Department of
Computer Science

Bachelor Thesis

Learning Efficient Programs

Natalie Höpperger
natalie.hoepperger@student.uibk.ac.at

11 March 2021

Supervisor: Univ.-Prof. Dr. Georg Moser

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Ich erkläre mich mit der Archivierung der vorliegenden Bachelorarbeit einverstanden.

____________________	____________________
Datum	Unterschrift

Abstract

The aim of inductive programming (IP) – also called program synthesis – is to learn programs from incomplete specifications. There exists a number of IP systems that are capable of synthesizing code sequences by using different approaches. However, most of them do not consider distinguishing between the efficiency of the synthesized algorithms. In this thesis *efficiency* stands for the runtime complexity of a program. This thesis investigates the IP tools *FOIL*, *Metaopt*, *MagicHaskeller* and *Hoogle+*, their underlying algorithms as well as their implementations. Furthermore those systems are tested by giving them input-output examples for ten simple programming tasks and analysing the output in terms of correctness and efficiency. It has been shown that three of the four systems are not even able to solve half of the defined tasks without predefining types, functions, relations, meta-rules or predicates. Only *MagicHaskeller* was able to find a satisfactory number of correct code sequences by using library functions as basis for the synthesis procedure.

Contents

1 Introduction **1**
1.1 Aim and Methodological Approach 2
1.2 Contributions 3
1.3 Overview 4

2 Inductive Programming **5**
2.1 Inductive Logic Programming 5
2.2 Inductive Functional Programming 7

3 Background and Related Work **9**
3.1 Foundation of ILP 9
3.2 Pioneer Work in IFP 11
3.3 Surveys of IP Systems 12

4 Applications of Program Synthesis **14**
4.1 Data Wrangling 14
4.2 Automated Program Repair 14
4.3 Code Suggestions 14
4.4 Superoptimization 15
4.5 Teaching Tool for Programming Novices 15

5 Tools, Frameworks and Systems **16**
5.1 MagicHaskeller 16
5.1.1 The Algorithm 17
5.1.2 Implementation – Web Interface 20
5.1.3 Implementation – Haskell Package 21
5.2 FOIL 23
5.2.1 The Algorithm 23
5.2.2 Implementation 26
5.3 Metaopt 28
5.3.1 The Algorithm 29
5.3.2 Implementation 30
5.4 Hoogle+ 31
5.4.1 The Algorithm 32
5.4.2 Implementation 35

6 Benchmark and Comparisons 37
6.1 Programming Tasks . . . 37
6.1.1 Sample Solutions . . . 37
6.2 Results . . . 40
6.2.1 MagicHaskeller . . . 41
6.2.2 FOIL . . . 41
6.2.3 Metaopt . . . 44
6.2.4 Hoogle+ . . . 46

7 Conclusion 48

Bibliography 50

1 Introduction

Automatically finding a code sequence or program just by telling the computer the desired output is surely a fundamental problem in computer science. And indeed, there is an entire research area on this topic. Program synthesis, as the name implies, has the goal to automatically synthesize programs "that satisfy user intent expressed in some form of constraints"[1]. This research area combines a lot of different communities and fields of computer science such as machine learning and artificial intelligence as well as various programming languages and paradigms.

One of the most prominent examples of program synthesis is a feature of Microsoft Excel called *Flash Fill* [6] that was first introduced in 2013's version of Excel. *Flash Fill* allows the user to automatically apply text manipulation on data after simply giving one input-output example. For example, the input might be one column where each row contains the full name of a person (first name and last name). The user now wants to have two additional columns where the name is split into first and last name. Normally the user would have to go through each row and manually split the names. By using the *Flash Fill* technology the user only has to give one example in the first row and then apply the feature to all remaining rows and thus gets the desired and split data. This fully automatic text manipulation is done via program synthesis. For each task millions of small programs – each consisting of 10 to 20 lines of code – are generated and then the synthesizer finds the one which is best-suited to complete the desired task.

Program synthesis is normally divided into two subcategories: *Deductive program synthesis* and *inductive program synthesis*. While deductive approaches are based on a complete formal specification, inductive approaches – also known as *inductive programming* – generate programs from incomplete information (e.g. input-output examples, constraints, traces, etc.). Deductive synthesis – due to the needed complete specification – is extremely complicated and not practicable, which is why the inductive approach seems to be more successful in the long term.

A lot of research has been done in the field of program synthesis over the past years. Nonetheless, the efficiency of the synthesized programs is still a quite rarely covered topic. Besides a few exceptions such as *Metaopt*, an inductive logic programming (ILP) system introduced by Muggleton and Cropper in 2019 [4], none of the existing program synthesis tools is able to distinguish between the efficiency of the generated program, much less synthesize an optimal program. This is why a goal of this bachelor thesis is to examine the efficiency of such synthesized programs.

In this thesis an *efficient* or *optimal* program is a code sequence with a minimal time complexity and therefore a low runtime when executing the program. Thus the *optimal*

[1] Gulwani et al. 2017 [9], 3

program is a code sequence where the complexity is as small as possible. For example, *quick sort* is considered to be one of the best sorting algorithms with a time complexity of $O(n \cdot \log n)$ (average case). If a system is then able to synthesize *quick sort*, this system is able to synthesize an *optimal program* for sorting.

1.1 Aim and Methodological Approach

The goal of this thesis is to give an overview of the existing research in the field of program synthesis, with a focus on comparing different inductive approaches and their corresponding tools. In addition to this, the efficiency of the synthesized programs will be considered. The research question consists of the following:

> *What is the state-of-the-art in the field of inductive programming when it comes to complex, efficient and recursive programs? Which tools for program synthesis based on an inductive approach do exist? How efficient are the emerging programs of these tools?*

For this thesis an overview of selected methods and tools is done through a literature study. Additionally the existing tools will first be theoretically compared through corresponding documentations and papers. In the course of this analysis a comparison between the characteristics of such tools, for example, the underlying programming paradigm or the programming language used by the developers, will be made. If possible, the tools will also be compared in an experimental way to examine the emerging programs. As already mentioned, the efficiency of these programs has priority and the term *efficiency* stands for the time complexity. Also the efficiency of the learning phase is considered, but not discussed in detail.

For this thesis the following four program synthesis tools were selected:

- *MagicHaskeller* (Katayama 2005) [16],
- *FOIL* (Quinlan 1990) [37],
- the ILP system *Metaopt* (Cropper and Muggleton 2019) [4],
- and a recently introduced system called *Hoogle+* (James et al. 2020) [14].

These four tools will be compared by the already mentioned aspects and criteria. The mentioned tools were selected based on different aspects. All systems emerge from open source projects and can be accessed via a web interface (*MagicHaskeller* and *Hoogle+*) or by downloading the source code (*Metaopt* and *FOIL*). Furthermore the systems were selected by their combination of *inductive logic programming* (Section 2.1) and *inductive functional programming* (Section 2.2) approaches. In this thesis older systems such as *FOIL*, *MagicHaskeller* that is somewhere in the middle and systems that were released quite recently such as *Metaopt* and *Hoogle+* are compared.

1.2 Contributions

The results were rather surprising as systems that initially seemed quite promising could not live up to expectations (see Table 1.1, further descriptions can be found in Chapter 6). For example, the recently introduced tool *Hoogle+* was not able to synthesize a single task except for the ones described in the corresponding paper. But on the other hand *MagicHaskeller* – the sole purpose of which is to help programming novices to learn how to program in Haskell – is able to synthesize nearly all kind of short code sequences for the defined tasks. *FOIL* was for sure a powerful tool when it was introduced 30 years ago but is not able to keep up with recent improvements in the field of inductive programming. The usage as well as the necessary preparation of input files (including constants, types and relations) is rather complex and time-consuming compared to the other tools. Further the emerging Horn clauses have to be translated into executable programs (e.g. Prolog code). *Metaopt* is able to synthesize efficient code sequences if the required meta-rules and predicates are predefined. Further the structure of the desired algorithm must be known in advance to use *FOIL* or *Metaopt* for inductive programming.

Problem / System	*MagicHaskeller*	*FOIL*	*Metaopt*	*Hoogle+*
Sort	✓	✓	–	–
Split names	✓	–	–	–
Get initials	✓	–	✓	–
Duplicate elements	✓	↯	–	–
Find duplicate	↯	–	✓	–
Remove duplicate	✓	–	–	✓
Remove element	✓	–	–	–
Reverse	✓	✓	↯	–
Reverse & append	✓	↯	–	–
Count elements	✓	↯	–	–

Table 1.1: Experimental results for all four systems:
✓: A correct output was produced.
↯: An incorrect output was produced.
–: No output was produced.

It must be said that some of the questions and goals described in Section 1.1 can not be answered at this point in time. This is due to the fact that most systems are not even able to synthesize simple and correct programs and much less *complex, efficient and recursive programs.*

As the know information (the *evidence*) is per se unsound and incomplete no inductive programming system is able to prove if a synthesized program is correct. That is why the correctness of a synthesized program must be checked by the user.

1.3 Overview

This thesis is structured as follows:

1. An introduction to *inductive programming* is given by explaining the difference between *inductive logic programming* (ILP) and *inductive functional programming* (IFP) in Chapter 2.
2. Background and related work is discussed in Chapter 3.
3. Common real-life applications of program synthesis are presented in Chapter 4.
4. Chapter 5 gives an overview of the theoretical background and implementation of the selected tools.
5. Experimental results are presented and compared in Chapter 6.

The supplementary materials used for this thesis such as data and code files along with a short README file can be found on GitLab[2].

[2] `https://git.uibk.ac.at/csas8322/learning-efficient-programs-bachelor-thesis`

2 Inductive Programming

In this chapter a short introduction to inductive programming and its subcategories *inductive logic programming* (ILP) and *inductive functional programming* (IFP) is given.

In computer science and especially in programming a *deductive* approach, which means having a general problem as starting point and finding its solution in order to solve a specific problem, is predominant [25]. On the contrary *inductive* methods try to solve a more general problem by first finding the solution to a specific problem, which is also called *inductive reasoning*. The goal is to find general patterns in data and then to produce a generalization of the given incomplete specification (e.g. examples).

In *inductive programming* (IP) the known information – also called the evidence – is per se unsound and incomplete which means that no IP system is able to prove the correctness of a synthesized program or algorithm [5, 25]. That being the case the obtained specifications are only *hypotheses*.

Inductive programming is a type of machine learning, but also goes beyond its focus on classification and regularities through restricted models such as decision trees and neural networks which are typical for machine learning approaches. Instead inductive programming covers the learning of general programs that also include more powerful programming concepts such as recursion and loops [25]. Furthermore IP can be seen as part of program synthesis. Other approaches to program synthesis are deductive or transformational approaches [42].

2.1 Inductive Logic Programming

Inductive logic programming (ILP) is based on (counter)-examples, more precisely positive and negative examples (the *evidence*) and uses first-order logic (e.g. Horn clauses) to represent data, examples and hypotheses [25]. It is an intersection of logic programming and inductive learning and uses machine learning as well as logic programming techniques [31].

ILP relies on inductive inference which means that logical conclusions are derived from known premises (*logical consequence* or *entailment*). For ILP systems only a few examples are needed to generalise whereas other techniques such as machine learning are data hungry and need large amounts of examples. Furthermore data is represented as logic programs which makes it possible to learn with complex relational information and to further integrate expert knowledge [2]. ILP systems are able to synthesize and generalise programs and algorithms that go beyond the initial examples. For instance, ILP applications for sorting and transforming strings can handle different input sizes and types [2].

To sum up, inductive logic programming is about selecting a hypothesis H which is an element of a set of possible hypotheses (definite programs) such that $H \cup B$ (where B is the consistent background knowledge) with respect to positive and negative evidence (E^+ and E^-) [25]. Out of the systems considered in this thesis *FOIL* [37] and *Metaopt* [4] use such an approach.

FOIL, an inductive logic programming system developed by John Ross Quinlan in 1990, can be used to first find an explicit representation of the target relation in the form of Horn clauses and then a more generalized functional representation by applying a information-based heuristic search [37]. As already mentioned the system needs positive and negative examples to do so. *FOIL's* approach is further described in Section 5.2.

Metaopt developed by Stephen Muggleton and Andrew Cropper in 2019 is another ILP system that extends their previous meta-interpretive learning (MIL) system $Metagol_O$. MIL is a form of inductive logic programming that also supports the learning of recursive algorithms and programs as well as the use of predicate invention for problem decomposition [3]. For MIL systems an adapted Prolog meta-interpreter is used to prove a set of goals which is done by “repeatedly fetching higher-order meta-rules whose heads unify with a given goal”[1]. As a result meta-substitutions can be obtained and applied onto their corresponding meta-rules whereby a hypothesis is formed. Further details on the implementation of *Metaopt's* approach can be found in Section 5.3.

FOIL for example, is able to learn an algorithm for *quick sort* by defining the following relations with a sufficient number of examples:

- *part(N,Xs,Ls,Rs)*: Takes an element N (the *pivot*), a list Xs and returns two lists (Ls, Rs), where all elements of Xs that are less than or equal to N are put into Ls and the rest is put into Rs. E.g. *part(1,[1,2]),Ls,Rs)* returns $Ls = [1]$, $Rs = [2]$.

- *append(Xs, Ys, Zs)*: Takes two lists Xs and Ys and returns a list Zs, that appends Xs to Ys. E.g. *append([1],[2],Zs)* returns $Zs = [1, 2]$.

- *components(Xs,N,Ys)*: Takes a list Xs and returns the head N and the tail of the list Ys. E.g. *components([1,2],N,Ys)* returns $N = 1, Ys = [2]$.

The sequence synthesized by *FOIL* can be found in Listing 2.1 and the corresponding Prolog code in Listing 2.2. First head and tail of the given list A are saved in C and D by calling *components*. Then D is divided into E and F around the pivot C by calling *part* and sorted by recursively calling *qsort* for the divided parts. The sorted lists are stored in G and H and eventually linked with the pivot to the resulting list B.

```
1 qsort(A,A) :- null(A)
2 qsort(A,B) :- components(A,C,D), part(C,D,E,F), qsort(E,G), qsort(F,H),
3         append(G,I,B), components(I,C,H)
```

Listing 2.1: An algorithm for *quick sort* found by *FOIL*.

