View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by DSpace at Tartu University Library

UNIVERSITY OF TARTU

Institute of Computer Science

Software Engineering Curriculum

Giorgi Gogiashvili

The State of the Art of Automatic Programming

Master's Thesis (30 ECTS)

Supervisor(s): Siim Karus

Tartu 2018

The State of the Art of Automatic Programming

Abstract:

Automatic programming or code generation is a type of computer programming where the code is generated using some tools allowing developers to write code at higher level of abstraction. Implementing these types of programs into the software development process is a good way to boost programmers' performance by focusing on the task at hand rather than implementation details. Current literature on the subject reviews single approach or method. Very few of them are reviewing state of the art in general. This paper reviews the state of the art of automatic programming by overviewing the existing literature on the topic using systematic literature review method. The paper overviews approaches and algorithms of the topic, examines issues and open questions in the field and compares the state of the art to the state of the practice. Of 37 relevant studies, 19 addressed general definitions and subtopics of automatic programming. 30 presented specific algorithms or approaches. 2 of proposed techniques were implemented in practice. Currently, the focus of automatic programming shifted from program synthesis to inductive programming, caused by a breakthrough in artificial intelligence. Definition of the term and subtopics is consistent between scholars. However, formulating correct specification and providing sufficient information for automation is still an open research question.

Keywords:

Automatic programming, program synthesis, inductive programming, code generation

CERCS: P170 - Computer science, numerical analysis, systems, control

Ülevaade automaatprogrmmeerimise hetkeseisust

Lühikokkuvõte:

Automaatprogrammeerimine või koodi genereerimine on teatud tüüpi arvutiprogrammide loomisviis, kus kood genereeritakse mõne tööriista abil, mis võimaldab arendajatel koodi kirjutada kõrgemal abstraktsioonitasemel. Selliste programmide rakendamine tarkvaraarenduse protsessis on hea viis programmeerijate produktiivsuse tõstmiseks, võimaldades neil keskenduda pigem käesolevale ülesandele kui implementatsiooni detailidele. Senises teaduskirjanduses on vaadeldud konkreetseid lähenemisi või meetodeid eraldi. Väga vähesed uurimustööd vaatlevad aga kogu valdkonna viimast taset. Käesolevas töös käsitletakse automaatprogrammeerimist olemasoleva kirjanduse süstemaatilise kirjandusülevaate meetodi abil. Töö teeb ülevaate teemaga seonduvatest algoritmidest, probleemidest ning uurimisvaldkonna avatud uurimisküsimustest ning võrdleb valdkonna hetketaset praktika hetketasemega. Vaaldeldud 37 asjakohasest uuringust tegelesid 19 automaatprogrammeerimise üldise määratlemise ja alateemadega. 30 pakkusid välja konkreetse algoritmi või lähenemisviisi. Esitatud tehnikatest rakendati 2 praktikas. Viimasel ajal on automaatprogrammerimise fookus nihkunud programmide sünteesilt induktiivsele programmeerimisele, mille on põhjustanud läbimurded tehisintellekti valdkonnas. Mõistete ja alateemade määratlus on teadlaste vahel ühtne. Õigete spetsifikatsioonide sõnastamine ja piisava teabe andmine automatiseerimiseks on endiselt lahtine uurimisküsimus.

Võtmesõnad:

Automaatprogrammeerimine, programmide süntees, induktiivne programmeerimine, koodi genereerimine

CERCS: P170 - Arvutiteadus, arvanalüüs, süsteemid, kontroll

Table of Contents

1 Introduction

Automatic programming or code generation is a type of computer programming where the code is generated using some tools allowing developers to write code at higher level of abstraction. During the last years, vast amount of papers have been published concerning automatic programming. Most of them are comprehensive but are focused on a single case, method or algorithm. Very few of them are reviewing state of the art of code generation in the whole [1]. Having research paper examining the state of the art of the field has a big scientific value as it will provide scholars base for the future research.

This paper analyses and reviews state of the art of automatic programming by overviewing existing literature and best practices of the topic.

The main aim is to review existing techniques and algorithms in this space find issues and open questions and compare it to the state of the practice, with an emphasis on the use of these methods for computer program generation.

The research questions this paper tries to answer to are following:

RQ1: How is automatic programming defined by different scholars and what are subtopics of the subject?

RQ2: What are the algorithms and approaches used in the field?

RQ3: What are the limitations and open questions on the topic?

RQ4: How does the level of the state of the practice compare to the state of the art or the subject?

As the main aim of the paper is to formalize the state of the art, many in-depth details will be omitted and simplified from all papers to achieve readability and briefness. Paper encourages the reader to see reference paper for a more exhaustive analysis of an algorithm or approach.

The paper structure is as follows: in Section 2 the background of the topic will be presented. Section 3 will address the research method used in this paper. Section 4 will overview the results of the literature review. In section 5 research question will be discussed and finally in section 6 paper will be summarized.

2 Background

This section of the paper will introduce the field of automatic programming by describing theoretical and technological concepts of the topic. At first, the concept of Metaprogramming and genetic programming will be overviewed. Second, origin and history of the automatic programming will be described. Finally, an overview of different types of automatic programming and used technologies will be presented.

2.1 Metaprogramming

Metaprogramming is a type of programming where computer programs are treating other programs as their data. This means that programs are designed in a way that they are able to generate, analyse, read or transform itself or other programs [2]. This allows developers to reduce codebase, have more flexibility, move computations from run-time to compile-time, to generate code during compilation. Metaprogramming language is called *metalanguage* and the language of the programs that are being manipulated *object language*. The main feature of Metalanguages is *reflection* or *reflexivity* that facilitates the whole concept of metaprogramming.

2.2 Genetic Programming

Genetic programming (GP) is a type of programming where programs are regarded as genes and then are evolved using genetic algorithms (GA). The main aim of GP is to improve the program in performing a predefined task. According to K. Becker [3], the Genetic algorithm takes as an input set of instructions or actions that are regarded as genes. Then a random set of this instructions are selected to form an initial chain of DNA. The whole genome is then executed as a program and results are scored in terms of how well the program solved the defined task. Then top scorers are mated together and offspring is rated again until the desired program is produced. To achieve diversification evolutionary techniques such as roulette selection, crossover, and mutation are used. Even though genetic programming is usually research subject of artificial intelligence its ideas and techniques are occasionally used in automatic programming as well [4].

2.3 History of Automatic Programming

There have been different approaches to this problem and the term definition itself was varying through time. From the early 1940s when term considered the description of the manual

process of punching paper tape. After some time when compilers started to appear, automatic programming was describing a translating process from high-level code to low-level (For example C to bytecode). Even today there is no deterministic formulation of the definition, but the generally accepted meaning of the term is that Automatic programming is programming in a higher level of abstraction than it is available to the programmer [5].

2.4 Automatic Programming

Automatic programming or code generation is a type of computer programming where the code is generated using some tools allowing developers to write code at the higher level of abstraction. There are different approaches to the automatic programming, one of them is *program synthesis* which aims to generate programs based on specifications that are usually “non-algorithmic statements of an appropriate logical calculus" [2]. In contrast to that *inductive programming* aims to derive computer programs from input-output examples or constraints, i.e. incomplete specifications. In order to learn missing specifications, inductive programming utilizes artificial intelligence and machine learning algorithms [5], [1].

3 Method

The review of the literature on this topic is done by systematic literature review and is based on the research protocol and in this section of the paper research-strategy, the sources, the studies selection, the selection execution and the data extraction will be defined.

3.1 Search Process

A search of the appropriate literature was carried out on the databases presented in Table 1 as well as manual search in references.

Table 1. List of databases and corresponding acronyms

Source	Acronym
ACM Digital Library	ACM
IEEE Xplore	IEEE
SpringerLink	SL

The search keywords were “automatic programming”, “inductive programming”, “program synthesis”, “automated refactoring”, “DSL” and “domain-specific language”.

After conducting a search on the above-mentioned databases and after applying inclusion and exclusion criteria, references from and to this papers were analysed to identify additional papers.

3.2 Inclusion and exclusion criteria

The Inclusion criteria were the following:

IC1 The study must be written in English.

IC2 The study must be about automatic programming.

IC3 The study should describe one or more algorithm or technique of automatic programming.

The exclusion criteria were the following:

EC1 The full-texts of the study was not accessible by the library proxy of the University of Tartu.

EC2 Study is non-peer reviewed research.

EC3 Study is about a topic that has already been included in the selection (in that case papers are evaluated compared to each other and the better one is selected. For comparison references to each other are checked and if one paper refers to another the former is excluded. If there are no relevant references then common reference on that topic is found and this paper is included instead of other two).

3.3 Study Selection

The Literature review was carried out by searching with the keywords defined in section 3.2 on databases in Table 1. The results of the initial first search are presented in Table 2.

Table 2. Initial search results by query and database

Query/Database	**automatic programming**	**inductive programming**	**automated refactoring**	**DSL & domain-specific language**	**Program Synthesis**	**Total**
ACM	253	15	56	326	154	804
IEEE	2,035	3	44	437	212	2,731
SL	451	33	10	179	97	770
Total	**2,739**	**51**	**110**	**942**	**463**	**4,305**

After the first iteration of searching, **4,305** papers were identified to apply inclusion and exclusion criteria.

Before applying inclusion criteria duplicates were rejected from each library as shown in Table 3.

Table 3. Duplicates identified in the results

Database	**Duplicates Removed**	**Total Left**
ACM	284	520
IEEE	338	2,393
SL	125	645
Total	**747**	**3,558**

Additionally, 412 duplicates were rejected from the results of all libraries combined, thus leaving only 3146 papers for applying inclusion and exclusion criteria on.

Inclusion criteria were applied one criterion at a time followed by exclusion criteria.

Results of applying inclusion criteria are following:

- 2969 papers satisfied IC1.
- 784 papers satisfied IC2 based reviewing their titles.
- 124 papers satisfied IC3 based on their abstracts.

Results of applying exclusion criteria are following:

- EC1 eliminated 8 papers, leaving 116.
- EC2 eliminated 10 papers, leaving 106.
- EC3 excluded 75 papers, leaving 31.

Then citations and references of selected papers were analysed manually to find additional 6 relevant papers that were not found by querying databases.

Finally, 37 papers were selected for the review. These papers are listed in Appendix I.

3.4 Data Extraction

Selected papers were further analysed and relevant data for the research questions were extracted. During extraction the following extraction criteria were taken into consideration:

EC1: Title, abstract, authors, publishing year

EC2: Theoretical overview (definitions, subtopics etc.)

