TECHNISCHE UNIVERSITEIT DELFT

MASTER OF SCIENCE THESIS IN COMPUTER SCIENCE

Constraint Propagation in Program Synthesis

Bart SWINKELS
B.J.A.Swinkels@student.tudelft.nl

Supervisors:
Dr. Sebastijan DUMANČIĆ
Tilman HINNERICHS

21st June 2024

Delft University of Technology

Constraint Propagation in Program Synthesis

Master's Thesis in Computer Science

Algorithmics group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Bart Swinkels

21st June 2024

Author
Bart Swinkels

Title
Constraint Propagation in Program Synthesis

MSc presentation
4th July 2024

Graduation Committee

Dr. E. Demirović (chair)	Delft University of Technology
Dr. S. Dumančić	Delft University of Technology
Dr. C. Poulsen	Delft University of Technology
Dr. S. Verwer	Delft University of Technology

Abstract

Program synthesis is often seen as the holy grail of computer science. A user only needs to provide program specifications and a computer will automatically generate the desired program. This often involves searching for the desired program in the program space, which is like searching for a needle in a haystack. Luckily, there is room for improvement here. Many programs in the program space are redundant and should never be considered. To filter out these redundant programs, we propagated grammar constraints during search using a novel built-in constraint solver. Only programs that satisfy these constraints will be considered as candidates for synthesis. To measure the effectiveness of the solver, we have enumerated several program spaces with and without constraints. Although the amount of filtering largely depends on the concrete grammar and constraints, the results show that constraints can often eliminate 99% of the program space. Accounting for the overhead of propagation, this comes down to a 50-fold improvement in runtime.

Preface

This thesis is the final step in obtaining my Master of Science degree in Computer Science at Delft University of Technology. After earning my bachelor's degree in Mathematics, I realized that what I enjoyed the most were courses related to coding. I love implementing ideas and bringing them to life through code. To pursue this passion, I applied for the computer science program and searched for a thesis project with a large engineering component.

My supervisors, Sebastijan Dumančić and Tilman Hinnerichs, had an open project in their program synthesis framework `Herb.jl`. They entrusted me with their codebase and gave me total freedom to implement my ideas. They also organized "hackathons", in which all Herb developers collaborated to work on Herb. These events were a fun way to learn the framework and meet the other developers (and eat pizza too!). Working on this framework has been a valuable experience, as this was the first large project I have ever worked on. Besides helping me with coding, I would like to thank my supervisors for their feedback, ideas, and patience during our (sometimes unusually long) meetings.

I also want to take this opportunity to express my gratitude to my friends and parents for supporting me outside of the thesis. In particular, I want to thank my roommates for distracting me with games, movies, and most importantly, good talks. Additionally, I want to thank my friends Sid and Thomas, whom I met through computer science, for joining me on campus when my motivation was low. And since with this thesis, I am concluding my time as a student, I would also like to take this moment to thank my friends from the bachelor's program, Max and Fos, for all the good times we had throughout our studies.

I am very grateful to have met you all!

Bart Swinkels

Delft, The Netherlands
21st June 2024

Contents

Chapter 1

Introduction

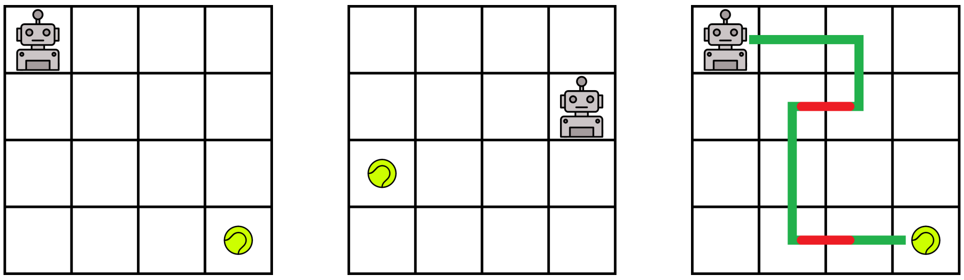
We are living in a technology-driven world, where the demand for software applications is higher than ever before. However, the ability to program remains a specialized skill possessed by a limited number of individuals. This creates a gap between the growing need for software solutions and the available workforce capable of fulfilling these demands. In response to this challenge, program synthesis emerges as a promising approach to bridge this gap. It allows developers to create programs without the need to write them manually. A program synthesizer only needs a description of the intended program behavior and will then automatically synthesize the desired program.

Compilers have been around for decades and can be seen as the first form of a program synthesizer. Expressing programs in modern programming languages allows developers to quickly construct programs without having to worry about low-level implementation details. Program synthesis takes this process to the next level by allowing users to describe intent in non-imperative ways. For example, by providing specifications, input-output examples, or even natural language [11].

A popular approach to program synthesis is the Programming by Example (PBE) paradigm [7, 8]. In this approach, the synthesizer aims to find a program that satisfies a set of user-provided input-output (IO) examples. The synthesizer also needs to be provided with a Context Free Grammar (CFG) [1] that holds the syntax of the target language of the program. Then, programs of a given language are enumerated and tested on the IO examples until one is found that satisfies all the examples.

The main challenge of program synthesis is the enormous amount of possible programs to consider [8]. Luckily, there is room for improvement here. Many programs are syntactically correct according to the grammar, but semantically redundant. We can save a lot of time by filtering these redundant programs out of the enumeration. For this purpose, we will introduce a notion of *constraints* to program synthesizers.

As a running example, consider a simple environment where a robot can move around a grid and can grab and drop balls. An IO example is defined as the state of the environment before and after executing the program, like in Figures 1.1a and 1.1b. The program space consists of all possible paths the robot can take and can be drastically reduced by imposing constraints on the grammar. For example, a constraint could ensure the robot never takes detours. Such a constraint will prevent programs like the one depicted in Figure 1.1c from ever being considered.



(a) Input (b) Output (c) Candidate Program

Figure 1.1: Figure (a) and (b) form an IO example for the robot environment. Figure (c) illustrates a valid, yet semantically redundant, candidate program. The red parts of the program represent an unnecessary detour.