[1] Cropper & Muggleton 2015 [3], p. 3425

```
1  qsort([],[]).
2  qsort(A,B) :- components(A,C,D), part(C,D,E,F), qsort(E,G), qsort(F,H),
3          append(G,I,B), components(I,C,H).
4
5  components([X|Xs],X,Xs).
6
7  part(Y,[],[],[]).
8  part(Y,[X|Xs],[X|Ls],Rs) :- X =< Y, part(Y,Xs,Ls,Rs).
9  part(Y,[X|Xs],Ls,[X|Rs]) :- X > Y, part(Y,Xs,Ls,Rs).
10
11 append([],Ys,Ys).
12 append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

Listing 2.2: Corresponding Prolog code to the algorithm found by *FOIL*.

2.2 Inductive Functional Programming

In inductive functional programming (IFP) input-output examples are first transformed into constructive expressions such as predicates and traces, where each predicate registers the structure of an input example and each trace calculates the associated output for a given input example [41]. Then these pairs of predicates and traces are searched for regularities which then are used for finding generalizations.

There are different approaches to IFP: an *analytical* and a *generate-and-test* approach. The analytical approach uses pattern matching while the generate-and-test approach generates as a first step programs and algorithms which then are reduced to those who satisfy the given condition [21]. Analytical approaches tend to be more efficient and therefore faster, while generate-and-test approaches tend to be slower but the given specification is not limited by the number and complexity of the I/O-examples.

MagicHaskeller [16] is an IFP system that uses a generate-and-test approach that is based on systematic search. First the systems generates all possible and type-correct Haskell expressions which then are tested by a given test function. Due to the fact that *MagicHaskeller's* program generator starts with the smallest expression possible and then tests if that expression suits the given specification, a program of minimal size is obtained first [21].

Another prominent example for inductive functional programming is the recently introduced system *Hoogle+* [14], which is able to find Haskell expressions when given a few input-output examples. Further the tool is able to synthesize a suitable type from these examples. The system first infers a type, which is needed to synthesize candidate programs by using a type-directed component-based synthesis approach. In a next step the emerging programs are sorted out and eventually test cases are generated to show the functionality of the synthesized code sequence.

MagicHaskeller, for example, is able to find a Haskell library function for sorting (see Listing 2.4) by giving one simple input-output example as shown in Listing 2.3. The function `sort` is implemented as *merge sort* and is part of the `Data.List` module[2].

[2] `https://hackage.haskell.org/package/base-4.14.1.0/docs/src/Data.OldList.html#sort`

```
1 f [5,2,1,3,4] == [1,2,3,4,5]
```

Listing 2.3: An input-output example for sorting given to *MagicHaskeller*.

```
1 f = sort
```

Listing 2.4: A function for sorting found by *MagicHaskeller*.

3 Background and Related Work

In this chapter an overview of existing research literature and the current state of research is given. First the origins of *inductive logic programming* and *inductive functional programming* are discussed. As Muggleton is considered to have established this field of research, ILP – or rather inductive inference – is explained as proposed by Muggleton et al. [31] by giving the process of synthesizing the sorting algorithm *quick sort* as an example. IFP is explained by presenting the algorithm of one of the first inductive functional programming systems *THESYS* [45], which was introduced by Summers in 1977.

The notations for clauses, expressions etc. used in the following sections are oriented towards those used by the authors in the corresponding papers.

3.1 Foundation of ILP

The term *inductive logic programming* was first used by Stephen Muggleton in 1991 to define a combination of *inductive learning* (or machine learning in general) and *logic programming* [28]. In the same paper he further describes theoretical foundations of ILP such as the general concept that *hypotheses* are formed by searching a consistent set of *examples* (Muggleton also called them *observations*) and consistent *background knowledge* [28]. Hypotheses (H) and background knowledge (B) must further entail the examples or observations (O) as shown in (3.1.1).

$$H \wedge B \vdash O \tag{3.1.1}$$

Another paper about theories and methods of inductive logic programming was published by Stephen Muggleton and Luc de Raedt in 1993, where three different examples about inductive inference (family relationships, Tweety and sorting) were used to describe how ILP works [31].

The following example[1] shows how *quick sort* is learned by ILP systems. The notation used to describe the algorithm is the notation used in logic programming which is based on formal logic. The definitions in (3.1.2) are used as background knowledge [31] and were already described in Section 2.1 for *FOIL*, which uses the same relations as background knowledge.

[1] cf. Muggleton & De Raedt 1994 [31], p. 633

$$B = \begin{cases} part(X, [], [], []) \leftarrow \\ part(X, [Y|T], [Y|S1], S2) \leftarrow Y =< X,\ partition(X, T, S1, S2) \\ part(X, [Y|T], S1, [Y|S2]) \leftarrow Y > X,\ part(X, T, S1, S2) \\ app([], L, L) \leftarrow \\ app([X|T], L, [X|R]) \leftarrow app(T, L, R) \end{cases} \tag{3.1.2}$$

The program also receives a set of positive (E^+) and negative (E^-) examples (the *evidence*) as shown in (3.1.3) and (3.1.4).

$$E^+ = \begin{cases} qsort([], []) \leftarrow \\ qsort([0], [0]) \leftarrow \\ qsort([1, 0], [0, 1]) \leftarrow \\ \ldots \end{cases} \tag{3.1.3}$$

$$E^- = \begin{cases} \leftarrow qsort([1, 0], [1, 0]) \\ \leftarrow qsort([0], []) \\ \ldots \end{cases} \tag{3.1.4}$$

The hypothesis shown in (3.1.5) should be generated by the algorithm if a sufficient number of positive and negative examples is given. For an empty list *qsort* returns the empty list. If the list is not empty, a *divide-and-conquer* approach is used, where first the tail of the list T is partitioned around the pivot element X (the head of the list) into two lists $L1$ and $L2$. After that *qsort* is recursively called for $L1$ and $L2$ and the resulting lists are stored into $S1$ and $S2$. As a last step $S1$ is appended to the pivot element X and $S2$ ($[X|S2]$) and the resulting list S is the sorted list.

$$H = \begin{cases} qsort([], []) \leftarrow \\ qsort([X|T], S) \leftarrow part(X, T, L1, L2), \\ \qquad\qquad\qquad\qquad qsort(L1, S1), \\ \qquad\qquad\qquad\qquad qsort(L2, S2), \\ \qquad\qquad\qquad\qquad app(S1, [X|S2], S) \end{cases} \tag{3.1.5}$$

The ILP systems *GOLEM* [30] as well as *FOIL* [37] are able to synthesize the described algorithm for *quick sort* from only a few examples.

Furthermore Muggleton has implemented a number of various ILP systems such as *GOLEM* (1990) [30], *PROGOL* (1995) [29] and *Metagol* [32] which is under active development since 2014 and recently he has introduced *Metaopt* (2019) [4], a system that is capable of synthesizing efficient programs. *Metaopt* and its implementation will be discussed in Section 5.3.

3.2 Pioneer Work in IFP

In 1977 Phillip D. Summers introduced a program synthesis system called *THESYS* [45], which is capable of deriving recursive LISP programs from examples without the need of performing a search for candidate programs [24]. The basic idea of the system's algorithm is to infer traces from examples and then fold these traces into a recursive program by using a trace-based programming method [5].

Definition 3.1 (McCarthy Conditional Expression [27]). A McCarthy-Conditional has the form $(p_1 \rightarrow e_1, ..., p_n \rightarrow e_n)$, where p are propositional expressions and e are arbitrary expressions. It can be read as "If p then e".

The system is based on two steps. First a trace as shown in (3.2.1) is formed. A trace consists of McCarthy conditional expressions (see Definition 3.1) where $p_i(x)$ is a predicate and $f_i(x)$ a program fragment.

$$F(x) = (p_1(x) \rightarrow f_1(x), ..., p_{k-1}(x) \rightarrow f_{k-1}(x), T \rightarrow f_k(x)) \tag{3.2.1}$$

In the second step recurrences between those predicates and program fragments are found and a recursive program is formed that generalizes those recurrent relations [25].

To construct a trace the following primitive LISP functions are used:

- `atom`: Returns *False* if an object is a list.
- `car`: Returns the first element of a list.
- `cdr`: Returns the list without the first element.
- `cons`: Combines two expressions to a list.
- `nil`: Represents the empty list.

The following example is used by Kitzelmann[2] to demonstrate the algorithm described above. The synthesized program should be able to compute the the initial sequence of a list. Input-output examples shown in (3.2.2) are used as input.

$$\langle (A), () \rangle, \langle (A,B), (A) \rangle, \langle (A,B,C), (A,B) \rangle, \langle (A,B,C,D), (A,B,C) \rangle \tag{3.2.2}$$

First the trace (see (3.2.3)) consisting of predicates (atoms) and program fragments can be derived from these examples. *cd...dr* stands for multiple calls of the function `cdr`.

$$\begin{aligned} F(x) = (&atom(cdr(x)) \rightarrow nil \\ &atom(cddr(x)) \rightarrow cons(car(x), nil) \\ &atom(cdddr(x)) \rightarrow cons(car(x), cons(cadr(x), nil)) \\ &T \rightarrow cons(car(x), cons(cadr(x), cons(caddr(x), nil)))) \end{aligned} \tag{3.2.3}$$

Then the recurrences shown in (3.2.4) and (3.2.5) can be identified.

$$p_{i+1}(x) = p_i(cdr(x))\ for\ i = 1, 2 \tag{3.2.4}$$

[2] cf. Kitzelmann 2008 [24], p. 89f.

$$f_{i+1}(x) = cons(car(x),\ f_i(cdr(x)))\ for\ i\ =\ 1,2,3 \tag{3.2.5}$$

As a last step the (recursive) LISP program as shown in (3.2.6) is synthesized by inductively generalise the recurrences in (3.2.4) and (3.2.5).

$$\begin{aligned} F(x) &= (atom(cdr(x)) \rightarrow nil \\ &\qquad\qquad\qquad T \rightarrow F'(x)) \\ F'(x) &= (atom(cddr(x)) \rightarrow cons(car(x), nil) \\ &\qquad\qquad\qquad T \rightarrow cons(car(x), F'(cdr(x))) \end{aligned} \tag{3.2.6}$$

3.3 Surveys of IP Systems

Although there exists a lot of research in the field of program synthesis, not much effort has been put into comparing existing tools.

In 2009 different inductive programming tools were compared by Hofmann et al. [13]. In this survey seven systems were analyzed: *ADATE* [34], *FFOIL* [38], *GOLEM* [30], *MagicHaskeller* [16], *FLIP* [12], *IGOR I* [26] and *IGOR II* [23]. Further a CCRS framework was designed to describe those IP systems. A CCRS is a *(conditional) combinatory rewrite system* that is based on a combination of term rewriting rules, conditional rules and meta-variables that allow for generalisation over functions with a given arity. All systems were first characterized, then classified into this framework and compared by testing their efficiency in form of runtimes on different problems (classifying numbers in even/odd, sorting, reversing a list, etc.). While, for example, FFOIL failed on nearly all problems by providing wrong results, *IGOR II* – a system designed by one of the authors Emanuel Kitzelmann – or *MagicHaskeller* proved to be rather successful in synthesizing programs [13].

A year later – in 2010 – a survey on different techniques and systems for inductive programming was done by Emanuel Kitzelmann [25] but due to the recent developments in program synthesis some of the more up to date tools such as *Flash Fill* [6] or *Metaopt* [4] were not covered in his paper. The compared tools were divided into the two already mentioned subcategories of inductive programming, ILP (see Section 2.1) and IFP (see Section 2.2) and further described, but no benchmarking of any kind was done. Furthermore the efficiency and performance of the tools was not evaluated and the capabilities as well as the limitations of the systems were not discussed.

Comparing such systems seems and proved to be rather difficult because mostly it is not possible to apply them to the same tasks because they were designed for different kind of problems and input, e.g., *Metaopt* can handle strings well while *FOIL* performs better on numbers. Nonetheless Pantridge et al. [35] performed a survey on a number of IP systems such as *Flash Fill* [6] , *MagicHaskeller* [16], *TerpreT* [7] and two forms of genetic programming methods, *PushGP* and *Grammar Guided Genetic Programming*. They used a benchmark suite for program synthesis with basic programming problems – taken from an introductory computer science textbook – to make a comparison possible.

This benchmark suite was designed by Thomas Helmuth and Lee Spector in 2015 [11] and contains 29 basic problems such as computing the sum of an integer and a float number, comparing string lengths or printing the smallest of four given integers. Pantridge et al. concluded that although the computation costs of genetic programming systems are rather high, they are more capable of successfully synthesizing programs [35].

Recently a comprehensive survey that focuses on describing inductive logic programming in detail as well as comparing inductive logic programming systems was presented in a preprint by Andrew Cropper and Sebastijan Dumančić [2]. The survey covers prominent examples of inductive synthesis tools such as *FOIL* [37], *PROGOL* [29] and *Metagol* [32]. Altogether 14 ILP systems were analyzed and compared on five different dimensions and four systems were described more precisely and in detail. The method that was used to compare all 14 systems differs from the usual methods described above as not only different programming problems but rather dimensions such as noise, optimality, infinite domains, recursion and predicate invention were used to compare the systems [2]. It is worth mentioning that Cropper and Dumančić define *optimality* not only as shortest runtime possible but rather include aspects such as a minimal hypothesis (minimal number of clauses, literals, etc.) in their analysis.

4 Applications of Program Synthesis

4.1 Data Wrangling

One of the most common use cases for program synthesis is data wrangling, which includes all transformations based on data. For example "cleaning, transforming, and preparing data"[1] in different kind of ways is *data wrangling*.

According to Gulwani et al. [9] data engineers and/or data scientists spend about 80% of their time on manipulating data in different kind of ways, which makes this application of program synthesis even more interesting. Especially when economical factors are considered.

A state of the art tool in this section is Microsoft's *Flash Fill* technology that was first released in in Excel 2013 [8] and which was already mentioned in the introduction of this thesis.

4.2 Automated Program Repair

Another use case for program synthesis is code repair, where bugs in code written by human programmers should automatically be fixed [9, 35]. Under the assumption that a program P with a specific number of bugs and a specification ϕ exists, the code repair tool computes a new program P' that satisfies ϕ [9]. This is mostly done by finding alternatives for the used expressions in the original program and then testing if the program satisfies the given specification after applying the generated expressions.

This could be used in many fields of computer science. One interesting application is using automated program repair for automatically correcting programming assignments from students. For example, the system *AutoProf* [43] is designed for a usage like this.

4.3 Code Suggestions

In the dimension of Software Engineering program synthesis is used for autocompletion or finding code suggestions [9]. Although such tools are currently only able to complete expressions after typing a few letters, they could be able to complete whole code blocks and not only a couple of tokens in the future [9].

There exist different techniques for designing such tools. While *statistical* approaches use probabilistic models and a web search for code snippets, *type-directed completion*

[1] Gulwani et al. 2017 [9], p. 15

approaches use primarily typing information but also a ranking function and space abstractions to synthesize code sequences [9].

More advanced autocompletion tools would for sure be an extension for every programmer especially in respect of programming languages where library functions are used to a large extend.

As nearly every IDE has its own tool for code completion there are many examples for the use of program synthesis in this area.

4.4 Superoptimization

In order to make machine code as optimal as possible, program synthesis is used [9]. The goal is to synthesize a *better* – or more efficient – program from a given piece of code. Therefor program synthesis is used for finding the shortest (machine) code sequence.

Although partial improvements are done by compilers, they do not produce optimal code. Most compilers just improve the code instead of really provide the most efficient code sequence for a given program. The goal of superoptimization is precisely to generate optimized machine code and therefore optimize the efficiency of code as much as possible.

Superoptimization is mostly used to optimize linear code fragments, but also for optimizing loops in the context of automatic vectorization as well as for bitvector programs [9].

4.5 Teaching Tool for Programming Novices

Learning how to program or even learning a new programming language can be really hard. The user hast to memorize the syntax as well as the semantic aspect and if available also some of the standard library functions. That is why program synthesis is also used in the context of education.

As an example the web-based automatic programming tool *MagicHaskeller* [16] supports users with learning and understanding functional programming. The tool is able to synthesize Haskell expressions by giving just one input-output example which makes it easy to use. Furthermore it is available to use online.

5 Tools, Frameworks and Systems

5.1 MagicHaskeller

MagicHaskeller[1] is an inductive functional programming tool developed by Susumu Katayama in 2005 and its algorithm as well as its implementation was described in multiple papers (e.g. [15, 16, 17, 20, 22]). Parts of the system were modified and improved several times [18, 19, 21], which makes it quite difficult to understand what kind of algorithm is used in the current and available version.

The goal of *MagicHaskeller* is to help users to learn the syntax and semantics as well as the standard library functions of Haskell. This is done by offering a web interface to quickly synthesize programs. Furthermore *MagicHaskeller* can be included and used as library within GHCi (*Glasgow Haskell Compiler's interactive environment*), as API for inductive program synthesis and it also comes with an executable for a standalone program synthesizer [22].

Only one input-output example is needed to use *MagicHaskeller* which makes it intuitive to use. Examples can either be an incomplete specification or a condition that must be satisfied by the emerging program. If an incomplete specification is used, it must be a Boolean-valued expression such as shown in Listing 5.1, where `f` is a free variable.

