$$told : \text{Table}[Employee]\{id : Int | salary : Int, addressNumber : Int\} \rightarrow$$
$$\text{Table}[Employee]\{id : Int | firstSalary : Int, addressNumber : Int\}$$

The first step is to invert the specification, so we apply rule I-Pi which adds the table argument to context and proceeds to invert the output type:

$$\Gamma, told : \text{Table}[Employee]\{id : Int | salary : Int, addressNumber : Int\}; \Sigma \vdash$$
$$\text{Table}[Employee]\{id : Int | firstSalary : Int, addressNumber : Int\}$$

Our current program is $\lambda told : told : \text{Table}[Employee]\{id : Int | salary : Int, addressNumber : Int\}.e$, which means that now we invert the return type to obtain the body term e. We cannot invert the atomic table type anymore nor create a new table (I-new) since there is an employee table in context. So we proceed by focusing on the context and we focus on operation rename whose type we define by R.

$$\Gamma, [rename : R]; \Sigma \vdash \text{Table}[Employee]\{id : Int | firstSalary : Int, addressNumber : Int\}$$

Focusing on rename (polymorphic) means that we will instantiate any polymorphic variable with a fresh existentially quantified variable (rules FOC-$\forall$ to FOC-$\forall_L$). The fresh variables are used as arguments to the polymorphic application in which we instantiate the function (later, by unification, they are replaced by types). We identify the instantiated function as rename' with type R' and focus on it.

Focusing on a function (FOC-Pi) means that we focus on the output type to see if we can produce a term with the goal type and then we invert the argument type to find an argument term for the function application. Let us start by focusing on the output type.

$$\Gamma, [x : \text{Table}[N'_{Sk}]\{keys'_{Sk} | L_{2_{Sk}} : T_{Sk}, rest'_{Sk}\}]; \Sigma \vdash$$
$$\text{Table}[Employee]\{id : Int | firstSalary : Int, addressNumber : Int\}$$

Since we are focusing on a table, the only rule we can apply is FOC-τ, in which we produce the unification constraint $\text{Table}[N'_{Sk}]\{keys'_{Sk} | L_{2_{Sk}} : T_{Sk}, rest'_{Sk}\} \sim \text{Table}[Employee]\{id : Int | firstSalary : Int, addressNumber : Int\}$, to check if there is a unifier for these types, which means that by applying this function we can get a term of the desired type. We now proceed by inverting the argument type:

$$\Gamma; \Sigma \vdash \{v : \text{Table}[N'_{Sk}]\{keys'_{Sk} | L_{1_{Sk}} : T_{Sk}, rest'_{Sk}\} | \textbf{not } L_{2_{Sk}} \textbf{ in } rest'_{Sk}\}$$

We invert the table base type and produce constraint $C \implies \textbf{not } L_{2_{Sk}} \textbf{ in } rest'_{Sk}$ to check whether the refinement is satisfiable (C stands for a possible constraint we may produce when inverting the base type).

$$\Gamma; \Sigma \vdash \text{Table}[N'_{Sk}]\{keys'_{Sk} | L_{1_{Sk}} : T_{Sk}, rest'_{Sk}\}$$

We focus on the specification's input told which is in context:

$$\Gamma, [told : \text{Table}[Employee]\{id : Int | salary : Int, addressNumber : Int\}]; \Sigma \vdash$$
$$\text{Table}[N'_{Sk}]\{keys'_{Sk} | L_{1_{Sk}} : T_{Sk}, rest'_{Sk}\}$$

All we can do now is to produce the unification constraint $\text{Table}[Employee]\{id : Int|salary : Int, addressNumber : Int\} \sim \text{Table}[N'_{Sk}]\{keys'_{Sk}|L1_{Sk} : T_{Sk}, rest'_{Sk}\}$, if we find a unifier between these two tables that means that we apply told to the instantiated rename function. To conclude the program, we are left with the unification contraints:

- $\text{Table}[N'_{Sk}]\{keys'_{Sk}|L2_{Sk} : T_{Sk}, rest'_{Sk}\} \sim \text{Table}[Employee]\{id : Int|firstSalary : Int, addressNumber : Int\}$
- $\text{Table}[Employee]\{id : Int|salary : Int, addressNumber : Int\} \sim \text{Table}[N'_{Sk}]\{keys'_{Sk}|L1_{Sk} : T_{Sk}, rest'_{Sk}\}$

We solve them to get the unifier: $keys'_{Sk} \sim Id : Int$, $rest'_{Sk} \sim addressNumber : Int$, $L1_{Sk} \sim salary$, $L2_{Sk} \sim firstSalary$, $N'_{Sk} \sim Employee$ and $T'_{Sk} \sim int$.

We apply the unifier to the refinement constraint $C \implies$ **not** $L2_{Sk}$ **in** $rest'_{Sk}$, and also remove C since we did not find any other constraint. Constraint **not** $firstSalary$ **in** $(addressNumber : Int)$ is satisfiable, so we obtain the program:

$$\lambda told : \text{Table}[Employee]\{id : Int|salary : Int, addressNumber : Int\}.$$
$$(rename[Employee][salary][firstSalary][addressNumber : Int][id : Int][int])(told)$$

6.1.3 Composition of Operations

Consider that now we want to apply more complex changes which require us to synthesize a composition of operations, since one cannot perform everything, we show how we can compose functions to achieve the desired type.