Other program synthesizers, like POPPER [3], offload the propagation of constraints to an external solver. Such synthesizers use an iterative framework where a generator proposes a candidate program, a tester checks the program against examples, and the constraint mechanism generates new constraints to guide the search in the next iterations. In particular, the generation and constraining of programs can be performed by established constraint solvers, such as Satisfiability (SAT) and Constraint Programming (CP) solvers [6], by utilizing the well-researched search mechanisms included in these solvers. Existing solvers, however, were not created for program synthesis. The encoding of a grammar and semantics of the language would become very large and complex, making this approach sub-optimal for solving.

A new program synthesis framework `Herb.jl`[9], is under development at the Delft University of Technology. It aims to propagate the constraints of a grammar within the framework itself, instead of offloading it to an external SAT solver. The advantage of a native solver inside the synthesizer is that it can (1) exploit the tree-shaped nature of a grammar and (2) deal with new decision variables during search.

The main contribution of this thesis will be a tailor-made constraint solver for `Herb.jl` that leverages well-tested practices used in CP solvers. The related research question is:

Which propagation techniques from the constraint programming paradigm can be leveraged to propagate constraints in program synthesis problems effectively?

This project serves as a solid baseline for a program synthesis constraint solver that other developers can easily extend and optimize.

Chapter 2

Background Knowledge

In this chapter, we will look more closely at what program synthesis is and how it can be implemented. Then, we will review constraint programming (CP), a paradigm for solving combinatorial problems. In the remainder of the thesis, we will combine these two fields by implementing constrained program synthesis.

2.1 Program Synthesis

Program synthesis is the task of constructing a program in a target language, according to a provided description of the program's desired behavior. [8, 7]. The main contribution of this thesis is to implement a constraint solver for the program synthesis framework `Herb.jl`[9], so we will look at program synthesis through the scope of this related work. Like many program synthesizers [8], Herb can be sub-divided into three components (See Figure 2.1):

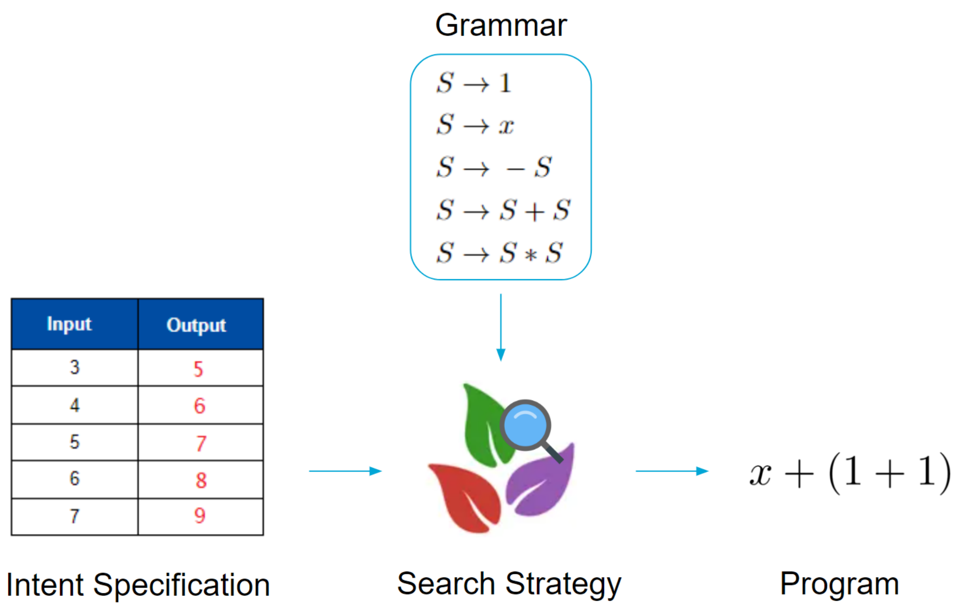
1. Intent specification. What should the program be able to do?
2. Program space. What kind of programs can syntactically be constructed?
3. Search strategy. How to traverse program space?

In the upcoming three sub-sections, we will go over each of these components in more detail and use the *Robot Environment* as a running example.

Example 1 (***Robot Environment***). States in the robot environment are defined by an $n \times n$ grid, the coordinates of a ball, the coordinates of a robot, and a boolean to indicate if the robot is currently holding the ball. An IO example defines the state of the environment before and after the execution of a program. Programs in the environment can be constructed using the CFG in Figure 2.3.

2.1.1 Intent Specification

To synthesize a program, it is crucial to specify what the target program intends to do. One way to achieve this is to provide input-output (IO) examples that should



Input	Output
3	5
4	6
5	7
6	8
7	9

Figure 2.1: Programming by Example (PBE). Programs in the program space are tested against the intended behavior and then a satisfying program is returned.

be satisfied by the target program. This is often referred to as Programming by Example (PBE).

Programming by Demonstration (PBD) [4] is an extended version of PBE. In this form of intent specification, IO examples are extended with an arbitrary number of intermediate states. These intermediate states provide additional information and can help the synthesizer to find a valid solution earlier and/or break symmetries. Such an extended IO example is called a *trace*. Figure 2.2 depicts an example of a trace in the robot environment. As of now, PBD and traces are not yet supported in Herb, but it would be a great extension.

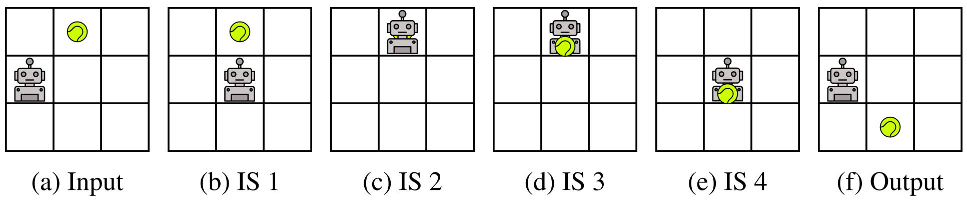


Figure 2.2: A trace, an IO example extended with intermediate states (IS).

