Evolving a Domain-Specific Language to Speed Up Program Synthesis

Philip Tempelman
Supervisor: Sebastijan Dumančić
EEMCS, Delft University of Technology, The Netherlands

23-6-2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

Program synthesis is used in various ways to automate repetitive tasks or to generate software automatically. Search-based program synthesis constitutes searching the space of candidate programs created from a given language. However, this form of program synthesis is very expensive in terms of computing power. By optimising the synthesiser's parameters on certain tasks, program synthesis can be made more efficient on other similar tasks. One of these parameters is the domain-specific language. Using a genetic algorithm, an optimised language was evolved for three different domains. This resulted in unnecessary language predicates being phased out and commonly used structures being introduced as new language predicates. Overall, using these evolved languages made program synthesis faster for all three domains.

1 Introduction

"Program synthesis is the task of automatically finding a program in the underlying programming language that satisfies the user intent expressed in the form of some specification. Since the inception of AI in the 1950s, this has been considered the holy grail of Computer Science." [1]. Now, in the 2020s the field of program synthesis is a lot more established. It has been used by software engineers to validate code quality [2] but also by others to automate repetitive tasks in the form of Excell's Flash Fill [3].

For this research, we go one step beyond this. Instead of synthesising a certain piece of code, we delve into the holy grail of the holy grail, meaning we will research how to synthesise a program synthesiser. If successful, one would be able to discover with relative ease how best to synthesise something for a certain category of tasks. Therefore not only the actual synthesised code is likely to perform well but the synthesising process will also be optimal. Once one is certain that the synthesising process is optimal for certain tasks, the same process can then be applied to similar tasks. By doing this, one can speed up program synthesis by making an investment once, instead of needing to tune a lot for different tasks.

Program synthesis uses a few components. One of them is a domain-specific language (DSL). A domain for program synthesis is a set of tasks that can be solved by programs that are constructed using the same set of predicates. For a domain that involves a robot and a map the predicates in the language probably include ones that allow the robot to move around. However, the language for a domain designed to simulate string manipulation probably includes predicates that can remove or change letters in some way.

The search space for program synthesis consists of all the programs that can be created using the DSL. This search space is infinitely big. However, for each task, a program synthesiser only searches this space until a program that solves that task has been found. This search space can be seen as a tree, where all programs are very specific branches. In this tree, the synthesiser is looking for the exact branch that represents the program that solves a task. If you add just one predicate to the DSL, at every single branching point a new branch has to be added. This makes a synthesiser's chance of finding the right branch a lot lower, which means a lot more time is needed.

To reduce a synthesiser's search time, it is important that every predicate in the DSL serves a clear purpose. More specifically, it should be essential for the task the synthesiser is trying to solve or it should contribute in another way to reducing the synthesis time. Figuring out which predicates contribute to slower or faster synthesis is something that can be done manually. However, this can be very difficult and eventually one will still not be sure whether the created DSL is actually optimal.

Ideally, one would try all the potential DSL's to be sure which one is optimal. However, like trying all programs for a task, this is not feasible. This is why we propose to use a genetic algorithm to evolve a language automatically for program synthesis. Genetic algorithms have been proven useful for optimisation problems in various fields. Using a combination of techniques like crossover and mutation over many generations an optimal solution can be approximated. Therefore, this paper aims to answer the question:

Can we evolve a programming language to speed up program synthesis?

This research has two main contributions. The first involves the general use of a genetic algorithm to evolve a language. We show a language can be evolved for certain synthesis tasks which speeds up synthesis on similar tasks. The second contribution is more specific and involves the creation of new composite language predicates. These composite language predicates are made up of multiple primitive predicates. In a program, a composite predicate involves the execution of one or more primitive predicates. The research shows that in some cases, composite language predicates can be added to the DSL to speed up program synthesis. To make these contributions, this paper answers the following research questions:

- **RQ1:** *How can we translate a DSL into a chromosome?*
- **RQ2:** *How can we add composite predicates to a DSL?*
- **RQ3:** *How can we use genetic programming techniques to evolve a DSL?*
- **RQ4:** *How does a program synthesiser using an evolved DSL compare to one using the standard DSL?*

For all three domains we considered, evolving a domain-specific language, proved to be successful. This means that with the resulting language, the search process was sped up considerably and more tasks could be solved within a certain time.

2 Background

In this section, we lay the foundation for our research. First, we delve into program synthesis and the main issue we are

trying to solve. Then, we describe why we use a genetic algorithm and go over the algorithm's components. Lastly, we go over previous work done on reducing the search space for program synthesis and predicate invention.

2.1 Program Synthesis

The purpose of programming is to find a sequence of instructions that automates a task. The motivation for program synthesis is derived from this; to automate the finding of a sequence that automates a task. To do this, program synthesis attempts to construct a program, so a sequence of instructions, that satisfies a certain specification.

Inductive synthesis

Inductive and deductive synthesis are the two main classes of program synthesis. "The inductive approach aims to derive the final program from some traces at a high level of specification, whereas the deductive approach aims to construct the final program from a type of specification that expresses a relationship between the input and output of a desired program." [[4–9] as cited in [10]].

Inductive synthesis itself can also be divided into a few paradigms. Example-guided synthesisers rely on a number of input and desired output examples to construct programs. Oracle-guided synthesisers use a form of an oracle to answer the learner's questions. Finally, component-based synthesis is used to generate loop-free programs using existing functions as building blocks.