```
1 f ["ABCDE", "DF","1234", ""] == ["", "DF","1234","ABCDE"]
```

Listing 5.1: An input-output example for *MagicHaskeller*.

MagicHaskeller synthesizes a program through applying an exhaustive and systematic breadth-first search. It uses a *generate-and-test approach* as briefly described in Section 2.2, which means that first all type-correct expressions for a type given by the user or inferred by the system are generated by the main part of the system – *the program generator*. The emerging expressions include function composition, λ-abstractions or functions from the Haskell standard library.

After generating all possible expressions bottom-up (smallest expressions with minimal program length first and then increasing the size) the emerging expressions are compiled and tested until termination or a timeout is reached. As a consequence the most generalized expression can be obtained and overfitting is prevented.

Due to the exhaustive search the problem size is limited (i.e. the system is not able to find solutions for more complex problems), which means that synthesizing big programs is not possible. However, also other systems that do not use such a search technique

[1] `http://nautilus.cs.miyazaki-u.ac.jp/cgi-bin/MagicHaskeller.cgi`

have problems to deal with big problem sizes. By applying an exhaustive and therefore complete search it is furthermore guaranteed that the solution is complete [13].

The way *MagicHaskeller* generates expressions for a given type conforms the way propositions are proved under Curry-Howard isomorphism [21]. The Curry-Howard isomorphism (also known as Curry-Howard correspondence) describes the relationship between type values and proofs of propositions. Similar to generating a proof tree for validating a proposition, *MagicHaskeller* generates programs that satisfy a given type by also generating a proof tree [19]. This makes the system basically to an extension of an automatic prover algorithm.

5.1.1 The Algorithm

To synthesize Haskell programs the system requires a so called component library, which consists of a Haskell source file that describes the available function set. The algorithm consists of the following four steps [15]:

1. First the component library (as Haskell source file) is read to the interpreter when the system gets invoked.
2. Then the specification given by the user – consisting of a type and an expression (the *property* or *property function*) – is read. If no type is defined by the user, it is inferred by using a conventional Hindley-Milner style type inference algorithm on the given input-output examples.
3. Then expressions that match the requested or inferred type are generated.
4. Finally, the synthesized expressions are tested. To test if a generated expression satisfies the specification the property function is executed with the synthesized program as the argument. If so, the interpreter returns *True* and the system terminates. If the return value is *False*, the described process is repeated until *True* can be obtained as return value [15].

A graphical representation of the structure of the system can be found in Figure 5.1.1.

Generation of Expressions

In the following the generation of expression is described by assuming the type given to the program generator is $\forall a\, b.[a] \rightarrow b \rightarrow Int$[2]. The algorithm then proceeds as follows:

1. First the type variables that are assumed to be universal quantifiers such as a and b are replaced by new, non-existent type constructors (free variables), for example $G0$ and $G1$. Thus we obtain $[G0] \rightarrow G1 \rightarrow Int$.
2. Next a function called `unifyingExprs`, whose type definition is shown in Listing 5.2, is invoked.

[2]cf. Katayama 2007 [17], p. 116f.

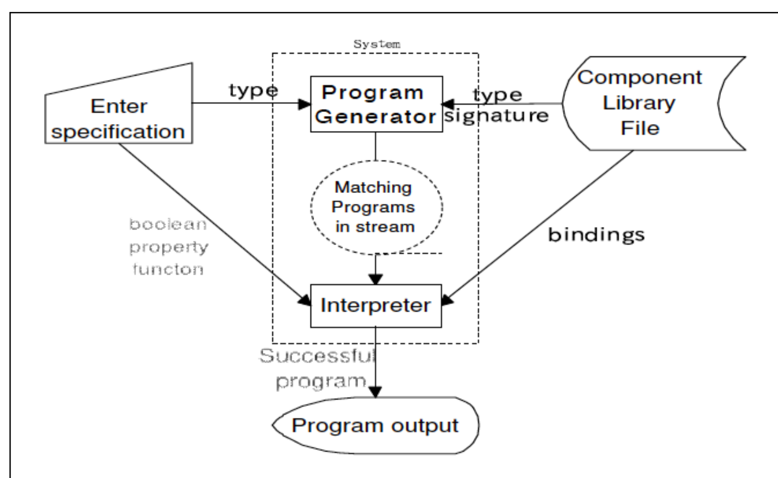

Figure 5.1.1: System structure of *MagicHaskeller* ([17], p. 126).

```
1 unifyingExprs :: [(Expression,Type)] -> Type -> TI Recomp Expression
```

Listing 5.2: The type of the function *unifyingExprs*.

The first argument `[(Expression,Type)]` represents a list of variables and their types from the component library, the second argument `Type` stands for the requested type. Note that `Expression`, `Type`, `TI` and `Recomp` are defined datatypes and monads. `TI` (see Listing 5.3) was defined as a monad for type inference with `Subst` as the current substitution and a parameter `Int` that represents the ID of the next fresh variable. If possible, it returns the inferred type `a`. `Recomp` is a more efficient implementation of Spivey's monad for breadth-first search [44]. To limit the consumption of heap space Katayama defined the monad `Recomp` as shown in Listing 5.4. In the function `unifyingExprs` both monads are used in the form of `TI Recomp Expression`.

```
1 newtype TI a = TI (Subst -> Int -> Maybe (a, Subst, Int))
2 newtype (Monad m) => TI m a = TI (Subst -> Int -> m (a, Subst, Int))
```

Listing 5.3: The type inference monad [16, 17].

```
1  newtype Recomp a = Rc {unRc :: Int -> Bag a}
2  instance Monad Recomp where
3      return x = Rc f where f 0 = [x]
4                            f _ = []
5      Rc f >>= g = Rc (\n -> [ y | i <- [0..n]
6                                , x <- f i
7                                , y <- unRc (g x) (n-i) ])
8
9  instance MonadPlus Recomp where
10     mzero = Rc (const [])
```

```
11      Rc f 'mplus' Rc g = Rc (\i -> f i ++ g i)
```

Listing 5.4: The Recomp monad that recomputes everything to prevent enormous heap consumption [16, 17].

```
1  module Library where
2
3  zero :: Int
4  zero = 0
5
6  inc :: Int -> Int
7  inc = \x -> x+1
8
9  nat_para :: Int -> a -> (Int -> a -> a) -> a
10 nat_para = \i x f -> if i == 0 then x
11   else f (i-1) (nat_para (i-1) x f)
12
13 nil :: [a]
14 nil =[]
15
16 cons :: a -> [a] -> [a]
17 cons = (:)
18
19 list_para :: [b] -> a -> (b -> [b] -> a -> a) -> a
20 list_para= \l x f -> case l of
21    [] -> x
22   a:m -> f a m (list_para m x f)
```

Listing 5.5: A simplified component library [16, 17].

To simplify the following explanations it is assumed that only expressions of type `Int` are required. The function `unifyingExprs` first generates a list of functions taken from the component library where the return type is also `Int`. When using a library consisting of the functions shown in Listing 5.5, we obtain as so-called "head candidates"[3] the functions `zero`, `inc`, `nat_para` and `list_para`. Now the most general unifier is applied as substitution to each function, e.g., for `list_para` we obtain the type signature shown in Listing 5.6 after applying $[a \mapsto Int]$.

```
1  list_para :: [b] -> Int -> (b -> [b] -> Int -> Int) -> Int
```

Listing 5.6: The type signature of *list_para* after applying the substitution $[a \mapsto Int]$.

3. Then a prioritized *bag* (multiset) of expressions is obtained by calling the function `unifyingExprs` recursively on the argument types of the head candidates. For `list_para` the argument types would be `[b]`, `Int` and `(b -> [b] -> Int -> Int)`. The obtained expressions are called "spine candidates"[4].

4. As a last step the spine candidates are applied to the corresponding head candidates

[3] Katayama 2007 [17], p. 116
[4] ib., p. 117

to obtain the final prioritized bag, a multiset of generated expressions and possible candidates for the final output program.

To reduce the number of generated expressions *MagicHaskeller* does not generate any expressions that are theoretically known to be equivalent to already generated expressions, which is done via pattern matching. For example, it is known that `foldr op x [] = x` and therefore the system avoids to construct expressions of the form `foldr _ _ []` [16]. Redundancies are then found by executing the expressions with random arguments (*random testing*) [18, 19]. To do so the Haskell testing library *QuickCheck* [1] is used.

5.1.2 Implementation – Web Interface

I first tested *MagicHaskeller* through the web interface. By giving one input-output example I tried to synthesize a sort function which is a rather simple problem. The tool was able to synthesize two expressions for the first example, which was an unsorted list with numbers from 1 to 10. The web interfaced as well as the synthesized expressions are shown in Figure 5.1.2. The tool was able to synthesize the standard library sort function which is exactly the expectable result. Due to the fact that all numbers from 1 to 10 occur in the given input-output example, *MagicHaskeller* synthesized another expression that returns a list with numbers from 1 to the length of the input list in ascending order.

Furthermore the tool offers the possibility to directly access the documentation of each part of each expression. By clicking on, e.g., the synthesized function `length` the entry on the predefined Haskell Prelude function `length` is opened and explained.

As it is shown in Figure 5.1.3 the web interface of *MagicHaskeller* comes with random examples to test the synthesized expression. Often these examples contain edge cases like empty sets or negative numbers. This feature is invoked by clicking on the *Exemplify* button after specifying a function and synthesizing it as it was done in Figure 5.1.2.

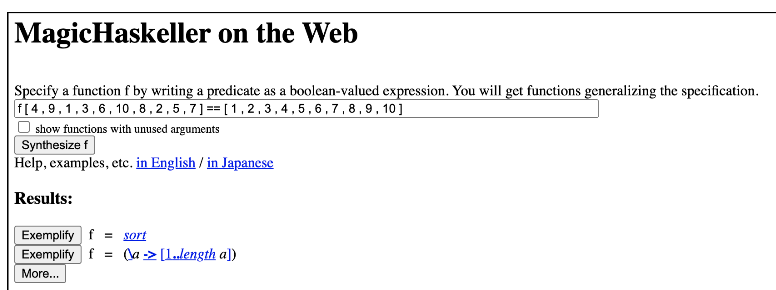

Figure 5.1.2: *MagicHaskeller* is able to quickly synthesize the sort function.

As a second input-output example for sorting numbers I passed the sequence shown in Listing 5.7, which led to an even better result because now the tool was able to only synthesize the sort function.

```
1 f [ 55, 23, 11, 75, 66, 17] == [11, 17, 23, 55, 66, 75]
```

MagicHaskeller on the Web --- input-output examples

The candidate expression
f = sort [Exemplify]
satisfies the following input-output relation:

f [4 , 9 , 1 , 3 , 6 , 10 , 8 , 2 , 5 , 7] == [1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10] [Synthesize f]
&& f [0,1,0] ~= [0,0,1] [Narrow search]
&& f [] ~= [] [Narrow search]
&& f [1] ~= [1] [Narrow search]
&& f [(-1),(-3),4,(-1)] ~= [-3,-1,-1,4] [Narrow search]
&& f [0,(-1),3] ~= [-1,0,3] [Narrow search]

Figure 5.1.3: The synthesized function can automatically be applied to random arguments.

Listing 5.7: An input-output example for sorting numbers with *MagicHaskeller*.

Since sorting numbers worked fine I tried to synthesize expressions that apply some sort of string manipulation to given strings. First I used some really simple examples like removing the first name from a full name, e.g., “Jonathan Moore” becomes “Moore”, which worked really well. After that I tried to synthesize a string manipulation that for example can be done really fast and effective by the already mentioned Microsoft Excel feature *Flash Fill*: splitting full names into first and last names. To synthesize this kind of program I used the input-output example shown in Listing 5.8, which lead to the synthesis of the expression shown in Listing 5.9, which indeed is a correct function to split full names into first and last names.

