Delft University of Technology
Master of Science Thesis in Computer Science

# Speeding up program synthesis using specification discovery

**Jacob de Jong**

# Speeding up program synthesis using specification discovery

Master of Science Thesis in Computer Science

Algorithmics Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands

Jacob de Jong

June 26, 2023

**Author**
Jacob de Jong

**Title**
Speeding up program synthesis using specification discovery

**MSc Presentation Date**
July 4, 2023

**Graduation Committee**

| | |
|---|---|
| Dr. Matthijs Spaan | Delft University of Technology |
| Dr. Sebastijan Dumančić | Delft University of Technology |
| Dr. Jesper Cockx | Delft University of Technology |

## Abstract

How convenient would it be to have an AI that relieves us programmers from the burden of coding? Program synthesis is a technique that achieves exactly that: it automatically generates simple programs that meet a given set of examples or adhere to a provided specification. This is often done by enumerating all programs in the search space and returning the first program that satisfies the requirements. However, these algorithms frequently enumerate redundant programs because of symmetries in the search space. We propose a new constraint discovery system that is able to detect these symmetries in a language and systematically generate symmetry-breaking constraints for them. To test these constraints, we implemented a novel, re-usable framework for program synthesis called *Herb.jl*. The generated constraints are shown to cut down search spaces to less than 25% of the original size and reduce the enumeration time by a factor of 3. Furthermore, this approach is extended to automatically discover semantic specifications without needing an expert. The effectiveness of these specifications is evaluated with an existing specification-based synthesizer, which shows that adding these specifications is an effective way to cut the synthesis time in half for domains where expert-defined specifications are not available. Together, these approaches demonstrate the effectiveness of extracting additional information from a language and applying it during enumeration.

# Preface

The thesis that lies before you has been written as the last step in obtaining my Master of Science degree in Computer Science at Delft University of Technology. The research, development and writing of this thesis spanned from November 2022 to June 2023.

During my studies at TU Delft, I mainly took an interest in the algorithmics and programming languages courses. Program synthesis is related to both of these fields and therefore it perfectly matched my interests. This thesis shows a way in which concepts from both fields can be combined. In the past seven months, I also got the opportunity to work on Herb.jl, a program synthesis framework that will be introduced in this thesis. It has been very motivating and fulfilling to see Herb.jl already getting used in several student projects, in which the framework is extended with new functionality.

I would like to thank Sebastijan Dumančić, Jesper Cockx and Tilman Hinnerichs for the excellent feedback and guidance I received, both during our meetings, in the rounds of feedback on this thesis and whenever I had questions. I would also like to thank Ivar de Bruin for his review of my thesis as someone that is not involved in program synthesis, allowing me to make the text accessible to a wider audience. Finally, I want to thank everyone with whom I interacted during my thesis. It was a pleasure to be able to work with you.

Jacob de Jong

Delft, The Netherlands
26th June 2023

# Contents

# Chapter 1

# Introduction

Program synthesis has been a topic since digital computers were just invented and it has roots in the fields of artificial intelligence and programming languages. The idea of an ‘automatic’ programmer is considered by many to be one of the ultimate goals of computer science. Ideally, there would be a program synthesizer that is able to quickly generate complicated programs from a simple specification or description. It is worth noting that the term ‘program’ in program synthesis is a very broad term. Next to traditional computer programs, program synthesis is employed in domains such as robotics to synthesize action sequences [23] and in 3D modelling for synthesizing 3-dimensional structures [10].

Recently, pre-trained large language models such as ChatGPT and GPT-4 have made some great advancements [28]. These models have the capability to rapidly synthesize relatively large and complex programs from a natural language description. The downside to these large language models is their reliance on statistical patterns rather than language definitions. Consequently, they cannot provide guarantees on the efficiency or correctness of the generated code and frequently create bugs [19].

Language definitions can be divided into two essential components: syntax and semantics [27]. Although this terminology originates from the field of linguistics, it is equally applicable to programming languages. Syntax pertains to how parts of a language can be combined to create well-formed programs. Semantics define the actual meaning and interpretation of these programs.

A common method for synthesizing programs is enumeration, where an algorithm generates every syntactically correct program in a language one by one and verifies if they meet the user’s requirements. Enumeration is particularly effective for synthesizing small programs in complex languages [16]. However, it is also a very inefficient method, since it does not utilize any information from the semantics of a language. This means that it solely relies on knowledge of how operators and literals can be combined, without considering their meaning and characteristics. Consequently, a significant number of semantically equivalent programs are explored.

To illustrate this, consider a language of simple arithmetic expressions. A program enumerator will enumerate the programs 4, $4+0$ and $4+(0\times 9)$. Although these programs differ in syntax, they all yield the same result of 4, demonstrating their semantic equivalence. The enumerator is unaware that these programs are semantically equivalent since it does not recognize that multiplication with 0 always equals 0, or that adding 0 to a number does not have an effect.

The goal of a program synthesis problem is to find a program for which the semantics satisfy a certain requirement. Instead of enumerating every syntactically different program, it is a lot more efficient to make use of semantic knowledge to instead enumerate every semantically unique program. The main question this thesis aims to answer is:

*How can we extract semantic knowledge from a language and utilize it to reduce redundant enumerations?*

The primary focus of this thesis is to automatically discover this semantic knowledge and to prevent redundant programs from being enumerated. Of course, users could also define this semantic information manually. However, this is a tedious and error-prone task that requires in-depth knowledge of a language. In addition, a big part of the program synthesis target audience does not have this knowledge of the language or library they are synthesizing from. Automating this process will therefore make synthesizing from arbitrary languages or libraries more accessible to a large group of users.

Other algorithms such as MORPHEUS [12] or NEO [13] make use of specifications, which express semantic knowledge by defining a relation between the input and the output of a certain operator in the language. The specifications are combined with problem-specific information to reduce the search space. Both MORPHEUS and NEO require a user to manually encode these specifications. This thesis will therefore also demonstrate how to discover specifications and compare the extracted information to the information defined by an expert.

Our approach to limiting the number of redundant programs makes use of constraints. Constraints that reduce redundant enumerations are called *symmetry-breaking constraints*. These constraints prevent certain programs from being enumerated by the enumeration algorithm. One could for example add a constraint that forbids multiplication with 0 since it will always produce 0 as a result, which has already been enumerated before. A constraint can thus be used to add some semantic information to the enumeration. We specifically focus on constraints that forbid a certain combination of operations and constraints that forbid a certain ordering of operands, since these can eliminate most symmetries. This thesis introduces a modular and reusable program synthesis framework called *Herb.jl*, which is able to enforce these constraints.

Formulating these constraints is still manageable for the case of integer arithmetic, but for more complex search spaces, it can be a tedious process in which it is easy for humans to make mistakes. For example, when synthesizing in a language with lists, `reverse(append(reverse(a), b))` is semantically equivalent to `append(reverse(b), a)`. This is already significantly more difficult to come up with, even though the semantics of `reverse` and `append` are well-known. Libraries often have more complex functions for which the definition is a lot less clear. Consequently, humans might create incorrect constraints which remove potential solutions from the search space. There might also be valuable constraints that the user is not aware of and therefore does not include, thus limiting the potential of the constraint system.

The second contribution of this thesis is therefore a new constraint discovery system that can automatically generate these constraints from the semantics of a program. Usually, the exact definitions of functions in a language are not available, so the constraint extraction system has to work without those. Instead, it uses an evaluator that can evaluate expressions in the search space and acts as an *oracle*. This constraint discovery system works in three steps. It first generates a bunch of hypotheses in the form of potential semantic equivalences in the language. In the second step, these hypotheses are tested for correctness using the evaluator and random input data. The third step converts the correct hypotheses into constraints that can be used by the constraint system. These constraints decrease the number of enumerated programs to less than 20% and the runtime to almost 25% of the unconstrained approach in some grammars.

An enumeration algorithm equipped with constraints still does not use any problem-specific data. Ideally, one would also incorporate information extracted from the specification or examples in the enumeration. However, information about inputs and outputs is only useful if it can eliminate options during enumeration. For example, when synthesizing a function that operates on lists, and the relation between the length of the input list and the output list is known, we also need information on how operations in the language modify the length of the list. This is again semantic information that has to be provided by the user. The third contribution of this thesis demonstrates how this kind of semantic data can be extracted from a language. Since there are a lot of parallels between discovering equivalences and identifying other semantic properties, this contribution can be seen as an expansion or modification of the constraint discovery system. Just like the constraints, the extracted specifications are a big improvement over not using any specifications at all. However, they do still fall short compared to the specifications that are defined by an expert.

In summary, this thesis makes the following key contributions:

1. We implement a new constraint system made specifically for enumerative program synthesis.
2. We introduce a constraint discovery system that detects semantic equivalences in a language and creates constraints to remove semantically duplicate programs.
3. We show how this constraint discovery system can be modified to extract other kinds of semantic data.
4. We introduce *Herb.jl*, a modular and reusable program synthesis framework that supports constraints.

The rest of this thesis will provide some background information, explain these contributions in more detail and evaluate their effectiveness.

Chapter 2

# Theoretical background

Program synthesis is the process of automatically generating (parts of) computer programs from some sort of description of what the program should do [16]. It is a topic that mainly occupies researchers with an algorithmic background or artificial intelligence background but also has connections to the field of programming languages.

In some sense, program synthesizers are quite similar to compilers; they transform a program or specification in a higher-level programming language into a program in a lower-level language. Compilers operate by applying a predetermined sequence of transformations, for instance, desugaring or translating into another (intermediate) representation. The distinguishing feature of program synthesizers is that they do not operate with those predetermined steps, but rather involve some kind of search [16]. This means that we can give them a specification of *what* we want the desired program to do, as opposed to giving it a specification of *how* the desired program should do it.

A program synthesis framework is mainly defined by three key components: the intent specification, the program space and the search [16]. This chapter gives relevant background information on the field of program synthesis by going over these components and introducing concepts wherever necessary.

$$\begin{aligned} sqrt(n) \Leftarrow\ & \text{find } z \text{ such that} \\ & \quad \text{integer}(z) \text{ and } z^2 \leq n < (z+1)^2 \\ & \text{where integer}(n) \text{ and } 0 \leq n \end{aligned}$$

(a) **Full specification**

$$\begin{aligned} 4 &\to 2 \\ 36 &\to 6 \\ 23 &\to 4 \\ 1 &\to 1 \end{aligned}$$

(b) **Specification by examples**

Figure 2.1: **Different forms of specification in program synthesis for an integer square root function [26].**

## 2.1 Intent specification

The intent specification is what communicates the wish of the user for what the program should do to the program synthesizer. It is sometimes also called the *problem specification* or more commonly just *specification*. To avoid confusion with the language specifications that will also be encountered in this thesis, we will keep calling it *intent specification*. Intent specifications can be classified into complete and incomplete intent specifications. A complete intent specification fully defines what the program should do. Therefore, the user's objective clear to the algorithm. Figure 2.1a shows a classic example of a full intent specification for synthesizing a square root function. Designing such a specification can be as difficult as writing the program itself, and one often needs to add restrictions that are not immediately obvious [16]. For example, it needs to be explicitly stated that both the input and the output of the integer square root function must be an integer and that it is not possible to take the square root of negative numbers. Complete intent specifications are used in deductive synthesis approaches.

In contrast, incomplete intent specifications are much more flexible and allow the user to be less precise, with the trade-off being that it might be ambiguous what the user wants since the problem is under-specified. A program synthesis algorithm for this kind of intent specifications, therefore, has the additional task of not just finding a program that satisfies the specifications, but also the program that the user envisaged. Incomplete specifications can only be used in inductive synthesis approaches. Therefore, algorithms that take incomplete intent specifications are sometimes also called inductive synthesizers. There are multiple kinds of incomplete intent specifications, of which input-output examples, traces and sketches are the most well-known.

Using input-output examples as a specification is perhaps the most accessible way of providing specifications in program synthesis. They are frequently used for data processing [15, 16]. Figure 2.1b illustrates how input-output examples are shaped by giving a specification for the integer square root function. Using this kind of intent specification in program synthesis is also called Programming By Example (PBE) [16].

Traces are essentially an extension of input-output examples [16, 35]. In addition to the input and the output, they also contain intermediate steps. For example, consider that one wants to synthesize the program $x \times 3 + 2$. Instead of having an example $5 \to 17$, we would have $5 \to 15 \to 17$. Using traces as an intent specification is also called Programming by Demonstration (PBD) [16, 24].

A third way of providing an incomplete specification is by sketching [35, 36]. A sketch is a program where some parts are not filled in yet. Constraints on how the program synthesizer has to fill these holes can be provided in the form of assertions. This way of giving specifications is designed to aid programmers by allowing them to give a high-level structure of a program and letting the program synthesizer figure out the low-level details.

## 2.2 Program space

The program space is the search space of a program synthesis problem. It consists of all syntactically correct programs in a certain language. A program in the program space is sometimes also called a '*hypothesis*' since it is a possible solution to a program synthesis problem [16]. The program space is usually infinite in size, with the exception of some rudimentary languages. Every program in the program space adheres to certain rules that define the program structure. These rules are called the *syntax* of a language [33]. The syntax does not consider the meaning of the operators or values in the language, but only how they can be put together. Just like in linguistics, the syntax can be defined using a grammar [4], in this case, a Context-Free Grammar (CFG).

A CFG is a collection of production rules, which are also called '*rewrite rules*'. These rules are built from terminal symbols and non-terminal symbols. A terminal symbol represents an operator or value that can appear in a program. Non-terminal symbols are placeholders that represent a set of combinations of terminal symbols. Each grammar has one special non-terminal symbol which is called the *start symbol.* This symbol represents all programs in the language.

A production rule in a CFG has a single non-terminal symbol on the left-hand side and a combination of terminal and non-terminal symbols on the right-hand side. The combination on the right-hand side is a possible replacement for the non-terminal symbol on the left-hand side. The language of a grammar is each combination of terminal symbols that can be created by recursively applying the production rules to non-terminal symbols, starting with the start symbol.

Figure 2.2a describes the simple integer arithmetic grammar that is used in Chapter 1. Int is the only non-terminal symbol in the language, and $+$, $\times$, $-$, $0, \dots, 9$ and $x$ are the terminal symbols. The first three symbols are the operators, and 0 to 9 are the literals of this language. $x$ is the input variable. When evaluating a program defined by this grammar, a value from an input example is assigned to this input variable. Since Int is the only non-terminal symbol, it is also the start symbol in this grammar.

Often, grammars have more than one non-terminal symbol. Such a grammar can be seen in Figure 2.2b, which represents a language consisting of lists of integers. In this grammar, List is the start symbol.

