GENERALIZE: A framework for evolving searching constraints for domain-specific languages in program synthesis

L.G. Kroes
Supervisor(s): Dr. S. Dumančić
EEMCS, Delft University of Technology, The Netherlands
18th June, 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

In this paper, we propose a method for eliciting constraints for arbitrary Domain-Specific Languages (DSL) in Program Synthesis search. We argue that we can successfully predict constraints using a form of attribute-based induction. We also provide a novel approach to constraint verification using genetic algorithms to optimize desired results. We implement our approach into *GENERALIZE*, a novel algorithm for reducing DSL size. *GENERALIZE* is tested and compared against the default Brute algorithm using 2 different program synthesis domains, robot planning and pixel art. These experiments show that *GENERALIZE* does not improve performance if good objective functions are available, because of a tendency to get stuck in local heuristic minima. It can increase performance if no such function is available.

1 Introduction

Program synthesis is the automatic generation of software from some specification. While various methods of program synthesis exist it can often be framed as a search problem over the space of possible programs. Using program synthesis can help people without programming experience to automate tasks, and it can help programmers to define program behaviour instead of program implementation.

The issue with program synthesis is that finding correct programs is often costly. The size of the search space of possible programs grows exponentially in regards to the program size [1]. Significantly reducing the size of the search space can therefore drastically increase search speed of existing algorithms.

Program synthesis solutions are often framed within the context of a Domain-Specific Language (DSL), a set of functional atoms that encode relevant actions for the specific problem type. In this work, we will propose a system to reduce the effective size of such DSLs by learning constraints from binary relations between member functions. By applying these constraints we attempt to restrict the size of the solution space of possible programs.

Programming by Example (PBE) [1] is a sub-field within program synthesis that defines the intent of the program using example input-output pairs. Within this paper, we will be working primarily with the Brute [2] search algorithm. Brute is a PBE synthesizer that uses an example-dependent loss function, also called an objective function, to evaluate partial programs. This means that unsuccessfully partial programs can be rated on how close they are to the expected solution. This technique makes it possible to learn programs up to 20 times larger than previous high-performance methods [2].

Other works have been done regarding constraints in program synthesis. Previous works have relied on the concept of observational equality reduction [3, 4], which is computed during the run-time of the synthesis algorithm. Other constraint systems designed for Inductive Logic Programming synthesizers [5] use the concept of entailment (whether a program provides the correct solution for some example) to remove partial programs that are either too general or too specific [6]. Since all these systems compute constraints during program search, there is a lack of generality concerning the search algorithm used and important information is lost after a solution is found.

To address these problems, we propose *GENERALIZE*, our novel method for pre-processing and deriving candidate constraints from DSLs. GENERALIZE finds function pairs where function-ordering does not matter, or function pairs that invert each other. From this it derives constraints on what potential solutions can look like. This approach allows us to significantly reduce the domain size before run-time.

The main goal of this paper is to explain the workings of the *GENERALIZE* system. Its main goals are to:

- Generate constraints for arbitrary problem domains as opposed to arbitrary problem instances
- Provide an optimal constraint allocation for arbitrary search conditions
- Apply these constraints in a computationally feasible manner

Our main insight is that candidate constraints can be determined observationally. Using a genetic algorithm a constraint assignment can be made that provides a complete and near-optimal system.

In this paper, we provide the following contributions: A short overview of previous works (Section 2), a motivating example for our work (Section 3), *GENERALIZE*: a novel algorithm for determining constraints for arbitrary DSLs, that uses a set of training examples to determine constraints (Section 4), a benchmark of *GENERALIZE* using different synthesis search algorithms and settings (Section 5). After this there is a conclusion with inspiration for future work (section 6). The last section focuses on ethical issues with AI, and reproducability of this work.

2 Related work

Program Synthesis: As mentioned in the introduction program synthesis is the problem of finding (or "synthesizing") a function based on some specification. To further specify this problem three elements are needed in the problem specification; The grammar G which specifies what a solution can look like; The search method S which specifies in what way we look through the space of possible solutions; and finally the intent specification P, which determines how we validate whether a program satisfies the intended specification.

Observational equality reduction: Observational equality reduction is a general method for reducing the size of the problem domain. [3, 4] It theorizes that two programs P and Q can be considered equivalent if they evaluate to the same output as any continuation from that state will be the same.

While Observational equality reduction has been shown to significantly reduce search time in synthesis [3], the size of the pruned space only increases as more of the problem space is uncovered. Furthermore, pruning can only be considered after program evaluation, while the static nature

of GENERALIZE means that the ways a state can be reached are limited from the start of program search.

Generalizations and Specialisations: Another method of pruning the search space is using generalizations and specialisations. This method uses mathematical subsumption relations between various candidate solutions to determine whether a possible program is either more general than a program that is already too general or more restrictive than a program that's already too specialised [6]. This same method is also used by Padmanabuhni [7] in the constraint elicitation process, where generalizations and specialisations are used to synthesize constraints for constraint satisfaction problems.

Unfortunately this is only a valid method in the context of Logic Programming, where functions are formulated as logical clauses. In our problem setting functions are formulated as mathematical functions and therefore this method cannot be applied.