2.1.2 Program Space

The synthesizer needs to generate a program in a target language. To realize this, languages are described using a Context Free Grammar (CFG) [1]. A CFG consists of the following 4 elements:

- Terminal symbols. A terminal symbol is a piece of code that is interpretable in the target language. (e.g. `state = initialState;` or `move- Right(state);`). Such pieces of code may also contain non-terminal symbols that need to be recursively expanded into terminal symbols. For example `if(C) T else T` expects that symbols `C`, `T` and `T` will be replaced with some terminal symbols.

- Non-terminal symbols. Non-terminal symbols represent a hole in a program that could be filled using any of the corresponding production rules. In the grammar depicted in Figure 2.3, the non-terminal T represents any tail of code and can be expanded using either a return statement (rule 2) or an operator (rule 3).

- Production rules. A production rule transforms a non-terminal symbol into a terminal symbol, along with an ordered sequence of non-terminal symbols as children. In Figure 2.3, rule 3 transforms the non-terminal T into a statement terminal with two non-terminal children O and T. Note that there can exist multiple production rules that transform the same non-terminal symbol. In section 2.1.3, we will see how a Herb decides which production rule to use.

- Starting symbol. The non-terminal symbol that is used to create any valid program in the target language.

All the programs that can be constructed using a CFG form the *program space*. The main issue of program synthesis is that program spaces are exponentially large. Searching this space for a program is therefore a very time-consuming and often practically impossible task.

Fortunately, it is possible to reduce the size of the program space by imposing constraints on the grammar, making it a Context Sensitive Grammar (CSG). We wish to add constraints to the grammar that reduce the program space, without removing the target program. Most of these constraints aim to break symmetries in the grammar. That is, remove equivalent alternatives of partial programs from the program space. For example, consider the following 3 programs:

1. `moveLeft(state);`
2. `moveLeft(state); moveRight(state); moveLeft(state);`
3. `if(C) moveLeft(state); else moveLeft(state);`

They are semantically equivalent programs, but they are all part of the program space. We are only interested in the shortest variant of these equivalent programs:

$$
\begin{array}{rrcl}
1: & S & := & \text{state = initialState; } T \\
\\
2: & T & := & \text{returnState(state);} \\
3: & T & := & O; T
\end{array}
$$

$$
\begin{array}{rrcl@{\qquad\qquad}rrcl}
 & & & & 12: & C & := & \text{atLeft(state)} \\
4: & O & := & \text{moveLeft(state)} & 13: & C & := & \text{atRight(state)} \\
5: & O & := & \text{moveRight(state)} & 14: & C & := & \text{atTop(state)} \\
6: & O & := & \text{moveUp(state)} & 15: & C & := & \text{atBottom(state)} \\
7: & O & := & \text{moveDown(state)} & 16: & C & := & \text{notAtLeft(state)} \\
8: & O & := & \text{grab(state)} & 17: & C & := & \text{notAtRight(state)} \\
9: & O & := & \text{drop(state)} & 18: & C & := & \text{notAtTop(state)} \\
10: & O & := & \text{if(}C\text{) } T \text{ else } T & 19: & C & := & \text{notAtBottom(state)}
\end{array}
$$

Figure 2.3: Production rules of a CFG for the robot environment. This grammar contains 4 non-terminal symbols: S for start, T for tail, O for operation and C for condition. All the other symbols are terminals and are valid syntax for the interpreter. The starting symbol is S.

`moveLeft(state);`. Therefore we can impose constraints that forbid the other patterns from arising anywhere in the program space. This has two advantages: (1) we reduce the size of the program space and (2) we will find a more concise program.

Herb can be used to automatically detect symmetries in a grammar [5]. This is done by first adding a special kind of terminal called a *pattern variable* to the grammar. Then, a number of subprograms is generated. These subprograms are compared by repeatedly assigning different values to pattern variables. Two subprograms are considered to be part of the same *equivalent class* if they produce the same output on all assignments. After grouping the subprograms into equivalence classes, one representative program will be selected to remain in the program space. For each of the other programs of the equivalence class, a constraint is generated to prevent the pattern from arising anywhere in the program space.

2.1.3 Search

Once the intent specification and program space are known, the next step is to find a program that satisfies the specification. The search component of Herb uses a top-down enumeration of all the programs (see Algorithm 1). For each candidate program provided by the enumerator, the evaluator checks if the program satisfies the input-output examples. If it does, it is deemed a solution, otherwise, it is ignored. We will now look into the enumeration of the candidate problems.

Herb starts the enumeration by taking the starting symbol of the grammar as the root node of the program tree. Nodes corresponding to non-terminal symbols are called *holes*. This name indicates that it is not a concrete part of the program yet, and needs to be expanded using a corresponding production rule. Any program tree with one or more holes, is called a *partial program* and needs to be repeatedly expanded until there are no holes left, making it a *complete program*. When expanding a partial tree, a hole is selected using a heuristic. Then the hole is expanded with a valid production rule from the grammar. When a hole has n valid production rules, n copies of the program tree will be made, and for each of the copies, the hole will be filled using a different production rule, making it a concrete *rule node*. Then, the copies are stored in a priority queue of programs that need to be considered in future iterations. A simple, yet effective, priority function could be the size of the program, as we ideally wish to find smaller programs before larger programs. This is called a breadth-first search (BFS).