The programs in the program space are represented as Abstract Syntax Trees (ASTs) [12, 13]. Sometimes, these are also called *derivation trees* or *program trees* [16]. ASTs define the structure of a program as a tree. Each node corresponds to a production rule in the grammar. Figure 2.3 shows two example ASTs from the integer arithmetic grammar in Figure 2.2a.

A grammar does not define the semantics of a language. Inductive program synthesis techniques usually do not have access to a full semantic specification of the language, but they do need access to an evaluator to evaluate any hypothesis they find.

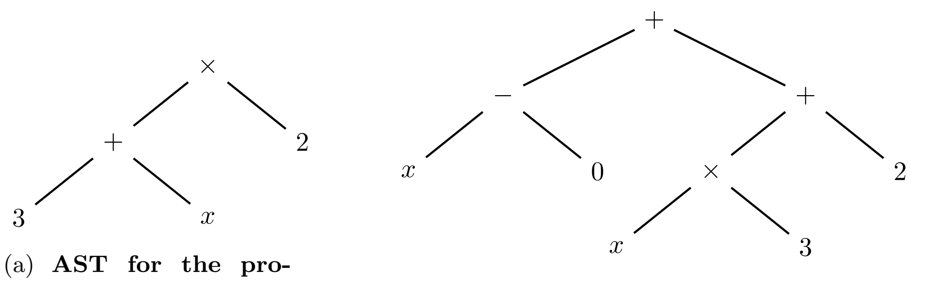
$$\begin{aligned}
\text{Int} &::= \text{Int} + \text{Int}\\
\text{Int} &::= \text{Int} \times \text{Int}\\
\text{Int} &::= \text{Int} - \text{Int}\\
\text{Int} &::= 0\\
&\vdots\\
\text{Int} &::= 9\\
\text{Int} &::= x
\end{aligned}$$

(a) **Integer Arithmetic**

$$\begin{aligned}
\text{List} &::= \texttt{[]}\\
\text{List} &::= \text{List} :: \text{Int}\\
\text{List} &::= \texttt{reverse}(\text{List})\\
\text{List} &::= \texttt{sort}(\text{List})\\
\text{List} &::= \texttt{append}(\text{List}, \text{List})\\
\text{Int} &::= 1\\
&\vdots\\
\text{Int} &::= 9\\
\text{Int} &::= x\\
\text{Int} &::= y
\end{aligned}$$

(b) **Lists**

Figure 2.2: **Grammar production rules for an integer arithmetic language and a list language.**



(a) **AST for the program** $(3 + x) \times 2$

(b) **AST for the program** $(x - 0) + ((x \times 3) + 2)$

Figure 2.3: **Examples of ASTs in the integer arithmetic grammar.**

Defining a grammar is relatively straightforward. The definition does nevertheless affect the performance of program synthesis. It is best to limit the number of equivalent programs that are allowed by the grammar. Different syntactic representations for semantically equivalent programs are called *symmetries* [35]. When symmetries are removed from the program space, we reduce our program space without eliminating any potential solutions. One of the most accessible ways of reducing symmetries is by modifying the grammar [35]. For example, the grammar below removes the symmetry caused by the associativity of addition from the integer arithmetic grammar.

$$\begin{aligned} \text{Int} &::= \text{Int}' \mid \text{Int} + \text{Int}' \\ \text{Int}' &::= x \mid \text{Int} - \text{Int} \mid \text{Int} \times \text{Int} \mid 0 \ldots 9 \end{aligned}$$

The original non-terminal symbol Int now has a variant $\text{Int}'$ that excludes addition. Using the new non-terminal symbol, the production rule for addition $(\text{Int} + \text{Int}')$ now only allows other addition rules as the left child. This enforces left-associativity of the addition operator, and thus no longer allows programs such as $(1 + (x + 3))$ to be generated. Since the program $((1 + x) + 3)$ is still in the grammar, there is no loss of expressive power in the program space. This method can be an effective and easy solution for eliminating symmetries. However, its applicability is limited, since it can not eliminate symmetries such as commutativity of operators. Furthermore, eliminating multiple symmetries by modifying the grammar can make the definition really complex.

Researchers working on constraint satisfaction or combinatorial optimization problems have been eliminating symmetries for decades with so-called symmetry-breaking constraints [11]. A symmetry-breaking constraint is a constraint on the search space that effectively forbids one-half of a symmetry. Symmetry-breaking constraints have not yet been applied to program spaces for program synthesis. The idea of breaking the symmetries in a program space using constraints is something that will be explored in this thesis.

## 2.3 Search

The search component of a program synthesis algorithm is responsible for finding a program in the program space that meets the given intent specification. Depending on the type of intent specification, the search component might look very different.

In deductive approaches, the search component rewrites and transforms the complete intent specification into a program from the program space [26, 35]. Deciding which transformations to apply is not always obvious, and thus a search algorithm is used.

A lot of inductive program synthesis algorithms are based on an enumeration search technique. The idea of enumeration is to systematically explore the program space for ASTs that satisfy the intent specifications. Naturally, this is not a very efficient approach. Enumeration can either be done top-down or bottom-up [35, 7]. Some approaches also combine these two methods [16].

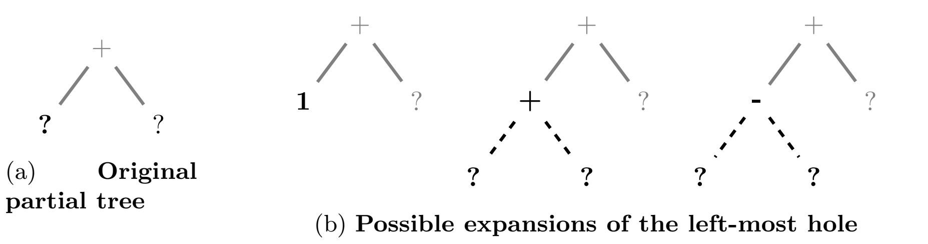

+
?
?
(a) **Original partial tree**
+
1
?
+
+
?
?
?
+
-
?
?
?
(b) **Possible expansions of the left-most hole**

Figure 2.4: **A partial tree and some possible expansions of its left-most hole.**

In *bottom-up enumeration*, the leaves (bottom) of the AST are first constructed and evaluated individually. Larger trees are then discovered by combining these leaves in various ways. By continually combining the lower-level components that were found, it is possible to enumerate arbitrarily large trees. For example, in the search space defined in Figure 2.2a, the programs $1, 2, ..., 9$, and $x$ will be enumerated first. After that, $+$, $\times$ and $-$ are used to get combinations of the previously enumerated rules, such as $5 + x$. Repeating this last step also enumerates for example $3 + (5 + x)$ or $(2 \times 5) - (5 + x)$.

In *top-down enumeration*, an AST is constructed top-down. This means that the main structure of the program is first discovered, and the lower-level components are filled in later. Lower-level components that are not filled in yet are called *holes* and an AST containing holes is called a *partial tree* or an *open hypothesis*, as opposed to *closed hypotheses* that do not have any holes. Consider the search space from Figure 2.2a again. Just like with bottom-up enumeration, $0, 2, ..., 9$, and $x$ are enumerated for the hole. However, some partial trees are also generated: $?+?, ?\times?$ and $?-?$. In the next iteration, one open hypothesis is picked, and one hole in the tree is expanded. Consider filling the left-most hole of the partial tree ?+?, also shown in Figure 2.4a. This hole can be filled using every production rule for the non-terminal symbol *Int*. As a result, the tree is expanded to 0+?, 1+?, ..., 9+?, $x$+?, (?+?)+?, (?$\times$?)+? and (?$-$?)+?. Figure 2.4b shows a selection of these expanded trees.

In this thesis, top-down enumeration will be used. However, the approaches to symmetry-breaking described in this thesis can also be applied to bottom-up enumeration. Some algorithms exploit the use of deduction in inductive synthesis to complement their enumeration [12, 13, 14]. This is covered in more detail in the related work in the next chapter.

# Chapter 3

# Related work

The program synthesis field lacks approaches that specifically address semantic information discovery or automated symmetry breaking. However, that does not mean that there is no relevant literature. This chapter primarily focuses on algorithms that leverage semantic knowledge to perform deduction in Section 3.1. These algorithms are the potential use cases of semantic information discovery. Additionally, Section 3.2 explores POPPER [8], an Inductive Logic Programming system that utilizes semantic knowledge that transfers between domains and hence does not require manual encoding. Lastly, two existing algorithms for extracting semantic knowledge from languages or libraries are discussed in Section 3.3.

## 3.1 Deduction in inductive synthesis

Section 2.2 showed that enumerative synthesizers generally only rely on syntactical information. The synthesizers that are covered in this section also incorporate semantic information in the search by performing deduction. The main purpose of deduction in a search is to use reasoning to 'reject' an open hypothesis, in other words, an AST with holes in it. Since exploring this open hypothesis will potentially lead to a large number of closed hypotheses, rejecting it early can cut off large parts of the search space. There are a couple of related works that make use of some form of constraint, semantic specification or learned lemma to do deduction in combination with their search [12, 13, 14, 18].

### 3.1.1 Semantic specifications

MORPHEUS [12] and NEO [13] are both algorithms that make use of semantic specifications. These specifications describe how certain properties of the output of an operator relate to the input.

The intended use case of MORPHEUS is table transformation tasks in the programming language R. Morpheus utilizes specifications for specific operators or functions that mainly enforce the size of the input and output of a particular operator. For example, the constraint for the `filter` operator is:

$$\mathrm{T}_{out}.\mathrm{row} < \mathrm{T}_{in}.\mathrm{row} \wedge \mathrm{T}_{out}.\mathrm{col} = \mathrm{T}_{in}.\mathrm{col}$$

$\mathrm{T}_{in}$ and $\mathrm{T}_{out}$ represent the input and output tables, and row and col are the numbers of rows or columns of a table. When a (partial) hypothesis is generated, the operator specifications are combined into a hypothesis specification, which contains placeholders for the intermediate tables:

$$\begin{aligned} ?_1.\mathrm{row} <?_3.\mathrm{row} \wedge ?_1.\mathrm{col} =?_3.\mathrm{col} \wedge \\ ?_0.\mathrm{row} =?_1.\mathrm{row} \wedge ?_0.\mathrm{col} <?_1.\mathrm{col} \end{aligned}$$

This is then combined with data from a specific example with input $x_1$ and output $y$ by adding the following:

$$\begin{gathered} x_1 =?_3 \ \wedge \ y =?_0 \wedge \\ x_1.\mathrm{row} = 3 \ \wedge \ x_1.\mathrm{col} = 4 \ \wedge \ y.\mathrm{row} = 2 \ \wedge \ y.\mathrm{col} = 4 \end{gathered}$$

A Satisfiability Modulo Theories (SMT) solver checks this function for satisfiability. If this formula is not satisfiable, the hypothesis can be rejected. If the hypothesis was partial, we do not have to fill it in any more. Morpheus is a synthesis tool that is optimized for table transformations. Nevertheless, it is capable of handling any constraint as long as it is encodable in first-order logic since this is what the SMT-solver accepts. It is however important that a constraint that enforces a certain property is present in multiple operators of the language for the constraint to be effective. If this is not the case, there is little propagation possible, and thus the probability that this constraint causes a hypothesis to be rejected is very low.

Neo [13] is more advanced than Morpheus. It uses similar operator specifications to Morpheus. After the search algorithm filled a certain hole, the following actions are taken:

1. **Propagate**: Any choices that can be inferred from the existing hypothesis are filled in. This is done with a SAT-solver, which simply checks for a certain hole if there is a single way of filling it.

2. **CheckConflict**: After propagation, the hypothesis is tested for any conflicts. This is quite similar to what Morpheus does, and also uses an SMT solver. However, if a conflict is detected, it is analysed to find the root cause of the conflict. This root cause is turned into a constraint that prevents further conflicts. For example, if the fact that the input and output of `reverse` have the same length causes a conflict, we already know that `sort` will also not work in that scenario.

One of the main strengths of Morpheus is that it improves the effectiveness of deduction with partial evaluation. Combined with its statistical language model, Morpheus is shown to be very effective in data transformation tasks. However, the specifications that are made Morpheus is specifically made for table transformation tasks, which makes it difficult to apply to other domains.

The propagate step of Neo is an optimization over Morpheus. The propagation prevents the search from trying multiple ways of filling the hole in a later iteration. Another big benefit is the ability to identify the root cause of a conflict since this provides insight into what can be done to prevent this kind of conflict later in the search. However, both algorithms suffer from the same restriction of having to manually provide specifications. Applying them to a different domain is therefore a tedious process.

### 3.1.2 Example propagation

Some program synthesizers use deduction to divide the problem of finding a suitable program into multiple sub-problems. A good example of such an algorithm is $\lambda^2$ [14]. It is able to divide a search into smaller parts by using inference rules to infer a new set of input-output examples for certain holes. In addition, it also checks for conflicts, similar to Neo and Morpheus. The inference rules either conclude unsatisfiability or translate the input-output examples. For example, consider a `map` function that takes an input list and a function $f$ and returns an output list. The input and output examples for synthesizing the function $f$ can be inferred from the input list and the output list of the `map` function. An inference rule defines how these examples can be inferred, or alternatively reject a hypothesis if the transformation is not possible using a `map` function. Solving the problem defined by the new set of examples is equivalent to filling the hole.

A more advanced example is DryadSynth [18]. In contrast to the aforementioned algorithms, this is not a programming-by-example synthesizer but rather a synthesizer that takes an (incomplete) specification as input. It relies on a divide-and-conquer approach to tackle the program synthesis problem. The algorithm starts with the input specification. Each time a new sub-problem is made, it is put through a deductive synthesizer that simplifies the problem as much as possible. This simplification is performed by exhaustively and repeatedly applying a set of predefined deductive rules, such as the following deduction rule for a language with conditionals:

$$f(\mathbf{e}) \geq e_1 \land f(\mathbf{e}) \geq e_2 \implies f(\mathbf{e}) \geq \texttt{if}(e_1 \geq e_2, e_1, e_2)$$

Here, `if` is the conditional if-then-else function. Note that $\texttt{if}(e_1 \geq e_2, e_1, e_2)$ is equivalent to the `max` function. This rule essentially describes that in case the output of a function should be greater than both inputs, a `max` function can be synthesized.

If the deductive synthesizer is not able to completely solve the problem, the problem is handed to the Divide-and-Conquer splitter. If a problem is indivisible, it is solved with an enumeration technique.