```
1 f ["Jonathan Moore", "Olivia Dunlop", "Gabrielle Mull"] == [["Jonathan", "Olivia",
      "Gabrielle"], ["Moore", "Dunlop", "Mull"]]
```

Listing 5.8: An input-output example for splitting strings with *MagicHaskeller*.

```
1 f = (\a -> transpose (map words a))
```

Listing 5.9: An expression found by *MagicHaskeller* for splitting strings into first and lastnames.

5.1.3 Implementation – Haskell Package

As a next step I tried to install and use the *MagicHaskeller* package. After some troubles the execution of the package worked and I tested the system by using the same input-output examples as described in the previous chapter.
There are three aspects I want to talk about in this section:

1. The documentation of the Haskell package.

2. Non-repeatable results.

3. Different results.

First of all, the documentation of the package leaves much to be desired. In fact, three sources are available: A short paper written by Katayama himself [20] called *MagicHaskeller: System demonstration*, the corresponding web page where the web application can be found and the package and module documentation on *hackage.haskell.org*. The paper contains an installation guide (that did not work for me) and some examples. On the website another installation guide and the web interface for MagicHaskeller can be found. The module documentation contains a list of implemented Haskell functions, but no description of how to use the system.

The examples described in the paper [20] show how to use the command line version of *MagicHaskeller*, but on the first attempt I was not able to execute those examples, which leads me to the next point: non-repeatable results. By simply following the steps for synthesizing a function that doubles characters in a given string, which is described in the system demonstration[5], only an error message was produced. Executing the input shown in line 1 of Listing 5.10 did not work. After changing the input to the expression shown in line 2 of Listing 5.10 I was able to obtain the results described in the paper [20].

```
1 \f -> f "abc" == "aabbcc"
2 printAll True $ \f -> f "abc" == "aabbcc"
```

Listing 5.10: Input statement for the web interface compared to the one for the usage in GHCi.

It was further not possible to reproduce the results I got for the expression shown in Listing 5.7, because the system did not terminate on this example. Splitting names into first and last names did not work too. The example in Listing 5.11 shows that the web interface produces a different, less complex output (first line) compared to using the *MagicHaskeller* as module in GHCi (second line).

```
1 f = length
2 \a -> list_para a 0 (\_ _ d -> (1 GHC.Num.+) d)
```

Listing 5.11: Output of the web interface compared to the one of the module in GHCi.

It is noticeable that the execution of examples in GHCi needs more time than executing them in the web interface. This is interesting considering that the computation is done on a backend server, thus bandwidth and delay could be an issue too.

The efficiency of the learning phase depends on the size of the search space. For example synthesizing a sort function for numbers or a split function for full names was rather fast (seconds) which shows that *MaigcHaskeller* is efficient in synthesizing such (small) programs. The tool performs well on list manipulations (strings or integer lists). However, due to the exhaustive search, synthesizing more powerful programs leads to a huge search space and therefore also the execution time increases steadily [13].

[5] cf. Katayama 2011 [20], p. 3

5.2 FOIL

FOIL (*First-Order Inductive Learner*) is an inductive logic programming system developed by John Ross Quinlan in 1990 that is able to synthesize function-free Horn clause definitions from examples [37]. It is based on an algorithm that uses a top-down approach, meaning that first a general hypothesis is formed which is then formalized. The system first looks for a literal that describes some of the positive examples (⊕ tuples, that are known to be in the target relation) and can be added to the clause, removes those examples and updates the set of tuples (the training set). Then the search is continued until there are no ⊕ tuples left in the training set.

A Horn clause is a formula of the form $P_1 \wedge P_2 \wedge ... \wedge P_n \rightarrow Q$ where $P_1, ..., P_n$, and Q are atoms, ⊤ or ⊥. The system was designed to learn definitions of a relation in terms of itself and other relations [39]. To show the concept of learning relations, *FOIL* is often described by using family relations: E.g., the system is able to learn the concept *grandfather(X, Y)* when given the relations *father(X, Y)* and *parent(X, Y)*. In the following section this will be further explained by using another simple example.

To use *FOIL*, a file where types are specified and relations are defined is needed. Additionally also test cases can be defined in this file. To obtain resulting Horn clauses the system must be invoked with the mentioned file as input parameter.

Although the system was developed thirty years ago and therefore one of the first program synthesis tools, it still seems to be quite powerful compared to other existing and much younger systems. Of course the obtained Horn clauses first have to be translated into executable code by using programming languages such as Prolog, which makes the systems kind of impracticable.

5.2.1 The Algorithm

As done by Quinlan[6] the algorithm of the system will be described by using an example that deals with the reachability of nodes in a network such as illustrated in Figure 5.2.1. The target relation of *FOIL* is based on a predicate $P(X_1, X_2, ..., X_k)$ such as *can-reach*(X_1, X_2) where X_1 is the starting node and X_2 is the node that can be reached through one or more paths when starting at node X_1.

The training set consists of tuples $(X_1, X_2, ..., X_k)$ where values are assigned to the variables. The tuples are labelled with either ⊕ (for positive examples; tuples that are known to be in the target relation) or ⊖ (for negative examples; tuples that are not part of the target relation) [37, 39]. For the relation *can-reach*(X_1, X_2) and the network illustrated in Figure 5.2.1 the 81 tuples shown in (5.2.2) and (5.2.3) are part of the training set T.

$$\begin{aligned} \oplus : &\langle 0,1\rangle, \langle 0,2\rangle, \langle 0,3\rangle, \langle 0,4\rangle, \langle 0,5\rangle, \langle 0,6\rangle, \langle 0,8\rangle, \langle 1,2\rangle, \langle 3,2\rangle, \langle 3,4\rangle, \\ &\langle 3,5\rangle, \langle 3,6\rangle, \langle 3,8\rangle \langle 4,5\rangle, \langle 4,6\rangle, \langle 4,8\rangle, \langle 6,8\rangle, \langle 7,6\rangle, \langle 7,8\rangle \end{aligned} \tag{5.2.2}$$

[6] Quinlan 1990 [37], p. 241ff.

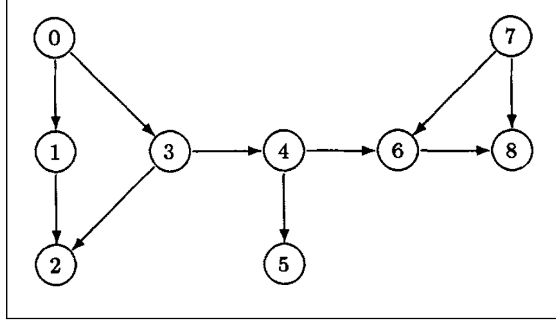

Figure 5.2.1: A simple network ([37], p. 241).

$$\begin{aligned}\ominus : &\langle 0,0\rangle, \langle 0,7\rangle, \langle 1,0\rangle, \langle 1,1\rangle, \langle 1,3\rangle, \langle 1,4\rangle, \langle 1,5\rangle, \langle 1,6\rangle, \langle 1,7\rangle, \langle 1,8\rangle,\\ &\langle 2,0\rangle, \langle 2,1\rangle, \langle 2,2\rangle, \langle 2,3\rangle, \langle 2,4\rangle, \langle 2,5\rangle, \langle 2,6\rangle, \langle 2,7\rangle, \langle 2,8\rangle, \langle 3,0\rangle,\\ &\langle 3,1\rangle, \langle 3,3\rangle, \langle 3,7\rangle, \langle 4,0\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle, \langle 4,4\rangle, \langle 4,7\rangle, \langle 5,0\rangle,\\ &\langle 5,1\rangle, \langle 5,2\rangle, \langle 5,3\rangle, \langle 5,4\rangle, \langle 5,5\rangle, \langle 5,6\rangle, \langle 5,7\rangle, \langle 5,8\rangle, \langle 6,0\rangle, \langle 6,1\rangle,\\ &\langle 6,2\rangle, \langle 6,3\rangle, \langle 6,4\rangle, \langle 6,5\rangle, \langle 6,6\rangle, \langle 6,7\rangle, \langle 7,0\rangle, \langle 7,1\rangle, \langle 7,2\rangle, \langle 7,3\rangle,\\ &\langle 7,4\rangle, \langle 7,5\rangle, \langle 7,7\rangle, \langle 8,0\rangle, \langle 8,1\rangle, \langle 8,2\rangle, \langle 8,3\rangle, \langle 8,4\rangle, \langle 8,5\rangle, \langle 8,6\rangle,\\ &\langle 8,7\rangle, \langle 8,8\rangle\end{aligned} \tag{5.2.3}$$

First the training set is established as shown above. Then the algorithm iterates through all $\oplus$ tuples and seeks for a function-free Horn clause of the form $P(X_1, X_2, ..., X_k) \leftarrow L_1, L_2, ...L_n$ that describes some of the $\oplus$ tuples. If such a clause is found, all tuples that satisfy the right-hand side of the detected clause are removed from the training set. This procedure is continued until there are no more $\oplus$ tuples left in the training set [37]. After that, the clauses are checked for any redundancies and reordered: Non-Recursive clauses (or *base cases*) must come before recursive clauses [39].

Clauses for our *can-reach*(X_1, X_2) problem are found by a greedy cover algorithm that can be described by the following steps:

1. The clause is initialized as shown in (5.2.4).

$$\textit{can-reach}(X_1, X_2) \leftarrow ... \tag{5.2.4}$$

2. A local training set T_1 (more general T_i where $i = 1$) is initialized (in the first iteration) to the training set T described above. This local training set contains all $\oplus$ tuples.

3. While T_i contains $\ominus$ tuples:
   - First the algorithm searches for an appropriate literal L_i that can be added to the right-hand side of the clause. FOIL searches for *gainful* literals, which means that a literal must help to remove $\ominus$ tuples from the training set. The gain of this literals must be close to the maximum gain that is possible (close

means ≥ 80% of it). The gain rate is determined by the formula in (5.2.5) with T_+ as the number of ⊕ tuples and $I(T)$ as the information provided by the training set.

$$T_+ * I(T) \tag{5.2.5}$$

The first literal L_1 that can be found for *can-reach*(X_1, X_2) is shown in (5.2.6).

$$\textit{can-reach}(X_1, X_2) \leftarrow \textit{linked-to}(X_1, X_2) \tag{5.2.6}$$

For the target-relation *can-reach* this leads to updated ⊕ tuples (see (5.2.7)) while the ⊖ tuples remain the same.

$$\oplus : \langle 0,2\rangle, \langle 0,4\rangle, \langle 0,5\rangle, \langle 0,6\rangle, \langle 0,8\rangle, \langle 3,5\rangle, \langle 3,6\rangle, \langle 3,8\rangle, \langle 4,8\rangle \tag{5.2.7}$$

This is why a new L_2 literal *linked-to*(X_1, X_3) to replace *linked-to*(X_1, X_2) (this literal causes no ⊖ tuples to be removed) as shown in (5.2.8) is selected.

$$\textit{can-reach}(X_1, X_2) \leftarrow \textit{linked-to}(X_1, X_3) \tag{5.2.8}$$

Now ⊖ tuples starting with $\langle 2, ...\rangle, \langle 5, ...\rangle$ and $\langle 8, ...\rangle$ can be removed.

- Then a new training set $(T_i + 1)$ that only contains the tuples of T_i that satisfy the found literal L_i from the previous step is produced. Now we obtain triples (due to the introduced variable X_3) and a new training set T_2 of triples can be generated as shown in (5.2.9) and (5.2.10).

$$\begin{aligned}\oplus :& \langle 0,2,1\rangle, \langle 0,2,3\rangle, \langle 0,4,1\rangle, \langle 0,4,3\rangle, \langle 0,5,1\rangle, \langle 0,5,3\rangle, \langle 0,6,1\rangle,\\ & \langle 0,6,3\rangle, \langle 0,8,1\rangle, \langle 0,8,3\rangle, \langle 3,5,2\rangle, \langle 3,5,4\rangle, \langle 3,6,2\rangle, \langle 3,6,4\rangle,\\ & \langle 3,8,2\rangle, \langle 3,8,4\rangle, \langle 4,8,5\rangle, \langle 4,8,6\rangle\end{aligned} \tag{5.2.9}$$

$$\begin{aligned}\ominus :& \langle 0,0,1\rangle, \langle 0,0,3\rangle, \langle 0,7,1\rangle, \langle 0,7,3\rangle, \langle 1,0,2\rangle, \langle 1,1,2\rangle, \langle 1,3,2\rangle,\\ & \langle 1,4,2\rangle, \langle 1,5,2\rangle, \langle 1,6,2\rangle, \langle 1,7,2\rangle, \langle 1,8,2\rangle, \langle 3,0,2\rangle, \langle 3,0,4\rangle,\\ & \langle 3,1,2\rangle, \langle 3,1,4\rangle, \langle 3,3,2\rangle, \langle 3,3,4\rangle, \langle 3,7,2\rangle, \langle 3,7,4\rangle, \langle 4,0,5\rangle,\\ & \langle 4,0,6\rangle, \langle 4,1,5\rangle, \langle 4,1,6\rangle, \langle 4,2,5\rangle, \langle 4,2,6\rangle, \langle 4,3,5\rangle, \langle 4,3,6\rangle,\\ & \langle 4,4,6\rangle, \langle 4,4,6\rangle, \langle 4,7,5\rangle, \langle 4,7,6\rangle, \langle 6,0,8\rangle, \langle 6,1,8\rangle, \langle 6,2,8\rangle,\\ & \langle 6,3,8\rangle, \langle 6,4,8\rangle, \langle 6,5,8\rangle, \langle 6,6,8\rangle, \langle 6,7,8\rangle, \langle 7,0,6\rangle, \langle 7,0,8\rangle,\\ & \langle 7,1,6\rangle, \langle 7,1,8\rangle, \langle 7,2,6\rangle, \langle 7,2,8\rangle, \langle 7,3,6\rangle, \langle 7,3,8\rangle, \langle 7,4,6\rangle,\\ & \langle 7,4,8\rangle, \langle 7,5,6\rangle, \langle 7,5,8\rangle, \langle 7,7,6\rangle, \langle 7,7,8\rangle\end{aligned} \tag{5.2.10}$$

- As a last step i is incremented.

There are still ⊖ tuples left which is why the inner loop is entered again and another literal *can-reach(X_3, X_2)* is selected. This leads to T_3, a training set that only contains ⊕ tuples, which is shown in (5.2.11).

$$\oplus : \langle 0,2,1\rangle, \langle 0,2,3\rangle, \langle 0,4,3\rangle, \langle 0,5,3\rangle, \langle 0,6,3\rangle, \langle 0,8,3\rangle, \langle 3,5,4\rangle, \langle 3,6,4\rangle, \langle 3,8,4\rangle, \langle 4,8,6\rangle \tag{5.2.11}$$

All ⊖ tuples have been removed and all ⊕ tuples are covered by one of those clauses. This leads to the final definition of *can-reach* (see (5.2.12)).

$$\textit{can-reach}(X_1, X_2) \leftarrow \textit{linked-to}(X_1, X_3),\ \textit{can-reach}(X_3, X_2) \tag{5.2.12}$$

5.2.2 Implementation

I had access to *FOIL* 6.0, implemented by Q. R. Quinlan in association with Mike Cameron-Jones in October 1993. The implementation is written in `ANSI C` (also known as `C89`).

The input to this command line implementation of FOIL can be divided into three parts – each followed by a blank line – whereby the last part and the corresponding as well as the previous blank line are optional. The following example comes with the implementation of FOIL and the goal is to find a clause that is able to tell if a number between 1 and 3 is a part (a *member*) of a given list that contains a maximum of three numbers.

First types and constants are declared as shown in Listing 5.12.