Algorithm 1 Program Enumerator: Enumerates program trees by iteratively filling holes with rule nodes.

```
root ← RuleNode(starting_symbol))
pq ← PriorityQueue(root)
while len(pq) > 0 and !stopping_criteria_met() do
    tree ← pq.pop()
    if tree.complete then
        yield tree
    end if
    hole ← hole_heuristic(tree)
    rules ← derivation_heuristic(hole.domain)
    for rule ∈ rules do
        new_tree = fill_hole(tree, hole, rule)
        propagate_constraints(new_tree)
        pq.enqueue(new_tree, priority_function(new_tree))
    end for
end while
```

As mentioned before, a grammar can have a set of constraints attached to it. These constraints are propagated whenever a new node is added to a tree, as can be seen in Algorithm 1. Propagators can check if a tree is consistent with the constraints and potentially remove impossible rules from the domains of holes. In the upcoming section (Section 2.2), we will review constraint propagation in constraint programming. Then, we will apply these techniques to implement constraint propagation in program synthesis in Chapters 4 and 5.

2.2 Constraint Programming

Constraint Programming (CP) is the practice of formulating and solving Constraint Satisfaction Problems (CSP) [16, 10, 12]. The objective of a CSP is to find an assignment for a given set of variables that satisfy a given set of constraints. Constraint programming has a wide amount of applications such as timetabling, resource allocation [13] and vehicle routing [14]. In the upcoming sections, we will review constraint programming techniques, so we can re-apply them for the purpose of program synthesis.

2.2.1 Specification

A Constraint Satisfaction Problem consists of three components: a set of decision variables X, their respective initial domains D, and any number of constraints C.

A decision variable $x \in X$ is a variable that can take values in its corresponding domain $D(x) \in D$. Whenever there is only 1 value a variable can take, that is $|D(x)| = 1$, we say a variable is *fixed*. A CSP is considered solved if all variables are fixed.

A constraint $c \in C$ can restrict the initial domains of variables. For example, we may include a constraint $c = (x > y)$ to enforce that x must always be larger than y.

$$\begin{aligned} X =& \{x, y, z\} \\ D =& \{D(x), D(y), D(z)\} = \{\{1, 5, 7\}, \{2, 3, 5\}, \{-1, 1, 3, 5\}\} \\ C =& \{x \geq y, z = x - y\} \end{aligned} \tag{2.1}$$

Equation 2.1 is an example of a CSP formulation with 3 decision variables and 2 constraints. Since this is a small example, we can solve it upon inspection and find a solution: $(x, y, z) = (5, 2, 3)$. Note that in this case, the solution is not unique and one may also find the second solution: $(x, y, z) = (7, 2, 5)$. Both are equally valid. If the user wishes to eliminate a solution, the problem should be more strictly constrained. Alternatively, the problem could also be extended to an optimization problem by adding a objective function $f(X)$ [10]. In that case, only the solution that, WLOG, minimizes the objective function will be returned.

2.2.2 Solving

Solving a given CSP is split up into two main components:

1. Propagation. Check for constraint violations and filter out impossible values from domains.

2. Branching. When no filtering can be done, split the problem into multiple sub-problems with smaller domains such that any solution of the original problem can be found in the union of the sub-problems.

Propagation

We will now review how constraint propagation is implemented in `Mini-CP` [10], an uncluttered CP solver for education purposes. Propagation is responsible for removing impossible values from domains according to the constraints. This process will continue until the constraints are unable to further reduce any of the domains. This is called a *fixed point*: continuing propagation has no effect. The fix-point algorithm 2 propagates a queue of scheduled constraints. [1]. The queue only contains constraints that can possibly make any deductions. That is, we skip constraints that are known to be currently unable to shrink any domains. Note that the queue may increase in size during a propagate method, as some constraints that were unable to make deductions, may be able to do so after another propagator made some deductions before it.

Algorithm 2 Fix-point

```
$Q \leftarrow$ PriorityQueue(constraints)
while len(Q) $> 0$ do
    propagate(Q.dequeue())
end while
```

Constraints have post and propagation methods.

Post. The post method is executed whenever the constraint is first imposed upon the problem. This is where the initial propagation of a constraint takes place and where constraints usually add themselves to notify lists of their corresponding variables. This is implemented using the observer design pattern. For example, the "$x \geq y$" constraint c will add itself to the `onBoundChange` notify list of both the x and y variables. Then, whenever the bounds of the domain of either x or y change, c will be scheduled for propagation.

Propagate. The propagate method tries to shrink the domains of its related decision variables using a constraint-specific filtering algorithm. For example, the "$x \geq y$" constraint may update the domains of x and y like so:
$D(x) := \{x \geq \max(D(y)) \mid x \in D(x)\}$.
$D(y) := \{y < \min(D(x)) \mid x \in D(y)\}$.
This update can trigger the scheduling of other constraints. For example, if the

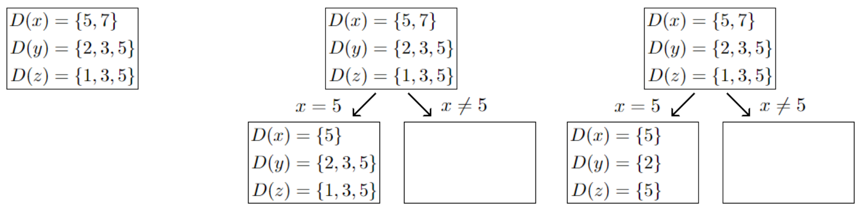
[1]Ordering the constraints by impact using a heuristic can significantly boost the performance of a solver [16]

bounds of $D(x)$ changed, constraints in the `onBoundChange` notify list of x will be added to the queue.

Branching

After executing the fix-point algorithm, we may not have found a solution yet. That is, there exists some variable $x \in X$ with $|D(x)| \geq 2$, and none of the constraint propagators can further reduce the size domain. In that case, we subdivide the problem into two or more sub-problems such that all solutions of the original problem can be found in the union of the solutions of the sub-problems. In other words, no solution is lost by splitting up the problem.

Figure 2.4 demonstrates how a solver may find a solution for the CSP from Equation 2.1. In state 2.4a, the fix-point algorithm is executed with the initial constraints. $x \geq y$ allows the removal of 1 from the domain of x. No further deductions can be made, so the problem must be divided into sub-problems. In this case, the simple *first-fail* branching scheme is used [10]. For the left branch, a new constraint is posted that fixes the value of the variable with the smallest domain. For the right branch, the negated constraint is posted, assuring that no solution is lost. In Figure 2.4b, the solver decides to search the left branch first and adds the corresponding branching constraint to the list of constraints. This triggers the fix-point algorithm and after execution, a solution has been found (Figure 2.4c). Now the solver can backtrack to the root node to potentially discover more solutions.