EC3: Algorithm, approach or technique researched in paper

EC4: Issues, problems and limitations of the EC3

EC5: Tools utilizing proposed algorithm, approach or technique

Data extracted from the papers with the criteria defined above was used to answer research questions and categorize papers. Specifically, Table 4 shows the mapping of research questions to corresponding data extractions criteria.

Table 4. Research question mapping with Data extraction criteria

Research Question	**Data extraction criteria**
RQ1	EC1, EC2, EC3
RQ2	EC2, EC3, EC5
RQ3	EC2, EC4
RQ4	EC5

Papers were classified into following criteria (Appendix I):

- Compilers
- Program Synthesis
- Inductive Programming

Next chapter will formalize the state of the art of the automatic programming based on the classification defined above.

4 Results

The state of art of the automatic programming can be split into three categories: *compilers*, *program synthesis* and *inductive programming* [1]. Compilers are usually considered separately as they do not tend to create new logic not defined in the original code but translate latter into a different programming language [6]. As for program synthesis and inductive programming, even though these two terms are sometimes used interchangeably, they are disparate and utilize contrasting principles for code generation.

To begin with, it is important to mention that setting unreal expectations to the automatic programming is a frequent issue and we need to be cautious when analysing purposes of the automatic programming. Claims, that automatic programming systems do not need domain knowledge or that it is possible to implement fully autonomous general-purpose programming framework has no empirical evidence [7]. Discussing automatic programming in that perspective is a philosophical debate that can be related to artificial general intelligence (AGI).

The first comprehensive paper on code generation was *Automatic Programming: A Tutorial on Formal Methodologies* by Alan W. Biermann (1985) who was the first one to divide concept into program synthesis and inductive programming and analysed multiple approaches from both sides [1].

In this section, the state of art of the automatic programming will be analysed based on the papers listed above. For the base of the discussion, this research will use Biermann's paper and supplement it with more recent works, as well as analyse completely different approaches to this topic. First, automatic programming will be reviewed form the perspective of compilers, followed by program synthesis and finally inductive programming.

According to Gulwani [8], Automatic programming problem can be categorized into three main dimensions: *intent specification*, *program space*, and *search strategy*. Intent specification implies how should the desired application be specified and what information should be included in the specification. Program space defined search space of the desired program with domain-specific insights such as DSLs. The search strategy is concerned with what approaches and methodologies should be used for actual synthesis (Deductive synthesis, divide and conquer of strategic approach, syntax-guided synthesis, etc.).

Domain-specific language

A domain specific language (DSL) is a is a special purpose computer language that is specialized for computations in a specific domain [9] One good example of DSL is Structured Query Language (SQL) which by itself does not comprehend anything that cannot be expressed using general purpose languages such as C or Java. SQL encompasses actions and functions required for communication with the database engine.

In the literature, there are two major types of DSLs: first-class language and embedded language as shown in Figure 1. The first-class language has its own interpreter and compiler and only operates in a predefined environment, while embedded languages are closely tied to the host language and utilizes latter's semantics, however, they may have distinct look and feel.

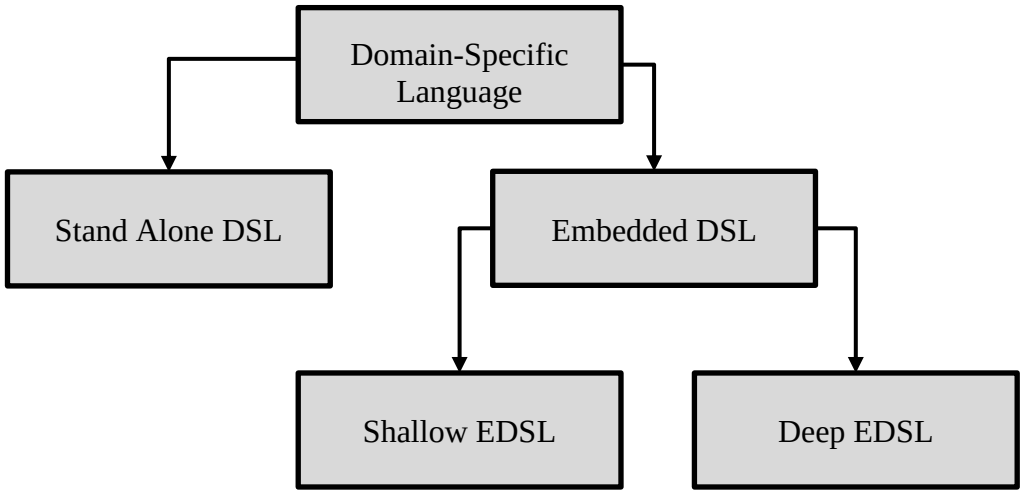

Figure 1. Type of domain-specific languages [9]

Embedded DSL (EDSL) has the look and feel and semantics of a language it is embedded to. But it is specialized for a specific domain. Since it resembles host language very much developing and maintaining DSL is easier. According to A. Gill [9], Haskell's concise syntax made developing the vast amount of EDSLs possible. Combining multiple terms in DSL gives us deeply embedded DSL which can be considered as a library of provided functions [9]. In contrast to shallow EDSL, where values are computed directly, deep EDSL constructs

syntax trees. The result of computation of shallow EDSL is valued while deep EDSL returns structure that can later be evaluated into a value.

4.1 Compilers

Computer programs are written in programming languages that specify specific command-set in their syntax. A computer processor, on the other hand, accepts sequences of instructions in contrast to program texts. The computer in order to be able to understand program text first needs to be translated into proper instructions. This process can be automated by a specific program referred as *compiler* and the translation process itself is called compiling [10]. More formally, a compiler is a computer program that is used to translate source code written in one language to another programming language [6].

There are many different types of compilers designed for different purposes. The term is usually referred to the software that translates high-level language code (e.g. C/C++) into a lower-level language (assembly, machine code). Other types of compilers include: cross-compiler that has ability to compile code for different CPU or architecture; bootstrap compiler that is written in the same language as the target [11]; decompiler that translates code from low-level language to high-level [12]; source-to-source compiler that translates between two high-level programming languages [6]; and compiler-compiler used for designing syntax analysers.

The compilers must follow two fundamental principles: *The compiler must preserve the meaning of the program being compiled* and *the compiler must improve the input program in some discernible way* [6]. The first principle cannot be ignored as if we allow change of the meaning, what will limit compiler to produce completely different program then we intended to write. The second principle is more practical than formal, if we define improvement as optimization, for example making the program perform faster, then the compiler can overlook this principle. But if improvement is understood as a creation of a value, for example, the creation of executable program from source code, then the compiler is indeed improving input.

The compiler usually adopts separation of concerns principle and is structured in two or three parts: *Front-end, Middle end* or *Optimizer* and *Back-end.* Difference between two and three-part structure is that two-part structure does not have optimization. Figure 2 presents the structure of a common compiler.

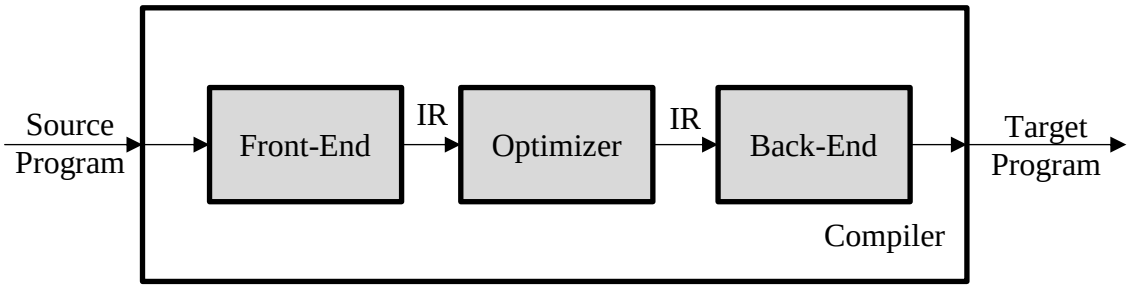

Figure 2. Structure of a common compiler [6]

Front-end receives source code and is responsible for analysing and building an *intermediate representation* (IR) of it. Usually, front-end process is divided into following parts: line reconstruction, pre-processing, tokenization, syntax analysis, semantic analysis.

Middle end or optimizer receives intermediate representation as an input runs it through series of optimization algorithms and outputs improved intermediate representation. The focus of each optimization algorithm varies from compiler to compiler. Following are the most frequent optimizer directions: speed, size, number of page faults, and energy consumption.

Back-end receives output from the optimizer, in case of a two-part compiler, or from the front end, in case of three-part compiler, and translates it to CPU and architecture specific instruction set. Its main aim is to maximize the efficiency of available resources utilization. The backend is a multi-stage process that includes: Machine dependent optimizations, which rewrites particular instructions of assembly language to be more efficient on the given processor; Code generation, which refers to translation from assembly language to machine language.

The benefit of multi-part compilers and separation of concerns principle is that front and back ends can be changed with different ones, to support numerous languages, processors and architectures, while maintaining optimizations. This is frequent practice and the biggest and most common example if GNU Compiler Collection (GCC) that supports multiple languages and platforms.

4.2 Program Synthesis

Program synthesis is a process of the deriving program from a given full specification. The desired program should be synthesized by specifying features or behaviours but not exact

algorithms. Usually, specification consists of human-provided insights and general description of the domain. Regardless it should be specified enough information to successfully derive desired program [13], [1]. Even though computers' main job is to derive the low-level details, human intervention by specifying additional insights is valuable [14]. More formally, the program synthesis addresses the problem of finding function *f* such that will satisfy function ϕ that checks the correctness of *f* in terms of specification (i.e. ϕ stating that *f* is a valid program) [13].

Many approaches of program synthesis relay on mathematical theorem proving algorithms as well as metaprogramming, however, more recent studies started researching different approaches to program synthesis such as A. Solar-Lezama's *Program Synthesis by Sketching* [14]. Moreover, an essential part of this subtopic relays on domain-specific languages.

In the following chapters, the current state of the art of the program synthesis will be formalized.

Strategical Approach

Strategical approach for program synthesis was first introduced by Bibel and Hornig in 1984 [1]. They proposed logical program synthesis system LOPS whose main features include acquiring specification, manipulating domain, proposing and proving critical theorems and last but not least constructing code.