```
1 X: 1, 2, 3.
2 L: [111], [112], [113], [11], [121], [122], [123], [12], [131], [132],
3    [133], [13], [1], [211], [212], [213], [21], [221], [222], [223],
4    [22], [231], [232], [233], [23], [2], [311], [312], [313], [31],
5    [321], [322], [323], [32], [331], [332], [333], [33], [3], *[].
```

Listing 5.12: Declaration of types and constants.

Here *L* represents the type *list*, while *X* represents an element of such a list. The second part of the input defines the relations by specifying the ⊕ tuples. ⊖ tuples can be specified too, but their specification is not necessary as FOIL is able to determine them by using the *closed-world-assumption* (all not specified tuples must therefore be ⊖ tuples). To follow the previous example, we have a relation *member(X, L)* with the two arguments *X* (a list element) and *L* (the list itself). The input defining the relations would look like the one shown in Listing 5.13 (positive examples) and Listing 5.14 (negative examples).

```
1 member(X,L)
2 1,[1]
3 3,[3]
4 1,[11]
5 1,[13]
6 ;
```

Listing 5.13: Positive examples for *member(X, L)*.

```
1 member(X,L)
2 1,[]
3 1,[3]
4 1,[33]
5 1,[333]
6 .
```

Listing 5.14: Negative examples for *member(X, L)*.

The semicolon is necessary to distinguish positive (⊕) from negative (⊖) tuples. The tuples defined in Listing 5.13 and 5.14 are only a small part of the complete input given for this example and can be looked up in the file *member.d* (see supplementary material[7]). Then also other relations can be defined as input such as shown in Listing 5.15

```
1 *components(L,X,L) #--/-##
2 [1],1,[]
3 [2],2,[]
4 [3],3,[]
5 [11],1,[1]
6 [12],1,[2]
7 [13],1,[3]
8 [21],2,[1]
9 [22],2,[2]
10 .
```

Listing 5.15: The *components* relation.

Unlike the first definition here the asterisk in front of the relation's name tells FOIL that no definition should be constructed for this relation with the three arguments L (a list), X (the head element of this list) and L (the tail of this list). The characters $\#--/-\#\#$ are a key like used in databases. Again the example above is not complete and can be looked up on GitLab.

The third (optional) part of the input are test cases where the first line indicates which relation should be tested and the following lines are the test arguments X and L separated by a comma and followed by a colon, then a space and a sign (+ or −) that indicates whether the tuples should satisfy the definition (+) or not (−). For example, test cases for the *member* relation could look like the one shown in Listing 5.16.

```
1 member
2 2,[]: -
3 3,[121]: -
4 3,[23]: +
5 3,[232]: +
6 .
```

[7] https://git.uibk.ac.at/csas8322/learning-efficient-programs-bachelor-thesis

Listing 5.16: Test cases for the member problem.

After executing FOIL with the input given in the file *member.d* and the command `foil6 < member.d` the clauses shown in Listing 5.17 can be derived within seconds.

```
1 member(A,B) :- components(B,A,C)
2 member(A,B) :- components(B,C,D), member(A,D)
```

Listing 5.17: Output clauses.

Those clauses can now be translated into Prolog code, which is shown in Listing 5.18

```
1 member(A,B) :- components(B,A,C).
2 member(A,B) :- components(B,C,D), member(A,D).
3
4 components([X|Xs],X,Xs).
```

Listing 5.18: Output of FOIL translated into Prolog code.

5.3 Metaopt

Metaopt is an inductive logic programming system implemented in Prolog that is capable of synthesizing *efficient* programs, meaning that it is able to learn lower cost Prolog programs through iteration [4]. In this case *lower cost programs* means programs that have a minimal time complexity, thus a low runtime. According to the authors Andrew Cropper and Stephen H. Muggleton *Metaopt* [4] is the first system that is able to learn such minimal cost algorithms.

The system induces efficient Prolog programs by finding meta-substitutions for positive examples – which is done by a meta-interpretive learner – that are applied to meta-rules. Before that the meta-rules are recursively proven by using meta-interpretation. In the end the resulting program is checked by applying negative examples [4].

Metaopt is the result of years of research in the field of program synthesis. In the last years Cropper and Muggleton focused on synthesizing efficient code sequences. *Metagol*$_O$ [3] – another system developed by Cropper and Muggleton in 2015 – is for example able to learn efficient robot strategies, which are dyadic logic programs consisting of predicates with two arguments for in- and output [4]. But the system is not able to induce programs of any kind and is further restricted in its usage, e.g, the user must specify the costs of predicates.

Metaopt on the other hand is able to learn minimal cost programs and also only a small number of examples (< 20) is necessary to do so. The system synthesizes efficient code sequences by further restricting the hypothesis space in each iteration which is a search procedure called “iterative descent”[8]. *Metaopt* seems to be particularly successful

[8]Cropper & Muggleton 2019 [4], p. 1065

when it comes to string transformations [4].

5.3.1 The Algorithm

Metaopt is an extension of Cropper and Muggleton's previous system $Metagol_O$ [3], that is based on the algorithm used for another system of Muggleton et al., $Metagol_D$ [33], which is why I will shortly describe the algorithm of this system.
$Metagol_D$ is based around a meta-interpreter that is able to invent predicates (*predicate invention*). To do so the user has to provide meta-rules (see Table 5.1), which are higher-order meta-rules that describe which clauses are permitted in the hypothesised programs.

Name	**Meta-Rule**	**Order**
Instance	$P(X,Y) \leftarrow$	*True*
Base	$P(x,y) \leftarrow Q(x,y)$	$P \succ Q$
Chain	$P(x,y) \leftarrow Q(x,z), R(z,y)$	$P \succ Q, P \succ R$
TailRec	$P(x,y) \leftarrow Q(x,z), P(z,y)$	$P \succ Q$, $x \succ z \succ y$

Table 5.1: Meta-rules as used for the meta-interpreter [33].

The meta-interpreter then tries to prove the given examples and the corresponding order constraints to each existing meta-rule to ensure that the proof terminates. If a proof is successful, the associated substitution is saved and later applied to the meta-rules which leads to an inductive generalisation of the given examples in form of a first-order definite program [33].
Metaopt uses meta-interpretive learning (MIL) to minimise the cost of a synthesized program [4]. The input for the MIL learner is a triple of the form (B, E, Φ) where:

- $B = B_C \cup M$, with B_C as a set of definite clauses and M as a set of meta-rules.
- $E = (E^+, E^-)$, a set of positive and negative examples.
- Φ is a program cost function.

The cost minimal MIL learner returns a program $H \in V_{B,E}$ such that $H \preceq_\Phi H'$ for all $H' \in V_{B,E}$ [4]. The size is defined by the number of clauses in H and programs are ordered by their efficiency ($\preceq_\Phi$) which is defined by the maximum cost of a program. The system measures the cost of a program by measuring the runtime or time complexity as a function of the SLD-tree that is being searched. SLD-resolution (= Selective Linear Definite clause resolution) is used in logic programming to formalise computation. For a definite program H the tree cost can be determined by the function shown in (5.3.1), where G is an initial goal, T a SLD-tree, the $branch_size(H,G)$ the number of resolutions prior to and including L (the leftmost successful branch of T) and the $tree_size(H,G)$ the number of resolutions of T, a finitely failed tree.

Input	Output
My name is John.	John
My name is Bill.	Bill
My name is Josh.	Josh
My name is Albert.	Albert
My name is Richard.	Richard

Table 5.2: Input-output examples for the learning phase of *Metaopt*.

$$tree_cost(H,G) = \begin{cases} branch_size(H,G) & \text{if } T \text{ has a successful branch} \\ tree_size(H,G) & \text{if } T \text{ is finitely failed} \end{cases} \quad (5.3.1)$$

Metaopt takes positive and negative examples (atoms) as input and tries to prove the positive ones. The cost of proving one atom is added to the overall proof cost and when a certain bound is exceeded, the proof is terminated because obviously the program is to inefficient. This bound is determined by a so called “iterative descend procedure”[9], an algorithm that searches for a minimal cost program H_i in an iterative depth-first way on the number of clauses.

As already described for *Metagol*$_D$ also *Metaopt* forms a logic program after proving all positive examples by applying the meta-substitutions to the associated meta-rules. As a final step the program is tested by checking it with the negative examples and if inconsistency is detected the system backtracks and searches in different branches of the SLD-tree [4].

5.3.2 Implementation

As already mentioned, *Metaopt* is implemented in Prolog and can be found on GitHub[10]. Cropper and Muggleton provide four different categories of examples: *converge*, *duplicate*, *strings* and *postman*. Each example folder comes with learning data, already synthesized programs and results. Further each example is automatically learned by all three systems (*Metagol*, *Metagol*$_O$ and *Metaopt*) while executing the learning phase.

I downloaded the whole project folder and tested the system by first rerunning the learning step for the string manipulation examples. Even after an execution time of more than 12 hours only a small part of the examples had been learned. To test the system I tried one of the string manipulation examples that the system was able to learn in a short period of time. The algorithm is able to detect names in the sentence “My name is...”. To do so, the system was provided with five (positive) examples as shown in Table 5.2.

Metaopt then was able to find the code sequence shown in Listing 5.19.

```
1 f(A,B):-tail(A,C),f_1(C,B).
```

[9] Cropper & Muggleton 2019 [4], p. 1071
[10] `https://github.com/andrewcropper/mlj18-metaopt`

```
2 f_1(A,B):-f_2(A,C),dropLast(C,B).
3 f_2(A,B):-f_3(A,C),f_3(C,B).
4 f_3(A,B):-tail(A,C),f_4(C,B).
5 f_4(A,B):-f_5(A,C),f_5(C,B).
6 f_5(A,B):-tail(A,C),tail(C,B).
```

Listing 5.19: The code sequence found by *Metaopt*

The synthesized algorithm can be tested by executing the code in SWI-Prolog in the command line as shown in Listing 5.20.

```
$ swipl -q programs/extractName.pl
?- ['e-metaopt'].
   true.
?- f([M, y,' ', n, a, m, e,' ', i, s,' ', N, a, t, a, l, i, e, .], X).
    X = [N, a, t, a, l, i, e].
?- f([M, y,' ', n, a, m, e,' ', i, s,' ', N, a, t, a, l, i, e, .], [J, o, h, n]).
   false.
?- f([M, y,' ', n, a, m, e,' ', i, s,' ', N, a, t, a, l, i, e, .], [N, a, t, h, a,
     l, i, e]).
   false.
?- halt.
```

Listing 5.20: Testing the code in SWI-Prolog.

5.4 Hoogle+

Hoogle+[11] is a web-based inductive functional programming (IFP) system that was recently introduced by James et al. [14]. The tool can be used by either specifying a type, giving input-output examples (called *tests*) or a combination of both and returns a list of synthesized Haskell programs that fulfill the initial goal defined by the input. *Hoogle+* uses Haskell library functions (the *component library*) to compose such programs. For example, when giving the input list `[1, 1, 1, 2, 2, 3]` and its corresponding output `[1, 2, 3]` to the web application, the Haskell program shown in Listing 5.21 that removes duplicates from a given list is generated.

```
1 \xs -> map head (group xs)
```

Listing 5.21: A function synthesized by *Hoogle+*, that removes duplicates.

Especially in functional programming it can sometimes be hard to find even small pieces of code. That is why the goal of this tool is to help programmers to discover algorithms for a given problem in the form of a type, one or more tests or both.

As the system synthesizes functions by combining already existing Haskell library functions an efficient search strategy is necessary. *Hoogle+* uses a type-directed search based on an already existing approach called *type-guided abstraction refinement* (TyGAR)

[11] `https://hoogleplus.goto.ucsd.edu/`

[10] which is why a type definition is needed. As programming novices might not know anything about types in Haskell, an algorithm that is able to infer types from the given tests (input-output examples) was developed. Further an heuristic that uses *property-based testing* to eliminate uninteresting programs was added to the system. To help the programmer decide which synthesized code snippet really solves the initial problem, the tool is able to automatically generate examples for each synthesized expression [14].

Furthermore the tool was tested on 30 people with different background knowledge in Haskell programming. The programmers had to solve different programming tasks in Haskell and were allowed to use *Hoogle+*. This study has shown that the users were able to solve more tasks and faster by using the tool [14].

5.4.1 The Algorithm

The system synthesizes functions by passing four different steps:

1. Type inference,
2. synthesis of candidate programs,
3. elimination,
4. and comprehension of the synthesized programs.

In the following section I will describe each of the above.

First a type is inferred from the given tests (input-output examples). This could be problematic as on the one hand tests like `"aabbab"` → `"abab"` and `[1, 1, 1, 2, 2]` → `[1, 2]` have obviously different types (`[Char]` → `[Char]` and `[Int]` → `[Int]`), therefore a polymorphic type is needed. On the other hand, if only one test with a concrete type is given, then the question is, if a more general type specification would be more useful or not. For example, for synthesizing a function that removes duplicates (deduplication) from a list, the input-output examples (tests) as shown in table Table 5.3 would be appropriate. The type inference is done by an algorithm called `TestToType` ([14], 205:9).

xs	output
“aaaabbbab”	“abab”
[1,1,1,2,2,3]	[1, 2, 3]

Table 5.3: Input for the dedup function in *Hoogle+*.

The input consists of the component library (or environment Γ) and a test suite $\bar{t}$. The output is a sequence of type specifications $\overline{T}$. After the input is received the algorithm proceeds like this:

- A *type inference oracle* $\Gamma \vdash e \Longrightarrow T$ computes the most general and concrete types $\overline{T_i}$ for each test.

- Then the function `AntiUnifyAll` is invoked and the least common generalization $T_\sqcup$ of the types $\overline{T_i}$ is computed. This is done by using anti-unification as described by Plotkin in 1970 [36].
- The function `Inhabited` then tries to find a set G which contains all generalizations of $T_\sqcup$ that are also likely to be suitable for the synthesis goal.
- Eventually the obtained types are ranked by a heuristic `TopK`.

For example, if no type is entered for the `dedup` function, but tests as defined in table Table 5.3 are given, the `TestToType` function is invoked which produces the proposals for the type specification of the function as shown in Listing 5.22.