Learning constraints: In previous work in the area of learning constraints, a framework for eliciting constraints using an inductive method in the context of Explicit Constraint Satisfaction Problem (ECSP) is provided [7]. To apply this in our context, some background knowledge about constraints in the DSL is needed. While *GENERALIZE* is effective at synthesizing this background knowledge, more specialised generalization methods could be derived from the mathematical properties of the derived constraints.

Genetic Algorithms: Genetic algorithms (GAs) are a subset of evolutionary algorithms that take inspiration from natural processes (mostly for efficient function optimization) [8]. GAs specifically take their inspiration from the way genes are passed around in nature, to determine a stochastic process for optimizing arbitrary functions [9]. GAs generally have a population of chromosomes, that encode some arbitrary entry of the solution space. Within these chromosomes, different genes encode the state of different features within the search space.

GAs use a fitness function to determine the "reproduction chance" of an individual chromosome. The genetic algorithm then takes one or two of these chromosomes to produce "offspring": chromosomes that inherit some properties from their parents. this offspring also has a chance to randomly mutate some of its values. finally, this new generation is evaluated and the process repeats until some finishing condition is met.

In this paper GAs will be used for computing the optimal constraint allocation to optimize search in a specific domain. GAs are used in this case not neccessarily because of the size of the search space, but because the computational cost of evaluating a single candidate-solution is very high. The efficient search nature of GAs is therefore needed.

3 Motivating example

To illustrate both the need and the functionality of *GENERALIZE* it is convenient to consider an example application as thought experiment.

Suppose that there is a robot that can move around on a board as in figure 1. This robot can move up, down, left and right. This robot can also pick up and drop off a ball. Now imagine that we set the end goal for this robot to pick up a ball, and drop it off at some drop off point. The goal of *GENERALIZE* would be to achieve this goal in as little steps as possible, and to designate as many possible paths as infeasible before executing any moves.

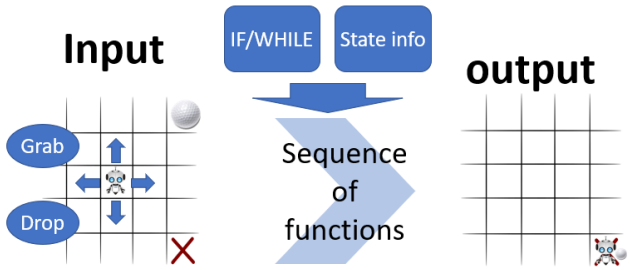

Figure 1: The ultimate goal of program synthesis: finding a sequence of functions to transform some input to some output state

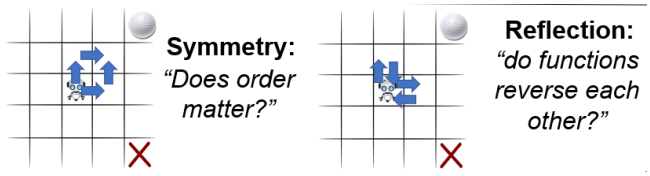

Figure 2: The ultimate goal of *GENERALIZE*: finding sequences that have no added benefit in program search

The main question *GENERALIZE* has to answer, is what options, or sequences of moves, we can remove from the space of possible move sequences without sacrificing the ability to get to any arbitrary state on the board. There are 2 such properties that we can easily identify in this example, and they are displayed in figure 2. For the symmetry property it is obviously irrelevant in which order the robot executes the moves right and up, as long as both functions are executed the robot will end up in the same problem state. The same goes for reflections; if the robot first moves right and then left, the robot will end up in the exact same state as it started. Therefore this is also a sequence that can arbitrarily be avoided in any good solution.

But why is it relevant to move a robot along an imaginary board? The honest answer of course is that this is irrelevant. The process of finding functions that map inputs to outputs, as performed in this thought experiment, is a very important issue to solve. With this functionality we can theoretically create any arbitrary function that transforms inputs to outputs. Imagine the possibilities in active areas of research in computer science, like classification and other machine learning or data wrangling.

4 Methodology

In this section, we will discuss our contribution to the problem of program synthesis. First we will discuss the mathematical framework of our constraint system. We will discuss the mathematical properties we are looking for, and the constraints we can derive from them, as well as combining these

Token Grammar:
start := sequence
sequence := token | token sequence
token := control | invented
invented := trans | trans trans
control :=
    if (bool) (invented) (invented) |
    while (bool) (invented) |
    while-then (bool) (invented) (invented)
trans := Domain-specific
bool := Domain-specific

Figure 3: Description of the program synthesis grammar

simpler constraints into more powerful variants, and finally we will discuss how we apply these constraints in a program synthesis setting. After this we consider how to generate a set of candidate constraints using observational techniques. Finally a system is proposed for automatic constraint validation and optimization.

4.1 tokens and environments: a brief introduction to our synthesis framework

Program synthesis can be done in various ways, using various different methods. The specific method can have a large impact on how the program functions as a whole, and more importantly, what constraints can be applied within the program.