DryadSynth is a good example of a program synthesizer that heavily relies on deduction. It is able to completely solve some problems without using search. A downside to both $\lambda^2$ and DryadSynth is that the deduction rules have to be expertly designed in order to transform an input specification into a program. For example, if the direction of the implication in the example deduction rule of DryadSynth was reversed, it would have counteracted the efforts of the algorithm to find a program for the specification. Small mistakes in the inference rules for $\lambda^2$ would lead to the algorithm confidently returning the wrong programs since it filled holes with subprograms that are generated with the wrong examples.

### 3.1.3 Discussion

All mentioned program synthesizers have to be provided with semantic specifications or deduction rules. The quality of the specifications or rules can have a big effect on the effectiveness of deduction. Providing something that is incorrect can remove potential solutions, and in the case of DryadSynth, providing correct but unhelpful rules can counteract the program synthesis. Furthermore, manually crafting specifications or deduction rules can be quite tedious and difficult, especially for people that are not familiar with the specific operators. Automating this task can make these tools accessible to a larger audience. For the purposes of this research, the approach taken by Morpheus is the most attractive since the simple specifications that are used are easier to automatically discover.

## 3.2 Popper

Popper is an Inductive Logic Programming (ILP) system [8]. ILP can be seen as a specific form of program synthesis that restricts itself to logic programs [7]. Instead of working with input-output examples, logic programs work with positive and negative examples. The goal is to synthesize the most general set of rules such that all positive and no negative examples are a logical consequence of the synthesized rules.

Popper makes use of the inherent semantic knowledge of the ILP problem definition to constrain the search space after an incorrect hypothesis was tested, which means that it learns from failures. Learning from failures closely relates to what Neo and Morpheus do. However, the benefit of Popper is that it is not specific to a certain domain. The knowledge that is used for creating the constraints comes purely from the failure and the semantics of ILP, and not from the domain to which ILP is applied.

## 3.3 Specification extraction

Specification extraction is a topic from the programming languages field. Automatically extracting specifications from function definitions and proving them is a useful feature for many applications, such as testing, proof assistants and documentation generation. This topic is also relevant to this thesis since equivalence specifications can be used to detect symmetries in a program space.

### 3.3.1 QuickSpec

A well-known tool for specification extraction is QuickSpec [5]. QuickSpec takes a set of functions and variables with their types and a data generator for each type. The algorithm is simple and elegant:

1. A set of terms is generated out of the functions and variables.
2. Every term is put into the same equivalence class. Essentially, this means that every term is considered to be equivalent.
3. Tests are generated by using a data generator for each variable. If programs of the same equivalence class have a different outcome for a test, the equivalence class is split into different new classes in such a way that every term in a class has the same outcome for every test executed so far. The testing continues until there is no change in the equivalence classes for a certain number of tests.
4. Every term is considered equal to any other term in the same equivalence class. Pruning is applied to filter redundant equations from the output.

This approach works very well in toy examples, but it fails to scale to real-world scenarios. This problem is addressed in the second version of QuickSpec [34]. Here, pruning and symmetry-elimination techniques are used *during* the testing phase to reduce the testing of redundant terms.

One shortcoming of QuickSpec is that it never has complete certainty about an extracted lemma. Terms are considered equal only if they show equal behaviour in a sufficient number of tests. Unless the terms are tested on every possible input, their equivalence cannot be ensured with an approach based purely on testing. Therefore, the lemmas that are found by QuickSpec should be considered conjectures.

### 3.3.2 HipSpec

We can give those conjectures to a proof assistant to try to automatically prove them. This is exactly what HipSpec [6] does; it uses QuickSpec to generate the conjectures and hands them off to Hip, a Haskell Inductive Prover [32]. Hip translates the conjectures to first-order logic and proves them by utilizing an external automated prover, such as Z3 [9]. A big limitation of HipSpec is its exclusive compatibility with Haskell. However, it does offer specifications accompanied by proof of correctness, which is something that QuickSpec does not provide. Ideally, these two approaches should be combined to get a system with the flexibility of QuickSpec and the proofs from HipSpec. However, this is a challenging endeavour, since the flexibility of QuickSpec originates from utilizing evaluators instead of relying on operator definitions. This means that one could for example provide it with a Python evaluator. In contrast, HipSpec requires definitions for the operators in Python in a specific format, which is more difficult to provide.

# Chapter 4

# Problem definition

This thesis aims to make semantic information more accessible by automatically discovering it. The first goal is to discover symmetries in the program space and eliminate those in the search. The second goal is to discover semantic specifications that can be combined with problem-specific knowledge to prune the program space. To reach these goals, three problems have been formulated. The first problem is the program symmetry problem, which addresses the need for a way of instructing the enumeration algorithm to remove part of a symmetry. The second problem is called the constraint discovery problem and focuses on automatically discovering the symmetries and generating constraints to eliminate them. The specification discovery problem is the last problem and it addresses automatically discovering semantic specifications. The rest of this chapter is dedicated to motivating and formulating these three problems more in-depth.

## 4.1 Program symmetry problem

Enumerative program synthesis algorithms use an enumerator to enumerate every program in the program space. However, the program space might have a lot of symmetries. For example, consider the running example of the integer arithmetic program space. A possible symmetry in this program space is multiplication with one. The operation takes a program $a$ and multiplies it with one to obtain the program $a \times 1$. Even though $a \times 1$ is different from $a$, it still has the same meaning. Another operation could be to multiply the program 0 with any other expression. Since multiplication with zero always results in 0, the behaviour of the transformed programs will not change. The operation that swaps the operands of multiplication or addition also defines a symmetry that is caused by commutativity.

Symmetries in a search space are usually undesirable. This also applies to the program space. Depending on the language, symmetries can make up large portions of the program space. This will become visible in the evaluation of this approach in Chapter 7. These programs all have to be enumerated and evaluated with the intent specification, but they do not contribute anything to the expressiveness of the program space. Section 2.2 already showed that other domains utilize symmetry-breaking constraints for this issue. However, the idea of symmetry breaking has not been explored with enumerative program synthesis. The question that we aim to answer in this thesis is:

*How can symmetries in the program space be broken?*

This thesis proposes a custom constraint system specifically for enumerative program synthesis that can be used to create symmetry-breaking constraints.

## 4.2 Constraint discovery problem

In a lot of applications of program synthesis, there is a human user that has to directly interact with the program synthesizer. Therefore, it is crucial to invest time and effort in the user experience and ease of use.

An inherent consequence of making program synthesis algorithms generalize to arbitrary user-defined languages is that language-specific knowledge such as symmetry-breaking constraints needs to be supplied together with the language. Section 3.1 showed that existing algorithms that make use of language-specific information rely on the user or an expert to provide this information. Manually producing language-specific information in the correct format is often a tedious and error-prone task that requires a profound knowledge of the semantics of the language. However, for many of the applications of program synthesis, the target audience does not necessarily have this knowledge. Moreover, individuals that have the necessary semantic knowledge might be more efficient by coming up with the program themselves instead of going through the tedious process of encoding this information.

In the constraint generation problem, the aim is to automatically detect symmetries in the program space by using the evaluator for a language as an oracle. Once a symmetry has been discovered, a symmetry-breaking constraint can be generated to eliminate the symmetry from the program space. This relieves end users of having to define the constraints themselves, while still enjoying the benefits of symmetry-breaking constraints. There are a couple important factors to take into account when generating symmetry-breaking constraints.

1. The discovered symmetries and corresponding constraints should be correct; incorrect constraints will remove potential solutions without leaving an equivalent program in the program space.

2. The combination of different constraints should not lead to both sides of an equivalence being removed.

3. Since the goal of a program synthesis problem is often to find the smallest program that satisfies the intent specification, the more complex half of a symmetry should be removed in favour of the simpler half.

Even though it is difficult to directly assess the quality of the discovered constraints, it is essential to take these considerations into account, since they have a big influence on the efficiency and correctness of the enumeration algorithm using the symmetry-breaking constraints.

The problem of generating symmetry-breaking constraints can be split into discovering symmetries and converting them to constraints. This gives us the following two questions:

*How can symmetries be discovered?*
*How can these symmetries be converted to symmetry-breaking constraints?*

Combining the answers to these questions should give a functioning system that automatically discovers symmetry-breaking constraints.

## 4.3 Specification discovery problem

Until now, we have focussed on language-specific knowledge in the form of symmetry-breaking. However, existing program synthesis solvers exploit other variants of language-specific knowledge. Morpheus [12] and Neo [13] use the relation between certain properties of input and output values of operators in the language. These relations are an over-approximate incomplete semantic specification of operators in the language.

A semantic specification (partially) describes the behaviour of a language. Combined with problem-specific knowledge, semantic specifications can prune large parts of the program space. Morpheus and Neo rely on the user to come up with these specifications. As motivated in Section 4.2, manually providing language-specific knowledge is cumbersome and requires a deep insight into the semantics of a language. Automating this step would make synthesizing with custom languages or libraries a lot more accessible to the average user. Therefore, the last question that needs answering is:

*How can semantic specifications be discovered?*

The next chapter will present the solutions to the questions that are posed in this chapter.

# Chapter 5

# A framework for discovering constraints and specifications

This section describes the way in which constraints and specifications are generated. Section 5.1 demonstrates how equivalences can be discovered in the program space. This is followed by an explanation of how these equivalences, which can be seen as symmetries, are converted into symmetry-breaking constraints in Section 5.2. The implementation of the constraints themselves is discussed in Section 5.3, as well as their application to a top-down enumeration algorithm. Finally, Section 5.4 shows how the equivalence discovery approach can be modified to discover semantic specifications.

## 5.1 Equivalence discovery

The structure of the equivalence discovery algorithm is based on QuickSpec [5]. QuickSpec generates equivalences for Haskell and Erlang programs in a few steps. First, a set of terms is generated and grouped inside a single equivalence class. This equivalence class is divided into smaller classes by repeatedly testing these terms on different inputs and splitting them according to their output. Once the testing has finished, the equivalence classes are converted to a list of equivalences. Finally, a pruning step removes equivalences that are implied by other equivalences and are thus redundant. Section 3.3.1 explains the algorithm in more detail.

To make the QuickSpec algorithm work for equivalence discovery for program synthesis, some modifications had to be made:

1. The equivalence discovery is made more general: it works for any grammar that a user can define. The only prerequisite is that the functions in the grammar must be *pure*, in other words, deterministic and without side effects. The algorithm also needs access to an evaluator for the language.

2. In the QuickSpec algorithm, a data generator has to be provided for

every type that is used in the grammar. The algorithm described in Section 5.1.2 is able to automatically create such a generator from the grammar, alleviating the user of this task.

3. The equivalence pruning step is replaced by a procedure tailored to creating symmetry-breaking constraints. QuickSpec prunes equivalences that can be derived by the remaining equivalences. In contrast, the pruning procedure in this algorithm is separated from the equivalence discovery, and executed after equivalences have been converted to constraints for the first time. The pruning prunes equivalences using the generated constraints and converts the remaining equivalences into a pruned set of constraints. Since this pruning step is no longer part of discovering equivalences, it is not covered in this section, but rather in Section 5.2.3.

The purpose of the equivalence discovery is to detect equivalences between certain programs in the program space. These equivalences can be seen as symmetries. The remainder of this section will explain the design of the equivalence discovery algorithm in more detail.

### 5.1.1 Grammar preparation

The grammar preparation step prepares a grammar for generating the terms between which equivalences will be detected. These terms could be generated directly from the language grammar, but this limits the generality of the discovered equivalences. To illustrate this, consider the integer arithmetic grammar again. Terms such as $1 + 2$ and $2 + 1$ will be generated, which causes the equivalence $2 + 1 \equiv 1 + 2$ to be generated at a later stage. Ideally, we would also discover the much more general equivalence of $a + b \equiv b + a$, showing the commutativity of the addition operator.

Terms such as $a + b$ or $b + a$ are called *patterns* because they contain variables that can match multiple sub-terms. These variables are different from the input variables already present in the language. An input variable is assigned values from input examples, or a value that a user provides once a program is being used. In the context of equivalence discovery, a pattern variable represents all sub-terms that are applicable to its location.

A pattern consists of reference nodes and variable nodes. A *reference node* references a specific production rule in the grammar. This node only matches this specific production rule in the program. For example, a reference node that references $+$ with two child nodes 1 and 2 only matches $1+2$ in the program. A *variable node* signifies a variable and contains an identifier symbol. Such a variable can match any production rule in the grammar. An example of this is the pattern $a + b$, which matches every addition, e.g. $2 + 3$ where $a = 2$ and $b = 3$. A single pattern tree can also have multiple instances of the same variable. In this case, all these instances must have the same assignment. For example, the pattern $a + a$ matches every addition of equal terms, such as $1 + 1$, where $a = 1$ and $(2 \times 4) + (2 \times 4)$ where $a = 2 \times 4$. To avoid confusion between pattern variables and input variables in the synthesized programs, we will use $a, b, c, ...$ for pattern variables and $x$, $y$ and $z$ for program variables in the remainder of this thesis.

$$
\begin{aligned}
\text{Int} &::= \text{Int} + \text{Int}\\
\text{Int} &::= \text{Int} \times \text{Int}\\
\text{Int} &::= \text{Int} - \text{Int}\\
\text{Int} &::= 0\\
&\vdots\\
\text{Int} &::= 9\\
\text{Int} &::= x
\end{aligned}
$$

(a) **Integer arithmetic grammar**

$$
\begin{aligned}
\text{Int} &::= \text{Int} + \text{Int}\\
\text{Int} &::= \text{Int} \times \text{Int}\\
\text{Int} &::= \text{Int} - \text{Int}\\
\text{Int} &::= 0\\
&\vdots\\
\text{Int} &::= 9\\
\text{Int} &::= a\\
\text{Int} &::= b
\end{aligned}
$$

(b) **Pattern grammar**

Figure 5.1: **The conversion of the integer arithmetic grammar to the pattern grammar for discovering constraints in the integer arithmetic grammar.**

Taking this into account, there are two changes that need to be made. Firstly, a user-defined number of pattern variables needs to be added for each non-terminal in the grammar. Secondly, all existing input variable production rules need to be disabled. These variables are not relevant to the patterns and since the equivalence extraction step does not always have access to problem-specific data such as input values, it is not possible to evaluate these variables. For example, in the integer arithmetic grammar from Figure 5.1a, the production rule Int $\rightarrow x$ would be disabled, and the production rules Int $\rightarrow a$, Int $\rightarrow b$, etc. would be added. Theoretically, the rule Int $\rightarrow x$ could be reused as a pattern variable, but this would be confusing. This results in the grammar given in Figure 5.1b. Theoretically, the rule Int $\rightarrow x$ could also be reused as a pattern variable. However, this would be very confusing, which is why it is removed instead.