```
1 xs: [a] -> [a]
2 (Eq a) => xs: [a] -> [a]
3 (Ord a) => xs: [a] -> [a]
```

Listing 5.22: Types inferred by *Hoogle+*.

In this example a polymorphic type is used as the input-output examples have different types and therefore in the end the synthesized function must be able to handle both or more input types.

After that programs are synthesized by using a component-based synthesis algorithm called `TyGAR` that generates Haskell programs by taking the type and a set of library functions as input. Component-based synthesis needs a library of components (such as Haskell functions) to generate programs. *Type-guided abstraction refinement* [10] (TYGAR) is an iterative process for “scalable type-directed synthesis over polymorphic datatypes and components”[12]. To find a matching code sequence an *abstract transition net* (ATN) [10] – a petri net of abstract types – is constructed. This ATN represents all possible types and hence all possible solutions which means that one of the paths in this petri net matches the solution to the initial synthesis problem. To find this path the algorithm uses already existing SMT-based (satisfiability modulo theories) techniques. If a path (i.e. a candidate term) is found, it is validated by a type checker. If the term is well-typed, the solution is found. If not, the ATN is refined and the search is repeated until a well-typed term is found [10]. This process is illustrated in Figure 5.4.1.

The algorithm is therefore based on a graph search, but due to the polymorphic components of Haskell such a graph could be endless, which is why the search is only done on abstract types that represent a set of concrete ones [10].

The last two steps elimination and comprehension use property-based testing as implemented in the Haskell library *SmallCheck* [40]. The synthesized programs are first reduced (elimination process) so that only *meaningful* and *unique* ones remain. *Meaningfulness* is defined as the existence of an input value for a given program that leads to termination and to an output value within a given time. On the other hand a *meaningless* program is defined as $\forall i. [\![p]\!](i) = \bot$, where p is the program and i the input

[12] Guo et al. 2020 [10], p. 12:2

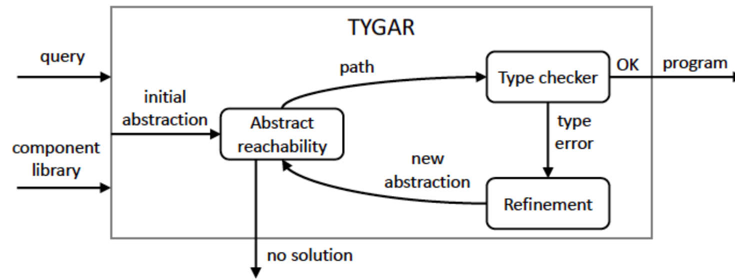

Figure 5.4.1: The synthesis process of TYGAR ([10], 12:3).

value. Uniqueness is defined as the absence of *observable equivalence*, therefore $p \not\equiv p'$ holds for each $p' \in P'$, where p and p' are programs of a set of synthesized programs P'. A program p is equivalent to another program p' if $\forall i. [\![p]\!](i) = [\![p]\!](i)$ (they produce the same output for all possible inputs) where i is again an input value [14].

As a last step the system generates three examples (test cases) that show how the synthesized piece of code works – or which output it produces – on a given input that is also generated by the system itself. These examples match the three fundamentals *meaningfulness*, *uniqueness* and *functionality*. *Meaningfulness* is shown by generating at least one example that is successful and sometimes also one that is not (hence fails). *Uniqueness* is guaranteed by only showing unique examples. To show the user the *functionality* of the synthesized program one ore more unique examples illustrate how the program works [14]. The synthesized program for the **dedup** example as well as the generated test cases are shown in Figure 5.4.2.

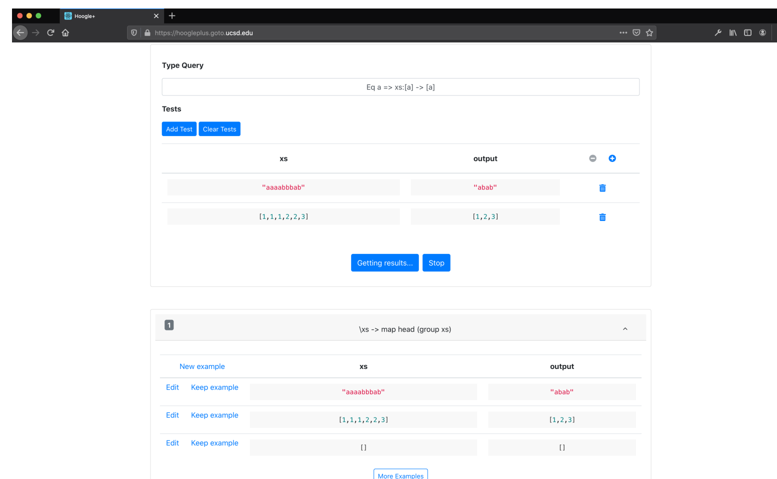

Figure 5.4.2: An expression for *dedup* found by *Hoogle+* with generated test cases.

5.4.2 Implementation

Hoogle+ is implemented as a web-based API system, but as already mentioned it does not terminate even on simple examples.

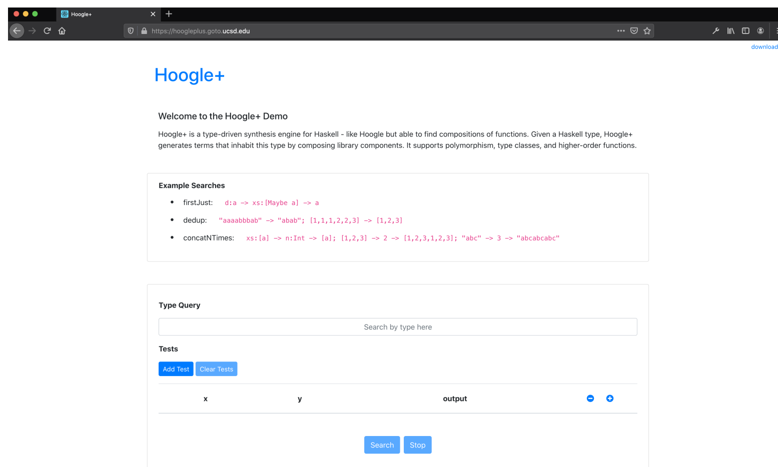

Figure 5.4.3: The interface of the web-based API *Hoogle+* with predefined examples.

In the previous section I described a function called `dedup` that removes duplicates from a given list or string. This examples is proposed by the developers in the corresponding paper and even on the web application of *Hoogle+*[13]. However, when running the example by clicking on the predefined test case the type inference indeed does work, but not the synthesizing itself.

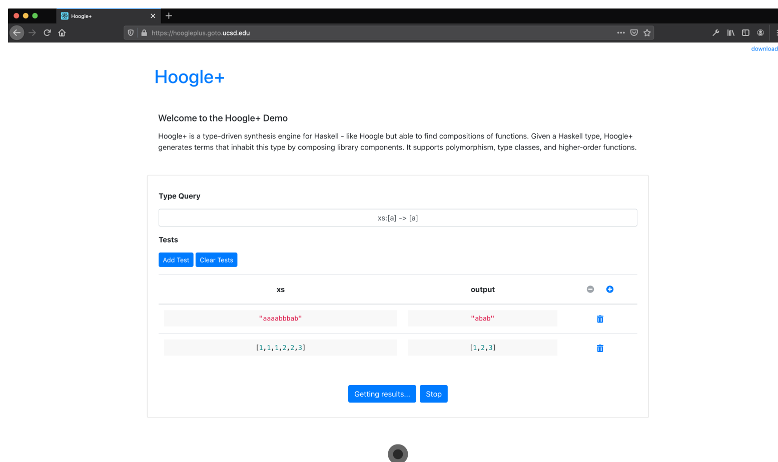

Figure 5.4.4: *Hoogle+* when trying to synthesize the *dedup* example.

[13] `https://hoogleplus.goto.ucsd.edu/`

Also other examples, such as sorting numbers and splitting strings do not work. The tool automatically cancels the computation after a specified timeout of 360 seconds. Nonetheless there is a way to synthesize the proposed "dedup" example which can be found in the corresponding paper. The system does find the code sequence `\x -> map head (group x)`, when given the type `Eq a => x:[a] -> [a]` and the input-output example `"abaa"` → `"aba"` (see Figure 5.4.2).

6 Benchmark and Comparisons

In this section the systems described in the previous chapter will be compared by the efficiency of the programs that they are able to synthesize. A synthesized program is efficient, if its runtime is low. The efficiency of the learning phase of each system was described in the previous chapter. A system is efficient, if its runtime during the synthesizing process is low.

To compare the tools, a fixed number of simple programming tasks was defined which will be described later in this chapter. First an attempt to synthesize those tasks is started for all four systems and in the first place it is checked, if the systems can produce an output. Then the obtained programs are compared by their correctness and their efficiency.

6.1 Programming Tasks

The tasks shown in Table 6.1 were selected for comparison. All examples are simple programming tasks that can be solved by programming novices within one to a few lines of code. Further these tasks are textbook examples that are used for educational purposes, for instance, as programming assignments for students. They also combine manipulation on lists and string manipulations. Some of the tasks were mentioned in the surveys described in Section 3.3, but it seems like the tasks typically given to ILP systems are different from those given to IFP systems. That is why I tried to combine tasks of those different fields and communities.

6.1.1 Sample Solutions

In the following subsection some of the task and their corresponding solutions as Haskell as well as Prolog code are discussed.

Sort

When looking at the first task *sort* there are obviously different solutions as sorting can be done by using different algorithms such as *merge sort*, *quick sort* or *bubble sort*. To find the most efficient solution, the first step is to compare the complexity of those sorting algorithms. *Quick sort* for example is considered to be one of the best sorting algorithms and has an average time complexity of $O(n \cdot \log n)$, the same applies to *merge sort*. Implementations of *quick sort* in Haskell and Prolog can be found in Listing 6.1 and Listing 6.2.

Task	Description	Example
Sort	The goal of this task is to sort a given list of numbers in ascending order.	[5, 2, 3, 1, 4] → [1, 2, 3, 4, 5]
Split names	The goal of this task is to split full names into first and last names.	[“Jonathan Moore”] → [[“Jonathan”], [“Moore”]]
Get initials	The goal of this task is to extract the initials of a single name or a list of names.	[“Jonathan Moore”] → [“JM”]
Duplicate elements	The goal of this task is to duplicate all elements given in a list or a string.	“abc” → “aabbcc” [1, 2, 3] → [1, 1, 2, 2, 3, 3]
Find duplicate	The goal of this task is to find the element that appears twice in a given list.	[1, 2, 3, 4, 5, 2] → 2
Remove duplicate	The goal of this task is to remove the element that appears twice in a given list.	[1, 2, 3, 4, 5, 2] → [1, 2, 3, 4, 5]
Remove element	The goal of this task is to remove a given element from a list.	([1, 2, 3, 4, 5], 3) → [1, 2, 4, 5]
Reverse	The goal of this task is to reverse a given list.	“abcde” → “edcba” [1, 2, 3, 4, 5] → [5, 4, 3, 2, 1]
Reverse & append	The goal of this task is to append the reversed list to the initial list.	“abc” → “abccba” [1, 2, 3] → [1, 2, 3, 3, 2, 1]
Count elements	The goal of this task is to count the number of elements in a given list or string.	“abcde” → 5 [1, 2, 3] → 3

Table 6.1: The tasks selected for comparison.

```haskell
1 quicksort :: (Ord a) => [a] -> [a]
2 quicksort [] = []
3 quicksort (x:xs) = quicksort [y | y <- xs, y <= x] ++ [x] ++ quicksort [y | y <-
    xs, y > x]
```

Listing 6.1: Sample Haskell solution for *quick sort* with a time complexity of $O(n \cdot \log n)$.

```prolog
1 quicksort([],[]).
2 quicksort([X|Xs],Ys) :- partition(Xs,X,Left,Right), quicksort(Left,Ls),
3         quicksort(Right,Rs), append(Ls,[X|Rs],Ys).
```

```
4
5 partition([],Y,[],[]).
6 partition([X|Xs],Y,[X|Ls],Rs) :- X <= Y, partition(Xs,Y,Ls,Rs).
7 partition([X|Xs],Y,Ls,[X|Rs]) :- X > Y, partition(Xs,Y,Ls,Rs).
8
9 append([],Ys,Ys).
10 append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

Listing 6.2: Sample Prolog solution for *quick sort* with a time complexity of $O(n \cdot \log n)$.

Get Initials

An algorithm that extracts the initials of given strings must iterate through the list and each string in this list must be filtered for upper-case letters. So in theory the complexity of such an algorithm is $O(n * k)$, where n is the number of strings and k the number of characters in each string. As k differs from string to string, the algorithm has a complexity of $O(n^2)$. An implementation of this algorithm in Haskell and Prolog can be found in Listing 6.3 and Listing 6.4.

```
1 getInitials :: [String] ->[String]
2 getInitials xs = map initials xs
3
4 initials :: String -> String
5 initials xs = [x | x <- xs, isUpper x]
```

Listing 6.3: Sample Haskell solution for the *get initials* task with a time complexity of $O(n^2)$.

```
1 isUpper(C):- member(C, ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L'
    , 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']).
2
3 initGetInitials([],[]).
4 initGetInitials([X|Xs],[A|Z]) :- string_chars(X, L), getInitials(L, A),
    initGetInitials(Xs, Z).
5
6 getInitials([],[]).
7 getInitials([X|Xs],[X|Z]):- isUpper(X),!, getInitials(Xs, Z).
8 getInitials([X|Xs],Z):- getInitials(Xs, Z).
```

Listing 6.4: Sample Prolog solution for the *get initials* task with a time complexity of $O(n^2)$.