This research makes use of all of these paradigms in different ways. The programs are synthesised for and verified on examples. The synthesiser is trying to minimise an example-dependent loss function, which is used as the oracle, as proposed by A. Cropper and S. Dumančić with Brute [11]. Finally, with our contribution, snippets from successful programs are used as building blocks for new programs. This will be described in Section 3.4.

The search space

The search space for inductive program synthesis is immense. If the wanted program size is known, it can be calculated using Equation 1.

$$search\_space\_size = dsl\_size^{program\_size} \quad (1)$$

Assume we have a language with only ten predicates and we know the resulting program can be made up of only ten predicates. Then there are already ten billion potential programs in the resulting search space.

In practice, most likely we do not know the wanted program size beforehand and our language will have more predicates, making the search space even more unfathomable. Because of the vast search space, searching through all potential programs is not a viable option.

This is why we use a loss function, as proposed with Brute [11], to guide the program synthesis process. For a perfect loss function, meaning that the final program is built on top of programs that each had the lowest loss in their respective stages, this reduces the search space so that it can be calculated by Equation 2.

$$search\_space\_size = dsl\_size * program\_size \quad (2)$$

However, even then, creating a program of length 30 with a language with 20 predicates leads to 600 potential programs to evaluate. Additionally, the loss function might not be as perfect as earlier stated. Furthermore, as search space is the most contributing factor to search time, reducing this search space will also reduce the search time. This will make program synthesis for the end-user less time-consuming. For these reasons, there is a lot to be gained by reducing this search space.

2.2 Genetic Algorithms

Genetic algorithms, like program synthesis, started being developed in the 1950s with the idea that evolution could be used as a tool for optimisation problems [12]. Now, they are commonly used to produce high-quality solutions to optimisation problems in various fields. J. Holland described a framework that mimics the process of evolution in nature [13]. We will briefly summarise the framework's components.

The **chromosome** is a collection of data one is trying to optimise to solve a certain problem. Each chromosome is a collection of genes, which are elements of the data.

The **fitness function** can be seen as the evaluation mechanism of the framework. It allows the algorithm to score the chromosomes. The parameters of the fitness function are the values you are trying to optimise for.

Crossover lays the foundation for how new chromosomes are created. In general, crossover can be thought of as combining one or more parent chromosomes with their offspring as the result. Chromosomes are selected for crossover with a certain probability.

Mutation is used to maintain diversity from one generation to the next. Without mutation, only combinations of initially generated chromosomes could be evaluated. Mutation methods randomly change one or more genes in a chromosome. Like crossover, chromosomes are also selected for mutation with a certain probability.

Selection is the mechanism that decides which chromosomes can pass on their genes and which cannot. In general, chromosomes with higher fitness have a higher chance of passing on their genes to the next generation, enforcing the principle of "survival of the fittest".

A **generation** is a virtual unit of time that encompasses one round of evaluation, selection, crossover, and mutation.

The **population** is the collection of chromosomes that is being evaluated in each generation. Usually, the population's fitness increases over time.

Elitism was first proposed by K. De Jong in [14] and it is used to always preserve the best performing candidates. It selects a certain amount of the best chromosomes for the next generation. These chromosomes can still be used for crossover but they will not be mutated. It is used to prevent the case where the algorithm accidentally mutates the best chromosomes into worse ones.

2.3 Related Work

With this research, we want to achieve two things: to reduce the search space and evolve a language for program synthesis. Although we are trying to reduce the search space by evolving a language, there are also approaches that attempt to directly reduce the search space. This section will discuss related research, including research that directly inspired our work.

Reducing the search space

A framework called Blaze, created by X. Wang, I. Dillig, and R. Singh, makes use of the abstract semantics of the underlying DSL. It *uses the semantics to find a program whose abstract behaviour satisfies the examples* [15]. It decreases the search space because multiple programs that do not produce the same concrete output can have the same abstract output. Therefore, as soon as Blaze knows a program with a certain abstract output is inconsistent with the abstraction of the examples, all programs with a similar abstract output can be discarded. A complication is that a program that is consistent with examples based on its abstract semantics may not actually solve the examples. To counter this, Blaze checks if a program is consistent with the examples using the concrete semantics, and if so, that program is the solution. If not, Blaze specifies the abstraction of the examples, making it inconsistent with many programs and therefore discarding them. It does this until it finds a program that satisfies the examples, or proves that no such program exists.

In [16], C. Smith and A. Albarghouti do something very similar. They *use a term rewriting system to deduce programs to their 'normal form'*. Then they use equivalence reduction on these rewritten programs to discard all but one of the programs with equal normal forms. Eliminating all but one of the programs with the same normal forms, also drastically reduces the search space. Though, it is hard to imagine how useful this framework and Blaze will actually be for domains where the abstract semantics or normal forms of the language predicates are very unclear.