(a) Fix-point in the root. (b) Branching. (c) Fix-point in the left child.

Figure 2.4: Solving the CSP from Equation 2.1 using a first-fail branching scheme.

After repeated application of the branching scheme, a tree structure appears. The nodes of this search tree are called *states*. Exploring new states in the search tree is typically done using a depth-first traversal. This search strategy is very memory efficient since only a single state needs to be maintained at a time [10]. Whenever a state is deemed solved or infeasible, we can backtrack through the parent states and impose an alternative branching constraint. Backtracking is done by keeping track of all changes between a parent state and its child and then reverting those changes.

Chapter 3

Related work

Surprisingly, there is no related work that combines program synthesis with a built-in constraint solver. However, there are program synthesizers that deduce constraints based on IO examples to aid the search. In this chapter, we will explore three of such systems.

1. Popper [3], an Inductive Logic Programming (ILP) based program synthesizer that leverages positive and negative examples to more general constraints.

2. Sketch [15], a SAT-based program synthesizer that generates constraints during search using Counter-Example Guided Inductive Synthesis (CEGIS).

3. Neo [6], a SMT-based program synthesizer that generates constraints during search using Conflict-Driven Clause Learning (CDCL).

Table 3.1 provides a quick overview of the differences between Popper, Sketch, Neo and Herb with respect to the type of constraints of each synthesizer. All related works offload constraint solving to external SAT or SMT solvers. This thesis presents a native constraint solver for Herb that can directly propagate constraints within the framework itself, without encoding the problem into SAT formulas.

	Popper	Sketch	Neo	Herb
Learns constraints from failures	**yes**	**yes**	**yes**	no
Supports an initial partial program	no	**yes**	**yes**	**yes**
Language bias (arity constraints)	**yes**	**yes**	**yes**	**yes**
Can fill holes with any valid expression	**yes**	no	**yes**	**yes**
Context-sensitive grammar constraints	no	no	**yes**	**yes**
Has a native constraint solver	no	no	no	**yes**

Table 3.1: Comparing constraints in four program synthesizers.

3.1 Popper

Popper [3] is a program synthesizer for logic programs [2]. Like many synthesizers, popper can be sub-divided into three components:

1. The program intent is given in the form of logical predicates that should either be true or false. These predicates are called positive and negative examples respectively.
2. The program space is defined using Background Knowledge (BK) and a syntactic bias.
3. The search is implemented as an inductive loop of generation, testing, and constraining 3.1.

In the test phase, hypotheses might be rejected by one of the examples. These *failures* will then be converted to a set of constraints that prevent similar hypotheses from being generated in future iterations. Then, in the next generate phase, the newly discovered constraints are passed to a SAT solver to generate new hypotheses.

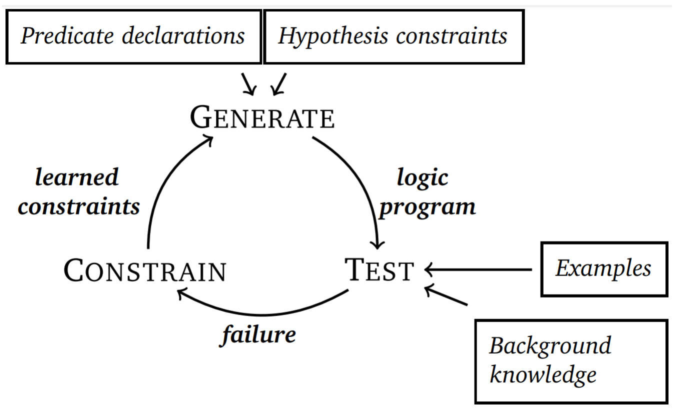


Figure 3.1: The generate, test, and constrain loop of Popper [3].

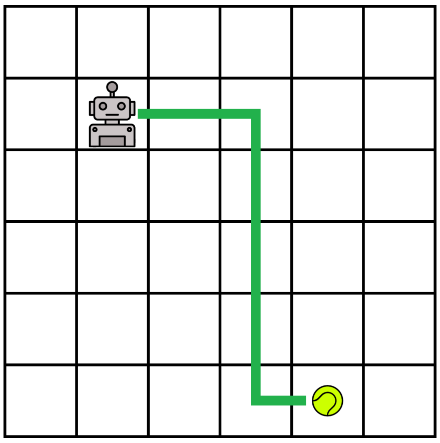
While Popper is great at synthesizing logic programs, it is much harder to effectively synthesize other kinds of programs, such as programs defined by a grammar. In that case, the grammar first needs to be encoded as a logic program system. To compare Herb and Popper, we will consider a logic program system encoding (See Appendix A.1) of the robot grammar from Figure 2.3. Popper can synthesize the hard-coded path depicted in Figure 3.2.

To synthesize the path in Figure 3.2, Popper was given the target positive example, along with a set of negative examples to prevent other paths and guide the search. In addition to the 2 negative examples from Figure 3.2a, Popper was given 100

```
pos(f(w(2,5),w(5,1))).
neg(f(w(1,5),w(4,1))).
neg(f(w(2,5),w(2,4))).

f(V0,V1):-
    move_right(V0,V2),
    move_right(V2,V3),
    move_down(V3,V4),
    move_down(V4,V5),
    move_down(V5,V6),
    move_down(V6,V7),
    move_right(V7,V1).
```

(a) Positive and negative examples and a logic program solution.



(b) Visual representation of the solution.