The main component of program synthesis is the solution grammar; the way a potential solution is represented. Within our framework there are two important parts to this representation: the way the problem state is encoded (*environments*) and the way the solution is encoded (*tokens*). An example of an environment can be seen in figure 1, where two instances of an environment are shown graphically. Figure 1 also shows instances of tokens, encapsulating the various possible actions discussed in section 3. Lastly figure 1 hints at the use of higher level control tokens, like if and while. The complete grammar is presented in figure 3. From this we can derive and enumerate a space of all possible solutions that we can then search using Brute [2].

Finally we can evaluate programs found by the search algorithm using some intent specification mechanism. In our case intent is specified using a set of input-ouput environments, where the intent is to find a program that maps all inputs to all outputs. This evaluation is done using an objective function, which not only checks whether the input is mapped to the correct output, but also how far off from the right solution the program is.

The final concept of importance within our framework is how tokens are modelled. Tokens are seen as a function of $environment \rightarrow environment$. They transform the state of the environment into another state where the token is executed. Within our framework, transitive tokens are stateless. They will always produce the same effect on any input state.

4.2 Relational constraints in program synthesis

To show how to derive constraints for just the transitive tokens, consider figure 2. Two cases are shown from which these constraints can be derived. If two functions inverse each other (Defintion 1), it is logical not to avoid solutions that put those two functions in sequence. Secondly we have functions where order does not matter (Definition 2).

If we want to reduce the size of the search space, it makes sense to only allow for one possible sequence of functions to achieve a state, so therefore it is reasonable to only allow one symmetry: Either first go up and then right, or first right and then go up. Only one of these sequences is necessary to keep a complete system.

Definition 1 (reflective tokens): Tokens that inverse each other: t1, t2 s.t. t1(t2(e)) = e

Definition 2 (independent tokens): Tokens where ordering does not matter: t1, t2 s.t. (t1(t2(e)) = t2(t1(e)) (independence)

These properties are practical because from them we can derive the following constraints:

Definition 3 (complete constraint): Constraint such that iff t1 and t2 are inversions of each other, disallow t1 after t2 and t2 after t1.

Definition 4 (partial constraint): Constraint such that iff t1 and t2 are independent of each other, either disallow t1 after t2 or disallow t2 after t1.

4.3 Deriving higher dimension constraints

These constraints by themselves are not very effective at reducing the size of the search space, as each constraint only sporadically decreases the amount of possible tokens to be added. The real reduction factor is achieved by grouping binary constraints into higher-arity constraints concerning multiple tokens.

To provide a framework for deducing these higher-arity constraints we can consider the case of the robot domain, and it's movement tokens. One can independently verify that, as long as no invalid transition is enacted, the ordering of a series of movement tokens does not matter because all these tokens are independent of each other. As a consequence of the definition of the independence property Theorem 1 holds, as any ordering can be rewritten to some main ordering of tokens (for a more formal proof, see appendix A).

Theorem 1. *If we have a set of tokens T=t1, t2 .. tn where each token is independent of all other tokens, any chain of tokens C where holds* $\forall t_i \in C$, $t_i \in T$ *can be arbitrarily permuted without changing the outcome of the function.*

Since any ordering of a set of independent tokens is equivalent, reducing the problem size comes down to implementing

an ordering constraint only allowing a single permutation to occur within the search problem.

Theorem 1 also has consequences for the complete identity constraints. Since any permutation of independent tokens is equal, if an inversion property were to hold between two of these tokens as well, that means that in some permutations these tokens would be next to each other. According to the definition of the identity property, the entire sequence can be rewritten by excluding these two tokens.

Theorem 2. *If we have a set of tokens T=t1, t2 .. tn where each token is independent of all other tokens, and a chain of tokens C where holds* $\forall\ t_i, t_j \in C,\ t_i, t_j \in T$. *Consider two arbitrary tokens* $t_p,\ t_q \in T,\ t_p, t_q \in C$ *invert each other. In this case shorter equivalent sequence C* of C can be constructed by removing pairs of* t_p, t_q *tokens.*

As a consequence of theorem 2 it is obvious that in a sequence of only movement tokens, if both move_left and move_right tokens appear, a shorter equivalent sequence can be by removing move_left and move_right pairs, as these cancel each other out.

4.4 Applying constraints

A final issue with this constraint system is that due to library token invention we are not dealing with just transitive tokens. Modern synthesis systems invent larger and more complex library tokens [2] to increase synthesis speed using if and while statements. This is a problem because for some parts of the functionality of GENERALIZE it is necessary that the functional output is the same for every environment. This does not rhyme with the nature of conditional logic introduced by if and while statements.

Constraints as state machines

To provide some insight into the nature of the problem at hand, it is useful to reason about it analogous to a state machine (SM) that takes as input a program sequence, and outputs a set of constrained tokens. Consider the two constraints presented as state machines presented in figures 4 and 5, derived from the property graphs shown in figures 7 and 8. In these two figures we encode the output of tokens to be constrained in the current state. Each token encountered in the input program sequence corresponds with taking the edge labelled with that note in the SM. If no such edge exists that means that the sequence is invalid and should have been constrained.

To explain why this model works for the constraints let us first point out some patterns in the two SMs.

First of all both SMs contain a "reset" edge, activated by tokens that are not independent of all other tokens. It is obvious from theorems 1 and 2 that tokens for which no independence relation holds break up the possibilities of constraining tokens and therefore reset the tokens to be constrained.