The main idea behind the strategical approach is to identify the relationship between the input and output. The system does this by searching for the portion of the input that can give any insight on output. If that kind of portion is found then system tries to find a repetition of the original specification with already processed part of input and it continues reducing the problem that way until all of the input is a process, thus completing the synthesis.

However, when required theorems are not available to process input further, synthesis blocks and invokes model exploration capability, which on the other hand tries to generate additional examples to derive missing theorem by generalizing the examples observed previously.

This approach resembles human reasoning to find new relationships in the domain and to generalize on the previous evidence. Theorem proving methodologies and generalization from examples capabilities are a crucial part of this approach [1].

Divide and Conquer

The widespread approach in developing software is the "top-down" design, where the original problem specification is divided into subtasks. Then each subtask is analysed, it can be trivial enough to be solved directly or still complex as original specification. If they are not primitive enough, they are further broken into parts, until all of the subtasks are simple enough to be solved individually. After that, the decomposed and solved subtasks are combined together to assemble a complete solution. The divide and conquer methodology of automatic programming as reviewed by A. Biermann [1] follows the strategy defined above. It partitions the problem input into multiple parts and tries to solve each sequence individually. In other words if the desired program is *f*, which accepts input *x*, with divide and conquer approach system will first check if *f* is simple enough to be solved directly, otherwise, it will split *x* into parts and repeats the process on each part of *x* individually. Finally, the results of each computation are collected together to produce the result of *f*. The main application of this approach is the class of sorting programs, where the different decomposition operator can generate different sorting programs.

Deductive Program Synthesis

There are several approaches to the program synthesis. One of them is deductive program synthesis pioneered by Z. Manna and R. Waldinger [15]. In this approach, program synthesis is considered as a problem of proving a mathematical theorem and utilizes theorem-proving methods such as transformation rules, mathematical induction and unification. The authors introduced *deductive tableau*, which is a two-dimensional structure used for encompassing the proof [16].

To overview proposed method we need to define formal terms and notions used in the paper. *The terms* encompass constants and variables, expressed in Latin symbols. The term can be constructed by application of different functions such as $f(a, g(a, x))$ [16]. Additionally, *if-then-else* is defined which is considered as a term constructor. *Atomic sentences* construction consists of applying predicate symbols (p, q, r) to terms. *Sentences* consist of atomic sentences and truth symbols *true* and *false*, their construction is based on the application of connectives $(\wedge,\ \vee,\ \neg, ...)$ and quantifiers $(\forall x, \exists x)$ to other sentences as well as conditional connective (*if-then-else*) for implication.

Paper also defined different types of expressions: *closed* which does not contain any free variables*; ground* which does not contain any expressions at all; *and herbrand* which is a

ground expression that does not contain any connectives, term constructors or equality symbols.

The total reflexive theory TR is defined by the following two axioms:

$$u \succcurlyeq u \ (reflexivity)$$

$$u \succcurlyeq v \vee v \succcurlyeq u \ (totality)$$

Table 5 presents basic deductive tableau, which is denoted by assertions and goals, rows and output columns. The tableau represents proof itself and derivation and the output columns are used for extracting a program. The authors present several features for deductive tableau: suiting, satisfying and equivalence that is then used in deriving deduction rules and finally deducting required program [16].

Table 5. Deductive Tableau [15]

Assertions	Goals	$f_1(a)$	…	$f_n(a)$
A_1		s_1		s_n
	G_1	t_1		t_n

rows

Output columns

Deduction rules add new rows to a tableau. They intend to preserve similarity but can violate equivalence. In other words, they leave primitive closed terms that satisfy the tableau untouched. Accordingly, the main feature of deduction rules is that by applying them to the program that satisfies the tableau, the program stays same.

The deductive tableau framework defined following deduction rules:

- Splitting rule, that splits rows into logical components.
- Resolution rule takes sub-sentence of two rows and performs truth case analysis on them.

- Equivalence rule that takes sub-sentence and replaces it with an equivalent sentence.
- Skolemization rule that removes quantifiers
- Equality rule that replaces sub-term with an equal term
- Mathematical induction rule that assumes the correctness of the desired program on inputs smaller than the given one.

To illustrate the complete process of deducting program Z. Manna and R. Waldinger [15] present the following example:

The desired program is to find two outputs for a given nonempty string *s*:

- the last character of *s* – *last(s)*
- all but the last character of *s* – *front(s)*

For example: for the *s ="example"* program should output O_1 *="example"* and O_2 *="e"*

The formal specification of this problems is as follows:

$$\langle front(s), last(s)\rangle \Leftarrow \begin{Bmatrix} find\ \langle z_1, z_2\rangle\ such\ that \\ if\ \neg(s = \Lambda) \\ then\ char(z_2) \wedge s = z_1 * z_2 \end{Bmatrix}$$

Table 6 represents complete deductive tableau of the defined problem.

Table 6. Complete Deductive Tableau

Nr	Assertions	Goals	$front(s)$	$last(s)$
1		$if\ \neg(s = \Lambda)$ $then\ char(z_2)$ $\wedge s = z_1 * z_2$	z_1	z_2
2	$if\ \neg(s = \Lambda)$			
3		$char(z_2) \wedge s$ $= z_1 * z_2$	z_1	z_2

4		$char(z_2) \wedge s = z_2$	Λ	z_2
5		$char(s)$	Λ	s
6		$char(u) \wedge char(y_2) \wedge s = u \cdot y_1 * y_2$	$u \cdot y_1$	y_2
7	$if\ x' \prec_w s$ $then\ if\ \neg(x' = \Lambda)$ $then\ char\big(last(x')\big)^- \wedge$ $x^i = front(x^i) * last(x^i)$			
8		$x \prec_w s \wedge \neg(x = \Lambda) \wedge char(u) \wedge char\big(last(x)\big)^+ \wedge s = u \cdot x$	$u \cdot front(x)$	$last(x)$
9		$x \prec_w s \wedge \neg(x = \Lambda) \wedge char(u) \wedge s = u \cdot x$	$u \cdot front(x)$	$last(x)$
10		$\neg(s = \Lambda) \wedge tail(s) \prec_w s^+ \wedge$	$head(s) \cdot front(tail(s))$	$last(tail(s))$

		$\neg(tail(s) = \Lambda)$ $\wedge$ $char(head(s))$		
11		$\neg(s = \Lambda) \wedge$ $\neg(tail(s) = \Lambda)$ $\wedge$ $char(head(s))^{+}$	$head(s)$ $\cdot front(tail(s))$	$last(tail(s))$
12		$\neg(s = \Lambda) \wedge$ $\neg(tail(s) = \Lambda)^{+}$	$head(s)$ $\cdot front(tail(s))$	$last(tail(s))$
13		$\neg(s = \Lambda)^{+} \wedge$ $char(s)$	$head(s)$ $\cdot front(tail(s))$	$last(tail(s))$
14		$\neg char(s)^{-}$	$head(s)$ $\cdot front(tail(s))$	$last(tail(s))$
15		$true$	$if\ char(s)$ $then\ \Lambda$ $else\ head(s)$ $\cdot front(tail(s))$	$if\ char(s)$ $then$ s $else\ last(tail(s))$

First, we begin with the basic goal line (1), then by applying if-split, we get a line (2) and (3). Then by the equality rule, we derive (4). With applying resolution rule to the reflexivity axiom we obtain (5). With applying equality rule to the axiom for concatenation we get (6). Then we apply induction rule on (1) to get (7). With applying equality rule to (7) and (6) we get (8). By applying resolution rule to (7) we get (9). With applying resolution rule to de-composition property of strings and to (9) we get (10). Then with the resolution rule twice we get (11) and (12). Then applying resolution rule to the trichotomy property of strings we get (13). By applying resolution rule to (2) we get (14). And finally, with the applying res-olution rule to the (5) and (14), we obtain (15), thus the final program.

Syntax-Guided Synthesis

In syntax-guided synthesis (SyGuS) a programmer specifies the outlines of a program, i.e. incomplete program and the synthesizer tries to fill in the missing details. To do so the system needs user-defined specifications and assertion [13].

According to R Alur, et al. [13] syntax-guided synthesis had many potential benefits compared to classical synthesis problem that consists of only the specification. Few of the potential advantages include:

- Candidate set *L*, which limits search space of possible implementations, thus gaining computational performance for solving the synthesis problem.
- Flexibility to specify the desired programme with the use of syntactic and semantic constraint.
- This approach can be considered for machine learning and inductive inference as in the end results in finding correct expression from the syntactic space.

Additionally, syntax-guided synthesis defines three constraints:

1. Background theory constraints logical symbols and their interpretation.
2. Background theory limits specification ϕ to a first-order formula and all its variables are universally quantified.
3. Syntactic expression in grammar restricts the universe of possible functions *f*

Accordingly, as R. Alur et al. [13] states, the syntax-guided synthesis problem can be formalized in the following way:

> *Given a background theory T, a typed function symbol f, a formula ϕ over the vocabulary of T along with f, and a set L of expressions over the vocabulary of T and of the same type as f, find an expression $e \in L$ such that the formula $\phi[f/e]$ is valid modulo T.*

Oracle-Guided Synthesis

Oracle-Guided Synthesis researched by Susmit Jha et al. [17] is an approach of automatic programming that is designed for loop-free programs. The method is a fusion of oracle-guided learning from examples and constraint-based synthesis which by itself uses satisfiability modulo theories (SMT) solvers. The main application of this approach is bit-manipulation programs and deobfuscation (converting difficult to understand program into simpler one) programs. The method takes as an input the following:

- A validation oracle V, that checks whether the synthesized program is the desired one.
- An I/O oracle I, that returns correct output for given input.
- A set of specifications $\{\langle \vec{I_\iota}, O_i, \phi_i(\vec{I_\iota}, O_i)\rangle | i = 1, \dots, N\}$ called *library*. Where $\vec{I_\iota}$ s list of input variables, O_i is output variable, and $\phi_i(\vec{I_\iota}, O_i)$ specifies the relationship between input and output.

The goal of the oracle-guided synthesis approach is to find program *P* that satisfies following requirements:

- can be checked with validation oracle *V*
- only utilizes components from the library
- takes $\vec{I}$ as an input and outputs *O*
- Uses $\{O_1, \dots, O_n\}$ as an intermediate or temporary variables
- Has form defined in (1)

$$P(\vec{I}): \quad O_{\pi 1} \coloneqq f_{\pi 1}(\vec{V}_{\pi 1}); \quad \dots; \quad O_{\pi n} \coloneqq f_{\pi n}(\vec{V}_{\pi n}); \quad return\ O_{\pi n} \quad (1)$$

Additional constraints for the form are that $\vec{V}_{\pi i}$ is either input variable or a temporary variable (C1); and $\pi_i, \dots, \pi_n$ is a permutation of $1, \dots, n$ (C2).