A. Cropper and S. Dumančić created Brute, *which reduces the search space by introducing a domain-specific loss function to guide program synthesis* [11]. It uses two stages: invent and search. In the invent stage, Brute creates composite predicates from primitive predicates, while eliminating pointless predicates. In the search stage, Brute builds a program using best-first search guided by a predetermined loss function. It starts from an empty program and selects the predicate that minimises the loss function. This predicate will then be added to the program. To measure the loss, the sequence of predicates is executed on an example, and the resulting state is compared with the desired final state. As soon as the loss is zero, an example-solving program was found. This means that instead of finding a complete program that either solves or does not solve an example, Brute only has to find the best predicate to add to the program at each stage. This drastically reduces the search space and it is the foundation for the synthesis method we use.

Evolving a language

The following two papers formed the inspiration for considering composite predicates as primitive predicates, which will be described in Section 3.4.

With Knorf [17], S. Dumančić, T. Guns, and A. Cropper proposed a knowledge refactoring system to allow machines to learn more efficiently. It *uses constraint optimisation to restructure a learner's knowledge base* to reduce its size and minimise redundancy. They show that compared to only adding or removing knowledge, restructuring knowledge can improve performance. Knorf supports predicate invention, which is the creation of new symbols together with formulas that describe them [18], and it invents predicates after learning. These new symbols can be thought of as new language predicates, with the describing formulas forming the underlying semantics.

K. Ellis et al. tried to simulate the human's ability to generalise knowledge gained on one domain for other domains [19] with a system named DreamCoder. Broadly speaking, DreamCoder does two things. First, it designs a suitable DSL from a more general set of predicates. Second, it trains a search algorithm that specifically exploits the created DSL. The final DSL has multiple layers of abstraction below it. This reduces the search space from one that is made up of all the low-level language elements, to one derived from only the abstracted domain-specific elements. Initially, Dreamcoder attempts to solve tasks with a very low-level programming language in the wake phase. Then in the abstraction phase, Dreamcoder *finds common program snippets and uses them in their entirety as new language predicates*. Lastly, in the dreaming phase, the neural network that searches programs is trained on so-called "fantasies". These fantasies are created by first randomly creating a program from the learned language. Then, DreamCoder constructs a task that would be solved by that program. Finally, a neural network is trained to predict the created program from the task it solves. The three phases are repeated until no improvements in synthesis and search performance are made anymore. DreamCoder and Knorf formed the inspiration for considering successful program snippets as new language predicates.

3 Methodology

In this section, we describe the methodology for our research. Throughout this section, some terminology will be used. Therefore, we first give some definitions of these terms. Then, we delve into how we implemented various genetic algorithm components. Finally, we give an overview of what happens in the generations at each point in time of the genetic algorithm.

3.1 Terminology

- **Example:** is a combination of an input, the starting state(s), and an output, the desired final state(s).
- **Domain:** is a set of examples that are concerned with the same general goal. E.g. turning one string into another, bringing a ball to a goal, or copying a drawing.
- **Environment** is a state of an example the programs interact with.
- **Token:** is a predicate that can interact with a state in the following ways:

- **Transform:** transforms an environment into another environment. E.g. the MoveRight token: move the robot one position to the right.
- **Observe:** observes a characteristic of the environment. E.g. the AtRight token: confirm if the robot is at the most right position.
- **Control:** deals with controlling the execution of other tokens. Takes observe tokens as input, and returns transform tokens. E.g. the LoopWhileThen token: while the robot is not at the right, move right.

- **Domain-specific language:** is a combination of transform and observe tokens that are different for each domain and that enable the program synthesiser to solve examples in that domain.
- **Program:** a sequence of tokens that makes a series of transformations on an environment. Generally, this series of transformations attempt to transform the input environment into the desired output environment.

3.2 Chromosome

Genetic algorithms try to optimise chromosomes. These chromosomes are representations of a collection of data. In our case, the collection of data we are trying to optimise is the DSL. A chromosome is made up of genes, for a DSL these are language predicates, also known as the tokens. To translate a DSL into a chromosome we simply create a collection of all the DSL's tokens, this answers **RQ1**.

3.3 Crossover

Various crossover techniques were considered. The first two methods are applicable to domains that have a clear division of transform and observe tokens, while the last method is applicable to all domains involving language tokens.

Exchanging transform and observe tokens

Every chromosome is made up of transform and observe tokens. The two parent chromosomes, A and B, exchange their transform and observe tokens. This results in two new chromosomes being created where one has the transform tokens of parent A and the observe tokens of parent B and the other has the opposite tokens, which is illustrated in Figure 1.

Figure 1: Transform and observe tokens crossover

Exchanging half per token type

This method is similar to the last one. However, instead of exchanging all of their transform and observe tokens, they exchange half. This results in two new chromosomes being created where one has half of the transform and observe tokens of parent A and B and the other one has the other halves, as can be seen in Figure 2.

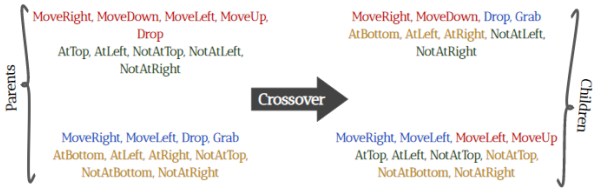

Figure 2: Half of token classes crossover

Exchanging random tokens

For this method, the transform and observe tokens are seen as one group. Then, two random halves of these groups from parents A and B are combined to create one chromosome. With the remaining halves, the other chromosome is created. Figure 3 is an illustration of this.