Secondly, all states in the state machines only have one specific token leading to them. For the complete constraints this makes sense as after going right once, the goal is to never allow going left until some non-independent token is used. For the partial constraints this is a method for ensuring that symmetries are all sequenced in one specific way. In this case: first all rights, then all lefts, then all downs, then all ups. Note that partial constraints do not care about the behaviour of complete constraints, and therefore have no issue with placing a right after a left.

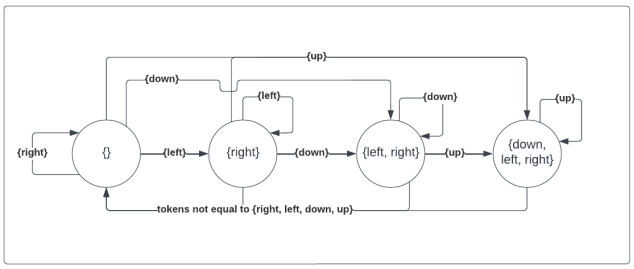

Figure 4: Partial constraint {right, left, down, up} modelled as state-machine

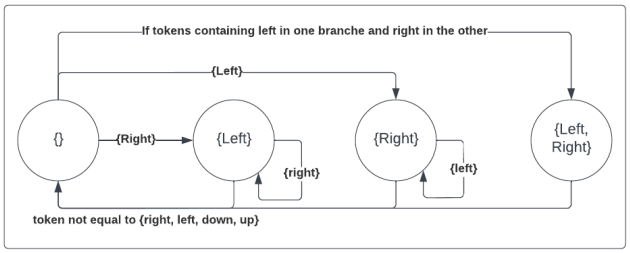

Figure 5: Complete constraint {right, left} within {right, left, down, up}-clique modelled as state-machine

Generalizing to complex tokens

Now we have derived a method for applying constraints in a simple context where no control or invented tokens exist. However as figure 5 already suggests, these constraints need to be applied in those contexts as well. To this end we will provide a method for applying partial- and complete constraints to if-tokens, and discuss how this generalizes to while- and while-then-tokens.

Calculating the constraint state for if-tokens can be done by running the SM on both branches of the statement in parallel. After this is calculated we can then combine these state machine's by choosing the least-permissive state as end state.

The least permissive state for a partial constraint is obvious. It is simply the state out of both branches that constraints the largest amount of tokens. For complete constraints the least permissive state is calculated as follows: the {left}- and {right}-state are both less permissive than the {}-state, however if one branch is in the {left} state and the other in the {right} state the states are combined to the {left, right} state blocking both lefts and rights.

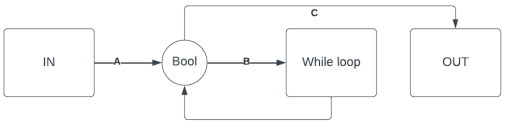

Figure 6: Transitions in a while-token

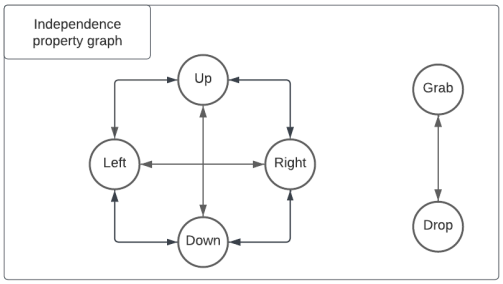

Figure 7: Modelling independence property relations between tokens as a graph problem for functions in the robot domain

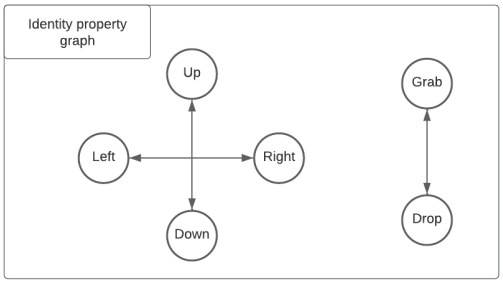

Figure 8: Modelling independence property relations between tokens as a graph problem for functions in the robot domain

For while tokens the same method can be applied. To this end we need to cover every possible transition-sequence in figure 6 once for every constraint state. This means we need to compute two constraint SMs in parallel. One that takes the transition from A to C directly, skipping the while loop entirely, and one that takes the transition from A to B, from B to B and then from B to C, effectively looping twice. This method covers all possible transitions within the token. A similar method can be constructed for the while-then token

Using this method a set of constraint state pre-conditions can be calculated for each token. We have a method for transforming the constraint state for a token in constant time, which means that we can check a sequence of tokens for validity in linear time. Since partial solutions are explored more than once, we can bring this down to expected constant time using memoization techniques for partial programs.

4.5 Generating the constraint space

Our approach for generating the space of possible constraints consists out of 3 stages. First *GENERALIZE* generates all binary properties that could possibly hold for a given *DSL*. Then *GENERALIZE* tests this set of properties against a set of test *Environments* to see which properties hold observationally. Finally *GENERALIZE* derives associated constraints from these properties, and abstracts these constraints to higher arities using combination rules derived from theorem 2 and 1.