The synthesis procedure is the following:

1. Encode the space of all possible programs with a formula.
2. Constraint formula based on input-output pairs.
3. Solve constraint defined previously.
4. If the solution is not the desired program then generate new input-output pair.
5. Constraint formula further and check again until the correct solution is found.

For encoding, authors define THEOREM 1 that state that there exist two formulas ψ_{wfp} and ϕ_{func}. First one represents *syntactically* well-formed programs, while the second formula embodies all *semantic* I/O behaviours of a well-formed program.

> *THEOREM 1. There exists a set of integer-valued location variables L, a well-formedness constraint* $\psi_{wfp}(L)$ *over L, a mapping* Lval2Prog, *and a functional constraint* $\phi_{func}(L, \vec{I}, O)$ *over* $L \cup \{\vec{I}, O\}$ *such that the following properties hold:*

- Lval2Prog *is a bijective mapping from the set of values L that satisfy the constraint* $\psi_{wfp}(L)$ *to the set of programs that satisfy constraints C1 and C2.*
- *Let* L_0 *be a satisfying assignment to the formula* ψ_{wfp}*. If* α *and* β *are any candidate input and output values, then the formula* $\phi_{func}(L_0, \alpha, \beta)$ *is true iff the program* Lval2Prog (L_0) *returns the value* β *on the input* α*.*

This paper will not the proof of this theorem for the sake of simplicity but it can be obtained from [17].

The key step in the oracle-guided synthesis is defining constraint (I/O-behavioral constraint) whose solution will guide us to the program desired.

I/O-Behavioral constraint is denoted with the following formula:

$$Behave_E(L) = \bigwedge_{(\alpha_j,\ \beta_j \in E)} \phi_{func}(L_0, \alpha_j, \beta_j) \quad (2)$$

Where E is a set of input-output examples.

Authors then define THEOREM 2 which gives us the ability to search for a candidate program in the finite search space of input-output pairs.

> *THEOREM 2 (I/O-behavioural Constraint). For any satisfying solution* L_0 *to the I/O-behavioural constraint, the input-output behaviour of the program* Lval2Prog (L_0) *matches all the input-output examples in the set E.*

Program synthesized with the above constraints then need to be checked with the validation oracle and since it is an expensive operation Susmit Jha et al. [17] define an additional *distinguishing constraint* that differentiates between two candidate programs:

$$Distinct_{E,L}(\vec{I}) = \exists L', O, O'\ Behave_E(L') \wedge \phi_{func}(L, \vec{I}, O) \wedge \phi_{func}(L', \vec{I}, O') \wedge O \neq O' \quad (3)$$

> *THEOREM 3 (Distinguishing Constraint). If* α *is a satisfying solution to the distinguishing constraint* $Distinct_{E,P}(\vec{I})$*, then there exists a program P' such that P and P' have different behaviours on input* α*, but have the same behavior on all the inputs in the set E.*

The synthesising procedure generates new programs in a loop that satisfy more and more inputs. In the beginning, only one input is selected corresponding program is synthesized

and then using distinguishing constraint different input is searched. If it is found then next iteration begins with expanded input set and the loop continues until the correct program is found. Otherwise, the system returns stating that not enough information is provided for successful synthesis. Additionally, Satisfiability Modulo Theory (SMT) is used for solving I/O behavioural and distinguishing constraint.

Program Synthesis by Sketching

The combination of high-level design and low-level algorithms is required for implementation of complex challenging programs. Even though computers' main job is to derive the low-level details, but without human intervention by specifying additional insights synthesizers cannot complete its tasks [14]. Therefore establishing synergy between the synthesizer and the programmer is one of the main challenges in program synthesis.

To approach this challenge, A. Solar-Lezama [14] in 2003 introduced a new strategy for synthesis *sketching* which tries to find a solution to the synergy problem. As it is already shown in the paper previous work on this subject was utilizing meta-programming or theorem proving algorithm to deduce an implementation form the specification. One of the problems with synthesized programs was that they were hard to read and understand and programmers needed to get used to the way of thinking to efficiently extend or reuse synthesized programs.

Sketching, on the other hand, removes that complexity form synthesis by allowing programmers to specify sketch, outline or high-level structure of an implementation, leaving low-level details to the synthesizer. Sketching utilizes Boolean satisfiability problem based synthesize which using a just small number of test cases can efficiently synthesize an implementation. Authors of this research also define counterexample guided inductive synthesis procedure (CEGIS) that uses the above-defined synthesizer as its core additionally adds validation procedures. This procedures can automatically generate test inputs and validate generated program correctness in regards to specification. Moreover, CEGIS can operate of concurrent program and is able to solve problems ranging from bit-level cypher to manipulation of linked data structures. As authors claim [14] programs that the system was able to synthesize complete optimized AES cypher, multiple concurrency problems such as a fine-locking scheme for a concurrent set, the solution to the dining philosophers problem, and sense reversing barrier.

4.3 Inductive Programming

Inductive programming is a subtopic of automatic programming which is focused on generating programs from incomplete specifications, like input-output examples, in contrast to program synthesis where specifications are usually complete. The research of inductive programming spans from programming to artificial intelligence and mainly research functional and recursive programs [18], [19].

There are multiple types of inductive programming depending on the programming paradigm. For example, inductive functional programming is used with the functional paradigm or inductive logic programming used in logic programming languages such as Prolog.

According to S. Gulwani et al. [20], the input of inductive programming system is an incomplete specification that usually consists of:

- Inputs and corresponding output examples, occasionally instead of outputs output-evaluation-function is being passed to the system. For describing intended behaviours.
- Action sequences describing the process of computing provided outputs
- Logical constraints to limit the search space of possible programs.
- Background knowledge, or domain specification
- Program templates, predefined functions and heuristics or other biases.

The output of an inductive programming system is a program consisting of any Turing-complete representations, such as loops, conditionals, recursive controls, etc.

In contrast to program synthesis where provided specification should be complete, inductive programming operates on an incomplete specification such as input-output examples, thus generated program should be correct in terms of provided partial specification [21].

The prominence of inductive programming is mostly based on its potential for enabling programmers to provide examples instead of full specifications. Examples are much easier to write than logical specifications as they can be incomplete. However, according to O. Polozov [22], inductive programming has three major limitations:

1. Requirements of deep domain-specific insight.
2. Extensive implementation efforts that can take up to 1-2 man-years.
3. Lack of extensibility as a small change in the underlying DSL causes non-trivial changes to the system implementation.

IGOR II

IGOR II [23] is the inductive programming system designed to learn recursive programs. To achieve full synthesis the system separates input partitioning or finding patterns and predicates in the input, and code synthesis that computes the desired output. IGOR II does partitioning completely and systematically instead of randomly or using greedy search and does this in parallel.

According to M. Hofmann [23], a search of the desired expression is complete and traceable even for a complex program. The reason as authors claim is the construction of hypotheses in IGOR II is data driver and that the system combines analytical program synthesis with the search.

The system defines *E* the set of example equations and *B* a set of background knowledge and both of them have constructor terms on their right-hand side and both describe input-output behaviours on their domain. This two notions together with the declaration of all used data types, the IGOR II system outputs a set of equations P which is correct in the following space:

$$\begin{cases} \forall (F(i) = o) \in E \\ F(i) \overset{!}{\rightarrow}_{P \cup B} o \end{cases} \qquad (1)$$

The result is constructor system which rewrites each left-hand side of equations form an example set *E* to its right-hand side.

Let's define P as the signature of the induced program. Then it is possible to define signature Σ of $E \cup B$ and $P = \Sigma \cup D_A$. Where D_A is a set of function not in the example of background sets, but defined through synthesis.

The system constraint auxiliary function is two ways:

- Input types of auxiliary function and function calling should be identical. This limits IGOR II to automatically infer auxiliary parameters
- Auxiliary function symbols are constrained with the right-hand side of calling function, i.e. these symbols cannot occur on the right-hand side.

These restrictions are called language bias [24]. By definition of property (1), there are infinitely many solution P, accordingly IGOR II, as almost all inductive programming methods, define preference bias that chooses the most appropriate solution form the set of possi-

ble solutions. The selecting criteria for the system are the number of subsets used for partitioning the example inputs, and since search is complete it is possible to systematically select the best solution.

The start state or the initial hypothesis is a constructor system (CS) with one rule for each target function in *E*. Next the best hypothesis is selected using preference bias, an unfinished rule is chosen and system replaces it with its successor rules. IGOR II contains multiple functions for computing successor rules, thus results of substation can be multiple and the best one is selected from this set using preference bias [24].

The system defined three different methods for successor rule replacement:

- The splitting rule by pattern refinement, which uses pattern refinement to split rules and replace rule *p* with at least to or more specific patterns. That way the system achieves a case distinction.
- Introducing auxiliary functions and generating new induction problems on, in other words, new example equation in set *E*.
- Function calls that either recursively calls itself or other defined function from target background, or auxiliary function sets. It should be noted that finding arguments for such functions are perceived as separate induction problem.

The process then continues to apply preference bias on the hypotheses and finishes when at least one of them is finished. At that point, induction is finalized and derived program is outputted [23].

Automatic Design of Algorithms through Evolution (ADATE)

Automatic design of algorithms through evolution (ADATE) is a framework for automatic programming that was introduced by R. Olsson [25]. The system induces functional programs containing recursion, invented auxiliary functions and numerical constants. ADATE's flexibility allows programmers to generate programs from scratch or improve existing ones.

ADATE works very well with high-performance applications where optimizations happen experimentally, such as implementations of heuristics. Since heuristics that give very high-quality results efficiently are difficult to design, automating this task is very beneficial [26] [27].

In more detail, ADATE maintains a population of programs structured hierarchically. The most important entities in the ADATE taxonomy are families which itself is divided into races which in turn consists of species. Species consists of similar programs that are derived from one single founding program using compound program transformations. As in genetic programming main principle for ranking population is that program should perform better than previously observed ones. After each transformation in varying combinations newly generated programs are ranked to be considered in the population. The transformation process is systematic and does not introduce any randomness [25].