The choice of the number of pattern variables can have a significant effect on the constraints that are generated. Pattern variables allow constraints to be more general. Including more pattern variables can make the equivalences and thus the constraints more general. Consequently, these general constraints often make multiple less-general constraints redundant, therefore reducing the total number of constraints. A good rule of thumb for setting the number of pattern variables per nonterminal is to make sure that every production rule can be filled with unique variables. For example, if there would be a production rule Int ::= Bool ? Int : Int, which contains the ternary operator, then at least two variables should be included for Int and at least one for Bool.

```
1: procedure GetAutoGenerator(Grammar $\langle V, \Sigma, R, S\rangle$, type $T$, evaluator $E$, max depth)
2:     Remove all variable production rules from $R$
3:     $H \leftarrow$ Enumerate($\langle V, \Sigma, R, T\rangle$, [], max depth, $\infty$)
4:     $O \leftarrow \{E(\langle V, \Sigma, R, S\rangle, x), x \in H\}$        ▷ Set of evaluated hypotheses
5:     return a function that randomly draws an item from $O$
6: end procedure
```

Algorithm 1: **Auto-generator creation**

### 5.1.2 Data generators

In QuickSpec [5], the user has to provide data generators for every type of the grammar. These data generators are used to generate values that will be assigned to the pattern variables. Defining data generators can be quite tedious and we should not make this an obligation for the user. In our implementation, it is not necessary to manually define data generators, even though it is still possible. When data generators are not provided, the algorithm creates an *auto-generator*. The auto-generator is created from the original grammar. Algorithm 1 outlines how such a generator is created. The first step is to remove all variable production rules from the grammar (line 2). The variable rules cannot be used because in order to evaluate them, there needs to be an assignment, which is not available by default. This modified grammar is then used to enumerate all possible hypotheses without variables up to a certain maximum depth in line 3. These hypotheses are then evaluated in line 4 to get their output values. The output values are stored in a set, meaning that duplicate values are removed. This set of output values is used to create the generator in line 5. The generator that is returned is a function that draws a random value from the set of values each time it is called.

An obvious downside of using auto-generators is that they only generate values that can be created using the grammar. If these values are not general enough, incorrect equivalences could be generated. This can become a problem at a later stage if input variables introduce values that are substantially different from the values in the grammar. To illustrate this problem, consider the lists grammar again. Since the grammar only has integers up to 9, and no arithmetic operators, the largest integer value that can be constructed is 9. As a result, the equivalence $push(sort(a), 9) \equiv sort(push(a, 9))$ is generated at a later stage in the equivalence discovery. This equivalence is valid as long as the input variables in the original grammar do not introduce values greater than 9. However, there might be cases where the data that can be generated by the grammar is not general enough for the possible assignments to the input variables.

The auto-generator has therefore been extended to also accept a list of possible values for certain input variables. These lists can be hand-made or taken from existing data sets. If such a list is provided for one or more of the input variables, these variables are not deleted from the grammar beforehand. Instead, whenever they have to be evaluated, a random value is sampled from this list. This allows users to also insert dataset-specific data into the generators

```
 1: **Input:** list of patterns $P$, evaluator $E$
 2: **Output:** Set of equivalence classes
 3: $Q \leftarrow \{P\}$
 4: **repeat**
 5:     $Q' \leftarrow \emptyset$
 6:     $T \leftarrow$ batch of $n$ test cases generated using $D$
 7:     **for** $q \in Q$ **do**
 8:         $O \leftarrow$ empty dict
 9:         **for** $p \in q$ **do**
10:             outputs $\leftarrow$ outputs of tests $T$ on $p$
11:             push $p$ to $O$[outputs]
12:         **end for**
13:         **for** $(\text{outputs}, P) \in O$ **do**
14:             Add $P$ as a new equivalence class to $Q'$
15:         **end for**
16:     **end for**
17:     $Q \leftarrow Q'$
18:     Remove equivalence classes with a size smaller than 2 from $Q$
19: **until** there has not been a split in the last $m$ iterations
```

Algorithm 2: **Equivalence extraction**

without having to define their own data generator.

### 5.1.3 Discovering equivalence classes

The next step after preparing the grammar and getting the data generators is to generate equivalence classes. An equivalence class is a set of hypotheses that are assumed to be extensionally equivalent. Algorithm 2 shows the procedure for extracting equivalences. The first step in obtaining equivalences is obtaining the set of patterns $P$ between which equivalences should be detected. This set can be constructed by utilizing a top-down enumeration algorithm (see Section 5.3.1). After that, every pattern is placed in the same equivalence class in line 3. This essentially means that we assume every pattern to be extensionally equivalent. Of course, this is not actually the case, and the goal of the remainder of this procedure is to falsify most of these assumptions by creating counter-examples.

This falsification starts by generating a batch of test cases in line 6. A test case consists of an assignment to each pattern variable. These assignments are generated using the data generators, that are either provided by the user or automatically made using Algorithm 1. The batch of test cases is used in line 10 to evaluate the patterns in each equivalence class. Since there are only pattern variables in a generated pattern, filling in those variables returns a program without variables that can directly be evaluated. The patterns are then grouped in line 11 based on their outcomes on the batch of test cases. Lines 13-15 turn these groups into new equivalence classes. After each pattern in each equivalence class has been evaluated on the batch of tests, any equivalence class that has only one element is removed (line 18). The goal is to detect equivalences between patterns, and if there is only one pattern in an equivalence class, it is not possible to generate an equivalence. The process of generating a batch of tests and splitting the equivalence classes is repeated until there have not been any splits in a user-defined number of iterations. Of course, there is never a point of absolute certainty about an equivalence. However, the confidence goes up with the evaluated number of tests.

### 5.1.4 From equivalence classes to equivalences

After there have been enough iterations that we are confident enough about the equivalence classes, it is time to turn the equivalence classes into equivalences of the form $A \equiv B$. The first step is to pick the least complex pattern in the equivalence class. This pattern represents the programs in the equivalence that will remain in the search space. The other patterns represent equivalent programs that can be removed. The goal of the search algorithm is generally to find the smallest program that satisfies the examples. Consequently, we want to prevent removing the smallest program from an equivalence class as much as possible. The least complex node in the order of complexity is therefore defined as follows:

1. It has the lowest number of nodes. This is important because this will cause the larger program to be removed from the program space. If the numbers of nodes are equal:
2. It has the lowest number of pattern variables. This is also essential since variables may represent a large part of the program in the program space, whereas the size of a literal is always equal to one. If the numbers of pattern variables are equal:
3. Any total order. If this step were to be omitted, the transitive property of the order would not be guaranteed and equivalences could prune each other in the pruning step.

Once the least complex pattern has been established, it gets combined with each other pattern in the equivalence class to form an equivalence. For instance, if the equivalence class has the patterns 0, $0 \times 1$, $0 \times a$, $a - a$, the pattern 0 gets chosen as the least complex pattern because it has the lowest maximum depth, and $0 \times 1 \equiv 0$, $0 \times a \equiv 0$ and $a - a \equiv 0$ are the equivalences that get generated. $0 \times 1$, $0 \times a$ and $a - a$ are the patterns that will be removed from the search space in favour of 0.

## 5.2 Constraint conversion

The constraint conversion step converts an equivalence to an actual constraint that can be used in the search. In most cases, converting an equivalence is quite straightforward. There already is an equivalence in the form of $X \equiv Y$, and it is known that $X$ is the least complex pattern in the equivalence class that generated this equivalence. For each equivalence, an attempt is made to convert it into a `Forbid` constraint. This constraint forbids a certain pattern from occurring in the program space, which is useful for removing symmetries such as multiplication with 0. In case this fails, the algorithm tries to create an `Order` constraint. The `Order` constraint enforces an order over variables. This can eliminate symmetries caused by the commutativity of operators. An important thing to keep in mind is that this constraint conversion step is not complete: there will be some equivalences that are not turned into constraints. However, the generated constraints are sound, meaning that they do not remove potential solutions from the program space without leaving an equivalent program.

### 5.2.1 Generating `Forbid` constraints

A `Forbid` constraint removes every occurrence of a pattern from the search space. In most cases, this is all that is necessary, but there is a complication in the case where $X$ and $Y$ are equal by variable renaming. To illustrate this problem, consider the commutativity equivalence again: $a + b \equiv b + a$. The pattern $a + b$ is able to match every addition. Adding this pattern in a `Forbid` constraint thus completely removes addition from the search space. In the case of this equivalence, it is rather obvious that a `Forbid` constraint does not work correctly. There are also equivalences where this is less obvious because it only wrongly removes a single program. For example, consider the equivalence $a + 6 \equiv 6 + a$. If the left-hand side of this equivalence is turned into a `Forbid` constraint, the program $6 + 6$ is removed from the search space, even though it is its own 'equivalent program'.

The underlying issue in these scenarios is that there are programs that match both the pattern on the right-hand side of the equivalence and the pattern on the left-hand side of the equivalence. In order to turn an equivalence into a `Forbid` constraint without accidentally removing part of the search space, we need to make sure that such a program does not exist for the equivalence. Determining the absence of such a program is not a problem that can easily be brute-forced. This problem is known as the *unification problem* [22].

Programs that match both patterns of an equivalence can be found by employing a unification algorithm. Such an algorithm finds an assignment to the variables in both patterns that make both patterns equal, which is equivalent to finding the program that matches both patterns. The following examples illustrate this process a bit more intuitively:

- Consider the equivalence $a + 6 \equiv 6 + a$. First, the left-hand side and the right-hand side are separated and the variables are renamed for clarity:

  $$a_l + 6 \qquad 6 + a_r$$

  These two equations can then match each other by assigning $a_l = 6$ and $a_r = 6$. This means that $6 + 6$ matches both sides and is not included in the search space anymore when this equivalence would have been turned into a `Forbid` constraint.

- For an example that can be turned into a `Forbid` constraint, consider $a + a \equiv a \times 2$. Again, the sides are separated and the variables are renamed:

  $$a_l \times 2 \qquad a_r + a_r$$

Here it is not possible to assign values to $a_l$ and $a_r$ to make the sides match. Therefore, there does not exist a program that matches both sides. The equivalence can hence safely be turned into a `Forbid` constraint.

As demonstrated, if the match attempt ends up being successful, there must be an instance of the right-hand side on which the left-hand side will match. Therefore, the equivalence should not be turned into a constraint since it might remove a program and all its equivalences from the search space.

This detection mechanism gives no false positives, meaning that it does not allow constraints to be produced that remove parts of the search space. However, there are some false negatives. For example, the equivalence for associativity of addition $(a + b) + c \equiv a + (b + c)$ is not turned into a `Forbid` constraint, even though it could be.

### 5.2.2 Generating `Order` constraints

An `Order` constraint can be created from a pattern and a list of variables in the pattern to which the constraint should apply. The `Order` constraint can enforce an arbitrary order between two variables in a pattern. If an equivalence cannot be turned into a `Forbid` constraint, an attempt is made to turn it into an `Order` constraint. The only prerequisite is that the left-hand side and the right-hand side of the equivalence must only be different by swapping two variables. So, the equivalence $(a + b) + c \equiv (b + a) + c$ will be turned into an `Order` constraint where $a$ and $b$ are ordered. The equivalence $(a + b) + c \equiv (c + a) + b$ will not be turned into an `Order` constraint, because more than two variables are swapped. This equivalence still contains a useful equivalence that we would like to make use of during the search. However, all the information can also be captured by other equivalences where only two variables are swapped. In other words, we can infer $(a + b) + c \equiv (c + a) + b$ from the equivalence $(a + b) + c \equiv (b + a) + c$ and $(a + b) + c \equiv (a + c) + b$. Therefore, we can discard the equivalences where more than two variables change place and only include three constraints:

1. `Order` $a$ `and` $b$ `in` $(a + b) + c$
2. `Order` $b$ `and` $c$ `in` $(a + b) + c$
3. `Order` $a$ `and` $c$ `in` $(a + b) + c$

This combination of constraints rules out any permutation of $a$, $b$ and $c$ except for the one that is ordered. There might be some exceptional cases in which these other equivalences are not generated because they are falsified during the falsification stage. However, in these cases, the original equivalence cannot be turned into an `Order` constraint without reducing the expressiveness of the program space, and therefore it should also be disregarded.

A downside to using three different constraints is that it causes the pattern $(a + b) + c$ to be checked multiple times in the search since each constraint is enforced separately. This problem is solved by including an extra step after the constraint generation that combines `Order` constraints. Every `Order` constraint that we have so far describes the equivalence of swapping two variables. The constraint removes one part of this equivalence by enforcing the variable assignments to be ordered. What is very helpful in this case is that the operation of ordering variables is transitive. Furthermore, how the variables are actually ordered is irrelevant. Therefore, using the transitive property of ordering, it is possible to combine `Order` constraints with the same pattern by ordering the union of the variables that the individual constraints ordered. Consider the constraints that order $[a, b]$, $[b, c]$ and $[a, c]$ again. By combining these constraints, a more general constraint is obtained that orders $a$, $b$ and $c$ in the pattern $(a + b) + c$.

### 5.2.3 Removing redundant constraints

The final step in the process of obtaining constraints is to prune any unnecessary or redundant constraints. Enforcing a constraint is a relatively expensive operation that happens for every constraint with every expansion in the search tree. Often, constraints can be removed in favour of a more general version that has also been discovered. For example, the programs that the `Forbid` constraint for $1 \times a$ removes is a superset of the programs that are removed with the `Forbid` constraint for $1 \times 2$. Therefore, the pattern $1 \times 2$ can be removed if $1 \times a$ is also discovered. This is a result of matching being a transitive operation; if a pattern $a$ is able to match another pattern $b$, $a$ also matches all patterns and programs that $b$ matches.

The pruning is applied to the equivalence classes, where redundant patterns are removed. This is done by viewing a pattern as a program and enforcing the generated constraints on it. Each equivalence is used separately to prune all remaining patterns in the equivalence classes. An equivalence is first converted to a constraint using the method described earlier in this section. After obtaining the constraint, it is applied to every pattern in every equivalence class. If the constraint can prune the pattern, the pattern is removed. There are some extra checks to make sure that a constraint does not remove the pattern on either side of the equivalence that generated the constraint. If these checks were to be omitted, every constraint would remove itself. At the end of the pruning step, the pruned equivalence classes are again converted to equivalences and constraints, and this pruned set of constraints is finally returned by the constraint discovery algorithm.

The essence of this pruning step is to retroactively apply the discovered constraints in the enumerator that enumerated the patterns before testing them. This means that equivalences between the patterns are detected and removed.