6.1.3.1 Multiple Updates.

Consider the case where we want the firstSalary and addressNumber columns to only allow positive values. We want to restrict the type of these columns from integer to $\{v : Int|v > 0\}$. We have two types of update functions, we will consider that we set v_1 as a default value and f as a function that transformes integer values into positive ones. The specification receives the previous table, v and f, to return the desired table type, as follows:

$$(told : \text{Table}[Employee]\{id : Int|firstSalary : Int, addressNumber : Int\}) \rightarrow$$
$$(v_1 : \{v : Int|v > 0\}) \rightarrow (f : (int \rightarrow \{v : Int|v > 0\})) \rightarrow$$
$$\text{Table}[Employee]\{id : Int|firstSalary : \{v : Int|v > 0\}, addressNumber : \{v : Int|v > 0\}\}$$

We will guide the synthesis towards using v_1 for addressNumber and f for firstSalary, even though there is no indication in the specification. The synthesis process can basically use anything in the context that helps to achieve the goal. We could for example, add the refinement $v = v_1$ to the type of addressNumber to force the use of v_1. We omit this refinement for simplicity.

Let us invert the specification function by applying I-Pi multiple times until we have every argument in context:

$$\Gamma, told : \text{Table}[Employee]\{id : Int | firstSalary : Int, addressNumber : Int\},$$
$$v_1 : \{v : Int | v > 0\}, f : (int \to \{v : Int | v > 0\}); \Sigma \vdash$$
$$\text{Table}[Employee]\{id : Int | firstSalary : \{v : Int | v > 0\}, addressNumber : \{v : Int | v > 0\}\}$$

Up to now the program we have is:

$$\lambda told : \text{Table}[Employee]\{id : Int | firstSalary : Int, addressNumber : Int\}.$$
$$\lambda v_1 : \{v : Int | v > 0\}. \lambda f : (int \to \{v : Int | v > 0\}).\ e$$

We continue to invert the return type to get e. Since we cannot invert the table anymore, we focus on updteColumnF (with type UF) in context (I-Focus):

$$\Gamma, [updateColumnF : \text{UF}]; \Sigma \vdash$$
$$\text{Table}[Employee]\{id : Int | firstSalary : \{v : Int | v > 0\}, addressNumber : \{v : Int | v > 0\}\}$$

We focus on the polymorphic function by instantiating the polymorphic variables, we use the instantiations as arguments for the polymorphic applications of updateColumnF. When then focus on the instantiated updateColumnF by focusing on the output table type:

$$\Gamma, [x : \text{Table}[N'_{Sk}]\{keys'_{Sk} | L'_{Sk} : T'_{2_{Sk}}, rest'_{Sk}\}]; \Sigma \vdash$$
$$\text{Table}[Employee]\{id : Int | firstSalary : \{v : Int | v > 0\}, addressNumber : \{v : Int | v > 0\}\}$$

Producing constraint $\text{Table}[N'_{Sk}]\{keys'_{Sk} | L'_{Sk} : T'_{2_{Sk}}, rest'_{Sk}\} \sim \text{Table}[Employee]\{id : Int | firstSalary : \{v : Int | v > 0\}, addressNumber : \{v : Int | v > 0\}\}$. If we can unify the types, then we can use updateColumnF.

We proceed by inverting the second argument type: $\Gamma; \Sigma \vdash T_{1_{Sk}} \to T_{2_{Sk}}$. We add $T_{1_{Sk}}$ to context (I-Pi) and invert $T_{2_{Sk}}$. Since the type is atomic, by I-Focus, we focus on context and more specifically we focus on function f.

$$\Gamma, [f : (int \to \{v : Int | v > 0\})]; \Sigma \vdash T_{2_{Sk}}$$

If we focus on $\{v : Int | v > 0\}$ with goal $T_{2_{Sk}}$, we obtain the unification constraint $\{v : Int | v > 0\} \sim T_{2_{Sk}}$. Inverting the argument int we end up focusing on $T_{1_{Sk}}$ and generating a unification constraint. We generate a fresh identifier x to identify the function's argument. We conclude inverting $T_{1_{Sk}} \to T_{2_{Sk}}$ and we constructed the argument function term $\lambda x : int. f(x)$.

Returning to focusing on the instantiated operation updateColumnF we still have to invert the first argument:

$$\Gamma; \Sigma \vdash \text{Table}[N'_{Sk}]\{keys'_{Sk} | L'_{Sk} : T'_{1_{Sk}}, rest'_{Sk}\}$$

We focus on the argument told with type Table$[Employee]\{id : Int | firstSalary : Int, addressNumber : Int\}$ and produce a unification constraint between the type and our goal.