The first step of the *GENERALIZE* algorithm generates a set of all candidate binary properties P^*. The second step then uses a set of test environments E^* to select the set of properties that hold observationally for those environments.

A key realisation is that not all possible properties can be evaluated using this method. One example of such a property is found in the robot domain: an independence property between Left and Grab.

These functions can only be performed in one specific order (if any) for all input environments. If the robot is on top of the ball it can only grab and then move left. If the robot is to the right of the ball, it can only move left and then grab. There exists no environment where both execution orderings are possible. This property is therefore considered absurd

Theorem 3. *a candidate property $p*$ is absurd in relation to a set of test environments E^* iff*
$\nexists\, e \in E^*$ *s.t.*
$p* .holdsFor(e) = True$ *or* $p* .holdsFor(e) = False$.

The concept of absurdity is of some importance when pruning the constraint space. There are many combinations of properties that are infeasible to test as they can never be meaningfully executed in a problem instance.

The last part of the constraint space generation is combining constraints to higher-arities, When considering the consequences of theorem 1, one can observe that if the constraints are modelled as edges in a graph as done in figure 7, where the tokens they apply to are seen as nodes. Finding the largest possible groups of tokens that have an independence relationship becomes an instance of the clique problem which we can use available methods to solve for [10]. These same cliques can be used to propagate complete constraints that fall within those groups as well (figure 7).

4.6 Evolving the optimal constraint allocation

In this section we will further elaborate on the use of genetic programming within *GENERALIZE*. First we will discuss why the constraint generation procedure as discussed in the previous section is not enough to provide an adequate constraint system. Then we will discuss how *GENERALIZE* implements a genetic algorithm to search over the provided constraint space.

Why further optimization is needed

Even though using carefully selected test environments to determine the constraints to be used within GENERALIZE can be sufficient to discern a correct constraint-set, especially for extensive or opaque domains it is harder to determine whether all properties are derived by the algorithm are valid. To automate the issue of constraint-set verification, *GENERALIZE* employs a genetic algorithm to search over the space of possible constraints and find the optimal allocation.

The use of the genetic algorithm has the added benefit that it optimizes the form of constraints for the problem set at hand. If e.g. the ordering of a partial-constraint matters to the performance of a synthesis algorithm the genetic algorithm can also be employed to find the optimal ordering of this partial constraint.

A last benefit of using a genetic algorithm is the fact that properties that do theoretically hold but do not improve the performance of the synthesis algorithm are also detected and removed from the final constraint set. This is because in evaluation the genetic algorithm only considers the performance

of the synthesis algorithm, not the actual validity of these properties.

Chromosomes and mapping function
The genetic algorithm used within *GENERALIZE* is of a rather simple design. The constraints derived in generating the constraint space are encoded using an integer. For complete constraints these genes can take either a 0 (disabled) or 1 (enabled). For partial constraints these genes can take either 0 (disabled), 1 (permutation 1) or 2 (permutation 2). Only after a selection is made in the genetic algorithm the higher order constraints are derived.

Constraint cliques as described before are only determined after a selection has been made. This means that the genetic algorithm does not inherently have control over the token order of these higher-arity constraints. In order to provide this control some mapping should be made so that the state of higer-arity constraints can be controlled by the algorithm.

Since the amount of permutations in a partial constraint are a product series in nature ($\prod_{k=0}^{n} k$ where n is the constraint size), and the amount of constraints within a clique are a sum series in nature ($\sum_{k=0}^{n} k$) a function needs to be designed to map the product-series to the sum-series.

This function has only one prerequisite: all possible permutations of the higher-order constraint should be mappable from the binary constraints. To this end we created a definite ordering of constraints, and created a binary string from the first k partial constraints where 1 mapped to 0, and 2 mapped to 1. In this case k was the amount of tokens in the higher order constraint. If this binary string resulted in a higher result than the number of permutations available our implementation rounds down to the highest possible value in the permutation list.

Crossover and mutation
The crossover method used in the selection algorithm was a form of uniform crossover, where every gene of a chromosome is randomly inherited from either parent with 50% chance for either parent. This crossover operator was chosen because there are no placement dependencies between different genes in the chromosome. The mutation method is also simple in design. Every gene in every chromosome has a chance p to mutate in every generation of the genetic algorithm. this chance p, in the algorithm defined as the $mutation\_chance$ is configurable at the start of running the genetic algorithm.

fitness
Within the genetic algorithm fitness is calculated as follows: a test set of problems is ran using a small training set of synthesis problems. The actual fitness of the algorithm is then calculated as follows:

$$fitness = \frac{1}{\sqrt{d_{max} \cdot d_{avg} \cdot s_{avg}}} * 100 - fitness_{normal}$$

Here d is the distance produced by the objective function, and s is the average run time of the algorithm and $fitness_{normal}$ is the fitness of the algorithm without any constraints enabled. This is done for two reason: in order to make sure that any constraints or constraints set that are worse than no constraints are not included.

5 Experimental Set-up and Results

Before evaluating the results of the *GENERALIZE* algorithm we will first discuss the data and settings used to test the set-up in various circumstances. We will first discuss the datasets we used both taken from [2]. After this we will discuss the set-ups we will be using to test *GENERALIZE*, and finally we will show some graphs with the outcomes of *GENERALIZE*.