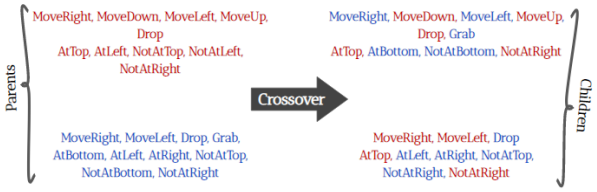

Figure 3: Half of random tokens crossover

3.4 Mutation

To preserve the diversity of chromosomes in each generation, multiple mutation methods were used simultaneously. The first two simple methods deal with primitive tokens, while the last deals with snippets from successful programs.

Add Random Token

Most chromosomes will not consist of all of the available tokens. They might need an extra token from the full domain-specific language to either solve more tasks or find the programs that solve tasks faster. Therefore, this function adds a random token from the full domain-specific language to the chromosome.

Remove Random Token

This method is similar to the last one. However, instead of adding a random token from the full domain-specific language, we now remove a random token from the chromosome. This is done because every extra token makes the potential program space a lot bigger. Therefore, tokens that do not contribute enough to solving tasks should be excluded.

Add Composite Token

The previous two mutation methods deal with the primitive tokens in the standard DSL. This method, however, deals with more complex tokens. These include sequences of the primitive tokens and LoopWhileThen-tokens with the primitive tokens as parameters. Imagine a task where something has to be picked up somewhere and dropped somewhere else. An

example of a program that the synthesiser can create to solve that task is then:

LoopWhileThen(NotAtBottom [MoveDown], [MoveUp]), [Grab, MoveRight, Drop]

This program states to go down until we are at the bottom, then move up, grab something, move right, and drop it. The 'LoopWhileThen' token and the sequence enclosed by '[..]' perform one or more of the primitive tokens which is why we refer to them as composite tokens.

The intuition for this special form of mutation stems from the fact that for a lot of tasks, programs with the same composite tokens are created. One can then look at all the programs that solved tasks to extract these composite tokens. The frequently successfully used composite tokens can then be added to a chromosome which might speed up the synthesis process and could even contribute to more tasks being solved within a certain timeout. This is because when these composite tokens are already in the language, the program synthesiser does not have to rediscover them for each task.

The selection of composite tokens is based on the frequency of appearance. They are picked randomly but the chance of being picked increases as they appear more frequently. While running experiments we noticed that some composite tokens are drastically more frequent than others. To still give a little less frequent composite tokens a decent chance of being picked the weights are scaled by taking the n-th root of all frequencies, where n is a parameter that can be tuned. Giving less frequent composite tokens a chance of being picked is important because it allows the algorithm to explore more combinations. This process answers **RQ2**.

3.5 Fitness

The fitness of a chromosome is calculated by running a program synthesiser on a set of tasks using that chromosome as the domain-specific language. The program synthesiser will attempt to create task-solving programs from the tokens in the language. A chromosome without any tokens can only solve a task where the input is already equal to the output. As you add tokens the options for the program synthesiser to create programs drastically get bigger, which likely results in the chromosome being able to solve more tasks. However, DSLs that are very big result in a very big program space which may make it harder to find a suitable program for a task within a certain time limit.

The fitness of a chromosome should therefore be a combination of the number of tasks solved versus the time it took to solve these tasks. A DSL solving more tasks, meaning it has a higher mean correct ratio (MCR), should have higher fitness. A DSL that takes more time per task, meaning it has a higher mean search time (MST), should have lower fitness. An issue with this way of calculating fitness is that some, usually small, chromosomes that only solve a few tasks but solve them extremely fast have exorbitantly high fitness values. In practice, it would not be useful to have a DSL that only solves a few very specific tasks. Therefore we combat these exorbitant fitness scores by checking how many tasks were solved. If the DSL (C) solves less than half of the tasks the standard DSL (S) solves, we set their fitness to zero. The fitness function we used, which covers these requirements, can be seen in Equation 3.

$$fitness = \begin{cases} 0 & MCR_C < MCR_S/2 \\ MCR_C/MST_C & \text{otherwise} \end{cases} \quad (3)$$

The way MCR is calculated depends on the domain's generalisability. For some domains, a task is a single training case. The resulting program after synthesising either solves or does not solve that single training case, which results in the simple Equation 4.

$$MCR = solved\_cases/total\_tasks \quad (4)$$

For other domains, each task might consist of multiple training cases. A program is created for these training cases and is then evaluated on similar test cases. Therefore, the resulting function for these domains is as shown in Equation 5.

$$MCR = \Big(\sum_{t=1}^{\#tasks} solved\_tests_t/total\_tests_t \Big) \Big/ total\_tasks \quad (5)$$

3.6 One Generation

In the first generation, we randomly create a population of chromosomes. These chromosomes have at least one token and at most all the available tokens in the standard DSL.

For each subsequent generation, we sort the chromosomes based on their fitness. We apply program synthesis with the chromosomes as the DSL to calculate their fitness. During this process, the composite tokens from programs that solved tasks are stored to be used for mutation later.

After sorting the chromosomes, we go over them. Starting from the fittest, we select them for crossover with a certain probability. The offspring along with a number of the fittest chromosomes that were not selected for reproduction are then included in the new population. All the chromosomes in the new population are subjected to mutation. Like crossover, chromosomes are selected for mutation with a certain probability. Following the principle of elitism, we select a number of the fittest chromosomes from the old population to be immediately included in the new population, without being subjected to mutation.