Now we have reached the point where we tried to use UpdateColumnF by proving that we can apply it to reach our goal and by synthesizing the arguments. Let us solve the unification constraints to get a unifier:

- Table$[N'_{Sk}]\{keys'_{Sk} | L'_{Sk} : T'_{2_{Sk}}, rest'_{Sk}\} \sim$ Table$[Employee]\{id : Int | firstSalary : \{v : Int | v > 0\}, addressNumber : \{v : Int | v > 0\}\}$
- $\{v : Int | v > 0\} \sim T'_{2_{Sk}}$
- Table$[Employee]\{id : Int | firstSalary : Int, addressNumber : Int\} \sim$ Table$[N'_{Sk}]\{keys'_{Sk} | L'_{Sk} : T'_{1_{Sk}}, rest'_{Sk}\}$

One possible unifier is: $T'_{2_{Sk}} \sim \{v : Int | v > 0\}$, $T'_{1_{Sk}} \sim int$, $N'_{Sk} \sim Employee$, $keys'_{Sk} \sim id : Int$, $L'_{Sk} \sim firstSalary$. The problem is that when we try to unify $rest'_{Sk}$ if we unify it with $addressNumber : \{v : Int | v > 0\}$ then we cannot unify the last constraint, if we unify it with $addressNumber : Int$ the dual happens.

We can conclude that told is not the argument we need, which means that we need to go back to inverting the first argument type:

$$\Gamma; \Sigma \vdash \text{Table}[N'_{Sk}]\{keys'_{Sk} | L'_{Sk} : T'_{1_{Sk}}, rest'_{Sk}\}$$

We now try to focus on the updateColumn operation to prove the current goal and argument of the updateColumnF operation (so that we can compose them). We instantiate the polymorphic variables so that we can focus on the instantiated function, note that the instantiation is done with fresh variables. Since the process is repetitive, we will summarize it.

We focus on the output type of updateColumn generating constraint Table$[N''_{Sk}]\{keys''_{Sk} | L''_{Sk} : T''_{2_{Sk}}, rest''_{Sk}\} \sim$ Table$[N'_{Sk}]\{keys'_{Sk} | L'_{Sk} : T'_{1_{Sk}}, rest'_{Sk}\}$. We invert the argument type $T''_{2_{Sk}}$ and end up focusing on v_1 which is in context with type $\{v : Int | v > 0\}$, so we produce a unification constraint between the two. The last step is inverting the argument table type which we will focus on told and generate constraint Table$[N''_{Sk}]\{keys''_{Sk} | L''_{Sk} : T''_{1_{Sk}}, rest''_{Sk}\} \sim$ Table$[Employee]\{id : Int | salary : Int, addressNumber : Int\}$.
Let us check the unification constraints again:

- Table$[N'_{Sk}]\{keys'_{Sk} | L'_{Sk} : T'_{2_{Sk}}, rest'_{Sk}\} \sim$ Table$[Employee]\{id : Int | firstSalary : \{v : Int | v > 0\}, addressNumber : \{v : Int | v > 0\}\}$
- $\{v : Int | v > 0\} \sim T'_{2_{Sk}}$
- Table$[N''_{Sk}]\{keys''_{Sk} | L''_{Sk} : T''_{2_{Sk}}, rest''_{Sk}\} \sim$ Table$[N'_{Sk}]\{keys'_{Sk} | L'_{Sk} : T'_{1_{Sk}}, rest'_{Sk}\}$

- ${T_2''}_{Sk} \sim \{v : Int | v > 0\}$
- $\text{Table}[N''_{Sk}]\{keys''_{Sk} | L''_{Sk} : {T_1''}_{Sk}, rest''_{Sk}\} \sim$
  $\text{Table}[Employee]\{id : Int | salary : Int, addressNumber : Int\}$

We can solve the constraints incrementally as:

- $\text{Table}[N'_{Sk}]\{keys'_{Sk} | L'_{Sk} : T'_{2_{Sk}}, rest'_{Sk}\} \sim \text{Table}[Employee]\{id : Int | firstSalary :$ $\{v : Int | v > 0\}, addressNumber : \{v : Int | v > 0\}\}$

  unifier: $N'_{Sk} \sim Employee$, $keys'_{Sk} \sim id : Int$, $L'_{Sk} \sim firstSalary$,
  $T'_{2_{Sk}} \sim \{v : Int | v > 0\}$ and $rest'_{Sk} \sim addressNumber : \{v : Int | v > 0\}$
- $\{v : Int | v > 0\} \sim \{v : Int | v > 0\}$
- $\text{Table}[N''_{Sk}]\{keys''_{Sk} | L''_{Sk} : {T_2''}_{Sk}, rest''_{Sk}\} \sim \text{Table}[Employee]\{id : Int | firstSalary : T'_{1_{Sk}},$ $addressNumber : \{v : Int | v > 0\}\}$