Figure 3.2: Popper's solution for a problem in the robot environment. The target path is from (2, 5) to (5, 1).

more negative examples. During the search, Popper created 623 constraints that prevent these negative examples from arising. However, these constraints only forbid particular hard-coded paths and do not exploit any higher-level semantics of the robot grammar. In fact, these constraints actually slow down the search procedure, as they need to be converted to SAT clauses without giving any strong inference. In Chapter 6, we will see how Herb's grammar constraints can filter out redundant problems for this environment more efficiently.

3.2 Sketch

Program synthesis by sketching [15] allows users to input a partial program with holes, called a *sketch*, to guide the synthesis. An example of such a sketch for the running robot example can be found in Figure 3.3. Sketches drastically reduce the program space, as candidate programs are restricted to derive from the provided sketch. Herb also supports synthesizing from partial programs, as we will see in Chapter 5.

Just like Popper, program synthesis by sketching is guided by failing examples. The core algorithm used in this search is called Counterexample Guided Inductive Synthesis (CEGIS) [8]. CEGIS uses a validation procedure that can automatically generate edge cases to test a set of candidate programs. Failing tests can be further reduced to a minimum observation set that captures the edge cases with the smallest possible inputs. This observation set is then converted to SAT clauses to constrain the program space of the next iteration of candidate programs. This cycle continues until the validation procedure is unable to generate a failing example.

```
void main(RobotState state){
    repeat (??) {
        state = {| move_down | move_up |}(state);
    }
    repeat (??) {
        state = {| move_left | move_right |}(state);
    }
    return state;
}

```

Figure 3.3: A sketch for a simple hard-coded path for the robot environment. The synthesizer will replace the questions marks with integer values and the regular expressions enclosed in brackets with one of the provided choices.

The counter examples and the provided sketch are the only types of constraints in program synthesis by sketching. It is not possible to constrain the grammar itself nor to fill holes with more complicated expressions other than just integers or the provided regular expression. This means that the synthesizer is highly dependent on the sketch provided by the user.

3.3 Neo

Neo [6] is a program synthesizer with a setup similar to Herb. Candidate program trees are generated according to a provided grammar and should satisfy the provided IO examples. Additionally, each production rule can be provided with semantic specifications to restrict the context in which that rule can be used. These specifications are passed to and propagated by an external SMT solver. Unfortunately, this requires the specifications to be encoded as grounded constraints. In Section 5.6, we will see that constraints presented in this thesis support first-order logic and can be used to forbid a larger class of programs.

Similar to Popper and Sketch, Neo also learns from failed IO examples by identifying and generalizing the root cause of the conflict. These constraints are encoded as SMT formulas and are used by the external solver to find a valid assignment of nodes to production rules.

Chapter 4

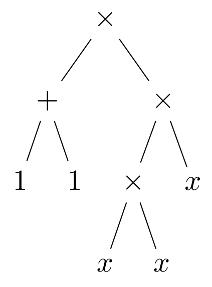
Problem definition

In this thesis, we are going to extend program enumeration with constraints. Recall from Chapter 2 that program enumeration is always with respect to a grammar. This chapter demonstrates the challenges of constrained program enumeration with the simple arithmetic grammar, defined in Example 2.

Example 2 (***Arithmetic Grammar***). A simple integer arithmetic grammar with 5 production rules (see Figure 4.1a). Programs are Abstract Syntax Trees (AST) and represent a simple unary function that takes an input value x. Figure 4.1b is an example of an AST derived from the grammar.

$$\begin{aligned}
1 &: \text{Int} := 1 \\
2 &: \text{Int} := x \\
3 &: \text{Int} := -\text{Int} \\
4 &: \text{Int} := \text{Int} + \text{Int} \\
5 &: \text{Int} := \text{Int} \times \text{Int}
\end{aligned}$$

(a) A simple arithmetic grammar

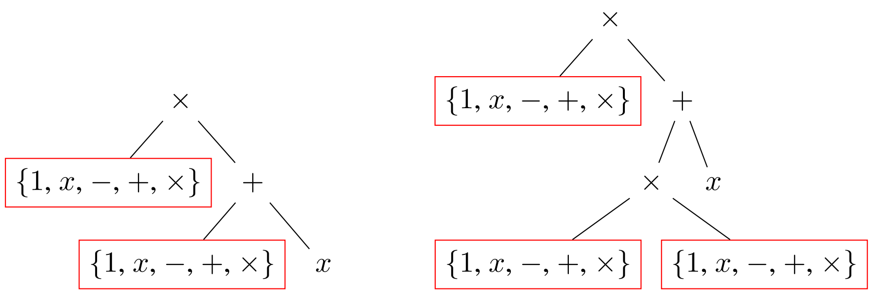


(b) An AST representing $2x^3$

Figure 4.1: A simple arithmetic grammar and an example of a complete program.

A CFG can be extended with *constraints*, making it a Context Sensitive Grammar (CSG). These constraints can eliminate some of the programs a grammar can produce by forbidding specific substructures in the program. In Section 4.3, we will go over some concrete examples of constraints.

Constrained program enumeration in program synthesis can almost be formulated as a CSP. The decision variables in program synthesis are called *holes*. Holes are nodes in a partial program tree that do not have a fixed value, but a domain of values. The domain of a hole contains the possible production rules that a hole can take. Whenever the domain of a hole contains exactly 1 value, a hole is considered *filled*. Filled holes will be referred to as *rule nodes*.



(a) An AST for $?? \times (?? + x)$ (b) An AST for $?? \times ((?? \times ??) + x)$

Figure 4.2: Two partial program trees. Holes in the tree are represented by a red box. The right tree (b) can be derived from the left tree (a) by filling in the bottom hole with an "$\times$". Note that the derived tree contains more decision variables than the original tree.

The major difference with a CSP is that new decision variables can appear during search. Figure 4.2 demonstrates how filling a hole adds two new holes to the set of decision variables. In a CSP, the amount of decision variables must be fixed before solving. This is why program enumeration cannot be formulated as a CSP and why we are developing a new constraint solver.