After the last generation, we compare the fittest chromosome with the standard DSL. We compare them on a, so far unseen, set of tasks that is similar in terms of complexity to the one the chromosomes were evolved for. Together with Sections 3.3, 3.4, and 3.5, this answers **RQ3**.

4 Experiments and Results

This section will describe the setup and motivation for our experiments and their results. We ran the experiments on the DelftBlue supercomputer [20]. Even while doing so, experiments took a long time. So first, we describe how we planned the whole experiment workflow. We then go into the findings for how best to run our genetic algorithm with regards to search and other parameters. Then, the domains for the final experiment are described. Finally, we compare the evolved DSLs to their original counterparts, discuss the results, and draw conclusions.

4.1 Order of Experiments

Ideally, one would try all combinations of all parameters to make sure no interdependencies between parameters are ignored. However, since a single run of the genetic algorithm can already take hours, trying all combinations is infeasible. Therefore, each parameter was optimised individually, so while keeping the others constant. This is relevant with regard to the reproducibility of the whole experiment workflow. To decide the order, the parameters were ranked based on importance and independence. Experiments for parameters that are important and on which a lot of other parameters depend were performed first. On the other hand, experiments for less important parameters that depend on earlier parameters were performed later. This was done to approximate the results you would get for running experiments with all potential combinations. The order of experiments is the same order as how they are documented in this paper.

4.2 Search

In the interest of time, all experiments regarding search were performed on one domain. Because it was the domain that had the heaviest search workload, we believe this domain is representative of all domains. A-star search (AS) [21] outperforms all of the other search algorithms that were tried, as can be seen in Figure 4. Compared to Brute [11], running the genetic algorithm takes >30% less time. Besides this, the resulting evolved language is able to solve >4% more tasks using A-star.

With A-star as the search algorithm, using the OptimizedAlignment loss function [22] allowed us to solve >42% more tasks than while using the second-best, the Levenshtein loss function [22].

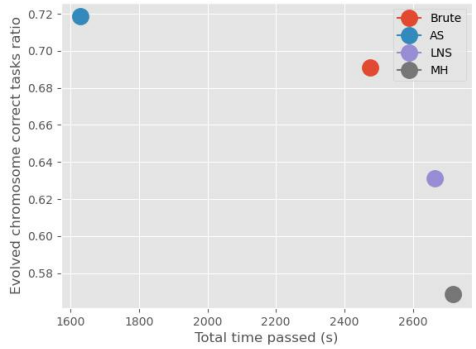

Figure 4: Performance of different search algorithms

A 2-second search time limit per task was chosen for future experiments as it allowed a relatively decent percentage of tasks to be solved, but without each generation taking very long. Higher time limits resulted in only a small fraction of extra tasks being solved while contributing very significantly to total execution time.

4.3 Genetic Algorithm Parameters

Various methods for crossover and mutations were created. Regarding **crossover** the options considered are exchanging two chromosomes' transform and observe tokens, exchanging half of each type, and exchanging a random half. There were no notable differences in performance. We decided to use the exchanging a random half method, as it is most generalisable to program synthesis methods that have no clear distinction between types of tokens. In general, we found that selecting chromosomes for crossover with a probability of 80% worked well.

The options for **mutation** are to add a composite token, add a primitive token, and remove a token. We used a combination of the tree, meaning adding a composite and adding a primitive token happen 25% of the time, and removing a token happens 50% of the time. Addition and removal are balanced so that the average chromosome length for the population is not affected by a bias that stems from mutation. Selecting a chromosome for mutation with a probability of 20% proved to be effective.

The population size and generation limit are very heavy contributors to the total run-time of the genetic algorithm. Therefore, optimising these parameters will decrease the time that is spent unnecessarily. Regarding the **population** size, a size of 30 allowed the best chromosomes to be created in the least amount of time.

Using that population size, after 30 **generations** no more improvements were observed. For the final experiments, being certain that the final results are optimal is more important than the time it takes. Therefore, the generation limit was increased by 10.

4.4 Three domains

To increase the likelihood of the research being applicable to other domains, we ran the final experiments on three different domains. These domains simulate real-world problems. They are also so different that a method that works for all of them, is likely to work for other domains as well. In this section, the three different domains will be described.

String transformation (string):

- **Task:** to transform a string into another string.
- **Input:** a string.
- **Output:** a different string.
- **Standard DSL:** {MoveRight, MoveLeft, MakeUppercase, MakeLowercase, AtEnd, NotAtEnd, AtStart, NotAtStart, IsLetter, IsNotLetter, IsUppercase, IsNotUppercase, IsLowercase, IsNotLowercase, IsNumber, IsNotNumber, IsSpace, IsNotSpace, Drop}

Robot planning (robot):

- **Task:** to move a robot to a ball, make it pick it up and bring it to the goal.
- **Input:** a grid with a robot, ball, and goal at various locations on that grid.
- **Output:** the same grid with the robot and ball at the same location as the goal.
- **Standard DSL:** {MoveRight, MoveDown, MoveLeft, MoveUp, Drop, Grab, AtTop, AtBottom, AtLeft, AtRight, NotAtTop, NotAtBottom, NotAtLeft, NotAtRight}