### 5.2.4 Duplicating `Forbid` constraints

One issue with applying the discovered constraints in the pattern enumerator is that sometimes the `Order` constraints prune patterns that generated valid `Forbid` constraints. To illustrate this, consider the following constraints being generated before pruning: `Forbid` $a \times 1$, `Forbid` $1 \times a$ and `Order` $a$ `and` $b$ `in` $a \times b$. The `Order` constraint will prune the `Forbid` $a \times 1$ constraint during the pruning phase since 0 is ordered before the pattern variable $a$ in the integer arithmetic grammar. However, $a$ is a pattern variable that can be filled in in multiple ways during the search. The assignment to this pattern variable might be ordered differently. Consequently, there could be programs such as $0 \times 1$ that are redundant, but no longer get removed because 0 and 1 are ordered, and the only `Forbid` constraint left is removing programs that match the pattern $1 \times a$. This is an inherent consequence of treating patterns as programs.

This issue only occurs when there are symmetric `Forbid` constraints, in this case, `Forbid` $a \times 1$ and `Forbid` $1 \times a$. If one of these patterns is kept during the pruning, the equivalent pattern needs to be added back. The simple solution is, therefore, to check each `Forbid` constraint for a symmetry that gets pruned by an `Order` constraint. If such a symmetry exists, we simply produce the other half of the symmetry for the pattern and turn it back into a `Forbid` constraint, while making sure that no duplicate constraints are generated.

After this step, we are left with a set of symmetry-breaking constraints that can be given to a program enumerator to break symmetries and reduce the number of redundant enumerations.

The diagram in Figure 5.2 provides an overview of the most important steps that are necessary to generate constraints. Section 5.1 covered the top half of this diagram, showing how equivalences can be discovered by testing candidate patterns. These candidate patterns were enumerated from a pattern grammar, which is obtained by modifying the original program grammar.

This section discussed how these equivalences, which represent symmetries in the program space, can be converted to symmetry-breaking constraints. First, constraints are generated from the equivalences. These constraints are then used to prune the equivalences. Finally, this pruned set of equivalences is again converted to obtain a pruned set of constraints. The next section will show how these constraints are implemented and how they can be enforced in an enumerative search algorithm.

## 5.3 Implementing constraints

Constraints are a means of limiting the search space. They can be used to rule out any undesirable program from being generated by an enumerator. This section shows how constraints fit into the context of an enumeration algorithm and how they are enforced. We conclude by showing how the overhead of constraint propagation can be limited by introducing global and local constraints.

### 5.3.1 Enumeration with constraints

Constraints are used in the top-down enumeration algorithm to limit the hypotheses that are enumerated. Algorithm 3 shows a simplified top-down enumeration algorithm. The input to the enumeration algorithm is the search space, defined by the grammar consisting of the nonterminals $V$, terminals $\Sigma$, production rules $R$ and start symbol $S$. Next to that, it also gets a list of constraints

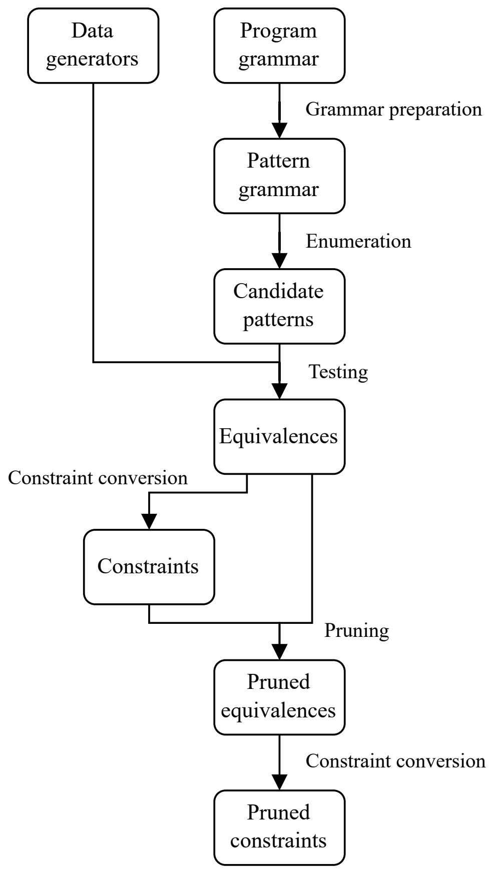


Figure 5.2: **Diagram showing the structure of the discovery process for symmetry-breaking constraints.**

that should be enforced in the enumeration procedure. The algorithm maintains a priority queue of possibly partial programs. In every iteration, the program with the highest priority is taken from this queue. If this program does not have any holes, it is returned by the enumerator, after which it can be evaluated on the intent specification. If this program is partial, one of the holes has to be expanded. To illustrate this, consider again the top-down enumeration from Section 2.3. The AST of the current program is ?+?. The `FindHole` procedure does a simple depth-first search of the AST to find a hole, which means that it finds the left-most hole. Once a hole has been selected, the grammar is consulted to obtain the set of production rules that can be used to fill the hole. This set of production rules is the *domain* of a hole. In the running example, the domain consists of every rule in the grammar, so the operators $+$, $-$ and $\times$, the numbers $0-9$ and the variable $x$.

The grammar does not take the constraints into account, so this set of production rules still needs to be pruned. Every constraint has an `Enforce`-procedure, which can take a set of production rules together with the entire tree and the position of the hole. This `Enforce`-procedure checks which production rules

```
 1: procedure ENUMERATE( grammar $\langle V, \Sigma, R, S \rangle$, constraints $C$, max depth,
    max size)
 2:     $Q \leftarrow$PriorityQueue((Hole($S$), PRIORITY(Hole($S$))))
 3:     $O \leftarrow$ []
 4:     while $Q$ is not empty do
 5:         $f \leftarrow$ tree with highest priority in $Q$
 6:         if $f$ does not contain a hole then
 7:             Append $f$ to $O$
 8:         else
 9:             $h \leftarrow$ FINDHOLE($f$)
10:             $t \leftarrow$ type of $h$
11:             $R' \leftarrow$ all rules of type $t$ from $R$
12:             for all $c \in C$ do
13:                 $R' \leftarrow$ ENFORCE($c, f, h, R'$)
14:             end for
15:             for all $r \in R'$ do
16:                 $f' \leftarrow$ replace $h$ in $f$ with $r$
17:                 if DEPTH($f'$) $\leq$ max depth and SIZE($f'$) $\leq$ max size then
18:                     Enqueue $f'$ with PRIORITY($f'$) in $Q$.
19:                 end if
20:             end for
21:         end if
22:     end while
23:     return $O$
24: end procedure
```

Algorithm 3: **Top-down enumeration**

would make the partial program violate the constraint and removes those rules. After enforcing each constraint, the only production rules that are left are the ones that do not violate any constraint when they replace the hole. If there would be constraints for removing addition with zero, then 0 would be removed from the domain in the example.

The final step is to make a copy of the partial tree for each production rule that is left and replace the hole in each copy with the corresponding production. For example, ?+? gets expanded to trees like 1+?, (?+?)+?, (?−?)+?, as can be seen in Figure 2.4b. Each copy can then be enqueued in the priority queue to get expanded again at a later point. The position in the queue is determined by the `Priority` function. In our implementation, the goal is to find the smallest program that satisfies the intent specification. Therefore, the priority function, uses the size of the program.

### 5.3.2 `Forbid` constraint

The `Forbid` constraint has a particular pattern that is not allowed in the search space. When a `Forbid` constraint is enforced at Line 13 in Algorithm 3, every rule in the domain that would complete the pattern is removed. A simple example of a pattern is $3 + 2$. This would remove every program that contains

$3 + 2$, such as $1 + (3 + 2)$, from the search space. It is also possible to use the variables introduced in Section 5.1.1. For example, multiplication with zero can be removed using the pattern $a \times 0$ and $0 \times a$, and addition of equal terms can be removed using the pattern $a + a$.

If a regular unification algorithm were to be used, we would have to first create a new tree for every rule in the domain and then match the pattern on this tree. This means that there is a linear relation between the number of production rules for a certain nonterminal and the number of match attempts. This adds a lot of overhead, especially for grammars with a large number of production rules and a small number of nonterminals. Therefore, a custom algorithm was developed that only has to do a single match attempt and can remove any production rule from the domain that violates the pattern. This algorithm traverses the tree and prunes the domain of a hole the moment it encounters it. If the attempt to match was successful, then this pruned domain is kept, otherwise, it is reset to the original domain.

### 5.3.3 `Order` constraint

Even though the `Forbid` constraint is quite flexible, it is still limited in what it can do. If we want to eliminate a symmetry from the search space that is caused by commutativity (e.g. $a+b \equiv b+a$), we need a different kind of constraint since the pattern $a + b$ matches every addition. The `Order` constraint can be applied in this scenario. For the `Order` constraint to work, an order must be defined over ASTs. The specific order is not important, as long as it is consistent and transitive. In our implementation, programs are ordered based on the index of the grammar rule defining the root node of the AST. If two ASTs have the same root node, we simply do a depth-first traversal of both trees to find a node that differs and compare this. This essentially means that there is a lexicographical ordering of the nodes in preorder. The only reason for choosing specifically this method is efficiency.

The `Order` constraint consists of a pattern with at least two variables, e.g. $a + b$. Furthermore, it also has a list with a minimum length of two, containing a subset of the variables in the pattern, defining how they should be ordered. For example, in the previously mentioned pattern, this list could be $[a, b]$. This constraint checks for every pattern match whether the assignments to the variables ($a$ and $b$) are ordered, so it checks $a \leq b$.

### 5.3.4 Constraint resolution

A match attempt can result in either a match, no match or an inconclusive result:

1. **Match**: The match attempt was successful. The assignments for the variable nodes are also returned.

2. **Inconclusive**: It is not yet possible to know if the pattern matches, because holes in the partial program have been encountered. The pattern might match if the holes are filled in.

3. **No match**: The pattern does not match, and there is no way to fill in the holes to make it match.

### 5.3.5 Global & local constraints

Every time a constraint containing a pattern is checked, a pattern match attempt must be executed at multiple points in the tree. However, the previous section showed that it is possible to know when a pattern will not match anymore. Using this information, it is possible to reduce the number of match attempts.

This is achieved by using local and global variants of the `Order` and `Forbid` constraints. *Local constraints* are associated with a specific partial program and also include a reference to a location within the AST of the program. The enforcement of this constraint is limited to this designated location in the specific AST. When enforcing the constraint, an attempt is made to align the root of the pattern with the node in the AST with which the AST is associated. There can be multiple local constraints for a single program. When a hole in the open program is expanded, it is not necessary to check every constraint. Since only the subtrees containing the hole are changing, we only have to check the constraints that reference a node on the path from the root to the hole that is being expanded. As mentioned in Section 5.3.4, the result of a pattern match is one of three options. With a local constraint, we can exploit this option. Since a constraint only has an effect if the output of the pattern match is successful, we can remove any local constraint for which we get 'No match' as an output. This also saves a lot of time, as pattern matching is an expensive operation that is done for every constraint at every expansion.

*Global constraints* are enforced in every tree and at every location in this tree. Just like with local constraints, it is not necessary to check the pattern of a global constraint on nodes that are not part of the path from the root to the hole we are expanding since nothing has changed in these subtrees. Still, doing a pattern match for every node on the path from the root to the hole that is being expanded adds considerable overhead. The global constraints that are used in this thesis, therefore, add a local variant of their constraint to the partial program, with a reference to the node that is being expanded. This is done at every expansion, so a local constraint that enforces the same property as the global constraint is created for every node. This has the same effect as enforcing the global constraint. However, we now have the additional feature of local constraints being deleted after it is known that they will no longer be of use.

## 5.4 Specification discovery

There is more useful information that can be extracted from the grammar. In addition to symmetry-breaking constraints, it is also possible to create specifications for operators in the grammar. These specifications can be utilized by a tool like MORPHEUS [12] to prevent programs that are infeasible for the problem to be generated at an early stage in the search, thereby improving the efficiency of the search. It is worth noting that the specifications do not have to be complete or precise, but they must be correct. MORPHEUS is specifically made for transforming data frames using functions from the DPLYR and TIDYR R libraries. Data frames can be seen as tables equipped with some extra features that are not relevant to this explanation. As Chapter 7 will evaluate this method with MORPHEUS, it will also serve as a running example in this section.

A specification creates a relationship between certain properties of input values and properties of the output value of a function. Since MORPHEUS works with data frames, the `row` and `col` property are used, representing the number of rows and columns respectively. Below you can find an example of the hand-made first-order specifications for the `filter` function in MORPHEUS:

$$T_{out}.row < T_{in}.row$$
$$T_{out}.col = T_{in}.col$$

These specifications state that the output of a `filter` function on a data frame always has the same number of columns and a lower number of rows than the input. Section 7.3.1 will discuss the correctness of this specification in more detail.

Two noteworthy observations can be made from these specifications. First of all, specifications only consider properties of a single operator, which negates the need for having an enumerator generate multiple combinations of operators. Secondly, the specifications are not limited to equivalences. This introduces a challenge since the algorithm from Section 5.1.3 can only discover equivalences. To address this, a new approach is adopted: instead of generating patterns and discovering equivalences between them, possible specifications are generated and the correct ones are identified in a testing phase. The rest of this section will delve into the details of the realisation of this approach.

### 5.4.1 Specification grammar

The first step is to generate the specifications, similar to how patterns were generated. The specifications can be generated separately for each operator in the program grammar. Therefore, it is not necessary to make use of the program grammar in this step. A user can instead provide a specification grammar, which defines the kind of specifications that can be found. In the case of `Morpheus`, the specifications need to be in first-order logic since they will be given to an SMT solver. Therefore, the following grammar is used for the specifications of operators operating on tables or data frames.

$$
\begin{aligned}
\mathit{Bool} &::= \mathit{InputInt} \leq \mathit{OutputInt} \\
\mathit{Bool} &::= \mathit{InputInt} < \mathit{OutputInt} \\
\mathit{Bool} &::= \mathit{InputInt} > \mathit{OutputInt} \\
\mathit{Bool} &::= \mathit{InputInt} \geq \mathit{OutputInt} \\
\mathit{Bool} &::= \mathit{InputInt} = \mathit{OutputInt} \\
\mathit{InputInt} &::= \mathit{InputDF.row} \\
\mathit{InputInt} &::= \mathit{InputDF.col} \\
\mathit{InputInt} &::= \mathit{InputInt} - \mathit{InputInt} \\
\mathit{InputInt} &::= \mathit{InputInt} + \mathit{InputInt} \\
\mathit{InputInt} &::= 1...9 \\
\mathit{OutputInt} &::= \mathit{OutputDF.row} \\
\mathit{OutputInt} &::= \mathit{OutputDF.col}
\end{aligned}
$$

The first few rules in the grammar describe how properties of the input and the output can be related to each other, which is done in this case with the four different comparison operators and the equality operator. This is followed by two rules that define which properties should be compared for the input data frames, which are the `row` and `col` properties in our example. There are also rules for addition, subtraction and literals, which allow for more complex specifications to be discovered. The last two rules of the grammar define that the `row` and `col` properties should also be used for the output data frames.