4.1 Search Nodes

Just like the search procedure in CP, constrained program enumeration consists of two components: (1) propagation and (2) branching. First, the constraints are propagated until no further deductions can be made. Then, the remaining partial program is split up into multiple smaller programs[1]. By repeatedly branching programs into smaller programs, a search tree[2] is constructed.

The leaves of a search tree correspond to all possible complete programs that derive from the grammar[3]. It is possible to check each complete program and discard any that do not satisfy the constraints. However, it would be better to discard any

[1]With *smaller*, we mean that the number of remaining solutions is less than or equal to the number of solutions deriving from the original partial program. Note that the smaller program might be larger in terms of rule nodes/holes.

[2]It is important to differentiate between a search tree and a program tree. The nodes of a *program* tree are rule nodes and holes, while the nodes of a *search* tree are program trees.

[3]In most grammars, this search tree grows infinitely larger. So we usually prune the search tree by setting a maximum program depth and/or size.

partial program that do not satisfy the constraints. By propagating constraints in the inner search nodes, we can prevent entire branches of the search tree from ever existing. This has the potential to exponentially reduce the number of future search nodes to consider (See Figure 4.3).

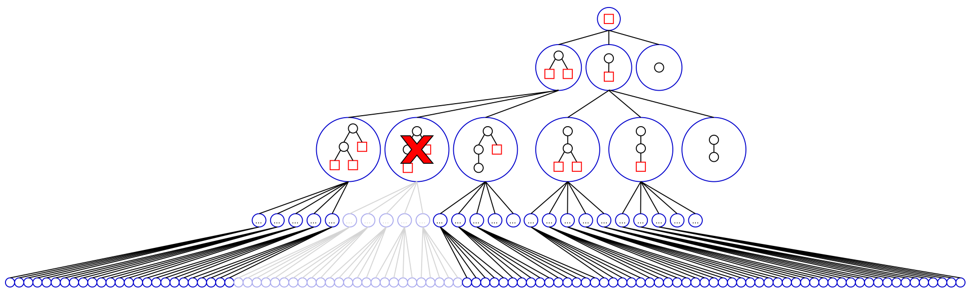


Figure 4.3: Constraint propagation spotted an inconsistency in any inner search node. This prevented an entire branch of the search tree from ever existing.

The number of nodes in the search tree is a good measure of the inference strength of constraint propagation. This is why this thesis aims to reduce the number of search nodes. The related research question is:

How can the number of nodes in the search tree be reduced?

It is clear that constraint propagation is most effective when done as high as possible in the search tree. This means that we should use a branching scheme that allows constraints to make deductions as early as possible.

The fact that new holes can appear during search can hinder constraints in making any deductions. For example, consider the Contains(x) constraint. This constraint enforces that an x should appear somewhere in the program tree. If there is only one hole left with an x and no x is in the program tree yet, this one can filled with an x. Unfortunately, new holes can appear during search, so this constraint typically has to stay dormant for a very long time before it can make any deduction. Figure 4.4a illustrates a partial program where no deductions can be made because the structure of the tree is unknown yet. Even though there is only a single hole with an x, the constraint cannot make any deductions. This is because new holes with an x could appear underneath the hole with non-terminal rules $+$ and $\times$. Only much later in the search (Figure 4.4b), a deduction can be made.

Just like the ”contains” constraint, many constraints can only make deductions if the structure of the program tree is known. This is why we should consider a branching scheme that fixes the shape of the program tree as early as possible.

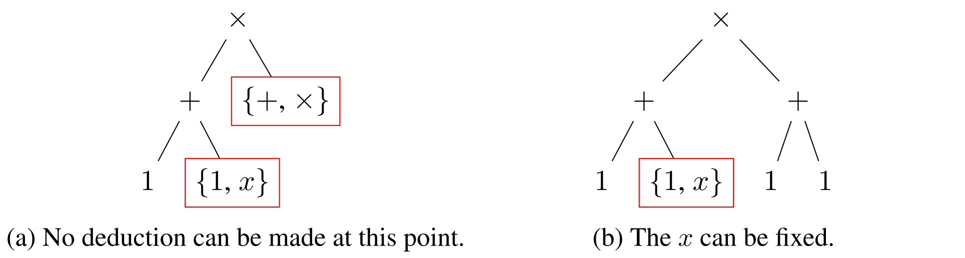


Figure 4.4: Two partial programs under the ”Contains(x)” constraint. In scenario (a) no deductions can be made as the x can appear in the left hole or underneath the right hole. In scenario (b), the hole with the x can be fixed as this is the only remaining place where an x can appear.

4.2 Backtracking

Existing CP solvers can be very memory efficient, as the structure of the problem is known before the search. This means that traversing the search tree can be implemented in a depth-first manner with backtracking.