Reverse

Reversing a list can be done in linear time ($O(n)$) and is easy to implement by using recursion. An algorithm that reverses a list must first iterate to the end of the list and then recursively appending the traversed elements. An implementation of this algorithm can be found in Listing 6.5 as Haskell code and in Listing 6.6 as Prolog code.

```
1 reverse :: [a] -> [a]
2 reverse [] = []
3 reverse (x:xs) = reverse xs ++ [x]
```

Listing 6.5: Sample Haskell solution for reversing a list with a time complexity of $O(n)$.

```
1 reverse([], Y, Y).
2 reverse([X|Xs], Y, Z) :- reverse(Xs, Y, [X|Z]).
```

Listing 6.6: Sample Prolog solution for reversing a list with a time complexity of $O(n)$.

6.2 Results

Table 6.2 shows which tasks could be solved by the systems. First it was tested if a system is able to produce an output and if so, if the output is correct. Details can be found in the following subsections.

Some users might be able to correctly synthesize some of the unsolved tasks for some of the systems, but even after an intensive examination of the systems, I was not able to induce code sequences for all tools.

Problem / System	*MagicHaskeller*	*FOIL*	*Metaopt*	*Hoogle+*
Sort	✓	✓	–	–
Split names	✓	–	-	–
Get initials	✓	–	✓	–
Duplicate elements	✓	↯	–	–
Find duplicate	↯	–	✓	–
Remove duplicate	✓	–	–	✓
Remove element	✓	–	–	–
Reverse	✓	✓	↯	–
Reverse & append	✓	↯	–	–
Count elements	✓	↯	–	–

Table 6.2: Experimental Results for all four systems[1].

The structure of the mentioned subsections might differ from section to section as the systems are in some ways quite difficult to compare and also the results differ a lot.

[1]✓: A correct output was produced.
↯: An incorrect output was produced.
–: No output was produced.

6.2.1 MagicHaskeller

To synthesize programs with *MagicHaskeller* I first used the web interface to obtain a Haskell code sequence which I then tested in GHCi. The system had no problem to synthesize nine out of ten programming tasks within seconds. All programs consist of only one function `f`, most of them are higher-order functions and use lambda expressions, but some also use just one library function such as `sort` for the sorting task, `nub` to remove duplicates or `reverse` to reverse a list (see Table 6.3).

Nonetheless, *MagicHaskeller* was not able to synthesize a correct code sequence for the *find duplicate* task. Normally the system needs only one input-output example, but for this task also two or more examples did not lead to a correct program.

The complexity of the used Haskell functions determines the efficiency of the synthesized expression. That is why to analyse the efficiency of the synthesized programs one must determine the complexity of the used Haskell standard library functions as well as the complexity of function compositions or λ-abstractions. The analysis of the synthesized programs also can be found in Table 6.3. The algorithms are analyzed by their runtime complexity, which is expressed by using the *big O notation*.

Problem	**Solution**	**Complexity**
Sort	`f = sort`	$O(n \cdot \log n)$
Split names	`f = \xs -> transpose (map words xs)`	$O(n^2)$
Get initials	`f = map (filter isUpper)`	$O(n^2)$
Duplicate elements	`f = \xs -> concat (transpose (replicate 2 xs))`	$O(n)$
Remove duplicate	`f = nub`	$O(n^2)$
Remove element	`f = \a b -> filter (/= b) a`	$O(n)$
Reverse	`f = reverse`	$O(n)$
Reverse & append	`f = \a -> a ++ reverse a`	$O(n)$
Count elements	`f = length`	$O(n)$

Table 6.3: Runtime complexity of the synthesized code sequences (9/10 tasks).

Most of the algorithms have a complexity of $O(n)$. As the `sort` function – which is part of the `Data.List` Haskell library – is implemented as *merge sort* it has a complexity of $O(n \cdot \log n)$. Compared against other sorting algorithms *merge sort* is rather efficient, but needs additional memory.

6.2.2 FOIL

Program synthesis with *FOIL* is rather difficult as the input data files are quite hard to create. The definition of types and constants as well as the specification of positive examples can be tedious, but are rather simple tasks compared to the definition of relations.

The system comes with a few pre-built data files, to solve the tasks *member*, *sort* and *reverse*. *Member* was already described in one of the previous chapters and checks if a given element is part of a given list. *Sort* and *reverse* are tasks as described in Section 6.1.

To know how to define such relations the user must have an idea of the structure of the final algorithm. This leads to input files with a few hundred or even thousands lines of data, e.g. the data file for the task *reverse* consists of 9266 lines of code and four relations (`components(L,E,L)`, `null(L)`, `list(L)` and `append(L,L,L)`) had to be defined to synthesize Horn clauses that describe the act of reversing a list. For the relation `append`, that describes how to link two lists, all possible combinations for input lists with numbers from 1 to 4 and sizes from an empty list to a list with four elements had to be defined, which led to 8991 examples. As a consequence it is nearly impossible to work with strings such as names as this would lead to very, very large number of constants that has to be declared. If an *undeclared constant* is used – in other words an element that was not defined in the set of constants – the system does not produce an output. The synthesized Horn clauses can be found in Table 6.4.

Problem	**Solution**	**Complexity**
Sort	See Listing 6.7 and Listing 6.8	$O(n^2)$
Duplicate elements	See Listing 6.9 and Listing 6.10	*Incorrect*
Reverse	See Listing 6.11 and Listing 6.12	$O(n^2)$
Reverse & append	`reverseAndAppend(A,A) :- null(A)`	*Incorrect*
Count elements	See Listing 6.13 and Listing 6.14	*Incorrect*

Table 6.4: Runtime complexity of the synthesized code sequences (5/10 tasks).

Even if relations are defined, *FOIL* is not able to synthesize correct Horn clauses for three out of five tasks (see Table 6.4). Of course this could be ascribed to the fact that relations were either incorrectly defined or relations that are crucial to solve the task are missing.

To exploit *FOIL's* full potential correct command line options must be used. For example for synthesizing the *duplicate elements* task as shown in Listing 6.9 the command `.\foil6 -v0 -N data < duplicateElements.d` must be used to obtain a result. When using e.g. the flag `-N`, the system does not consider any negative literals except negated equality literals. Further instructions on how to use the various command line options are given in the corresponding manual.

The algorithm for sorting found by *FOIL* (see Listing 6.7 and Listing 6.8) is correct, but more complex than it needs to be and thus has a complexity of $O(n^2)$.

```
1 sort([],[]) :-
2 sort(A,B) :- components(A,C,[]), sort([],E), components(B,C,E)
3 sort(A,B) :- components(A,C,D), components(B,E,F), sort(D,G), components(B,C,G),
      components(F,I,J), lt(E,I)
4 sort(A,B) :- components(A,C,D), components(B,E,F), sort(D,G), components(G,E,H),
      lt(E,C), components(I,C,H), sort(I,F)
```

Listing 6.7: A correct solution for sorting a list found by *FOIL*.

```
1 sort([],[]).
2 sort(A,B) :- components(A,C,[]), sort([],E), components(B,C,E).
3 sort(A,B) :- components(A,C,D), components(B,E,F), sort(D,G), components(B,C,G),
      components(F,I,_J), E < I.
4 sort(A,B) :- components(A,C,D), components(B,E,F), sort(D,G), components(G,E,H),
5         E < C, components(I,C,H), sort(I,F).
6
7 components([X|Xs],X,Xs).
```

Listing 6.8: Corresponding Prolog code to the sorting algorithm found by *FOIL*.

FOIL did find clauses for duplicating the elements of a list (see Listing 6.9), but it is not correct which can easily be seen when looking at line 1 or 2 where comparisons between elements are made. The corresponding Prolog code can be found in Listing 6.10.

```
1 duplicate(A,B) :- components(A,C,D), append(E,D,A), append(D,E,F),
2 components(F,G,H), A<>E, E<>H
3 duplicate(A,B) :- A<>B, list(B)
4 duplicate(A,B) :- append(C,D,A), append(D,C,E), components(E,F,C),
5 duplicate(C,H), list(H)
```

Listing 6.9: *FOIL's* attempt to synthesize the *duplicate elements* task.

```
1 duplicate(A,B) :- append(E,D,[C|D]), append(D,E,[G|H]), [C|D] \= E, E \= H.
2 duplicate(A,B) :- A \= B, list(B).
3 duplicate(A,B) :- append(C,D,A), append(D,C,E), components(E,F,C), duplicate(C,H),
      list(H).
4
5 components([X|Xs],X,Xs).
6
7 append([],Ys,Ys).
8 append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
9
10 list([]).
11 list([_|Xs]) :- list(Xs).
```

Listing 6.10: Corresponding Prolog code to *FOIL's* incorrect solution for duplicating.

The algorithm for reversing a list found by *FOIL* (see Listing 6.11 and Listing 6.12) is correct, but again more complicated than necessary and has a complexity of $O(n^2)$. As shown in Section 6.1.1 reversing a list can be done in linear time.

```
1 reverse(A,A) :- null(A)
2 reverse(A,B) :- components(A,C,D), reverse(D,E), append(F,D,A), append(E,F,B)
```

Listing 6.11: A correct solution for reversing a list synthesized by FOIL.

```
1 reverse([],[]).
2 reverse(A,B) :- components(A,C,D), reverse(D,E), append(F,D,A), append(E,F,B).
3
4 components([X|Xs],X,Xs).
5
6 append([],Ys,Ys).
7 append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

Listing 6.12: Corresponding Prolog code to the reversing algorithm found by *FOIL*.

The clauses found by *FOIL* (see Listing 6.13) for counting the number of elements in a list are also incorrect as the algorithm simply returns the second element of the list. For example when calling `count([1,2,3], X)` (implemented in Listing 6.14) the function returns `X = 2`. The incorrectness of the algorithm gets more obvious when a list of characters is passed as first argument, e.g. `count(['a', 'b'], X)` returns `X = b`.

```
1 count([],B) :- ~components(C,B,[])
2 count(A,B) :- components(A,C,D), components(E,B,A), components(D,B,F)
3 count(A,B) :- components(A,C,D), count(D,E), B<>E, components(D,F,G), count(G,H),
      components(I,B,G), B<>H, ~count(I,C)
```

Listing 6.13: *FOIL's* attempt to synthesize the count function.

```
1 count([],B) :- \+ components(_C,B,[]).
2 count(A,B) :- components(A,_C,D), components(_E,B,A), components(D,B,_F).
3 count(A,B) :- components(A,C,D), count(D,E), B \= E, components(D,_F,G),
4     count(G,H), components(I,B,G), B \= H, \+ count(I,C).
5
6 components([X|Xs],X,Xs).
```

Listing 6.14: Corresponding Prolog code to *FOIL's* incorrect solution for counting.

6.2.3 Metaopt

As already mentioned *Metaopt* comes with a large number of examples divided into four categories: *converge*, *duplicate*, *strings* and *postman*.
Converge and *duplicate* both cover the *find duplicate* problem, where the goal is to find a number in a list of numbers that appears twice in it. The difference is, that the examples and data in *converge* should show that *Metaopt* converges on minimal cost programs after given a certain number of training examples, which is the case after approximately 15 examples according to Cropper and Muggleton [4]. *Duplicate* on the other hand just compares the efficiency of *Metaopt*, *Metagol* and *Metagol*$_O$ when synthesizing code sequences to solve the *find duplicate* problem as described above.
Examples and data in *strings* cover real-world string transformations, such as extracting a first name from the sentence "My name is..." or from a given e-mail address. Further there a examples that can handle strings with numbers in it such as dates, for example formatting a sequence of numbers such that it is equal to a common date format (e.g. 1112011 ⇒ 1/11/2011).

Last *postman* covers the postman experiment described in a previous paper of Cropper and Muggleton [3]. The goal here is to synthesize a process of collecting and delivering letters as done by postmen. As this is a rather specific scenario I will not cover this problem further.
The process of synthesizing other examples than the given ones proved to be difficult as it is nowhere described how to do so. I analyzed the data file for the string problems to find out how new problems can be synthesized by *Metaopt*. It transpired that to synthesize code sequences for the given problem tasks only three examples have to be provided. Further meta-rules (such as *curry* and *chain*) and predicates (such as *is_letter/1* or *is_uppercase/1*) have to be defined. Nonetheless I tried to synthesize programs for the tasks described in Section 6.1 by solely defining five input-output examples, but no meta-rules or predicates. The results can be found in Table 6.2.

Problem	**Solution**	**Complexity**
Get Initials	Listing 6.15	$O(n)$
Find duplicate	Listing 6.16	$O(n \cdot \log n)$
Reverse	Listing 6.17	*Incorrect*

Table 6.5: Runtime complexity of the synthesized code sequences (3/10 tasks).

It is not surprising that by simply giving input-output examples *Metaopt* was not able to solve most of the problems. It seems like the system merely works for tasks defined by the developers. According to the Readme file, the results can be reproduced with new data but there is no explicit mention of synthesizing programs for new tasks. When observing the results, it becomes pretty obvious that synthesizing new tasks is nearly impossible without defining predicates because none of the synthesized functions uses built-in predicates. The predicates used in Listing 6.15, 6.16 and 6.17 are implemented by the developers and can be found in *bk.pl* (see supplementary material[2]). Those were used to help analyze the complexity of the synthesized code sequences as done in Table 6.5.

```
1 f(A,B):-filter(A,B,is_uppercase).
```

Listing 6.15: The code sequence for extracting initials found by *Metaopt*.

```
1 f(A,B):-mergesort(A,C),f_1(C,B).
2 f_1(A,B):-head(A,B),f_2(A,B).
3 f_1(A,B):-tail(A,C),f_1(C,B).
4 f_2(A,B):-tail(A,C),head(C,B).
```

Listing 6.16: The code sequence for finding a duplicate a list found by *Metaopt*.