Depending on the operator that is being evaluated, some variables for the input- and output values of this operator need to be added. The `filter` function takes a data frame and a predicate function for filtering. It returns a data frame in which the rows are filtered by the predicate function. Therefore, three input variables are automatically added by the procedure:

$$
\begin{aligned}
\mathit{InputDF} &::= \mathit{FilterInputDF}_1 \\
\mathit{InputPredicate} &::= \mathit{FilterInputPredicate}_1 \\
\mathit{OutputDF} &::= \mathit{FilterOutput}
\end{aligned}
$$

For the first two input variables, data generators have to be provided. The last input variable represents the output of the `filter` function and does not need a data generator.

Once this step is done, hypothesis specifications such as $\mathit{FilterInputDF}_1.\mathit{col} + 1 = \mathit{FilterOutput.row}$ or $\mathit{FilterInputDF}_1.\mathit{col} = \mathit{FilterOutputDF}_1.\mathit{col}$ can be generated.

### 5.4.2 Specification testing

Just like in the constraint discovery algorithm, a test is an assignment to the input variables. For the variables that correspond to the input values of the operator that is being evaluated, the assignment is taken from the data generators. For the variable corresponding to the output value, the operator (`filter` in this example) is evaluated on the input values to obtain the filtered list as an output value.

With these value assignments, each specification can be evaluated. Specifications that evaluate to false can immediately be discarded; they do not hold. In theory, only the correct constraints should remain after enough iterations. Of course, this has the same limitations as the constraint discovery: if the input data is not general enough, there is a probability that incorrect constraints are generated.

### 5.4.3 Pruning

At this point, there is a set of specifications that are correct with a very high probability. However, there is still a lot of redundancy in the specifications. Some specifications are implied by other specifications, which becomes apparent in the following example for the `filter` function:

$$\begin{aligned} FilterInputDF_1.row &\geq FilterOutput.row \\ FilterInputDF_1.row + 1 &\geq FilterOutput.row \\ FilterInputDF_1.row + 1 &> FilterOutput.row \\ FilterInputDF_1.row + 2 &\geq FilterOutput.row \\ &\vdots \end{aligned}$$

The topmost specification is the one that is the most specific and thus contains the most information. All other specifications are weaker versions and can be implied by the first specification. These implied specifications need to be pruned since they will unnecessarily slow down the deduction inside `Morpheus`. This pruning is achieved by passing the specifications to a symbolic reasoning library called SymPy [37]. SymPy is able to simplify the set of specifications and remove every specification that is implied by (a combination of) other specifications. In the example above, only the first specification is kept.

After pruning, the specifications are ready to be used. In our use case, they need to be rewritten in the format that is accepted by Morpheus. This results in the following specifications for our original example considering the `filter` function:

$$T_{out}.row \leq T_{in}.row$$
$$T_{out}.col = T_{in}.col$$

Observant readers might notice a small difference with the hand-made specification, where the '<' operator is used instead of the '≤' operator for comparing the number of rows. Section 7.3.1 will discuss this difference and explain why both versions are correct.

Unfortunately, based on our anecdotal evidence, the runtime of SymPy grows exponentially with the number of specifications that need to be pruned. This makes the pruning step the bottleneck of the entire algorithm, and it limits the size of the specifications that can be discovered. To combat this, we used the constraint discovery system to generate symmetry-breaking constraints for the relation grammar. While this did have an effect, it did not completely solve the issue. Section 8.2.3 will discuss other ways of overcoming this problem.

This chapter demonstrated how symmetry-breaking constraints for program synthesis can be generated by discovering equivalences in the program space and converting these to constraints that can be used during enumeration. The `Forbid` and `Order` constraints are introduced as a way of breaking symmetries. Furthermore, it was shown how the constraint discovery approach can be modified to discover semantic specifications. Chapter 7 will evaluate the effectiveness of these approaches.

Chapter 6

# Herb.jl: A program synthesis framework

Chapter 5 already demonstrated how constraints and top-down enumeration can be implemented. However, a fully functional program synthesis algorithm consists of several other components. Currently, there is little reuse of these components, even though they are not necessarily unique to specific program synthesis techniques. To address this, a novel program synthesis framework called *Herb.jl* has been developed alongside this thesis [17].

Specific implementation details will not be discussed in this thesis. However, the code for Herb.jl is open-source and can be found online[1]. This chapter does explain a bit more about the philosophy behind Herb.jl in Section 6.1, and motivates the important design choices that were made in Section 6.2. Finally, we will demonstrate how Herb.jl can be used to synthesize programs in Section 6.3.

## 6.1 Philosophy

Herb.jl was designed to unify program synthesis algorithms and optimizations. Currently, there are a lot of exciting and promising techniques in program synthesis, but it is very difficult to explore the potential of combining these techniques. When they are developed within a single framework, the data structures are automatically shared and experimenting with the combinations of these approaches is suddenly a viable option. For instance, it would require relatively little work to combine the constraint system and discovery described in this thesis with stochastic search techniques or search heuristics. Another benefit of a common framework is that developing a new technique does not require reimplementing data structures, helper functions or other shared functionality, which can save valuable time that can now be spent on the relevant and interesting parts of an algorithm.

The goal is to create a framework with the following properties:

1. **Easy to use**: People with minimal knowledge of program synthesis should

[1]`https://github.com/Herb-AI/Herb.jl`

be able to use it for simple tasks. This is important for end users that might want to solve a program synthesis problem using the framework.

2. **Easy to extend**: It should be easy to implement existing and future program synthesis techniques in Herb.jl.

3. **General**: There should be minimal restrictions on the kinds of program synthesis Herb.jl can support. For example, it should also be capable of solving ILP problems.

4. **Reasonably efficient**: Even though efficiency is not (yet) the main focus of Herb.jl, it should not be significantly slower than existing approaches, since this will limit its usefulness and appeal.

Even though a lot of Herb.jl has been developed in parallel with this thesis, it is not made solely for the purpose of this thesis. It is an open-source framework to which everyone is welcome to contribute. Herb.jl is also being used in some course projects already, where students work on adding new functionality and techniques.

## 6.2 Design choices

The most important design choices are made before the first line of code is written. These choices dictate how well the goals mentioned in the previous section can be met, and therefore it is important to put some thought into them. This section explains the most important design choices behind Herb.jl, namely the programming language and the architecture.

### 6.2.1 Programming language

Herb.jl is being implemented in Julia since the Julia programming language fits the goals well [2]. The following features of Julia are especially helpful for us:

1. **The syntax is quite similar to that of Python [31]**. Therefore, students, researchers and other users do not have to spend a lot of time getting familiar with the programming language. Furthermore, the language is high-level enough that the time investment for implementing features stays low.

2. **Julia is quite a fast language [29]**. Program synthesis often relies on enumerative techniques, and Julia has significantly more efficient loops compared to e.g. Python, R or Java, making it a logical choice.

3. **Julia also has strong support for meta-programming [30]**. This is used in Herb.jl to make it easy for users to define their search space. The Julia evaluation function is also used by default in Herb.jl.

### 6.2.2 Multi-module architecture

To make Herb.jl easy to extend, it is designed within a multi-module architecture. This makes it easy to extend or swap out individual components without needing to change functionality inside other modules. For the purposes of this thesis, the most important modules are:

1. **Herb.jl**: This is the main module that connects all other modules and provides an interface to the user.
2. **HerbData.jl**: HerbData.jl is a small module that defines the data structures that can be used to specify program synthesis problems.
3. **HerbGrammar.jl**: This is one of the most important modules, as it is used by almost every other module. It provides functionality to the user to easily define the search space by providing a grammar. It also includes data structures to efficiently represent programs in the search space and lots of related utility functions. Some of the functionality in HerbGrammar.jl is taken from the ExprRules.jl Julia package [25].
4. **HerbSearch.jl**: Perhaps the most interesting module is HerbSearch.jl. This module interacts with a lot of other modules. The module contains various algorithms for systematically searching and enumerating the search space.
5. **HerbConstraints.jl**: - HerbConstraints.jl contains several constraints that can be imposed on the search space. It also provides structures that can be extended for defining custom constraints.
6. **HerbEvaluation.jl**: - This module contains useful evaluators. It is not necessary to use this module, since one can also provide custom evaluators.

## 6.3 Example usage

This section demonstrates how Herb.jl can be used to perform a simple program synthesis task. Please note that the development of Herb.jl is ongoing, and the presented code might not work in future versions. More complete and up-to-date tutorials can be found in HerbExamples.jl[2].

### 6.3.1 Grammar definition: HerbGrammar.jl

Using Julia's powerful metaprogramming, the user is able to write their grammar in an intuitive syntax. The integer arithmetic grammar that is given in Section 2.2 can be defined as follows in Herb.jl:

```
1 grammar = @csgrammar begin
2   Int = Int + Int
3   Int = Int - Int
4   Int = Int × Int
5   Int = |(0:9)
6   Int = x
7 end
```

The first few lines add some operators. Line 5 adds the numbers 0-9 to the grammar. Line 6 adds the input variable `x`. For these variables, we expect assigned values in a problem definition from HerbData.jl. The default evaluator does not need definitions for the operators, since the Julia definition is used by default. In fact, it is also possible to define custom operators and functions in the scope where the grammar is declared and use those in the grammar. The

[2]`https://github.com/Herb-AI/HerbExamples.jl`

evaluator will make use of these custom definitions.

An important prerequisite for being able to use this syntax is that the grammar should be able to be parsed by Julia's built-in parser. If this is not the case, and a custom evaluator is being used, it is also possible to use Julia strings and string interpolation to represent the syntax.

### 6.3.2 Adding constraints: HerbConstraints.jl

`HerbConstraints.jl` is the module that is responsible for handling the constraints. The most important constraints it contains are the `Forbid` constraint and the `Order` constraint. To illustrate this, we will add two constraints that together forbid addition with the integer literal 0:

```
addconstraint!(grammar, Forbidden(MatchNode(1, [MatchNode(5),
    MatchVar(:x)])))
addconstraint!(grammar, Forbidden(MatchNode(1, [MatchVar(:x),
    MatchNode(5)])))
```

A constraint is added to a grammar. The `Forbid` constraint contains a pattern defined as a tree of `MatchNode`s (reference nodes) and `MatchVar`s (variable nodes). `MatchNode`s correspond to a specific rule in the grammar and are thus defined using the index of this rule in the grammar. This can be quite confusing to new users. For example, the index for 0 in the grammar is 5. Changing this to a more user-friendly way is something that is planned for the future[3]. `MatchVar`s define a variable that can match any rule. If a pattern has multiple instances of the same variable, they must correspond to the same tree. Section 5.1.1 gives a more detailed explanation of patterns.

### 6.3.3 Problem definition: HerbData.jl

HerbData.jl provides data structures for defining the program synthesis problem that should be solved. The main data structure is a `Problem` consisting of a list of examples and a name. The most common type of example is the `IOExample`, which represents input-output examples.

```
problem = Problem(
  [
    IOExample(Dict(:x => 0), 1),
    IOExample(Dict(:x => 2), 5),
    IOExample(Dict(:x => 4), 17)
  ],
  "My first problem!"
)
```

An `IOExample` consists of an input and an output, as the name suggests. The input is a dictionary with an assignment for each variable in the grammar. The output is a single value.

### 6.3.4 Search algorithms: HerbSearch.jl

After defining the program space and the requirements for the program that should be synthesized, the synthesis can begin. HerbSearch.jl contains various

[3]`https://github.com/Herb-AI/HerbGrammar.jl/issues/11`

search algorithms for finding the program in the program space that satisfies (the most) examples. The basic search procedure just uses a simple breadth-first search that finds the smallest program that satisfies all examples.

```
solution = search(grammar, problem, :Int)
```

The search algorithm is provided with the grammar, the problem and the starting symbol of the grammar. After running this code, the program $1 + x \times x$ is returned. This program does indeed solve all of the provided examples.

# Chapter 7

# Evaluation

The goal of this thesis is to extract semantic knowledge from a language and apply it in a program synthesis algorithm to reduce redundant enumeration. We discussed two different methods with which this can be achieved. For the first method, which involves removing symmetries, we asked how symmetries can be broken, how they can be discovered and how a discovered symmetry can be converted into a constraint. We evaluate whether our proposed solution from Chapter 5 is an appropriate answer by investigating the following follow-up questions:

1. Can the constraint discovery discover correct constraints?
2. How much of the program space is pruned by the constraints?
3. Does the benefit of a smaller program space outweigh the overhead of enforcing the constraints?

The second method is to discover the semantic specifications of the operators in a language. In Chapter 4, we asked how these specifications can be discovered. This chapter will show that the proposed solution from Section 3.3 answers our question, and evaluate their usefulness and performance. To do this, we answer the following questions:

1. Does the specification discovery discover correct specifications?
2. Do the discovered specifications improve the performance of Morpheus without specifications?
3. How do the discovered specifications compare to the expert-defined specifications?

However, we first discuss the setup for the experiments. After that, these six questions will be answered one by one.

## 7.1 Setup

The results are gathered on a system running an Intel i7-8750h processor with 16 GB of RAM. Runtime data is gathered using the BenchmarkTools.jl Julia package [20]. The runtime results that are displayed are the median of $n = 5$ samples. It is not necessary to use multiple samples for determining the search space size, since this is deterministic and thus constant between runs.

For the results of the constraints and the constraint discovery contributions, two benchmarks are used. The first benchmark uses the integer arithmetic grammar, given in Figure 2.2a. The second benchmark uses the lists grammar from Figure 2.2b. Performance on these grammars is tested by enumerating every program that is smaller than a certain maximum number of nodes.

Morpheus is used to assess the performance of the discovered specifications. As a benchmark, we use the dataset of 50 problems that can be found in the GitHub repository that contains the implementations of both Neo and Morpheus[1] [12, 13]. These problems are real data analysis tasks taken from StackOverflow. Specifications were added by removing existing specifications for the number of rows and columns from the `specs` files and replacing them with the discovered specifications.