ASCII art (pixel):

- **Task:** to move to the relevant grid locations and draw pixels there.
- **Input:** an empty grid, can be thought of as a canvas.
- **Output:** the same grid with some pixels drawn; the ASCII art.
- **Standard DSL:** {MoveRight, MoveDown, MoveLeft, MoveUp, Draw, AtTop, AtBottom, AtLeft, AtRight, NotAtTop, NotAtBottom, NotAtLeft, NotAtRight}

4.5 Evolved versus Standard Language

In this section, we compare the DSL evolved by the genetic algorithm and the standard DSL. For these experiments, the DSLs were evolved on a subset of all the available tasks. To measure their performance, they were then tested on a different subset with similar tasks. We compared the evolved and standard DSLs on three criteria: average search time, percentage of tasks solved, and average created program length. We also observed if the complexity of the tasks had an impact on the DSLs created. The results from these experiments answer **RQ4** and thereby also the main research question.

Search time

We start with arguably the most important results, the average search time. For each task, the timer starts the moment the synthesiser starts looking for programs and stops when it finds a task-solving program or hits the time limit. As can be seen in Figure 5, the search time was reduced significantly in all domains.

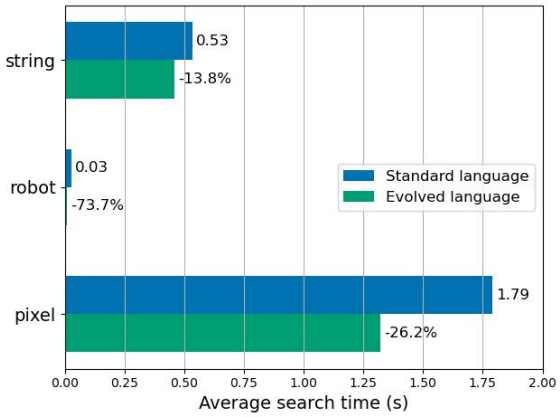

Figure 5: Standard vs evolved DSL average search time

For the string domain, the genetic algorithm evaluated 646 possible DSLs. The best candidate, when compared to the standard language, had five tokens (AtStart, IsLowercase, IsNotLowercase, IsSpace, and IsNotSpace) taken out. For some, it is easier to reason why than for others. For instance, taking out AtStart seems logical, especially when you have NotAtStart that can be used in a LoopWhileThen-token. Since then you can specify to move the cursor to the left until you are at the start and you never need to verify whether you are actually at the start. Besides this, knowing whether letters were lowercase or not was deemed not important, probably because we already have tokens to check whether letters are uppercase. The same is true for knowing whether a symbol is a space or not, which is probably unnecessary because we can already check if something is a letter or number.

The most significant reduction in search time was observed in the robot domain. Out of the 601 candidates explored, the best candidate had some significant changes. Namely, all the observe-tokens were taken out, keeping only the transform-tokens (Drop, Grab, MoveDown, MoveLeft, MoveRight, MoveUp). Apparently, the creation of LoopWhileThen-tokens was not important enough to keep any observe-tokens to do so.

For the pixel domain, 540 DSLs were evaluated. The evolved DSL is comparable to the one for the robot domain. It also only kept the transform-tokens (Draw, MoveDown, MoveLeft, MoveRight, MoveUp).

Tasks solved

We wanted to make sure that the evolved DSLs, although faster, did not contribute to a significant loss in the number of tasks that could be solved. As can be seen in Figure 6, there was no loss in any domain.

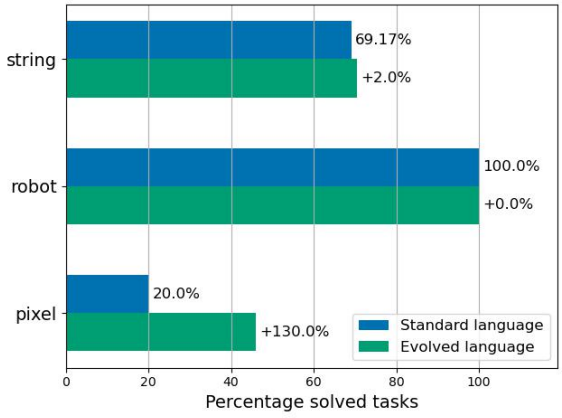

Figure 6: Standard vs evolved DSL percentage tasks solved

In fact, for the string domain, we see a slight increase, and for the pixel domain, more than twice as many tasks were solved using the evolved DSL. The increase in the number of tasks solved is inherently correlated to the search time. This is because of two reasons. First, as the average search time is lower, it is more likely a task-solving program can be found within the set time limit. Second, unsolved tasks are attributed a search time that is equal to the time limit. This means the search time is skewed to the time limit as fewer tasks are solved.

Program length

An interesting observation can be made with regard to the program length. As you can see from Figure 7, synthesis with the evolved DSL creates longer programs in every domain. This indicates that fewer LoopWhileThen-tokens are created. For the robot domain, and even more so for the pixel domain, the differences are significant. For both, it is a direct result

of not having the observe-tokens. For the string domain, each task often has multiple examples for which a general program must be found. Therefore you would expect more tokens that can be used in a general context, like LoopWhileThen-tokens, instead of very specific sequences of primitive tokens.