Object-Oriented Design and Genetic Programming

The system proposed by N. Pillay and C. Chalmers [4] consists of two components: a rule-based expert system and a genetic programming system, and aims to induce object-oriented programs from specifications and input-output examples. The expert system uses problem specification to generate an object-oriented design (OOD), while GP system induces the methods of each class.

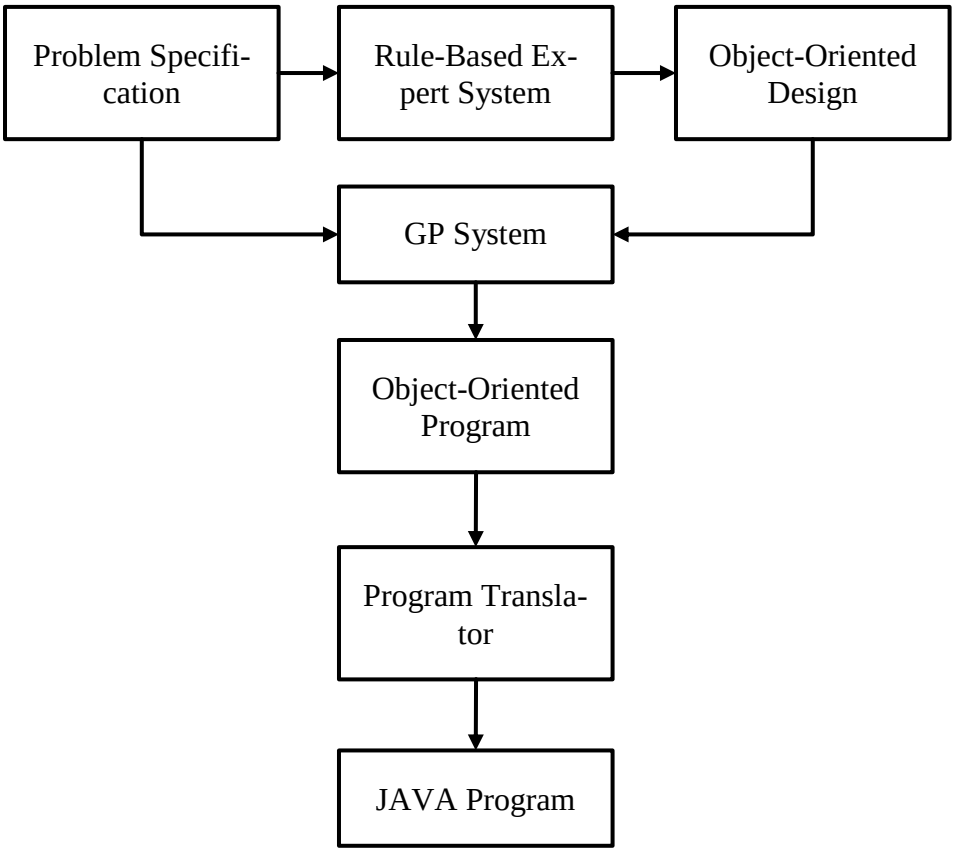

Figure 3. Overall Process [4]

The system takes problem specification as an input, the latter consists of the application domain represented as descriptions of entities and several input-output examples. With this method, programs are generated in an internal representation language but creating a translator for converting it into convenient programming languages. The overall process is visualized in Figure 3. The specification of each entity consists of its characteristics and behaviours. For example, if we have bank cheque as an entity "CHEQUE" its characteristics would be account number, profile and balance, while its behaviours would be a deposit and withdraw. In the specification, behaviours are represented as an input and output values. The specification also supports constants for defining application domain. It should be noted that the system is typed, accordingly in the specification types of each characteristic and input-outputs should be defined. Figure 4 shows example specification for the CHEQUE entity.

```
CHEQUE
accountnumber balance profile
integer real character

*Deposit*
#Input amount Real 100 , 50 , 200
#Input balance Real 1000 , 2300 , 400
#Output balance Real 1100 , 2350 , 600

*Withdrawal*
#Input amount Real 100 , 50 , 200
#Input balance Real 1000 , 2300 , 400
#Output balance Real 900 , 2250 , 200
```

Figure 4. Example Specification [4]

The actual synthesis is done using genetic programming. For each method of the entities that need to be induced, the specification is passed to the genetic programming system,

which employs the generational control model [4]. The system then generates an initial population of programs that are constructed by randomly selecting elements from the functions and terminal sets. Then this population is refined by using genetic programming techniques such as evaluation, selection and regeneration, to generate better programs. This process continues until the desired program is synthesized or until a predefined number of iterations is exhausted.

Data-Driven Domain-Specific Deduction

Data-driven domain-specific deduction (D^4) is novel approach by O. Polozov and S. Gulwani [22] that unifies deductive, syntax-guided, and domain-specific synthesis approaches into one meta-algorithm by utilizing strengths of each approach:

- D^4 operates over a DSL, thus its program space is syntactically restricted.
- D^4 reduces synthesis problem into smaller sub-problems by utilizing deductive synthesis techniques.
- D^4, in contrast to deductive synthesis, were program space is searched with a complete logical specification, operates on input-output examples thus it is data-driven.

In D^4 uses I/O examples for specifying intent, utilizes syntax-guided synthesis for reducing search space and for search deductive strategy is employed. Since program space is limited with the provided DSL, as in SyGuS, it drastically speeds up a search of the desired program (as authors claim synthesis of commons real-life tasks take less than a second [22]). The easiest form of specification is input-output specifications, this approach also employs a similar strategy with the addition of output properties, which is input states map to some properties of the output. Search strategy for D^4 is a novel *deductive inference* that is based on witness functions. These functions capture the inverse semantic of underlying DSL operator. Moreover, the deductive inference is combined with enumerative search.

FlashMeta

FlashMeta is a declarative framework based on D^4, which facilitates design, implementation and maintenance of efficient inductive program synthesizers [22]. The system operates on DSL that synthesizer developer should parameterize and provide as an input. Then FlashMeta automatically generates an inductive synthesizer that will utilize provided DSL. Additionally, the system provides a predefined library of witness functions and generic operators that can be reused by the developers with any conformant DSL. As authors claim [22],

a predefined modelling of properties of operators, it becomes a lot easier to develop synthesizer as only required task for a developer is an exploration of various design choices in DSL structure. The most notable examples of using FlashMeta framework in the industry is FlashFill and FlashExtract, shipped with Microsoft Excel and Windows PowerShell respectively [22].

FlashFill is a system for synthesizing string transformation is spreadsheets from input-output examples. It defined DSL L_{FF} which takes a tuple of user inputs as an input and outputs some transformation of input. It should be noted, that FlashFill was implemented manually by S. Gulwani in 2011 [28] but rewriting it with the help of FlashMeta framework resulted in 7 times less time and efforts and the system discovered optimizations that were not exploited in the original implementation.

FlashExtract is a system for synthesizing scripts for extracting data from unstructured documents [29]. Currently, it is integrated into PowerShell 3.0, shipped with Windows 10, and Azure Operational Management Suite for analysing logs. As FlashFill, FlashExtract operates on specific DSL L_{FE} that takes the textual document as an input and outputs sequence of spans in that document. Selection of spans is done using *Filter* and *Map* functions applied to the document provided.

Neuro-Symbolic Program Synthesis

Neuro-Symbolic Program Synthesis (NSPS) is a novel technique by E. Parisotto [30] that can be trained to generate the desired program incrementally without the need for an explicit search. According to authors NSPS is capable of synthesizing programs based on input-output examples provided a test time. The system utilizes two novel architectures of the modular neural network:

- Cross-correlation I/O network, that produces a continuous representation of input-output examples.
- Recursive-Reverse-Recursive Neural Network (R3NN), that takes input result of cross-correlation I/O network and by incrementally expanding partial programs synthesizes desired program. R3NN has a tree-based architecture and by utilizing rules from a context-free grammar (the DSL) constructs a parse tree.

The formal definition of the NSPS is as follows:

Given a DSL *L*, The system learns a generative model of programs in the *L*. The model is trained on input-output examples to constraint the search space of a consistent program. Figure 5 illustrates the training phase of NSPS which uses a large training set of programs form the DSL with equivalent input-output examples.

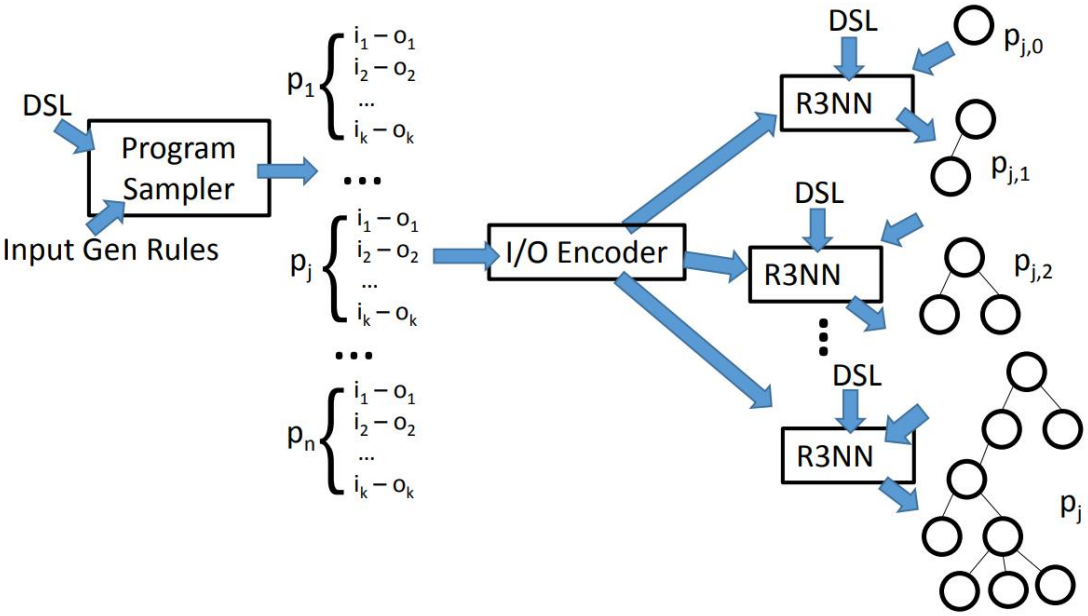

Figure 5. NSPS Training Phase [30]

For a neural network to be accurate it needs large training set to train on. The rule-based strategy is used to compute input string for uniformly samples programs form the provided DSL. Outputs then are generated by running inputs on the selected programs. These tree components sample set of programs, inputs, and outputs combines the training set used for NSPS.