## 7.2 Constraint discovery

### 7.2.1 Discovering correct constraints

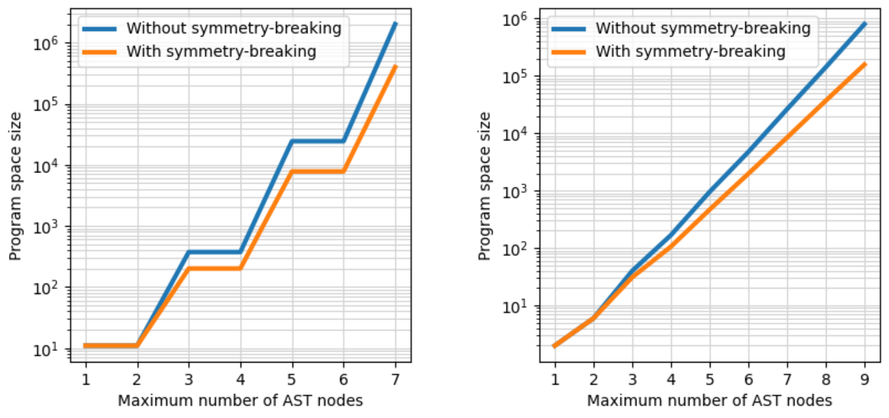
Generating the constraints for the integer arithmetic grammar and the lists grammar is an effortless process. The procedure is configured as follows:

1. The maximum number of nodes in a pattern is 5.
2. There is a maximum of 2 different variables in the patterns for the integer arithmetic grammar and a maximum of 3 different variables for the lists grammar. See Section 5.1.1 for the considerations on picking this value.
3. The automatic data generators consider expressions with a maximum size of 5 nodes when generating programs for the integer arithmetic grammar. For the lists grammar, programs are allowed to have a maximum of seven nodes. This slightly higher limit is due to the recursively defined lists, requiring a greater number of nodes to generate adequately long lists.

Executing the procedure for the integer arithmetic grammar and the lists grammar takes 3.12 and 3.72 seconds respectively (median from $n = 10$ samples). The following constraints are extracted for the integer arithmetic grammar:

| | |
|---|---|
| `Forbid` $a \times 0$ | `Forbid` $a \times 3 - a$ |
| `Forbid` $a - a$ | `Forbid` $0 + a$ |
| `Forbid` $a - 0$ | `Forbid` $0 \times a$ |
| `Forbid` $a + 0$ | `Forbid` $1 \times a$ |
| `Forbid` $a \times 1$ | `Order` $[a, b]$ `in` $a + b$ |
| `Forbid` $a \times 2 - a$ | `Order` $[a, b]$ `in` $a * b$ |
| `Forbid` $a + a$ | |

[1] `https://github.com/utopia-group/neo/tree/master/problem/Morpheus-PLDI`



(a) **Integer arithmetic** (b) **Lists**

Figure 7.1: **Results of running enumeration with and without constraints on the program space size**

The lists grammar has slightly more constraints:

Forbid append($a$, [])
Forbid append([], $a$)
Forbid reverse(reverse($a$))
Forbid sort(reverse($a$))
Forbid sort(sort($a$))
Forbid sort(push([], $d$))
Forbid append($a$, push($b$, $d$))
Forbid append($a$, append($a$, $b$))
Forbid sort(push($a$, 9))
Forbid append($a$, append($c$, $b$))
Forbid append(reverse($b$), reverse($a$))
Forbid sort(append($b$, reverse($a$)))
Forbid sort(append($b$, sort($a$)))
Forbid sort(push(sort($a$), $d$))
Forbid append(push([], $d$), $a$)
Forbid reverse(push([], $d$))
Forbid reverse(sort(push($a$, 0)))
Forbid reverse(append(reverse($a$), $b$))
Forbid reverse(append($b$, reverse($a$)))
Order $[a, b]$ in sort(append($a$, $b$))

A noteworthy observation in these constraints is the presence of the `Forbid` $sort(push(a, 9))$ and `Forbid reverse(sort(push(`$a$, 0`)))` constraints. These constraints are generated based on the assumption that the minimum value in a list is 0, and the maximum value is 9, as this assumption holds true for all values generated by the data generators. These constraints remain valid as long as the input variable values are inside this range. If the input values from the data set deviate from this range, they have to be passed to the data generator to make the constraints more general, as explained in Section 5.1.2. A good thing to keep in mind is that the discovered constraints are only as general as the data from the data generators. The effectiveness and applicability of the constraints are heavily dependent on the representativeness of the underlying data.

### 7.2.2 Comparing the program space

In the first test, the effect of the automatically generated constraints on the size of the program space is assessed. Figure 7.1 shows the size of the program space for an increasing limit on the number of nodes in the AST of the program. The y-axis is in a logarithmic scale since the size of the program space increases exponentially with the size limit on programs. The figures show that the inclusion of symmetry-breaking constraints does have a significant effect on the size of the program space for larger programs.

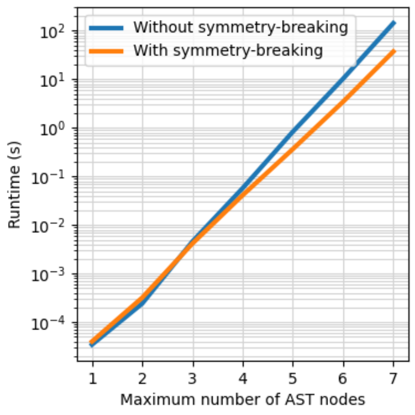
Figure 7.3 shows the program space size with constraints as a ratio of the program space size without constraints. This paints a clearer picture of the benefit of the constraints. In the integer arithmetic grammar limited to programs with at most 7 nodes in the AST, the constraints reduce the program space to 19.84% of the original size. In the lists grammar, the program space is reduced to 19.79% when limited to programs of size 9 or smaller. From these figures, it is clear that the constraints do have a significant positive effect on the size of the search space.

An interesting observation in the search space size of the integer arithmetic grammar is that the search space size has ‘plateaus’ between each odd number and the following even number. The cause for these plateaus lies solely in the grammar that was chosen. Since every production rule either has 0 children or 2 children, every AST from this grammar is a so-called *full binary tree.* An interesting property of full binary trees is that they always have an odd number of nodes. Consequently, every AST produced by the enumerator for this grammar has an odd number of nodes, and thus there are just as many enumerations for an even maximum number of nodes as there is for the previous odd maximum number of nodes.

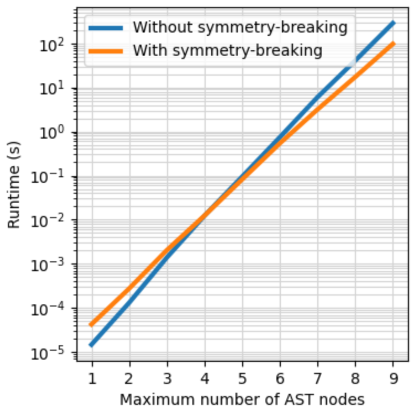
### 7.2.3 Comparing the runtime

The runtime of enumeration is also influenced by constraints. Constraints reduce the size of the search space, and therefore also the number of (partial) programs that are generated. However, the constraints also have to be enforced at every expansion step in the enumeration procedure, which can take up a significant amount of time. Figure 7.2 shows that this overhead makes the enumeration with constraints perform worse than enumeration without constraints for small programs. Again, this is captured in more detail in Figure 7.3. The runtime with constraints as a ratio to the runtime without constraints is higher than one for small programs. However, the same figures also show that for larger programs, there is a big improvement in the runtime. In the arithmetic grammar, the runtime of enumeration with constraints is only 26.03% of the runtime of enumeration without constraints of programs of at most 7 nodes. In the lists grammar, the relative runtime is 33.88% for programs up to 9 nodes. This shows that the discovered symmetry-breaking constraints can also significantly reduce the runtime of enumeration.

An important thing to note is that in a program synthesis problem, each enumeration is also evaluated several times. Evaluation can be expensive, depending on the number of examples and the grammar. Since the cost of evaluation depends on the number of enumerated programs, constraints might be even more useful in this scenario.



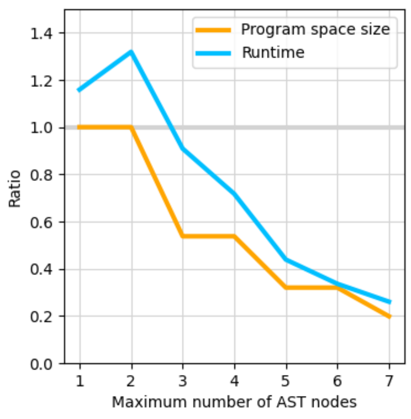
(a) **Integer arithmetic**



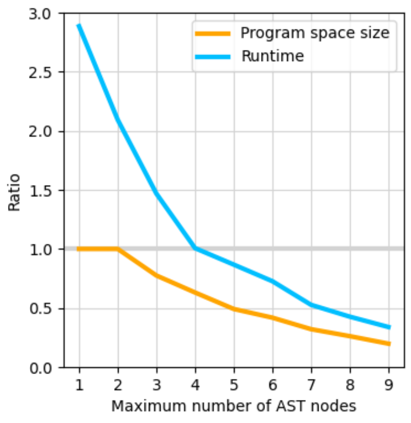
(b) **Lists**

Figure 7.2: **Results of running enumeration with and without constraints on the runtime**

Appendix A contains the exact values for the results presented in this section. Now that we investigated the correctness, program space size and runtime of the versions with and without symmetry-breaking on two grammars, it becomes apparent that the symmetry-breaking has a significant positive effect, especially when synthesizing non-trivial programs. Coming back to the questions from Chapter 4, we can conclude that the proposed symmetry-breaking constraints, as well as the symmetry discovery and constraint conversion algorithms, do work.



(a) **Integer arithmetic**



(b) **Lists**

Figure 7.3: **The cost of enumeration with constraints, represented as a ratio of the cost of enumeration without constraints.**

| Lib | Component | Description | Expert specification | Discovered specification |
|---|---|---|---|---|
| tidyr | `spread` | Spread a key-value pair across multiple columns. | $T_{out}.row \leq T_{in}.row$ <br> $T_{out}.col \geq T_{in}.col$ | $T_{out}.row \leq T_{in}.row$ <br> $T_{out}.col \geq T_{in}.col$ |
| | `gather` | Takes multiple columns and collapses into key-value pairs, duplicating all other columns as needed. | $T_{out}.row \geq T_{in}.row$ <br> $T_{out}.col \leq T_{in}.col$ | $T_{out}.row \geq T_{in}.row$ <br> $T_{out}.col \leq T_{in}.col$ |
| dplyr | `select` | Project a subset of columns in a data frame. | $T_{out}.row = T_{in}.row$ <br> $T_{out}.col < T_{in}.col$ | $T_{out}.row = T_{in}.row$ <br> $T_{out}.col \leq T_{in}.col$ |
| | `filter` | Select a subset of rows in a data frame. | $T_{out}.row < T_{in}.row$ <br> $T_{out}.col = T_{in}.col$ | $T_{out}.row \leq T_{in}.row$ <br> $T_{out}.col = T_{in}.col$ |
| | `inner join` | Takes two tables and joins them on a certain column, discarding rows that do not have a corresponding row in the other table. | $T_{out}.col \leq T_{1,in}.col + T_{2,in}.col - 1$ | $T_{out}.col \geq T_{1,in}.col$ <br> $T_{out}.col \geq T_{2,in}.col$ |

Table 7.1: **Sample specifications from Morpheus and the specification discovery algorithm for a few components**

## 7.3 Specification discovery

### 7.3.1 Discovering specifications

Discovering specifications is more involved than discovering constraints. This is mainly caused by the fact that more data generators are needed. The data generator for data frames is easy to define; it just has to return a random data frame from the dataset. However, for `filter`, the function that filters rows from a data frame, a predicate function needs to be defined that decides which rows to keep. Instead of coming up with a wide range of functions, it is also possible to provide a random predicate function that does a coin toss for each row. The probability of keeping a row is also picked at random. Using simple tricks like these made creating definitions for data generators a lot less tedious. The specification extraction algorithm was configured to find specifications with a maximum size of 5 nodes. This limit is necessary because otherwise, the pruning step is not able to finish the pruning within a reasonable amount of time. As a result, only very basic specifications are discovered.

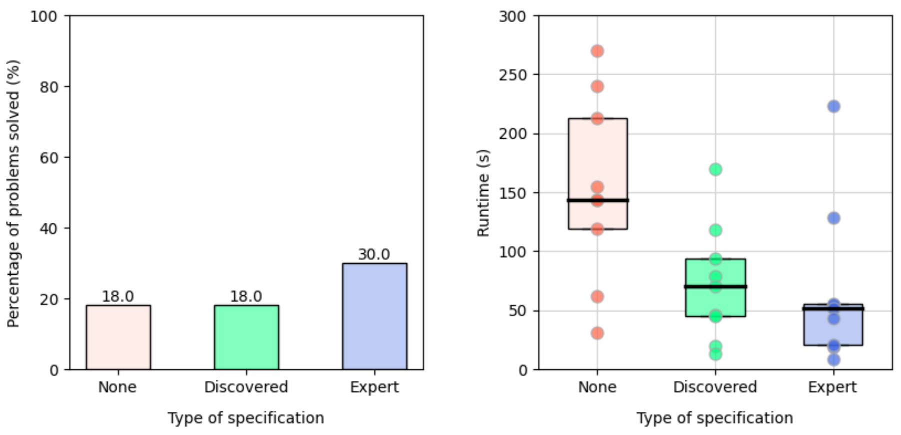
Table 7.1 shows a selection of the specifications that are discovered in the last column. Two interesting differences can be found when comparing the discovered specifications with the specifications defined by an expert. Firstly, the discovered specifications for the inner join operator are quite different from the expert-defined specification. This stems from the imposed specification size limit of five nodes. As a consequence, two less accurate alternatives are discovered instead of finding the more precise specification. These alternatives are still valid, but some information is lost. Another interesting difference can be seen in the specifications for the `select` and `filter` operators. For the `filter` operator, the expert defined the specification $T_{out}.row < T_{in}.row$. However, if the `filter` operator is supplied with a predicate function that always returns true, it returns exactly the input data frame. Hence, the discovered specification correctly states $T_{out}.row \leq T_{in}.row$. It is worth noting that the expert specification is technically also correct, and arguably more optimal, since having a `filter` operator that does not filter anything is redundant for the specific input-output example. Since MORPHEUS always works with a single input-output example, the `filter` can thus be omitted.

### 7.3.2 Comparing to other levels of specification

Figure 7.4 illustrates the performance of MORPHEUS under different scenarios:

1. without using any specifications on the number of rows and columns,
2. using the discovered specifications, and
3. using the specifications that are defined by an expert.