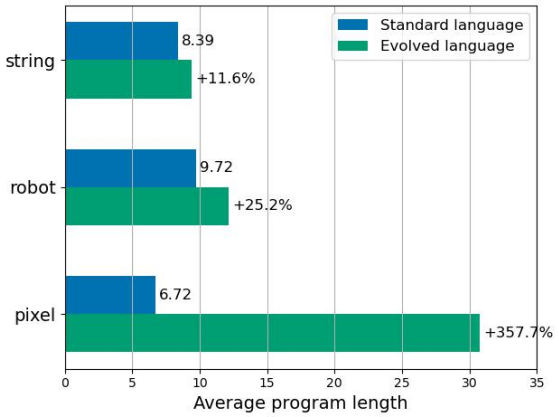

Figure 7: Standard vs evolved DSL program length

Evolving for specific complexity

In general, the tasks were grouped based on complexity. Of course, for more complex tasks, the search times were higher. Furthermore, fewer of these tasks were solved, and the created programs were bigger. One could also imagine that for more complex tasks, a different DSL would be evolved. For instance, for the robot domain, a more complex task has a bigger area for the robot to traverse. Perhaps, because it allows the synthesiser to create LoopWhileThen tokens, this would result in the observe-tokens being in the optimal language. This is because a LoopWhileThen-token could reduce the probability of the synthesiser needing to find a very long sequence of the same tokens. In practice, this was not the case. The DSLs that were evolved specifically for very easy and very complex tasks were almost completely identical. From this, we can conclude that the benefit of the ability to create a LoopWhileThen does not outweigh the increase in size of the search space resulting from these extra tokens.

Composite tokens

In some cases, DSLs with composite tokens allowed for faster program synthesis than just using the standard DSL. However, the optimal DSL for each domain did not have any composite tokens. Therefore, it could be that DSLs with these composite tokens outperformed the standard DSL despite having composite tokens, instead of because of it. For each domain, we discuss why this could be.

For the robot and pixel domain, the reason is similar. Take for instance the robot domain. To search a program for this domain, a loss function was used that took into account the distance from the ball to the robot and from the robot to the goal. When a synthesiser is trying to solve a task, it builds up a sequence of tokens and evaluates which token to add next, based on which minimises the distance to the next target. Only for some steps in some tasks will a LoopWhileThen token decrease this loss function. Namely, when the next target location is very close to an edge of the grid. In contrast, one of the primitive tokens always decreases the loss, since going in the right direction always minimises the distance to the next target.

For the string domain, it is a bit more complicated. A task in the string domain has multiple inputs and outputs for which a general program must be found. This is why DSLs for this domain need to be able to create the general LoopWhileThen tokens. However, apparently, the same configuration of this token is not used frequently enough for it to be worth it to add to the DSL. This means that not having to rediscover this token does not weigh up against the increase in search space, and thus search time.

5 Conclusion

Evolving a domain-specific language (DSL) can make inductive program synthesis more efficient. Applying an evolved DSL to a program synthesis task can lower synthesis time and can increase the probability of a task being solved within a certain time limit. We have shown that a genetic algorithm works adequately to discover which language predicates contribute to lower or higher search times. The programs created during program synthesis had some frequently used snippets, in the form of composite predicates. Introducing these composite predicates as new language predicates sped up program synthesis in some cases. We have shown that, for the three domains we used, the optimal languages did not use these composite predicates.

6 Future Work

In this section, we make two suggestions for future work. First, during each run of the genetic algorithm, many pointless DSLs were evaluated which contributed heavily to the total runtime. For some domains, some of the language predicates are essential. For instance, when the task is to draw something, the predicate 'Draw' is essential. Without these predicates in the language, no task will be solved. Still, the search algorithm would attempt to find a program until the time limit is reached. To reduce the probability of evaluating these pointless DSLs, one could use weight-based mutation. The genetic algorithm could keep track of individual predicates that contribute to solving many tasks fast. Then when mutating a DSL, removing these predicates would be discouraged, while adding them would be encouraged.

The second suggestion has to do with a certain predicate type. Control predicates control the execution of one or more primitive predicates. The control predicates used for creating loops were found frequently in successful programs, but the control predicate for if-statements was never found. These unnecessary control predicates increase the search space very significantly, just like primitive predicates. One could try to evolve a language that is made up of all the available primitive and control predicates. It would be interesting to see if this could be used to further speed up program synthesis.

7 Responsible Research

The best research is reproducible, credible, and complete. In this section, we highlight some of the positives of our research regarding these factors. However, to provide full disclosure, we also identify some issues and propose potential solutions.

7.1 Reproducibility

All the experiments can be run through the published codebase [23]. Instructions on how to run the experiments can be found in the README file. In case one wants to experiment with different parameters this can be easily done, and the experiments are designed in a way that allows this. This involves parameters for program synthesis, the genetic algorithm, and how results are displayed.

The experiments were run on the DelftBlue supercomputer [20]. For this, we used a 'compute' node: an Intel Xeon node which parallelised the workload over 48 cores with 4 GB of memory. For program synthesis tasks, we use a time limit for finding a task-solving program. Using the same time limit on a system with more or less performance might result in more or less solving programs being created before that limit is reached. This may also impact the results for the average search time. To mitigate this, ideally one would run the experiments on the same supercomputer using the same configuration. However, as one might not have access one could also tune the synthesis time limit down or up depending on whether they solve more or fewer tasks than indicated in this research.