The system treats DSL as a context-free grammar that consists of a start symbol *S*, set of non-terminals and corresponding expansion rules. The most straightforward way to run a search is to start synthesis with the start symbol S and then randomly choose non-terminals to extend tree until every leaf of the tree is a terminal. The NSPS, on the other hand, assigns probabilities to the non-terminals and expansion rules to optimize search for complete derivations. The generative model of the system utilizes Recursive-Reverse-Recursive Neural Network (R3NN) for partial tree encoding. Each node in the partial tree encodes global insight on every other node in the tree. The vector representation is assigned to every symbol

and expansion rule in the grammar. Then given a partial tree, the system applies vector representations to the leaves and recursively backtracks to the root node to encode the global insight. In the end, reverse recursive pass starting from the root node is invoked to update global representations of each node in the tree.

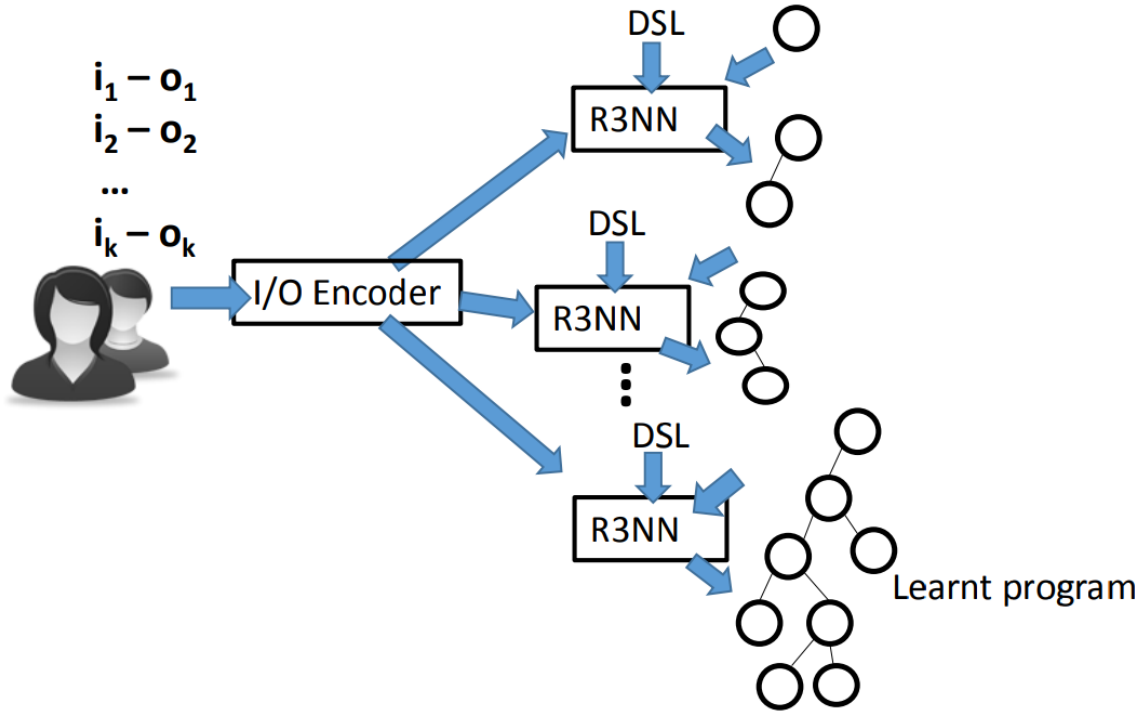

Figure 6. NSPS Testing Phase [30]

After the model is trained it is tested on the testing set of DSL and input-output examples, as shown in Figure 6. Depending on the approaches used for encoding or distributing probabilities the accuracy varies but in the optimal selection as authors claim accuracy is industry applicable [30].

Microsoft research group utilized NSPS to implement RobustFill that leverages data-driven approach to remove any hand-crafted rules from the synthesis [31]. RobustFill uses the *attentional sequence-to-sequence neural network* to synthesize the program from the input-output examples. For the example, authors used FlashFill real-life dataset, trained neural network and then evaluated the outcomes. The system turned out to have 92% accuracy.

4.4 Additional Approaches

The current state of the art of automatic programming consists of additional approaches that cannot be categorized with above-mentioned criteria, i.e. they do not completely fit in the compiler, program synthesis or inductive programming specifications but are still employing some kind of automation. Accordingly, it was decided to include them here as the main aim of this paper is to review the state of the art of automatic programming in depth.

The **AI Programmer** introduced by K. Becker and J. Gottschlich [3] is solely dependent on genetic programming (GP) algorithms but in contrast to previously mentioned approaches AI Programmer can operate only on a tightly constrained programming language consisting of just a few instruction sets, thus drastically limiting the search space and making GP feasible with a minimum human intervention. Table 7 shows the AI Programmers instruction set and gene map.

Table 7. AI Programmer Instruction Set and Gene Map [3]

Instruction	**Gene Range**	**Operation**
>	(0, 0.125]	Increment the pointer
<	(0.125, 0.25]	Decrement the pointer
+	(0.25, 0.375]	Increment the byte at the pointer
-	(0.375, 0.5]	Decrement the byte at the pointer
.	(0.5, 0.625]	Output the byte at the pointer
,	(0.625, 0.75]	Input a byte and store it at the ptr
[	(0.75, 0.875]	Jump to matching] if current 0
]	(0.875, 1.0]	Jump back to matching [unless 0

It should be noted that AI Programmer's language is Turing-complete, i.e. is theoretically capable of performing any programming task (in the scope of the single-taped Turing machine) given unlimited time and memory [3].

Another application of automatic programming is **refactoring legacy code** to utilize new features of the language. The paper by R. Khatchadourian [32] provides an automated approach for translating legacy java code to employ new Java enumeration types. Their algorithm for transformation produces code that is simple, easy to understand, type-safe and free from brittleness problem. The system uses interprocedural type inferencing to track the lifecycle of the enumerated values. Authors claim that their approach can successfully refactor large legacy java projects with a large number of fields.

One more application of automatic programming is in **web-service composition** and **generating API adapter** for cloud-based APIs. The approach by A. Omer and A. Schill [33] proposes methods for solving runtime problems that are occurring in service-oriented architecture (SOA) type environment by composing (semi-) automatically crucial points of the system. Authors also deal with dependencies that are emerging from combinations of independently developed web services. As for generating API adapters, the main problem defined by E. Hossny et al. [34] is that in the cloud environment where multiple services and applications are dependent on each other and are communicating via API calls, slight change of API specification causes big overhead in updated adapter for different dependants. To solve this problem authors propose automatic adapter generation method that is constructed upon semantic annotations and search. This method requires that conceptual meanings of inputs and outputs are specified based on a common domain ontology.

Other less important approaches include **Automatic Model Generation from Documentation for Java API Functions** by J. Zhai et al. [35] this approach takes API documentation of Java libraries whose source code representation in unknown and generate simple model from this documentation allowing developers to easily analyse and understand the functionality that could be difficult to process solely by documentation. The core of the generation is natural language processing (NLP) that has received big attention in the recent years.

One more application of automatic programming is Code completion suggestions to make the development process more efficient. There are several approaches to this problem [36] [37] [38], the most popular one is Being Developer Assistant (BDA) [36] that uses data mining techniques to collect analyse and store sample codes form public repositories such

as GitHub. Authors provide a plugin for the Microsoft Visual Studio that enables developers to automatically, based on their code get sample suggestions mined with the BDA. Another approach for code completion is Statistical Language Models [37] that uses several automatic programming methods and synthesizes code completions based on the partial implementation of the program, it uses machine learning and data mining techniques as well as treats program as a natural language and analyses it with big data collected from public repositories using NLP algorithms, thus providing high accuracy sample codes to fill in the holes in the implementations.

4.5 Limitations and Open Questions

Even though automatic programming is capable of synthesizing complex programs it still has many limitations and open questions for further research. This subsection will overview this issues with regards to RQ3.

To begin with, it should be noted that most of the automatic programming methods are operating on the tightly constrained domain-specific languages and scaling it to even slightly modified language requires rewriting the systems from the scratch. Thus the main limitation of approaches overviewed is the limited scope of operation [20].

Deductive synthesis is limited to first-order input-output relations and only applies to an applicative program set with only output and no side effects. Generally, it is important to research deductive program synthesis for high-level specification such as a description of the properties, not just input-output relations [16].

Divide and Conquer strategy requires very high-level specification that, in some cases, can be regarded as an executable program itself [1].

Deductive and Syntax-Guided Synthesis both require complete formal specification. Writing it is, in most, cases as complex as the implementation itself thus advantages of automatic programming cannot be utilized [30].

Benchmarking and comparison are difficult because most of the automatic programming approaches operate on a very specific narrow domain, thus it is impossible to find a benchmarking task that most of them would be able to work on [39].

For inductive programming the following limitations are apparent:

Domain change: Today most of the IP approaches operate only on a specific DSLs thus changing to a new domain is a hard challenge. The cause of this is that IP approaches are

relatively new to the real-world application and there is not enough experience in exploring the space of the application. In the long term, meta-synthesizers will appear as a dominant approach in IP thus allowing synthesizers to be scalable across different domains [21].

Noise Tolerance: Since inductive programming takes input-output examples as an input, in the real world it is difficult to provide accurate data, thus provided information will always have some noise in them. Currently available approaches cannot identify this noise, accordingly, synthesis accuracy lowers as nose level raises [21]. Only RobustFill, that utilizes Neuro-Symbolic Program Synthesis, is able to treat noise at some level, still returning accurate results and deriving correct desired programs to the end users [31].

Neuro-Symbolic Program Synthesis currently utilizes supervised-learning approach, that assumes the availability of desired programs during training, but in real life, there can be the cases where sample programs are not provided, or there is an oracle that returns correct output for the input. Current research does not give us insight into this problem, indicating that further research is required in that direction [30].

4.6 The State of the Practice

Even though automatic programming is popular research subject since the 1970s [1], but this research remained in the scope of artificial intelligence and did not transfer into the industry. The main reason for that could be that current systems lack usability and are very hard to transfer synthesizer implemented in each approach to a different domain.

First and most notable application of automatic programming in industry level software was FlashFill and FlashExtract shipped with Microsoft Excel [29], [22]. Other applications that provide user friendly automatic programming tools are TerpreT that is a domain specific language for expressing program synthesis problems [40], PushGP which is a family of programming languages suited for evolutionary computations, i.e. genetic programming [41], MagicHaskeller that is an inductive functional programming system for Haskell that uses systematic search for induction [42].