5.1 Verification data

Robot domain: The first domain we will be using is a set of problems where a robot on an nxn grid needs to navigate towards a ball located on the board, grab it, and drop it at some goal location. To that end, there are various actions that the robot can undertake: The robot can move up, down, left and right on the board, and the robot can grab and drop the ball. There is also some information about the robot's location made available to itself: namely, whether the robot is at the top, bottom, left or right of the board, along with their logical inverse. The domain contains multiple tasks, each task entailing only one example. That means that the synthesiser can find hardcoded solutions instead of solutions that generalize to similar problems.

Pixel domain: The second domain provided for verification is the pixel domain. This problem domain concerns writing out certain patterns on an nxn-grid. To this end, the program synthesizer has a pointer located on the grid. The synthesizer can move this pointer up, down, left and right, and can draw on any location on the board. The state information made available to the synthesizer is similar to that of the robot domain. This problem domain also only has tasks containing singular examples, which causes the same possible issues as discussed with the robot domain.

5.2 Experiment set-up:

For the experiment we have a set-up where we will be running Brute using the three testing domains as described before with 3 different objective functions specific to the domain. The 3 objective functions for each domain are:

Robot Domain: Greedy distance metric (manhatten distance between current and desired robot and ball position + whether the robot is holding the ball) (G), optimized step amount (absolute amount of steps needed to go to the desired state) (O), entailment (whether a state and the desired state are equal) (E)
Pixel Domain: Hamming distance (hamming distance between drawn pixels) (G), Hamming + movement distance (hamming distance + movement distance between pixels) (O), entailment (whether a state and the desired state are equal) (E)

We will run the *GENERALIZE* algorithm in 2 stages using each combination of these settings on a small training set.

First we will train the genetic algorithm on a test set of 10 problems, with a timeout of 0.1s and mutation rate of 0.1 to promote local search. The bias factor was disabled for both domains when using the "O" objective function as no constraint allocation performed better then no constraints. After training is complete we will then verify the generated constraints by running the constrained- and normal search algorithm on a large test set of 90 problems for the robot and pixel domain with a variety of time-out settings (0.1, 0.5, 1, 10).

5.3 Results

As seen in figures 10 and 9 a variety of constraint allocations were computed, and the form of that constraint allocation is in many ways reliant on both the objective function used and the domain.

Generated constraints:

The robot domain with entailment (E) includes almost all available constraints, excluding only a partial constraint between right and down, and a complete constraint between grab and drop. With the Greedy distance function (G) only a small subset of constraints were chosen, interestingly enough stopping the robot from going left or right after going up. For the optimzed steps (O) all constraints were included. This is remarkable as in verification the unconstrained algorithm was vastly superior to the constrained algorithm.

The pixel domain with entailment (E) includes a smaller set of constraints. with Hamming distance (G) all constraints available were applied. With hamming + movement distance (O) as function again a large subset of constraints were applied.

In general the constraint set for both domains with entailment are random in nature. For neither domains any of the training problems were solved in the evolution process, so the eventual allocation was decided by the total run time of the problems in which many induced randomness factors exist.

For the "G" distance function for the pixel domain the genetic algorithm had a clear preference for chromosomes with more constraints. For the robot domain however the chromosomes with only a few constraints performed best in general.

for the "O" distance functions no chromosome performed better than no constraints in training as well as evaluation. Because of this the bias factor in the fitness function was disabled in testing. Chromosomes with only a few constraints still performed worse than chromosomes with many constraints causing the algorithm to find a local minimum with many enabled constraints.

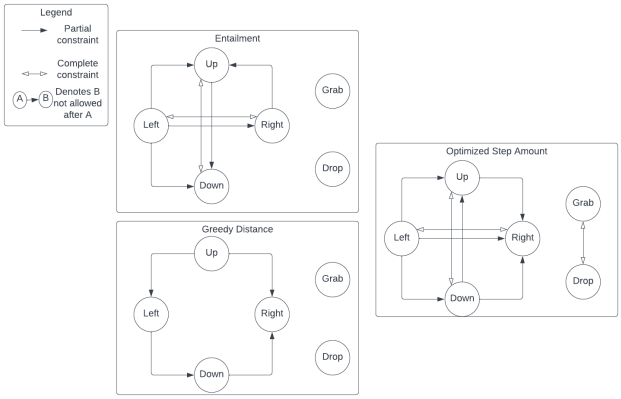

Figure 9: A graph representation of the evolved constraints for the robot domain for various objective functions

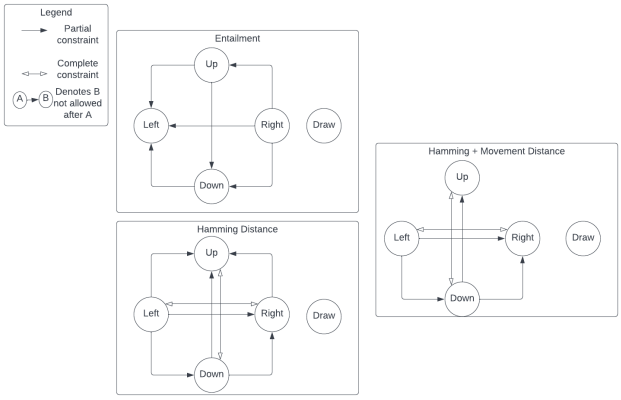

Figure 10: A graph representation of the evolved constraints for the pixel domain for various objective functions

Evaluation results:

The summarized results of evaluation of the constrained and normal system is shown in figures 11 and 12. The data shows a big overall decrease in performance when using GENERALIZE with "O" functions, with an almost 70% decrease in accuracy for hard problems in the robot domain and a slight increase performance is shown in the robot domain with Entailment with 15% more . Domains with "G" heuristics perform on average slightly worse.