  unifier: $N''_{Sk} \sim Employee$, $keys''_{Sk} \sim id : Int$, $rest''_{Sk} \sim firstSalary : T'_{1_{Sk}}$,
  $L''_{Sk} \sim addressNumber$ and ${T_2''}_{Sk} \sim \{v : Int | v > 0\}$
- $\{v : Int | v > 0\} \sim \{v : Int | v > 0\}$
- $\text{Table}[Employee]\{id : Int | addressNumber : {T_1''}_{Sk}, firstSalary : T'_{1_{Sk}}\} \sim$
  $\text{Table}[Employee]\{id : Int | salary : Int, addressNumber : Int\}$

  unifier: ${T_1''}_{Sk} \sim int$ and $T'_{1_{Sk}} \sim int$

To summarize, our final program thus results from the composition of operations updateColumn and updateColumnF. The program's body has the updateColumn function instantiated and applied with arguments told and v_1. The resulting term from this application we use as argument for updateColumnF plus function $\lambda x : int.f(x)$. Thus, our final program is:

$$\begin{array}{l}
\lambda told : \text{Table}[Employee]\{id : Int | firstSalary : Int, addressNumber : Int\}. \\
\lambda v_1 : \{v : Int | v > 0\}.\lambda f : (int \to \{v : Int | v > 0\}). \\
\quad (updateColumnF[Employee][firstSalary][addressNumber : \{v : Int | v > 0\}] \\
\quad [id : Int][int][\{v : Int | v > 0\}]) \\
\quad (updateColumn[Employee][addressNumber][firstSalary : Int] \\
\quad [id : Int][int][\{v : Int | v > 0\}](told)(v_1)) \\
\quad (\lambda x : int.f(x))
\end{array}$$

The type of the function's body is $\text{Table}[Employee]\{id : Int | firstSalary : \{v : Int | v > 0\}, addressNumber : \{v : Int | v > 0\}\}$.

6.1.3.2 Drop and Create.

Consider another example, where we want to normalize the employee table by relocating the addressNumber column to a separate address table. We specified this changed by a function which receives the current employee table and produces a pair (representing two outputs), where we have the address table with column addressNumber and then the employee table without this column and with a column 'address' referencing the primary key of the address table, as follows:

$$\begin{aligned}
&told : \text{Table}[Employee]\{id : Int | firstSalary : \{v : Int | v > 0\},\\
&\qquad addressNumber : \{v : Int | v > 0\}\} \rightarrow\\
&v : Int \rightarrow\\
&(\text{Table}[Address]\{id : Int | addressNumber : \{v : Int | v > 0\}\},\\
&\text{Table}[Employee]\{id : Int | firstSalary : \{v : Int | v > 0\}, address : Int\})
\end{aligned}$$

As we have been doing, we add the arguments to context (rule I-Pi) and invert the output type.

$$\begin{aligned}
&\Gamma, told : \text{Table}[Employee]\{id : Int | firstSalary : \{v : Int | v > 0\},\\
&\qquad addressNumber : \{v : Int | v > 0\}\}, v : Int; \Sigma\\
&\vdash (\text{Table}[Address]\{id : Int | addressNumber : \{v : Int | v > 0\}\},\\
&\text{Table}[Employee]\{id : Int | firstSalary : \{v : Int | v > 0\}, address : Int\})
\end{aligned}$$

We now have a pair, by rule I-Pair, we invert both components of the pair. This is a case of a choice where the order in which we invert does not matter, since we will follow both options.

The program right now is $\lambda told : \text{Table}[Employee]\{id : Int | firstSalary : \{v : Int | v > 0\}, addressNumber : \{v : Int | v > 0\}\}. \lambda v : Int. (e_1, e_2)$, we have to invert both components to get e_1 and e_2.

To invert the first component, we invert table $\text{Table}[Address]\{id : Int | addressNumber : \{v : Int | v > 0\}\}$ and by rule I-new the program term is New $\text{Table}[Address]\{id : Int | addressNumber : \{v : Int | v > 0\}\}$. The second component envolves removing the address-Number column, so to invert the table we move to focusing and focus on the operation dropColumn with type D:

$$\Gamma, [dropColumn : D]; \Sigma \vdash \text{Table}[Employee]\{id : Int | firstSalary : \{v : Int | v > 0\}\}$$

After instantiating the function we invert it. We focus on the output type $\text{Table}[N'_{Sk}]\{keys'_{Sk} | rest'_{Sk}\}$ and produce constraint $\text{Table}[N'_{Sk}]\{keys'_{Sk} | rest'_{Sk}\} \sim \text{Table}[Employee]\{id : Int | firstSalary : \{v : Int | v > 0\}\}$.

We invert the argument type and focus on told in context. To use told as argument we produce constraint $\text{Table}[N'_{Sk}]\{keys'_{Sk} | L'_{Sk} : T'_{Sk}, rest'_{Sk}\} \sim \text{Table}[Employee]\{id : Int | firstSalary : \{v : Int | v > 0\}, addressNumber : \{v : Int | v > 0\}\}$.