Recently Microsoft launched its new product Meta-synthesizer Microsoft Program Synthesis using Examples (PROSE) that encapsulates FlashMeta [22] framework as well as RobusFill [31] system to provide high level, user-friendly meta-synthesizer that is capable of generating synthesizers for user-defined DSL and train it on input-output examples [43].

5 Discussion

In this chapter evaluation of the state of art and state of practice of automatic programming will be provided.

5.1 Definition and Subtopic

This subsection answers RQ1: How is automatic programming defined by different scholars and what are subtopics of the subject?

As shown in Appendix I, out of 37 papers reviewed, 22 (59%) were on inductive programming, 19 (51%) were on program synthesis and only 5 (14%) were on compilers.

This distribution can be explained by analysing trends and semantics of each subtopic. For example, compilers are usually regarded separately as they do not tend to create new logic not defined in the original code but translate latter into a different programming language [6]. Accordingly, selection keywords for this literature review were more targeted on the papers that were focused on deriving additional insights from the specification, not just translation or compilation.

As it is evident, that inductive programming and program synthesis are selected equally. This can be conditioned for several reasons. First of all, we need to look into the distribution of term trends as shown in Figure 7.

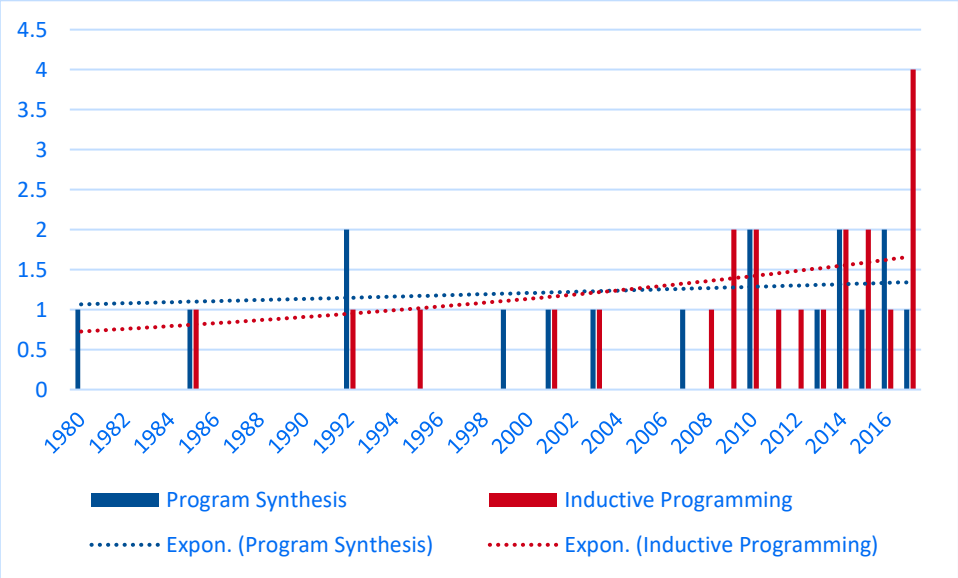

Figure 7. Distribution of research papers on automatic programming

From the graph, if we take a closer look at the trend lines it is easily seen that popularity of inductive programming was more rapidly growing than program synthesis. But since most of the inductive programming approaches utilize program synthesis to some extent they are still reviewing that topic, thus the current indicator of the popularity of program synthesis might be distorted. One reason for that trend is that in the 1990s, artificial intelligence techniques were not researched in an in-depth manner and machine learning, deep learning or neural networks were not that popular research subject, thus the time was spent mostly thinking how to create mathematical models for generation or how to model specifications that can be full and easy to write at the same time. Moreover, it should be noted that last year number of research papers on IP was far above the average number of papers per year, 4 and 1 respectively. 19 papers were discussing general definition or subtopics of automatic programming, out of which 10 (53%) were defining inductive programming, 12 (63%) were on program synthesis and 4 (21%) were on compilers. This distribution is in line with the previous reasoning.

5.2 Algorithms and Methods

This subsection answers RQ2: What are the algorithms and approaches used in the field?

Out of the selected 37 papers, 30 (81%) were focused on one or more approaches to the automatic programming. 6 (16%) were reviewing more than one algorithm or method and 24 (65%) were dedicated solely on a single approach. 2 (5%) papers were an improvement or enhancement of previously introduced research.

5.3 Limitations and Issues

This subsection answers RQ3: What are the limitations and open questions on the topic?

Out of the selected 37 papers, 9 (24%) were discussing the issues, limitations or open questions in automatic programming. From this 9 papers, 3 were the original work for the approach and 6 were overviewing one or more methods.

The limited scope of the operation and the complexity of domain change was mentioned in the 5 (%) papers.

The limitation of deductive synthesis to operate only on applicative programs, as well as, the issue with program synthesis and the specification complexity, discussed in Section 4.5, was stated in 3 papers.

Noise tolerance and applicable scope for neuro-symbolic program synthesis were reviewed only in one paper. The reason behind this is the lack of literature on the approach as that neuro-symbolic program synthesis was introduced recently and it still it requires comprehensive research.

5.4 Comparison to the State of the Practice

This subsection answers RQ4: How does the level of the state of the practice compare to the state of the art or the subject?

From the approaches reviewed in the selected papers, only a few of them were actually transferred to practice (FlashMeta, FlashExtract and RobustFill), other applications listed in section 4.6 were not explicitly analysed but were mentioned in several papers as a similar method. The reason behind that low transfer rate is that current systems lack usability and are very hard to transfer synthesizer implemented in each approach to a different domain. This limits possible applications to a single fixed domain but in real life, domains change rapidly and the required time for keeping up synthesizer to the domain is not worth the benefit automatic programming could give. But current approaches of meta-synthesizers will most probably promote the use of inductive programming in the industry.

6 Conclusion

This thesis has presented the state of the art of automatic programming by reviewing existing literature on the topic.

From the results of literature review, we can conclude, that automatic programming is generally categorized into three sub-topics: compilers, program synthesis and inductive programming.

Program synthesis was a much more popular research subject in the late 90s and early 2000s. There are many reasons for that, one of them is that at that time available computational power did not allow researchers to consider expensive operations (deep learning, neural networks, etc.), thus the time was spend mostly thinking how to create mathematical models for generation or how to model specifications that can be full and easy to write at the same time. But as the technology developed and with the advance of research in artificial intelligence inductive programming became more popular as its domain of deriving desirable programs from input-output examples or incomplete specifications matched the domain of machine learning and artificial intelligence.

The extensive research on the automatic programming throughout time introduced many interesting and breakthrough approaches. For program synthesis the state of the art approaches are Strategical Approach, Divide and Conquer, Deductive Program Synthesis, Syntax-Guided Synthesis, Oracle-Guided Synthesis, and Synthesis by Sketching. For Inductive programming, the state of the art approaches is IGOR II, ADATE, Object-Oriented Design and Genetic Programming, Data-Driven Domain-Specific Deduction (that inspired FlashMeta) and Neuro-Symbolic Program Synthesis. Additional approaches that cannot be regarded as any of upper mention sub-types include: AI Programmer (utilizing genetic programming), automatic refactoring, web-service composition and generating API adapter, and Automatic Model Generation from Documentation

The current most important limitations and open questions regarding automatic programming as identified in the literature are different for the sub-topics. For program synthesis, writing specification that by definition should be formally complete that is most of the times as complex as writing the actual implementation. For inductive programming, several issues arise, one of them is extending synthesize to different domains when actually a small change in the domain can lead to rewriting the whole synthesizer. The approaches using machine learning and neural networks are early birds in the subject thus their potential is not yet fully

embraced. For example, Neuro-Symbolic Program Synthesis currently utilizes supervised-learning approach currently does not give us insight on reinforced learning that is the essential case in the domain, indicating that further research is required in that direction.

As for the comparison to the state of the practice, the current situation in that direction is marginal that is only a few approaches (FlashMeta, MagicHaskeller, etc.) end up in the industry level software and are accessible for users. Recent efforts by the Microsoft research group provided PROSE framework that will further catalyse this transfer from research to practice, as it is providing tools for easily building inductive synthesizers.

7 References

[1] A. W. Biermann, “Automatic Programming: A Tutorial on Formal Methodologies,” *Journal of Symbolic Computation,* vol. Volume 1, no. Issue 2, pp. 119-142, 1985.

[2] K. Czarnecki and U. W. Eisenecker, Generative and Component-Based Software Engineering, Erfurt: Springer-Verlag, 1999.

[3] K. Becker and J. Gottschlich, “AI Programmer: Autonomously Creating Software Programs Using Genetic Algorithms.,” *CoRR, abs/1709.05703.,* 2017.

[4] N. Pillay and C. K. A. Chalmers, “A hybrid approach to automatic programming for the object-oriented programming paradigm,” in *Proceedings of the 2007 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, Port Elizabeth, 2007.

[5] D. L. Parnas, “Software aspects of strategic defense systems,” *Communications of the ACM,* pp. 1326-1335, December 1985.

[6] K. D. Cooper and L. Torczon, Engineering a Compiler, Burlington, MA: Morgan Kaufmann, 2012.

[7] C. Rich and R. C. Waters, “Approaches to Automatic Programming,” *Advances in Computers,* vol. 37, pp. 1-57, 1992.

[8] S. Gulwani, “Dimensions in Program Synthesis,” in *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming (PPDP '10),* New York, 2010.

[9] A. Gill, “Domain-Specific Languages and Code Synthesis Using Haskell,” *Queue,* vol. 12, no. 4, pp. 30-45, 2014.

[10] N. Wirth, Compiler Construction, Redwood City: Addison Wesley Longman Publishing Co., 1996.

[11] P. Terry, Compilers and Compiler Generators, 1996.

[12] J. Miecznikowski and L. J. Hendren, “Decompiling Java Bytecode: Problems, Traps and Pitfalls,” in *CC '02 Proceedings of the 11th International Conference on Compiler Construction*, London, 2002.

[13] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak and A. Udupa, “Syntax-Guided Synthesis,” in *Formal Methods in Computer-Aided Design*, Portland, 2013.

[14] A. Solar-Lezama, “Program Synthesis by Sketching,” 2003.