These results seem to indicate that something is wrong with the approach taken in this paper. To understand this consider the visual in figure 13. Of the 4 partial programs shown only 1 of those programs is still able to get to the goal in the constrained system. In the final version of *GENERALIZE* no way of checking whether the end goal is still achievable is implemented. Because the heuristic value of the partial program (located in the upper right corner of the squares) is still high while no actual solution can be achieved. Brute is spending large amounts of time searching local optima. This also explains why this problem is not encountered in entailment domains, as those local optima do not exist.

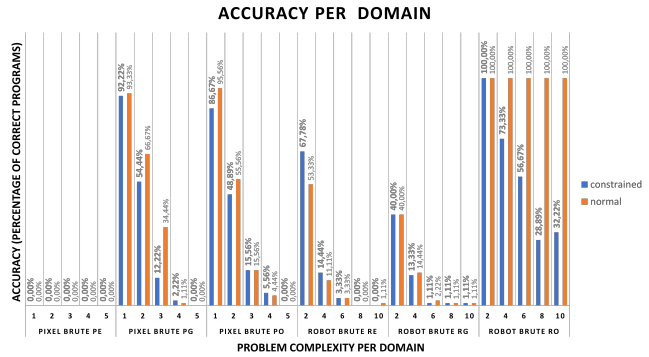

Figure 11: problem solving accuracy data for various domains and settings using a timeout of 10 seconds

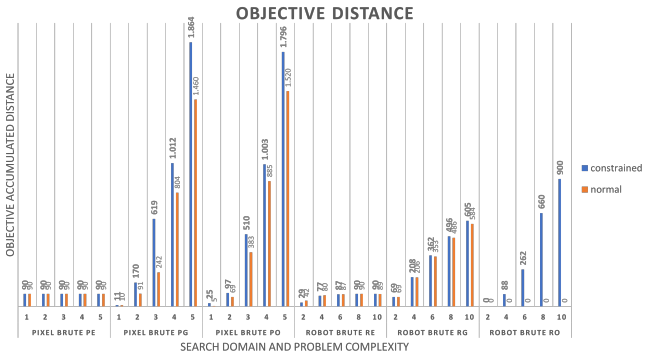

Figure 12: Objective distance data for various domains and settings using a timeout of 10 seconds

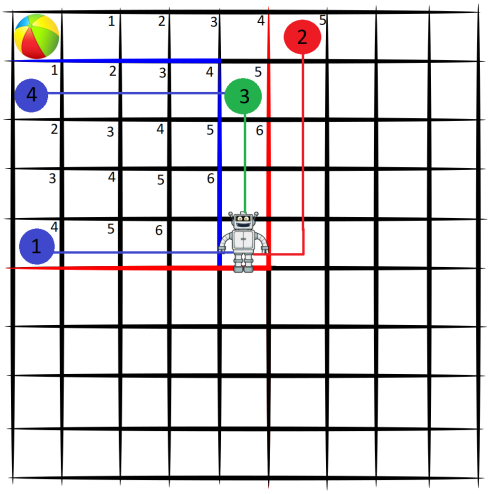

Figure 13: A visual representation of how the robot is constrained, with any solution ending up on the other sides of the blue (partial constraints) or red (complete constraints) lines being unable to get to the final solutions.

6 Conclusion

To summarize our contributions: In this paper we have shown some possible constraints to be applied in a program synthesis context (4.2), provided a method for generating higher order constraints from a set of smaller constraints (4.3), shown that these constraints can also be applied in a more complex context (4.4), shown a method for learning these constraints using a small training set (4.5) and finally we have automated validation of the learned constraints using a genetic algorithm (4.6). The results of this work have been evaluated in section 5.

From our results we can conclude that arbitrarily reducing the search space does not work. In general the constraints should cause the search algorithm to find more successful than unsuccessful programs when compared to the unconstrained system, and sadly *GENERALIZE* has achieved the opposite in this setting.

This does not mean *GENERALIZE* has no merit at all. The results achieved on the entailment domain, do signify some measure of speed up. If a constraint aware heuristic could be used in brute or if a method for calculating whether a program still has a continuance to a solution (completely removing the red and blue zones in figure 13) speed improvements could still be substantial.

The constraint system itself does limit expressive power to some extent. Consider the function $WhileThen\ \ (NotAtBottom)\ \ (Down)\ \ (Up)$ which moves the robot down to the bottom of the board and then moves up one row. This is a very expressive method of getting to the second to last row. With complete constraints enabled between Down and Up however, this function will always be constrained causing a loss in expressivity.