7.2 Credibility

Many of the experiments involve random factors. Ideally, to exclude this randomness completely from the results you would run the experiments many times and take the averages of the results. However, as the experiments take many hours if not days, more time is needed to achieve this.

Also in the interest of time, interdependencies between parameters were estimated to decrease the number of total experiments needed to approximate optimal results. However, these interdependencies are often very hard to estimate accurately. To mitigate this, an interdependency analysis should be done that is substantiated by actual data. This would still decrease the total amount of experiments to run and would result in more accurate optimal results.

7.3 Complete Results

To prevent the duplication of work as much as possible, researchers should publish all the relevant results they found, not only the positive ones. During this research and while showcasing the results this was kept in mind. Domains and metrics for which the methods used had little to no impact were still included. Also, when methods, in general, did not provide great results, this was still documented. An example of this is the technique of adding composite predicates to the language.

7.4 Acknowledgements

I would like to thank my supervisor Sebastijan Dumančić and my peers Nicholas Efthymiou, Lucas Kroes, Michal Okon, and Fabian Radomski for their collaboration on this research. Lastly, I want to thank the Delft High Performance Computing Centre for allowing us to use the DelftBlue supercomputer [20] for this research.

References

[1] S. Gulwani, O. Polozov, and R. Singh, "Program Synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[2] C. David, D. Kroening, and M. Lewis, "Using Program Synthesis for Program Analysis," in *Logic for Programming, Artificial Intelligence, and Reasoning* (A. Davis Martin
}and Fehnker, M. Annabelle, and V. Andrei, eds.), (Berlin, Heidelberg), pp. 483–498, Springer Berlin Heidelberg, 2015.

[3] S. Gulwani, "Automating String Processing in Spreadsheets Using Input-Output Examples," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, (New York, NY, USA), pp. 317–330, Association for Computing Machinery, 2011.

[4] E. Kitzelmann and U. Schmid, "Inductive synthesis of functional programs: An explanation based generalization approach," *Journal of Machine Learning Research*, vol. 7, 2006.

[5] E. Kitzelmann, "Inductive programming: A survey of program synthesis techniques," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5812 LNCS, 2010.

[6] S. Gulwani, J. Hernández-Orallo, E. Kitzelmann, S. H. Muggleton, U. Schmid, and B. Zorn, "Inductive programming meets the real world," 2015.

[7] Y. Korukhova, "An approach to automatic deductive synthesis of functional programs," in *Annals of Mathematics and Artificial Intelligence*, vol. 50, 2007.

[8] M. Hofmann, E. Kitzelmann, and U. Schmid, "A unifying framework for analysis and evaluation of inductive programming systems," in *Proceedings of the 2nd Conference on Artificial General Intelligence, AGI 2009*, 2009.

[9] N. Dershowitz and U. S. Reddy, "Deductive and inductive synthesis of equational programs," *Journal of Symbolic Computation*, vol. 15, no. 5-6, 1993.

[10] A. F. Subahi, "Cognification of program synthesis—a systematic feature-oriented analysis and future direction," *Computers*, vol. 9, no. 2, 2020.

[11] A. Cropper and S. Dumancic, "Learning large logic programs by going beyond entailment," in *IJCAI International Joint Conference on Artificial Intelligence*, vol. 2021-January, 2020.

[12] M. Melanie, *An introduction to genetic algorithms*, vol. 32. 1996.

[13] J. H. Holland, "Genetic Algorithms," *Scientific American*, vol. 267, no. 1, pp. 66–73, 1992.

[14] K. A. De Jong, *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, Ann Arbor, Mich., 1975.

[15] X. Wang, I. Dillig, and R. Singh, "Program synthesis using abstraction refinement," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, 2018.

[16] C. Smith and A. Albarghouthi, "Program synthesis with equivalence reduction," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11388 LNCS, 2019.

[17] S. Dumančić and A. Cropper, "Knowledge Refactoring for Program Induction," 2020.

[18] S. Kok and P. Domingos, "Statistical predicate invention," in *ACM International Conference Proceeding Series*, vol. 227, 2007.

[19] K. Ellis, C. Wong, M. Nye, M. Sablé-Meyer, L. Cary, L. Morales, L. Hewitt, A. Solar-Lezama, and J. B. Tenenbaum, "DreamCoder: Building interpretable hierarchical knowledge representations with wake-sleep Bayesian program learning," *arXiv*, 2020.

[20] D. H. P. C. Centre (DHPC), "DelftBlue Supercomputer (Phase 1)," 2022.

[21] B. Jenneboer, "Program Synthesis with A*," tech. rep., TU Delft Electrical Engineering, Mathematics and Computer Science, 2022.

[22] F. Azimzade, B. Jenneboer, N. Matulewicz, S. Rasing, and V. Wieringen van, "BEP Program Synthesis Systems Code ." https://github.com/victorvwier/BEP_project_synthesis, 2022.

[23] N. Efthymiou, L. Kroes, M. Okon, F. Radomski, and P. Tempelman, "BEP: Evolving a Program Synthesiser ." https://github.com/FabianRadomski/EvolvingProgramSynthesisers, 2022.