[15] Z. Manna and R. Waldinger, “A Deductive Approach to Program Synthesis,” *ACM Transactions on Programming Languages and Systems*, vol. 2, no. 1, pp. 90-121, 1980.

[16] Z. Manna and R. Waldinger, “Fundamentals of Deductive Program Synthesis,” *IEEE Transactions on Software Engineering*, pp. 674-704, 1992.

[17] S. Jha, S. Gulwani, S. A. Seshia and A. Tiwari, “Oracle-Guided Component-Based Program Synthesis,” in *32nd ACM/IEEE International Conference on Software Engineering*, New York, 2010.

[18] J. Voigtländer, “Ideas for connecting inductive program synthesis and bidirectionalization,” in *ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, Philadelphia, 2012.

[19] P. Flener and D. Partridge, “Inductive Programming,” *Automated Software Engineering*, pp. 131-137, April 2001.

[20] S. Gulwani, O. Polozov and R. Singh, Program Synthesis, Hanover: now Publishers Inc, 2011.

[21] S. Gulwani, J. Hernández-Orallo, E. Kitzelmann, S. H. Muggleton, U. Schmid and B. Zorn, “Inductive programming meets the real world, Volume 58 Issue 11,” *Communications of the ACM*, pp. 90-99, 2015.

[22] O. Polozov and S. Gulwani, “FlashMeta: a framework for inductive program synthesis,” in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Pittsburgh, 2015.

[23] M. Hofmann, E. Kitzelmann and U. Schmid, “Analysis and Evaluation of Inductive Programming Systems in a Higher-Order Framework,” in *31st annual German conference on Advances in Artificial Intelligence*, Berlin, 2008.

[24] M. Hofmann and E. Kitzelmann, “I/O guided detection of list catamorphisms: towards problem specific use of program templates in IP,” in *ACM SIGPLAN workshop on Partial evaluation and program manipulation*, Madrid, 2010.

[25] R. Olsson, “Inductive functional programming using incremental program transformation,” *Artificial Intelligence,* pp. 55 - 81, March 1995.

[26] A. Løkketangen and R. Olsson, “Generating meta-heuristic optimization code using ADATE,” *Journal of Heuristics,* vol. 16, no. 6, pp. 911-930, 2010.

[27] J. Olsson and D. M. W. Powers, “Machine Learning of Human Language through Automatic Programming,” *International Conference on Cognitive Science,* pp. 507-512, July 2003.

[28] S. Gulwani, “Automating string processing in spreadsheets using input-output examples,” in *38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, 2011.

[29] V. Le and S. Gulwani, “FlashExtract: A Framework for Data Extraction by Examples,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, New York, 2014.

[30] E. Parisotto, A.-r. Mohamed, R. Singh, L. Li, D. Zhou and P. Kohli, “Neuro-Symbolic Program Synthesis,” in *Proceedings of 5th International Conference on Learning Representations (ICLR 2017)*, Toulon, 2017.

[31] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed and P. Kohli, “RobustFill: Neural Program Learning under Noisy I/O,” in *Proceedings of the 34th International Conference on Machine*, Sydney, 2017.

[32] R. Khatchadourian, “Automated refactoring of legacy Java software to enumerated types,” *Automated Software Engineering,* 2016.

[33] A. M. Omer and A. Schill, “Web Services Composition using Input/Output,” in *Proceedings of the 3rd workshop on Agent-oriented software engineering challenges for ubiquitous and pervasive computing*, London, 2009.

[34] E. Hossny, S. Khattab, F. A. Omara and H. Hassan, “Semantic-based generation of generic-API adapters for portable cloud applications,” in *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*, London, 2016.

[35] J. Zhai, J. Huang, S. Ma, X. Zhang, L. Tan, J. Zhao and F. Qin, “Automatic Model Generation from Documentation for Java,” in *38th IEEE International Conference on Software Engineering*, Austin, 2016.

[36] H. Zhang, A. Jain, G. Khandelwal, C. Kaushik, S. Ge and W. Hu, “Bing developer assistant: improving developer productivity by recommending sample code,” in *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, New York, 2016.

[37] V. Raychev, M. Vechev and E. Yahav, “Code completion with statistical language models,” in *35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, 2014.

[38] M.-A. Storey and A. Zagalsky, “Disrupting Developer Productivity One Bot at a Time,” in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Seattle, 2016.

[39] E. Pantridge, T. Helmuth, N. F. McPhee and L. Spector, “On the difficulty of benchmarking inductive program synthesis methods,” in *Genetic and Evolutionary Computation Conference Companion*, Berlin, 2017.

[40] 51alg, “TerpreT,” [Online]. Available: https://github.com/51alg/TerpreT. [Accessed 20 May 2018].

[41] L. Spector, “Evolutionary Computing with Push,” [Online]. Available: http://faculty.hampshire.edu/lspector/push.html. [Accessed 20 05 2018].

[42] “MagicHaskeller: Automatic inductive functional programmer by systematic search,” [Online]. Available: https://hackage.haskell.org/package/MagicHaskeller. [Accessed 20 05 2018].

[43] “Microsoft Program Synthesis using Examples SDK,” [Online]. Available: https://microsoft.github.io/prose/. [Accessed 20 05 2018].

Appendix

I. Selected Papers

ID	Title	Author	Date	Topic Area
S1	Automatic Programming: A Tutorial on Formal Methodologies [1]	Alan W. Biermann	1985	Inductive Programming, Program Synthesis
S2	Fundamentals of Deductive Program Synthesis [16]s	Zohar Manna, Richard Waldinger	1992	Program Synthesis
S3	Inductive functional programming using incremental program transformation [25]	Roland Olsson	1995	Inductive Programming
S4	A Hybrid Approach to Automatic Programming for the Object-Oriented Programming Paradigm [4]	Nelishia Pillay, Caryl K. A. Chalmers	2007	Inductive Programming
S5	Web Services Composition using Input/output Dependency Matrix	Abrehet Mohammed Omer, Alexander Schill	2009	Inductive Programming
S6	I/O Guided Detection of List Catamorphisms: Towards Problem Specific Use of Program Templates in IP	Martin Hofmann, Emanuel Kitzelmann	2010	Inductive Programming, Generative Programming
S7	Oracle-Guided Component-Based Program Synthesis [17]	Susmit Jha et al.	2010	Inductive programming, program synthesis
S8	Ideas for connecting inductive program synthesis and bidirectionalization	Janis Voigtlander	2012	Inductive Programming

S9	FlashMeta: A Framework for Inductive Program Synthesis [22]	Oleksandr Polozov, Sumit Gulwani	2015	Inductive Programming
S10	Inductive Programming Meets the Real World [21]	Sumit Gulwani et al.	2015	Inductive Programming
S11	Semantic-Based Generation of Generic-API Adapters for Portable Cloud Applications	Eman Hossny et al.	2016	Inductive Programming
S12	Automated refactoring of legacy Java software to enumerated types	Raffi Khatchadourian	2016	Program Synthesis
S13	Disrupting Developer Productivity One Bot at a Time	Margaret-Anne Storey, Alexey Zagalsky	2016	Other
S14	AI Programmer: Autonomously Creating Software Programs Using Genetic Algorithms [3]	Kory Becker, Justin Gottschlich	2017	Inductive Programming, Machine Learning
S15	On the Difficulty of Benchmarking Inductive Program Synthesis Methods [39]	Edward Pantridge et al.	2017	Inductive Programming, a Tool comparison
S16	Program Synthesis [20]	Sumit Gulwani et al.	2017	Inductive Programming, Program synthesis
S17	Generative And Component-based Software Engineering [2]	Krzysztof Czarnecki, Ulrich W. Eisenecker	1999	Generative Programming

S18	Automatic Model Generation from Documentation for Java API Functions	Juan Zhai et al.	2016	Program Synthesis
S19	Inductive Programming [19]	Pierre Flener	2001	Inductive Programming, Program synthesis
S20	Analysis and Evaluation of Inductive Programming Systems in a Higher-Order Framework [23]	Martin Hofmann et al.	2008	Inductive Programming
S21	Compilers and Compiler Generators [11]	P.D. Terry	1996	Compilers
S22	Automating String Processing in Spreadsheets Using Input-Output Examples [28]	Sumit Gulwani	2011	Inductive Programming
S23	Bing Developer Assistant: Improving Developer Productivity by Recommending Sample Code	Hongyu Zhang et al.	2016	Inductive Programming, Code [Data] mining
S24	Compiler Construction [10]	Niklaus Wirth	1996	Compilers
S25	Code Completion with Statistical Language Models	Veselin Raychev et al.	2014	Inductive Programming, NLP
S26	Program Synthesis by Sketching [14]	Armando Solar-Lezama	2003	Program Synthesis
S27	Syntax-Guided Synthesis [13]	Rajeev Alur et al.	2013	Program Synthesis
S28	Engineering a Compiler [6]	Keith D. Cooper, Linda Torczon	2012	Compilers

S29	Approaches to Automatic Programming [7]	Charles Rich, Richard C. Waters	1992	Inductive Programming
S30	Domain-Specific Languages and Code Synthesis Using Haskell [9]	Andy Gill	2014	Program Synthesis
S31	RobustFill: Neural Program Learning under Noisy I/O [31]	Jacob Devlin et al.	2013	Program Synthesis
S32	A Deductive Approach to Program Synthesis [15]	Zohar Manna, Richard Waldinger	1980	Program Synthesis
S33	Neuro-Symbolic Program Synthesis [30]	Emilio Parisotto et al.	2017	Inductive Programming
S34	Machine Learning of Human Language through Automatic Programming [27]	Roland Olsson, David Powers	2003	Inductive Programming
S35	Generating meta-heuristics optimization code using ADATE [26]	Arne Løkketangen, Roland Olsson	2009	Inductive Programming
S36	Dimension in Program Synthesis [8]	Sumit Gulwani	2010	Program synthesis, Inductive Programming
S37	FlashExtract: A Framework for Data Extraction by Examples [29]	Vu Le, Sumit Gulwani	2014	Inductive Programming

II. Abbreviations

Abbreviation	**Description**
TR	Total reflexive theory
car(x)	A function that returns the first element of the list
cdr(x)	A Function that returns the list without the first element
GP	Genetic Programming
SMT	Satisfiability modulo theorem
SAT	Boolean satisfiability problem
OOD	Object-Oriented Design
SyGuS	Syntax Guided Synthesis

III. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Giorgi Gogiashvili**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

   The State of the Art of Automatic Programming,

   supervised by Siim Karus,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **21.05.2018**