Employee	
id	Int **PK**
salary	Int
addressNumber	Int

Table 6.1: Creation of Employee Table

Employee	
id	Int **PK**
firstSalary	{v:Int | v > 0}
address	Int **FK**

Address	
id	Int **PK**
addressNumber	{v:Int | v > 0}

Table 6.2: Table Normalization into Employee and Address Tables

Unifying L'_{Sk} with addressNumber and T'_{Sk} with $\{v : Int|v > 0\}$ means that the result table will not have this column, since the result type is Table$[N'_{Sk}]\{keys'_{Sk}|rest'_{Sk}\}$. The rest of the unifier is easy to understand. Note that since we have the extra field 'address', we cannot directly use the dropColumn operation with told as argument but we have to repeat the sequence with the insertColumn operation, to then compose both. The final program is:

$\lambda told : \text{Table}[Employee]\{id : Int|firstSalary : \{v : Int|v > 0\}, addressNumber : \{v : Int|v > 0\}\}.$

$\lambda v : Int.$

$(New\text{Table}[Address]\{id : Int|addressNumber : \{v : Int|v > 0\}\},$

$dropColumn[Employee][addressNumber][firstSalary : \{v : Int|v > 0\}, address : Int]$

$[id : int][\{v : Int|v > 0\}]$

$(insertColumn[Employee][address][firstSalary : \{v : Int|v > 0\}, addressNumber : \{v : Int|v > 0\}]$

$[id : Int][Int](told)(v)))$

At first sight, the problem of normalizing a table may seem less complex than, for example, applying multiple updates but, we will explain in the Section 6.2 that there are a few intricate questions regarding the row's data.

6.1.4 Resulting Script

Gathering all the operations resulting from the above synthesis section, provides us with a script that creates the table in 6.1 and does multiple updates until we get two tables like in 6.2.

We can now have a look at the script (Listing 6.1) containing all the operations:

$\text{new}\ \ \text{Table}[Employee]\{id : Int | salary : Int, addressNumber : Int\}$

$\lambda told : \text{Table}[Employee]\{id : Int | salary : Int, addressNumber : Int\}.$
$\quad (rename[Employee][salary][firstSalary][addressNumber : Int][id : Int][int])(told)$

$\lambda told : \text{Table}[Employee]\{id : Int | firstSalary : Int, addressNumber : Int\}.$
$\lambda v_1 : \{v : Int | v > 0\}.\lambda f : (int \to \{v : Int | v > 0\}).$
$\quad (updateColumnF[Employee][firstSalary][addressNumber : \{v : Int | v > 0\}][id : Int][int][\{v : Int | v > 0\}])$
$\quad (updateColumn[Employee][addressNumber][firstSalary : Int][id : Int][int][\{v : Int | v > 0\}$
$\quad (\, told \,)\,(\, v_1 \,)\,)$
$\quad (\lambda x : int\, .\, f(x))$

$\lambda told : \text{Table}[Employee]\{id : Int | firstSalary : \{v : Int | v > 0\}, addressNumber : \{v : Int | v > 0\}\}.$
$\quad (New\text{Table}[Address]\{id : Int | addressNumber : \{v : Int | v > 0\}\},$
$\quad dropColumn[Employee][addressNumber][firstSalary : \{v : Int | v > 0\}, address : Int][id : int][\{v : Int | v > 0\}])$
$\quad ((insertColumn[Employee][address][firstSalary : \{v : Int | v > 0\}, addressNumber : \{v : Int | v > 0\}][id : Int]$
$\quad [Int](told)(v))$

Listing 6.1: Final Script of Operations

6.2 Further Discussion

Our goal with this case study is to act as a proof of concept. We demonstrated that the framework we developed is capable of synthesizing a wide range of database operations that can be compiled into real operations (such as inserting, dropping, updating columns, or even normalizing a table).

The fact that the set of library operations is extensible means that we can easily add new operations to express future needs. Even though we have this possibility, we consider that the set we defined is quite expressible by itself. Not only we can apply any of the polymorphic functions, by instantiation, to the example we are dealing with, we can also synthesize the composition of operations.

One guarantee that we can have with type-directed synthesis is that if we synthesize a program then the program satisfies the specification since it was guided by it. This kind of certainty is very useful at the level of important operations such as the ones that deal with databases. Besides that, the synthesis procedure tries to discard wrong solutions as early as possible.

We have seen that, at the schema level, the framework is expressive enough, now let us discuss the inherent data. We deal with the data by imposing, in the insertColumn and updateColumn operations, the need for a default value or an update function to fill the column's rows. The subject of future work is the expression of more conditions on data, for example, defining which columns from the input table remain the same and which do not (and how do we change the data).