```
1 f(A,B):-myreverse(A,C),f_1(C,B).
2 f_1(A,B):-dropWhile(A,B,is_uppercase).
```

[2] urlhttps://git.uibk.ac.at/csas8322/learning-efficient-programs-bachelor-thesis

Listing 6.17: *Metaopt's* attempt for finding a code sequence that reverses a list.

The solution for *get initials* can not be compared with the solution produced by *MagicHaskeller* as *Metaopt* was only able to extract the initials of a single name and not for a list of names where each name is a list itself.

The synthesized code sequence for finding an element that appears twice in a list (*find duplicate*) is rather impressive as the system was able to find a code sequence that first sorts the list and then compares all elements until two adjacent elements are found that are the same. Due to the fact that sorting was implemented as *merge sort* this algorithm is quite efficient and according to the authors it is also the most efficient for the hypothesis space of the system [4]. Of course there are even more efficient ways to find a duplicate (e.g. using a *hash map*).

Unfortunately the system was not able to synthesize a correct solution for *reverse*, although a predicate called *myreverse/2* is implemented. The code sequence shown in Listing 6.17 only produces a correct result if a list is not empty and contains more than one element and no upper-case letters.

6.2.4 Hoogle+

Hoogle+ seems quite promising on paper, but when using the web interface to synthesize other algorithms than the five predefined and presented tasks all of a sudden every query reaches the timeout limit. In most cases the system is able to infer a matching type, but fails on finding a code sequence. Further most of the predefined examples do only work when the correct type is either manually given or chosen by the user. "Simple" types such as `x:[a] → [a]` (proposed by the tool and ranked first by the system's heuristic function) do not lead to a result, e.g. for the *remove duplicate* task. Instead the type `Eq a ⇒ x:[a] → [a]` (ranked fifth) must be chosen to get a (correct) result for the above mentioned task.
Hoogle+ was tested by performing a user study with 30 participants, which had to rate themselves as either Haskell-novices, intermediate-level users or experts. According to the authors 12 participants were novices, 10 intermediates and 8 experts. The users were given four tasks plus one training example to get familiar with the tool. 73 out of 120 task were completed and only one out of 73 solutions was wrong [14]. When analyzing the results of this study and after trying out the examples, it is hardly surprising that tasks that require an exact type definition as described above could only be solved by just a few of the participants. More precisely, only six participants were able to solve *Task B* of the study (the *remove duplicate* task).
Taken into account all of the problems mentioned above it is not surprising that *Hoogle+* was only able to solve one of the tasks. This task is further one of the predefined and well-described examples. When choosing the input exactly as it is documented in the corresponding paper a code sequence for the *remove duplicate* example can be synthesized (see Table 6.6), but if e.g. a more general type is chosen, the systems again reaches the

timeout.

Problem	**Solution**	**Complexity**
Remove duplicate	`\xs -> map head (group  xs)`	$O(n)$

Table 6.6: Runtime complexity of the synthesized code sequence (1/10 tasks).

Compared to the solution of *MagicHaskeller* (see Table 6.3), where the in `Data.List` predefined `nub` function is used to remove elements that appear twice in the given list, the solution of *Hoogle+* (see Table 6.6) is more efficient because it is a linear time algorithm, while *MagicHaskeller's* solution has quadratic behaviour. It is more likely that this is a coincidence and not intended by the developers as there was no mention of a focus on efficiency in the corresponding paper.

7 Conclusion

After intensively analysing and testing various systems for program synthesis, it can be concluded that there are some promising examples of tools for inductive programming. It has been shown that systems that use already existing libraries (such as *MagicHaskeller*) are quite successful when it comes to synthesizing single-line code sequences or more precisely singe-line functions.

Especially *MagicHaskeller* is extremely simple to use and therefore a tool that can even help people with no experience in programming to write simple Haskell programs. Although it was implemented 15 years ago it is still competitive in comparison with other systems as, for example, the search for a suitable code sequence is of short duration. Its main advantage is that for most of the problems only one input-output example is needed, but if necessary more examples can be given. The implemented *exemplify* feature gives on the one hand instant feedback if the synthesized function is correct for a number of test cases and makes it on the other hand possible to narrow the search by automatically adding more input-output examples. Further the system can be used by simply accessing the web interface which makes it available for the general public.

Hoogle+ is quite similar in its implementation and application but unfortunately not yet ready to use as the system does not terminate in most of the cases that were tested. It seems like this might be due to the inability of the system to find a code sequence when given a type that is too general and therefore the search through the abstract transition net (ATN) as described in subsection 5.4.1 can not be finished before a timeout is reached. Of course this assumption should be further examined, but this would go beyond the scope of this thesis. Nonetheless the accuracy of the type inference feature and the corresponding heuristic function that ranks inferred types of *Hoogle+* are quite impressive.

Metaopt has the potential to synthesize powerful and efficient programs if the respective meta-rules and predicates are defined by the user or the developers. To do so the user has to know the structure of the algorithm in advance. Then and only then the system is able to find correct and efficient Prolog code sequences by giving three to five examples. Further it must be said that currently predefined tasks (including meta-rules and predicates) purely focus on string transformations. Moreover it is poorly documented how to use the system and how new tasks can be added. An intensive analysis of the available files and rewriting bash scripts as wells as Prolog files were necessary to only start an attempt on synthesizing programs for new problems. That said, *Metaopt* is far apart from being used by programming novices or people that are not able to program on their own.

FOIL was for sure an impressive system when it was introduced in 1990 and certainly helped to draw attention to the field of inductive logic programming. Nowadays and in comparison with other ILP systems it probably seems slightly impracticable as it is

only able to induce Horn clauses and not executable programs or at least code sequences. Due to the fact that constants, types and relations have to be defined by the user, programming novices might not be able to use the system but on the other hand it is well documented how to use it. Nonetheless some algorithms require to predefine several relations and thousands of input-output examples. To define such relations one must further know the structure of the algorithm in advance as in the case of *Metaopt*.

When it comes to the efficiency of the synthesized programs it must be said that at this moment research should probably focus on developing systems that are able to induce all kind of (small) programs without having the need of predefining parts of the algorithm such as types, meta-rules, predicates or relations. Nonetheless all of the described systems are able to synthesize programs in some kind of way, but only *MagicHaskeller* seems usable with little to no effort.

Bibliography

[1] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In M. Odersky and P. Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM, 2000.

[2] A. Cropper and S. Dumancic. Inductive logic programming at 30: a new introduction. *CoRR*, abs/2008.07912, 2020.

[3] A. Cropper and S. H. Muggleton. Learning efficient logical robot strategies involving composable objects. In Q. Yang and M. J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 3423–3429. AAAI Press, 2015.

[4] A. Cropper and S. H. Muggleton. Learning efficient logic programs. *Mach. Learn.*, 108(7):1063–1083, 2019.

[5] P. Flener and U. Schmid. An introduction to inductive programming. *Artif. Intell. Rev.*, 29(1):45–62, 2008.

[6] D. Gantenbein. Flash fill gives excel a smart charge. `https://www.microsoft.com/en-us/research/blog/flash-fill-gives-excel-smart-charge/`, 2013. Accessed: 2020-05-02.

[7] A. L. Gaunt, M. Brockschmidt, R. Singh, N. Kushman, P. Kohli, J. Taylor, and D. Tarlow. Summary - terpret: A probabilistic programming language for program induction. *CoRR*, abs/1612.00817, 2016.

[8] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In T. Ball and M. Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 317–330. ACM, 2011.

[9] S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Found. Trends Program. Lang.*, 4(1-2):1–119, 2017.

[10] Z. Guo, M. James, D. Justo, J. Zhou, Z. Wang, R. Jhala, and N. Polikarpova. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang.*, 4(POPL):12:1–12:28, 2020.

[11] T. Helmuth and L. Spector. General program synthesis benchmark suite. In S. Silva and A. I. Esparcia-Alcázar, editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015*, pages 1039–1046. ACM, 2015.

[12] J. Hernández-Orallo and M. J. Ramírez-Quintana. Inverse narrowing for the induction of functional logic programs. In J. L. Freire-Nistal, M. Falaschi, and M. V. Ferro, editors, *1998 Joint Conference on Declarative Programming, APPIA-GULP-PRODE'98, A Coruña, Spain, July 20-23, 1998*, pages 379–392, 1998.

[13] M. Hofmann, E. Kitzelmann, and U. Schmid. A unifying framework for analysis and evaluation of inductive programming systems. *Proceedings of the 2nd Conference on Artificial General Intelligence, AGI 2009*, pages 74–79, 06 2009.

[14] M. B. James, Z. Guo, Z. Wang, S. Doshi, H. Peleg, R. Jhala, and N. Polikarpova. Digging for fold: synthesis-aided API discovery for haskell. *Proc. ACM Program. Lang.*, 4(OOPSLA):205:1–205:27, 2020.

[15] S. Katayama. Power of brute-force search in strongly-typed inductive functional programming automation. In C. Zhang, H. W. Guesgen, and W. Yeap, editors, *PRICAI 2004: Trends in Artificial Intelligence, 8th Pacific Rim International Conference on Artificial Intelligence, Auckland, New Zealand, August 9-13, 2004, Proceedings*, volume 3157 of *Lecture Notes in Computer Science*, pages 75–84. Springer, 2004.

[16] S. Katayama. Systematic search for lambda expressions. In *Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005*, pages 195–205, 2005.

[17] S. Katayama. Systematic search for lambda expressions. In M. C. J. D. van Eekelen, editor, *Trends in Functional Programming*, volume 6, pages 111–126. Intellect, 2007.

[18] S. Katayama. Efficient exhaustive generation of functional programs using monte-carlo search with iterative deepening. In T. B. Ho and Z. Zhou, editors, *PRICAI 2008: Trends in Artificial Intelligence, 10th Pacific Rim International Conference on Artificial Intelligence, Hanoi, Vietnam, December 15-19, 2008. Proceedings*, volume 5351 of *Lecture Notes in Computer Science*, pages 199–210. Springer, 2008.

[19] S. Katayama. Recent improvements of magicHaskeller. In U. Schmid, E. Kitzelmann, and R. Plasmeijer, editors, *Approaches and Applications of Inductive Programming, Third International Workshop, AAIP 2009, Edinburgh, UK, September 4, 2009. Revised Papers*, volume 5812 of *Lecture Notes in Computer Science*, pages 174–193. Springer, 2009.

[20] S. Katayama. MagicHaskeller: System demonstration. *Proceedings of AAIP 2011 - 4th International Workshop on Approaches and Applications of Inductive Programming*, pages 63–70, 01 2011.

[21] S. Katayama. An analytical inductive functional programming system that avoids unintended programs. In O. Kiselyov and S. J. Thompson, editors, *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, Philadelphia, Pennsylvania, USA, January 23-24, 2012*, pages 43–52. ACM, 2012.

[22] S. Katayama. MagicHaskeller: Automatic inductive functional programmer by systematic search. `https://hackage.haskell.org/package/MagicHaskeller-0.9.6.6.1`, 2017. Accesed: 2020-07-28.

[23] E. Kitzelmann. Data-driven induction of recursive functions from input/output-examples. In E. Kitzelmann and U. Schmid, editors, *Proc. of the ECML/PKDD 2007 Workshop on Approaches and Applications of Inductive Programming*, pages 15–26, 2007.

[24] E. Kitzelmann. Analytical inductive functional programming. In M. Hanus, editor, *Logic-Based Program Synthesis and Transformation, 18th International Symposium, LOPSTR 2008, Valencia, Spain, July 17-18, 2008, Revised Selected Papers*, volume 5438 of *Lecture Notes in Computer Science*, pages 87–102. Springer, 2008.

[25] E. Kitzelmann. Inductive programming: A survey of program synthesis techniques. In U. Schmid, E. Kitzelmann, and R. Plasmeijer, editors, *Approaches and Applications of Inductive Programming, Third International Workshop, AAIP 2009, Edinburgh, UK, September 4, 2009. Revised Papers*, volume 5812 of *Lecture Notes in Computer Science*, pages 50–73. Springer, 2009.

[26] E. Kitzelmann and U. Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *J. Mach. Learn. Res.*, 7:429–454, 2006.

[27] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960.

[28] S. Muggleton. Inductive logic programming. *New Gener. Comput.*, 8(4):295–318, 1991.

[29] S. Muggleton. Inverse entailment and progol. *New Gener. Comput.*, 13(3&4):245–286, 1995.

[30] S. Muggleton and C. Feng. Efficient induction of logic programs. In S. Arikawa, S. Goto, S. Ohsuga, and T. Yokomori, editors, *Algorithmic Learning Theory, First International Workshop, ALT '90, Tokyo, Japan, October 8-10, 1990, Proceedings*, pages 368–381. Springer/Ohmsha, 1990.

[31] S. Muggleton and L. D. Raedt. Inductive logic programming: Theory and methods. *J. Log. Program.*, 19/20:629–679, 1994.

[32] S. H. Muggleton, D. Lin, N. Pahlavi, and A. Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Mach. Learn.*, 94(1):25–49, 2014.

[33] S. H. Muggleton, D. Lin, and A. Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. *Mach. Learn.*, 100(1):49–73, 2015.

[34] R. Olsson. Inductive functional programming using incremental program transformation. *Artif. Intell.*, 74(1):55–8, 1995.

[35] E. R. Pantridge, T. Helmuth, N. F. McPhee, and L. Spector. On the difficulty of benchmarking inductive program synthesis methods. In P. A. N. Bosman, editor, *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*, pages 1589–1596. ACM, 2017.

[36] G. Plotkin. *Lattice Theoretic Properties of Subsumption.* MIP-R-. Edinburgh University, Department of Machine Intelligence and Perception, 1970.

[37] J. R. Quinlan. Learning logical definitions from relations. *Mach. Learn.*, 5:239–266, 1990.

[38] J. R. Quinlan. Learning first-order definitions of functions. *J. Artif. Intell. Res.*, 5:139–161, 1996.

[39] J. R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In P. Brazdil, editor, *Machine Learning: ECML-93, European Conference on Machine Learning, Vienna, Austria, April 5-7, 1993, Proceedings*, volume 667 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 1993.

[40] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In A. Gill, editor, *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, pages 37–48. ACM, 2008.

[41] U. Schmid. *Inductive Synthesis of Functional Programs, Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*, volume 2654 of *Lecture Notes in Computer Science.* Springer, 2003.

[42] U. Schmid, E. Kitzelmann, and R. Plasmeijer, editors. *Approaches and Applications of Inductive Programming, Third International Workshop, AAIP 2009, Edinburgh, UK, September 4, 2009. Revised Papers*, volume 5812 of *Lecture Notes in Computer Science.* Springer, 2010.

[43] R. Singh. *Accessible programming using program synthesis.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2014.

[44] J. M. Spivey. Combinators for breadth-first search. *J. Funct. Program.*, 10(4):397–408, 2000.

[45] P. D. Summers. A methodology for LISP program construction from examples. *J. ACM*, 24(1):161–175, 1977.