Using *GENERALIZE* with a different constraint elicitation method could also have a vast impact on performance. The reason the results are negative is mainly because the underlying constraint system decreases performance. If a constraint elicitation technique were to be used that provides more effective constraints results could improve.

GENERALIZE might also perform better with search methods less prone to stalling on local minima. More research could be done on *GENERALIZE* with probabilistic methods like MCTS or Genetic Programming. Some care should be taken however in how the constraints are implemented in these probabilistic search procedures.

A last method for increasing the observed performance is by better tailoring genetic training problems and -settings to fit the specific domain. No chromosome in either entailment domains were able to solve a single training example. In contrast almost all chromosomes in the robot domain with ”O” function were able to solve all training examples. If more diverse tasks and a longer search time was possible in training, problems with reaching local minima could potentially be signalled in training instead of only in validation.

7 Responsible Research

Within this paper the best possible effort was made that results were reproducible and verifiable and that the work was ethical. For the sake of reproducibility and verifiability we

will first discuss the research code used and the dataset produced by it. After this we will briefly touch on the subject of ethical application of AI.

7.1 Reproducibility and Verifiability

All code used for this paper can be found on github[1]. For generating the dataset used for validation, CMain.py was ran. This file takes 3 arguments as input. First a designation for the search algorithm, then P, R or S for running the pixel, robot or string domain respectively. For the last argument G, O or E designates with which objective function the domain should be ran with.

Trials within our program synthesis framework are not completely deterministic. The fact that a time-out was used means that the accuracy is influenced both by hardware and circumstance. In order to still verify that the data summarised in this paper is represented accurately the entire dataset used is included on the github page.

The dataset was generated on the DelftBlue Supercomputer [11]. Both training and verification happened on an Intel XEON E5-6248R 24C 3.0GHz processer with 4GB of RAM. Each combination of domain and objective function ran on their own processor.

7.2 Ethics and AI

Program synthesis is a subfield of AI. AI provides many very powerful techniques and opportunities to solve large problems if done correctly. AI also provides many ways of making problems worse if used improperly. When applying any Artificial Intelligence techniques, using *GENERALIZE* or else, proper validation should be done before and while applying it to situations that affect people. Just like people, AI is in no way infallible and systems should always be in place to correct faulty decisions discovered after the fact.

References

[1] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Foundation and Trends in Programming languages*, vol. 4, pp. 1–119, 2017.

[2] A. Cropper and S. D. Dumančić, "Learning large logic programs by going beyond entailment," 2020.

[3] A. Albarghouthi, S. Gulwani, and Z. Kincaid, "Recursive program synthesis." Springer, 2013, pp. 934–950.

[4] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur, "Transit: specifying protocols with concolic snippets," *ACM SIGPLAN Notices*, vol. 48, pp. 287–296, 2013.

[5] S. Muggleton, "Inductive logic programming," *New Generation Computing*, vol. 8, pp. 295–318, 1991.

[6] A. Cropper and R. Morel, "Learning programs by learning from failures," *Machine Learning*, vol. 110, pp. 801–856, 5 2021. [Online]. Available: http://arxiv.org/abs/2005.02259

[7] S. Padmanabuhni, J.-H. You, and G. Aditya, "A framework for learning constraints: Extended abstract," 1996, pp. 1–13.

[8] J. H. Holland, "Genetic algorithms," *Scientific American*, vol. 1, pp. 66–73, 1992.

[9] S. Sivanandam and S. Deepa, "Genetic algorithms," pp. 15–37, 2008.

[10] E. Tomita, A. Tanaka, and H. Takahashi, "The worst-case time complexity for generating all maximal cliques and computational experiments," *Theoretical Computer Science*, vol. 363, pp. 28–42, 10 2006.

[11] Delft High Performance Computing Centre (DHPC), "DelftBlue Supercomputer (Phase 1)," https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1, 2022.

[1] https://github.com/FabianRadomski/EvolvingProgramSynthesisers

Appendix A: Proof of theorem 1

Proof.

1. Let 2 arbitrary functions f and g be independent s.t. $f(g(x)) = g(f(x))$
2. Observe that any function f is independent of itself as $f(f(x)) = f(f(x))$
3. Consider a set of functions $F = f1, f2 \dots fn$ s.t. $f1 \dots fn$ are all independent of each other.
4. Consider an application chain $C* = f1(f2(..(fn(x)))$
5. To prove: Any permutation of the application chain $C*$ will produce the same result, and is therefore equivalent
6. Proof by construction: Without loss of generality take an arbitrary permutation of $C*$ $C = fi_1(fi_2..fi_n(x))$.
7. Take the function in C corresponding to f1. If it is applied as the last function in C move to the next function, else let f_j denote the function applied after $f1$.
8. Since $f1$ and f_j are independent we can rewrite it as applying f_j first and then f1 (line 1) thus constructing C' with one inversion.
9. Keep doing this until f1 is the last function applied, thus giving $C' = f1(fi_1(fi_2..fi_n(x)))$
10. Do the same for f2 until f2 is the second to last function applied.
11. Keep doing this until every function is in the same position as in C*
12. Since we can construct C* from an arbitrary permutation C any permutation C of C* is equivalent to C*.

□