For instance, considering table normalization, the specification was:

$$\begin{aligned}&told : \text{Table}[Employee]\{id : Int | firstSalary : \{v : Int | v > 0\},\\&\quad addressNumber : \{v : Int | v > 0\}\} \to\\&v : Int \to\\&(\text{Table}[Address]\{id : Int | addressNumber : \{v : Int | v > 0\}\},\\&\text{Table}[Employee]\{id : Int | firstSalary : \{v : Int | v > 0\}, address : Int\})\end{aligned}$$

As we mentioned, this example has challenging details at the level of data. When separating the table employee into two tables, we want to be able to specify in the output type that the id and firstSalary columns of employee remain the same and that the data in the addressNumber comes from the column also in the employee table. The idea that we have on this topic for future research would be to define predicates (or refinements) expressing these changes (which could be encoded as uninterpreted functions to discharge to Z3). Imagine that we define a predicate (e.g. named foo) whose definition (let us just see for the firstSalary case) is:

$$\begin{aligned}&foo : \text{Forall r in v. Exists x in told.firstSalary .}\\&\qquad (r.\text{firstSalary} = x)\end{aligned}$$

The predicate foo would express that the values of the firstSalary column come from the input table told. In this way, we could refine the output employee table with foo like $\{v : \text{Table}[Employee]\{id : Int | firstSalary : \{v : Int | v > 0\}\} | foo\}$ as a way of expressing the data's source. We could see the expression of the values in the addressColumn coming from the input also like this, as well as when we receive a default value/update function and want to connect it to the final table's data.

Another topic that we want to discuss is the expression of conditions on types (through refinement types). We defined the membership formula (l **in** r) to impose a condition on the specification of some tables as a way of defining that a label l is not present in that table. Future work could extend this direction into, for example in the normalization scenario, defining a condition that states that the column address in employee is a foreign key referring to the primary key of the address table. Moreover, one can also extend the expressibility of refinements to define integrity constraints and access control/validation rules.

7 Conclusion

Our work was motivated by the need to continuously grow and evolve data representations (and its properties), to accommodate the rich feature set that is expected of data-centric applications. Since data-centric applications have widespread use in modern commercial software, this work is very relevant. With these applications, data is the main focus, while applications are temporary, which means that to introduce new functionality there is the need to continuously (and uniformly) evolve the data schema. To introduce new functionality, one must consider the available data, its structure , dependencies and the ability to grow/evolve the data schema.

The automation of programs is the focus of GOLEM, our work is integrated in GOLEM and focuses on the data layer. We approached this problem by defining a synthesis framework based on richly-typed high-level specifications and using a type-directed synthesis procedure, synthesizes high-level operations that define and evolve the data schema and the inherent data.

Two of the main challenges were to understand how to specify the creation and evolution of the data representation, as well as what could be the target of the procedure. These are not straightforward problems, it was an intellectual challenge to define how much information did we need to synthesize correct programs and how can we express that same information (what can be the pipeline from specification to the final program). To overcome those challenges, we decided that our specifications would consist of richly-typed specifications with dependent and refinement types which enables us to express more complex properties, and we also defined (through the use of polymorphism and based on a case study) a library of base operations on tables that capture the scenario we considered.

As a result, we implemented a synthesis framework that can synthesize a wide variety of operations such as the creation of a data schema, single operations that insert, rename and drop columns (among other operations), as well as the composition of operations. The framework can also deal with other situations such as table normalization. The framework developed constitutes a basis for future work, since it will still be improved along the future years of the GOLEM project, and it has a lot of potential to, in the future, be able to define more complex properties on data and also integrity constraints, plus validation/access control rules.

7.1 Future Work

We categorize the future work resulting from this dissertation into research topics and implementation related.

In the area of implementation, there is space for optimizing the synthesis mechanism. Since synthesis is a combinatorial problem with the possibility of entering infinite loops, an interesting task here would be to look for possible optimizations.

For example, we use the SMT Solver to obtain satisfiability results on constraints. A possible optimization could be to try to gather more information from the solver such as models for satisfiable constraints or proofs for unsatisfiable constraints, and to investigate how this information can help guide the search process.

There is a necessary exploratory process involved in the optimization of the synthesis procedure to find ideas to break cycles, discover wrong solutions early or guide the search.

On the side of research topics, we can visualize two possible threads. The first involves researching how to connect the data from the input tables of the specification to the output tables, such as indicating which column's data remains the same, which data changes, and how. Using the expressibility of refinement types, it would be interesting to design refinements related to the data that then can be encoded and discharged to the SMT Solver possibly as uninterpreted functions.

The second thread relates to integrity constraints on data and access control/validation rules. We consider that synthesizing code that defines and evolves a data schema layer, but also is capable of dealing with integrity constraints, access control/validation rules brings the framework even closer to the user's real needs.

We conclude the future work discussion by stating that the framework was established from scratch involving several ideas and research throughout this period. The ideas presented here are just a snippet of the possible future research and new ideas that may come in the next years of the GOLEM project.

Bibliography

[1] 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%5C_18 (cit. on pp. 11, 14).

[2] R. Alur et al. “SyGuS-Comp 2016: Results and Analysis”. In: *Electronic Proceedings in Theoretical Computer Science* 229 (Nov. 2016), pp. 178–202. issn: 2075-2180. doi: 10.4204/eptcs.229.13. url: http://dx.doi.org/10.4204/EPTCS.229.13 (cit. on pp. 8, 9).

[3] R. Alur et al. “Syntax-Guided Synthesis”. In: *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. Oct. 2013, pp. 1–17 (cit. on pp. 7–9, 12–15).