In Figure 7.4a, it can be seen that adding the discovered specifications does not affect the number of solved problems. Both the versions without specifications and with the discovered specifications are able to solve a mere 9 out of 50 problems within a timeout of 5 minutes. MORPHEUS with the expert-defined specifications can solve 15 of the problems. Based on this data, it seems that the discovered specifications do not have an effect.



(a) **Percentage of solved problems within a 5-minute timeout**

(b) **Runtime on the 9 examples that all specifications solved**

Figure 7.4: **The performance of Morpheus without specifications, with the discovered specifications and the original specifications defined by an expert**

However, when comparing the runtime of Morpheus on the nine problems that every version can solve within the timeout, this effect is a lot more apparent. Figure 7.4b shows this comparison. When Morpheus operates without specifications, the median runtime for these problems is 143.3 seconds. The version with the discovered specifications is on average more than twice as fast, solving them with a median time of 70.0 seconds. The expert-defined specifications are still the fastest with a median runtime of 51.3 seconds. Hence, it becomes evident that the discovered specifications do indeed have an effect, and are superior to not having any specifications at all. Nonetheless, they are not yet suitable for replacing specifications defined by a domain expert.

In Chapter 4, we asked how the symmetry-breaking approach can be modified to discover semantic specifications from a language. The experiments in this section demonstrated that the proposed solution from Section 3.3 produces specifications of decent quality, which shows that the outlined approach works.

Chapter 8

# Conclusion & future work

## 8.1 Conclusion

This thesis presents a novel approach to eliminating symmetries in enumerative program synthesis by utilizing semantic knowledge. The proposed approach is able to automatically generate symmetry-breaking constraints by generating hypothetical equivalences, empirically testing them to obtain equivalences and converting these equivalences into constraints. The experimental results demonstrate that this approach significantly reduces the size of the program space and enhances the runtime efficiency of the enumeration algorithm. However, it is essential to acknowledge a pitfall of this approach, since the discovered constraints are only as general as the data that is used for discovering equivalences. Consequently, it is crucial to carefully configure the constraint discovery procedure and provide it with data that is sufficiently general.

Another contribution is the adaptation of the equivalence discovery procedure, aiming to discover semantic knowledge in the form of specifications. The performance of these specifications is assessed in MORPHEUS, and they prove to be a significant improvement compared to not having specifications. However, the expert-defined specifications still outperformed the discovered specifications, highlighting the need for further advancements in this area.

Finally, we also introduced Herb.jl, a new program synthesis framework that is modular and reusable. Herb.jl still needs a lot of time and effort before becoming a mature program synthesis framework. However, we were already able to use it for this thesis by equipping it with a constraint system. Herb.jl has also found some use in course projects, where students were able to efficiently create new functionality. This functionality can also be used in combination with other techniques since everything uses the same underlying data structures.

## 8.2 Future work

### 8.2.1 Proving the correctness of constraints

One important shortcoming of the constraint discovery procedure is that it is not completely certain that a certain constraint is correct. In practice, the approach has been very reliable, provided it is configured correctly. Nevertheless, complete certainty about the correctness of the constraints might be a desired property in some scenarios. To achieve this, it might be worth looking into connecting to HipSpec [6] for Haskell programs. It is important to note that this does require access to the implementation of the operators in the grammar, which the procedure currently does not have.

### 8.2.2 Discover runtime errors

Some programs are syntactically correct but result in a runtime error when evaluated. Examples include taking the `head` function of an empty list or dividing by zero. The constraint discovery procedure could be extended to also detect programs that always result in a runtime error. These can then be turned into `Forbid` constraints that prune programs that will always cause runtime errors from the program space.

### 8.2.3 Improved specification pruning

The pruning step for specification discovery currently is a bottleneck which limits the size of the specifications that can be discovered. A possible solution could be to include another testing step to find out which specifications imply other specifications. Specifications would be evaluated with data from data generators. The output data does not have to be the valid output for the operator that is being tested. If a certain specification $A$ evaluates to true on a superset of the tests on which another specification $B$ evaluates to true, we can conclude that $B$ implies $A$, and thus $A$ can be pruned from the specifications. Another option could be to simplify the specifications using an SMT solver such as Z3. Z3 has some built-in rudimentary simplifiers, but it is also possible to define custom ones [3].

### 8.2.4 Improved deduction

The current implementation of constraints does not deduce information from the domains of other holes in the tree when enforcing constraints. Adding this could mean that partial trees get removed from enumeration at an earlier point in time, thus reducing the number of expansions the enumerative search algorithm has to perform. To perform the deduction from the hole domains, an SMT solver could be employed. Further research should investigate if the benefits of this approach outweigh the added overhead of deduction.

### 8.2.5 Herb.jl

The contributions to Herb.jl are a big part of this thesis. Nevertheless, Herb.jl still needs a lot of development before becoming a mature program synthesis toolbox. This thesis only uses a basic working version of Herb.jl and extended this with constraints. In a lot of other aspects, Herb.jl is still very basal, and even the constraints are not handled very efficiently.

One important area that needs work is the enumeration. It could be made significantly more efficient by optimizing the code and parallelizing it. Another goal is to add implementations of multiple existing algorithms such as MORPHEUS [12], NEO [13] and DEEPCODER [1]. Please note that Herb.jl is an open-source project, and contributions are always welcome!

# Bibliography

[1] Matej Balog et al. “Deepcoder: Learning to write programs”. In: *arXiv preprint arXiv:1611.01989* (2016).

[2] Jeff Bezanson et al. “Julia: A fresh approach to numerical computing”. In: *SIAM review* 59.1 (2017), pp. 65–98. URL: https://doi.org/10.1137/141000671.

[3] Nikolay Bjorner. *Simplifiers — Online Z3 Guide*. URL: https://microsoft.github.io/z3guide/docs/strategies/simplifiers (visited on 15/06/2023).

[4] Noam Chomsky. *Three models for the description of language*. 1965.

[5] Koen Claessen, Nicholas Smallbone and John Hughes. “QuickSpec: Guessing Formal Specifications Using Testing”. In: *Tests and Proofs*. Ed. by Gordon Fraser and Angelo Gargantini. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 6–21. ISBN: 978-3-642-13977-2.

[6] Koen Claessen et al. “Automating inductive proofs using theory exploration”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7898 LNAI (2013), pp. 392–406. ISSN: 03029743. DOI: 10.1007/978-3-642-38574-2_27.

[7] Andrew Cropper and Sebastijan Dumančić. “Inductive Logic Programming at 30: A New Introduction”. In: *Journal of Artificial Intelligence Research* 74 (2022), pp. 765–850. ISSN: 10769757. DOI: 10.1613/jair.1.13507. arXiv: 2008.07912.

[8] Andrew Cropper and Rolf Morel. *Learning programs by learning from failures*. Vol. 110. 4. Springer US, 2021, pp. 801–856. ISBN: 1099402005934. DOI: 10.1007/s10994-020-05934-z. arXiv: 2005.02259. URL: https://doi.org/10.1007/s10994-020-05934-z.

[9] Leonardo De Moura and Nikolaj Bjørner. “Z3: An efficient SMT Solver”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 4963 LNCS (2008), pp. 337–340. ISSN: 03029743. DOI: 10.1007/978-3-540-78800-3_24.

[10] Tao Du et al. “InverseCSG: Automatic conversion of 3D models to CSG trees”. In: *SIGGRAPH Asia 2018 Technical Papers, SIGGRAPH Asia 2018* 37.6 (2018). ISSN: 15577368. DOI: 10.1145/3272127.3275006.

[11] Torsten Fahle, Stefan Schamberger and Meinolf Sellmann. "Symmetry breaking". In: *Principles and Practice of Constraint Programming—CP 2001: 7th International Conference, CP 2001 Paphos, Cyprus, November 26–December 1, 2001 Proceedings 7*. Springer. 2001, pp. 93–107.

[12] Yu Feng et al. "Component-based synthesis of table consolidation and transformation tasks from examples". In: *ACM SIGPLAN Notices* 52.6 (2017), pp. 422–436. ISSN: 0362-1340. DOI: `10.1145/3140587.3062351`. URL: `https://doi.org/10.1145/3140587.3062351`.

[13] Yu Feng et al. "Program synthesis using conflict-driven learning". In: *ACM SIGPLAN Notices* 53.4 (2018), pp. 420–435. ISSN: 15232867. DOI: `10.1145/3192366.3192382`. arXiv: `1711.08029`.

[14] John K. Feser, Swarat Chaudhuri and Isil Dillig. "Synthesizing data structure transformations from input-output examples". In: *ACM SIGPLAN Notices* 50.6 (2015), pp. 229–239. ISSN: 0362-1340. DOI: `10.1145/2813885.2737977`.

[15] Sumit Gulwani. "Automating string processing in spreadsheets using input-output examples". In: *ACM Sigplan Notices* 46.1 (2011), pp. 317–330.

[16] Sumit Gulwani, Oleksandr Polozov and Rishabh Singh. *Program synthesis*. Vol. 4. 1-2. 2017, pp. 1–119. ISBN: 9781680832921. DOI: `10.1561/2500000010`.

[17] Herb-AI. *Herb.jl*. URL: `https://github.com/Herb-AI/Herb.jl` (visited on 08/06/2023).

[18] Kangjing Huang et al. "Reconciling enumerative and deductive program synthesis". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2020), pp. 1159–1174. DOI: `10.1145/3385412.3386027`.

[19] Naman Jain et al. *Jigsaw: Large Language Models meet Program Synthesis*. Vol. 2022-May. 1. Association for Computing Machinery, 2022, pp. 1219–1231. ISBN: 9781450392211. DOI: `10.1145/3510003.3510203`. arXiv: `2112.02969`.

[20] JuliaCI. *BenchmarkTools.jl*. 2022. URL: `https://github.com/JuliaCI/BenchmarkTools.jl/tree/v1.3.2` (visited on 09/05/2023).

[21] Pepijn Klop. "Augmenting Program Synthesis with Large Language Models". MSc thesis. 2023.

[22] Kevin Knight. "Unification: A multidisciplinary survey". In: *ACM Computing Surveys (CSUR)* 21.1 (1989), pp. 93–124.

[23] Hadas Kress-Gazit, Morteza Lahijanian and Vasumathi Raman. "Synthesis for Robots: Guarantees and Feedback for Robot Behavior". In: *Annual Review of Control, Robotics, and Autonomous Systems* 1 (2018), pp. 211–236. ISSN: 25735144. DOI: `10.1146/annurev-control-060117-104838`.

[24] Tessa Lau et al. "Programming by demonstration using version space algebra". In: *Machine Learning* 53 (2003), pp. 111–156.

[25] Ritchie Lee and Mykel Kochenderfer. *sisl/ExprRules.jl: Functions for declaring and working with grammars and expression trees in Julia*. URL: `https://github.com/sisl/ExprRules.jl` (visited on 22/06/2023).

[26] Zohar Manna and Richard Waldinger. "A Deductive Approach to Program Synthesis". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2.1 (1980), pp. 90–121. ISSN: 15584593. DOI: 10.1145/357084.357090.

[27] Bertrand Meyer. *Introduction to the theory of programming languages*. Prentice-Hall, Inc., 1990.

[28] OpenAI. *GPT-4*. 2023. URL: https://openai.com/product/gpt-4 (visited on 23/05/2023).

[29] The Julia Project. *Julia Micro-Benchmarks*. 2023. URL: https://julialang.org/benchmarks/ (visited on 05/05/2023).

[30] The Julia Project. *Metaprogramming - The Julia Language*. 2022. URL: https://docs.julialang.org/en/v1/manual/metaprogramming/ (visited on 05/05/2023).

[31] The Julia Project. *Noteworthy differences from Python*. 2022. URL: https://docs.julialang.org/en/v1/manual/noteworthy-differences/#Noteworthy-differences-from-Python (visited on 05/05/2023).

[32] Dan Rosén. "Proving equational Haskell properties using automated theorem provers". Master's thesis, University of Gothenburg, Sweden, 2012.

[33] Kenneth Slonneger and Barry L Kurtz. *Formal syntax and semantics of programming languages*. Vol. 340. Addison-Wesley Reading, 1995.

[34] Nicholas Smallbone et al. "Quick specifications for the busy programmer". In: *Journal of Functional Programming* 27.January (2017). ISSN: 14697653. DOI: 10.1017/S0956796817000090.

[35] Armando Solar-Lezama. *Introduction to Program Synthesis*. 2018. URL: https://people.csail.mit.edu/asolar/SynthesisCourse/index.htm (visited on 04/01/2023).

[36] Armando Solar-Lezama. "The sketching approach to program synthesis". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5904 LNCS (2009), pp. 4–13. ISSN: 03029743. DOI: 10.1007/978-3-642-10672-9_3.

[37] SymPy Development Team. *SymPy*. URL: https://www.sympy.org/en/index.html (visited on 03/06/2023).

# Appendix A

# Exact results

## A.1 Integer arithmetic grammar

| | **Without constraints** | | **With constraints** | |
|---|---|---|---|---|
| Max program size | Search space | Runtime (s) | Search space | Runtime (s) |
| 1 | 11 | 3.46E-05 | 11 | 4.01E-05 |
| 2 | 11 | 2.43E-04 | 11 | 3.20E-04 |
| 3 | 374 | 4.59E-03 | 201 | 4.17E-03 |
| 4 | 374 | 5.75E-02 | 201 | 4.12E-02 |
| 5 | 24332 | 8.38E-01 | 7786 | 3.68E-01 |
| 6 | 24332 | 1.03E+01 | 7786 | 3.46E+00 |
| 7 | 2000867 | 1.43E+02 | 397056 | 3.73E+01 |

## A.2 Lists grammar

| | **Without constraints** | | **With constraints** | |
|---|---|---|---|---|
| Max program size | Search space | Runtime (s) | Search space | Runtime (s) |
| 1 | 2 | 1.48E-05 | 2 | 4.27E-05 |
| 2 | 6 | 1.32E-04 | 6 | 2.75E-04 |
| 3 | 40 | 1.39E-03 | 31 | 2.04E-03 |
| 4 | 168 | 1.23E-02 | 106 | 1.24E-02 |
| 5 | 950 | 9.47E-02 | 467 | 8.22E-02 |
| 6 | 4706 | 7.41E-01 | 1967 | 5.39E-01 |
| 7 | 26128 | 5.99E+00 | 8375 | 3.16E+00 |
| 8 | 140272 | 4.02E+01 | 36684 | 1.72E+01 |
| 9 | 789498 | 2.85E+02 | 156227 | 9.66E+01 |