[4] J. Andreoli. “Logic Programming with Focusing Proofs in Linear Logic”. In: *J. Log. Comput.* 2 (1992), pp. 297–347 (cit. on p. 56).

[5] D. W. Barowy et al. “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, pp. 218–228. isbn: 9781450334686. doi: 10.1145/2737924.2737952. url: https://doi.org/10.1145/2737924.2737952 (cit. on p. 15).

[6] C. Barrett, P. Fontaine, and C. Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Available at www.SMT-LIB.org. Department of Computer Science, The University of Iowa, 2017 (cit. on p. 68).

[7] C. Barrett and C. Tinelli. “Satisfiability Modulo Theories”. In: *Handbook of Model Checking*. Ed. by E. M. Clarke et al. 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 (cit. on p. 12).

[8] P. J. Denning. “The Field of Programmers Myth”. In: *Commun. ACM* 47.7 (July 2004), pp. 15–20. ISSN: 0001-0782. DOI: 10.1145/1005817.1005836. URL: https://doi.org/10.1145/1005817.1005836 (cit. on p. 1).

[9] J. Devlin et al. “RobustFill: Neural Program Learning under Noisy I/O”. In: *CoRR* abs/1703.07469 (2017). arXiv: 1703.07469. URL: http://arxiv.org/abs/1703.07469 (cit. on pp. 14, 15).

[10] E. W. Dijkstra. “Guarded Commands, Nondeterminacy and Formal Derivation of Programs”. In: *Commun. ACM* 18.8 (Aug. 1975), pp. 453–457. ISSN: 0001-0782. DOI: 10.1145/360933.360975. URL: https://doi.org/10.1145/360933.360975 (cit. on pp. 17–19, 24).

[11] E. W. Dijkstra. “The Humble Programmer”. In: *Commun. ACM* 15.10 (Oct. 1972), pp. 859–866. ISSN: 0001-0782. DOI: 10.1145/355604.361591. URL: https://doi.org/10.1145/355604.361591 (cit. on pp. 1, 2).

[12] I. Drosos et al. “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, pp. 1–12. ISBN: 9781450367080. DOI: 10.1145/3313831.3376442. URL: https://doi.org/10.1145/3313831.3376442 (cit. on p. 15).

[13] J. Dunfield and N. Krishnaswami. “Bidirectional Typing”. In: *ACM Comput. Surv.* 54.5 (May 2021). ISSN: 0360-0300. DOI: 10.1145/3450952. URL: https://doi.org/10.1145/3450952 (cit. on p. 44).

[14] A. Fariha and A. Meliou. “Example-Driven Query Intent Discovery: Abductive Reasoning Using Semantic Similarity”. In: *Proc. VLDB Endow.* 12.11 (July 2019), pp. 1262–1275. ISSN: 2150-8097. DOI: 10.14778/3342263.3342266. URL: https://doi.org/10.14778/3342263.3342266 (cit. on p. 16).

[15] Y. Feng et al. “Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples”. In: *SIGPLAN Not.* 52.6 (June 2017), pp. 422–436. ISSN: 0362-1340. DOI: 10.1145/3140587.3062351. URL: https://doi.org/10.1145/3140587.3062351 (cit. on pp. 9–11, 15).

[16] S. Gulwani. “FlashExtract: A Framework for Data Extraction by Examples”. In: *PLDI ’14, June 09 - 11 2014, Edinburgh, United Kingdom*. June 2014. URL: https://www.microsoft.com/en-us/research/publication/flashextract-framework-data-extraction-examples/ (cit. on pp. 10, 15).

[17] S. Gulwani. "Programming by Examples (and its Applications in Data Wrangling)". In: *Verification and Synthesis of Correct and Secure Systems*. IOS Press, Jan. 2016. URL: https://www.microsoft.com/en-us/research/publication/programming-examples-applications-data-wrangling/ (cit. on p. 9).

[18] S. Gulwani, A. Polozov, and R. Singh. *Program Synthesis*. Vol. 4. NOW, Aug. 2017, pp. 1–119. URL: https://www.microsoft.com/en-us/research/publication/program-synthesis/ (cit. on pp. 7, 8, 10, 13–16).

[19] M. Hinchey et al. "Software Engineering and Formal Methods". In: *Commun. ACM* 51.9 (Sept. 2008), pp. 54–59. ISSN: 0001-0782. DOI: 10.1145/1378727.1378742. URL: https://doi.org/10.1145/1378727.1378742 (cit. on p. 1).

[20] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259 (cit. on pp. 2, 17).

[21] C. Jenkins and A. Stump. "Spine-Local Type Inference". In: *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages*. IFL 2018. Lowell, MA, USA: Association for Computing Machinery, 2018, pp. 37–48. ISBN: 9781450371438. DOI: 10.1145/3310232.3310233. URL: https://doi.org/10.1145/3310232.3310233 (cit. on p. 44).

[22] S. Jha et al. "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, pp. 215–224. ISBN: 9781605587196. DOI: 10.1145/1806799.1806833. URL: https://doi.org/10.1145/1806799.1806833 (cit. on pp. 7, 9, 12, 13, 15, 16).

[23] C. Le Goues et al. "GenProg: A Generic Method for Automatic Software Repair". In: *IEEE Transactions on Software Engineering* 38.1 (2012), pp. 54–72 (cit. on pp. 14, 16).

[24] W. Lee et al. "Accelerating Search-Based Program Synthesis Using Learned Probabilistic Models". In: *SIGPLAN Not.* 53.4 (June 2018), pp. 436–449. ISSN: 0362-1340. DOI: 10.1145/3296979.3192410. URL: https://doi.org/10.1145/3296979.3192410 (cit. on pp. 14, 15).

[25] *Linear Logic*. https://www.cs.cmu.edu/~fp/courses/15816-f01/handouts/linear.pdf. Accessed: 2021-08-30 (cit. on pp. 55, 62).

[26] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. vii).

[27] Z. Manna and R. Waldinger. “A Deductive Approach to Program Synthesis”. In: *ACM Trans. Program. Lang. Syst.* 2.1 (Jan. 1980), pp. 90–121. ISSN: 0164-0925. DOI: 10.1145/357084.357090. URL: https://doi.org/10.1145/357084.357090 (cit. on pp. 7, 16, 20, 24).

[28] Z. Manna and R. J. Waldinger. “Toward Automatic Program Synthesis”. In: *Commun. ACM* 14.3 (Mar. 1971), pp. 151–165. ISSN: 0001-0782. DOI: 10.1145/362566.362568. URL: https://doi.org/10.1145/362566.362568 (cit. on pp. 16, 20, 24).

[29] M. Mayer et al. “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, Nov. 2015. URL: https://www.microsoft.com/en-us/research/publication/user-interaction-models-for-disambiguation-in-programming-by-example/ (cit. on p. 10).

[30] L. de Moura and N. Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and J. Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3 (cit. on p. 68).

[31] 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, pp. 619–630. ISBN: 9781450334686. DOI: 10.1145/2737924.2738007. URL: https://doi.org/10.1145/2737924.2738007 (cit. on pp. 16, 21–23, 25, 26, 43).

[32] 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 (cit. on pp. 10, 11).

[33] B. C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091 (cit. on p. 54).

[34] B. C. Pierce and D. N. Turner. “Local Type Inference”. In: *ACM Trans. Program. Lang. Syst.* 22.1 (Jan. 2000), pp. 1–44. ISSN: 0164-0925. DOI: 10.1145/345099.345100. URL: https://doi.org/10.1145/345099.345100 (cit. on p. 43).

[35] 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, pp. 522–538. ISBN:

9781450342612. DOI: 10.1145/2908080.2908093. URL: https://doi.org/10.1145/2908080.2908093 (cit. on pp. 11, 16, 21–26, 43).

[36] 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, pp. 107–126. ISBN: 9781450336895. DOI: 10.1145/2814270.2814310. URL: https://doi.org/10.1145/2814270.2814310 (cit. on pp. 7, 8).

[37] 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 (cit. on pp. 26, 27).

[38] E. Schkufza, R. Sharma, and A. Aiken. "Stochastic Superoptimization". In: *SIGPLAN Not.* 48.4 (Mar. 2013), pp. 305–316. ISSN: 0362-1340. DOI: 10.1145/2499368.2451150. URL: https://doi.org/10.1145/2499368.2451150 (cit. on pp. 13, 16).

[39] 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 (cit. on pp. 1, 2).

[40] 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 (cit. on pp. 10, 15).

[41] 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 (cit. on pp. 8, 9, 11–13, 15, 27).

[42] S. Srivastava, S. Gulwani, and J. S. Foster. "From Program Verification to Program Synthesis". In: *SIGPLAN Not.* 45.1 (Jan. 2010), pp. 313–326. ISSN: 0362-1340. DOI: 10.1145/1707801.1706337. URL: https://doi.org/10.1145/1707801.1706337 (cit. on pp. 16, 19).

[43] 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, pp. 135–152. ISBN: 9781450324724. DOI: 10.1145/2509578.2509586. URL: https://doi.org/10.1145/2509578.2509586 (cit. on p. 13).

[44] M. E. P. Valdez. "A Gift from Pandora's Box: The Software Crisis". Order No: GAXD–82988. PhD thesis. GBR, 1988 (cit. on pp. 1, 2).

[45] 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, pp. 452–466. ISBN: 9781450349888. DOI: 10.1145/3062341.3062365. URL: https://doi.org/10.1145/3062341.3062365 (cit. on pp. 9, 11, 16).

[46] Y. Wang et al. "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, pp. 286–300. ISBN: 9781450367127. DOI: 10.1145/3314221.3314588. URL: https://doi.org/10.1145/3314221.3314588 (cit. on p. 26).

[47] 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, pp. 224–234. ISBN: 9781479902156. DOI: 10.1109/ASE.2013.6693082. URL: https://doi.org/10.1109/ASE.2013.6693082 (cit. on p. 16).