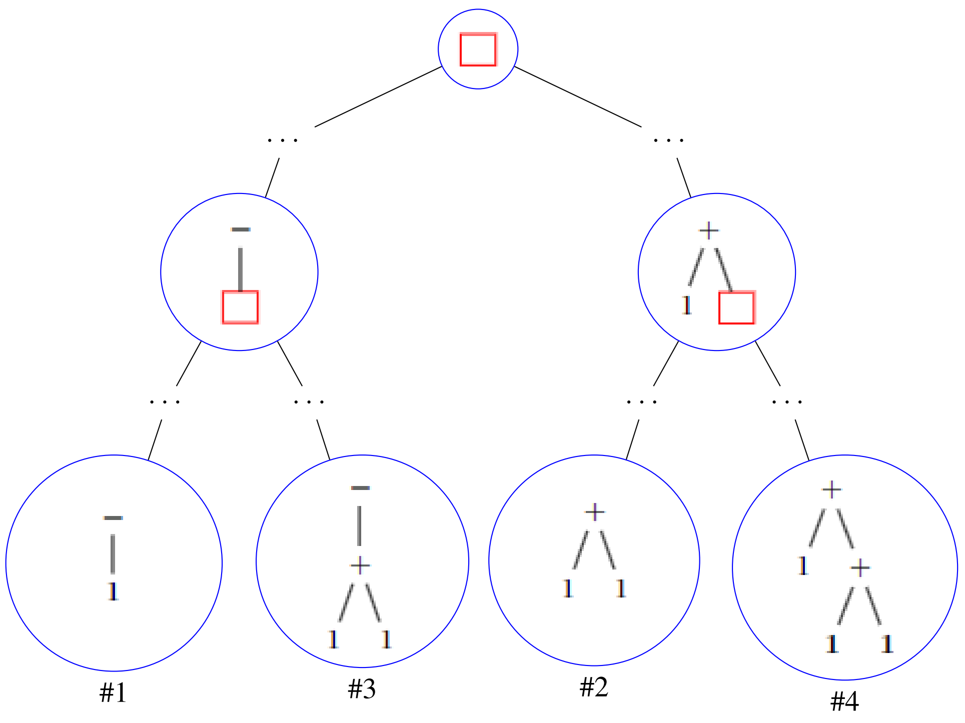


Figure 4.5: A search tree for the arithmetic grammar. The leaves correspond to candidate programs and should be considered in increasing order of program size.

In program synthesis, however, a depth-first search may never halt, as filling a hole can cause the creation of more holes. Furthermore, we have a natural bias towards the smallest satisfying program, so we want to consider candidate programs in increasing order of size. This means we cannot traverse the search tree in a depth-first manner (see Figure 4.5). Instead, we have to maintain multiple partial programs and switch back and forth. One of the challenges of this thesis is to switch between these different tree shapes in a memory-efficient manner.

> *How to use memory-efficient backtracking techniques while still considering candidate programs in increasing order of size?*

4.3 Grammar Constraints

In CP, constraints are defined with respect to a known set of decision variables. But in program synthesis, constraints are defined with respect to the grammar and not with respect to specific hole instances. Consider the partial program tree illustrated in Figure 4.6. Suppose we are propagating a constraint that forbids the sub-expression $1 \times x$. A tree manipulation has just occurred: a hole on the bottom left has just been filled with the value **1**. This manipulation should trigger constraint propagation. The constraint should react to the tree manipulation and propagate locally around the location where the tree manipulation occurred.

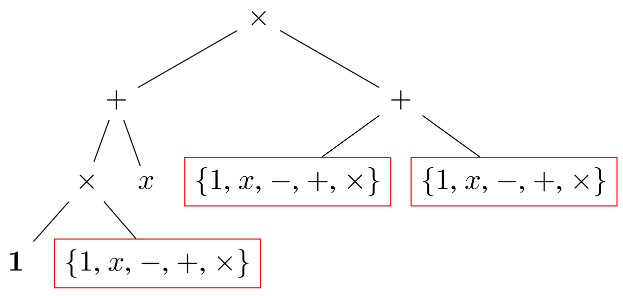


Figure 4.6: A partial program tree with a forbidden sub-tree constraint: $1 \times x$. A hole left has just been filled with a '**1**'. A propagator for this constraint should automatically detect that 'x' can be removed from the bottom hole.

When a tree manipulation occurs, it would be very inefficient to propagate the forbidden constraint on the entire tree. Instead, the constraint should only be considered on local parts of the tree that are potentially affected. We only want to propagate constraints *when* needed and *where* needed. This means that they should somehow relate to actual hole instances and react to tree manipulations accordingly. The question we aim to answer is:

> *How can grammar constraints be propagated efficiently?*

4.4 First-Order Constraints

Other constrained program synthesizers only make use of *grounded* constraints. A grounded constraint is defined for a particular assignment of values. To illustrate this, let's consider the *forbidden* constraint. A forbidden constraint forbids a given sub-tree from appearing anywhere in the program tree. Suppose we set up two constraints to forbid two sub-trees: Forbid($1 \times x$) and Forbid($x \times x$). These are called grounded constraints, as they each forbid a particular sub-tree.

To improve the inference strength, we can combine multiple grounded constraints into a single *first-order* constraint. A first-order constraint is defined over a set of values, instead of a particular assignment. In our example, we would can define a single forbidden constraint over 2 values: Forbid($\{1, x\} \times x$). The question we aim to answer is:

How can grounded constraints be combined into first-order constraints?

In Section 5.6, we will define the constraint types and explain how a *template tree* can be used to define a whole class of constraints at once.

Chapter 5

Methods

This thesis aims to implement a constraint solver for the program synthesis framework `Herb.jl`. This chapter begins with a general overview of the components needed to synthesize a program using this framework. The other sections of this chapter will explain each of the components related to constraint propagation in more detail. Finally, we will look at some concrete implementations of constraints in Section 5.6.

5.1 Overview

In program synthesis, a program iterator is responsible for iterating programs until the target program has been found. Different kinds of program iterators exist, each using a different search strategy. This chapter introduces a generic constraint solver that shrinks the program space by propagating constraints and is independent of the search strategy. As an optimization, a uniform solver is introduced to exploit a depth-first search once the shape of the tree is known.

To illustrate how these two constraint solvers work together, we will consider a top-down program enumeration as the search strategy. This approach uses the generic solver to enumerate different program shapes and the uniform solver to enumerate concrete programs from each shape.

Figure 5.1 presents a high-level overview of the constrained program synthesis this chapter aims to implement. To synthesize a program, the user has to define the program space using a grammar, constraints on the grammar, and optionally a sketch to give the synthesizer a jump start. These inputs are passed to a program iterator that yields valid candidate programs.

The top-down iterator uses a data structure called *uniform holes* to shape the form of the tree (See Section 5.2). Then, the *generic solver* reacts to changes made to each tree and propagates the constraints accordingly (See Section 5.3). If a program tree does not violate the constraints, any remaining holes will also be partitioned into uniform holes using the same procedure.

If a program tree does not contain any non-uniform holes anymore, it is considered a *uniform tree* and is sent to a *uniform solver* (See Section 5.4). This solver enumerates candidate programs that derive from the uniform tree and satisfy the constraints. Finally, each candidate program is tested against the intent specification until the target program is found.

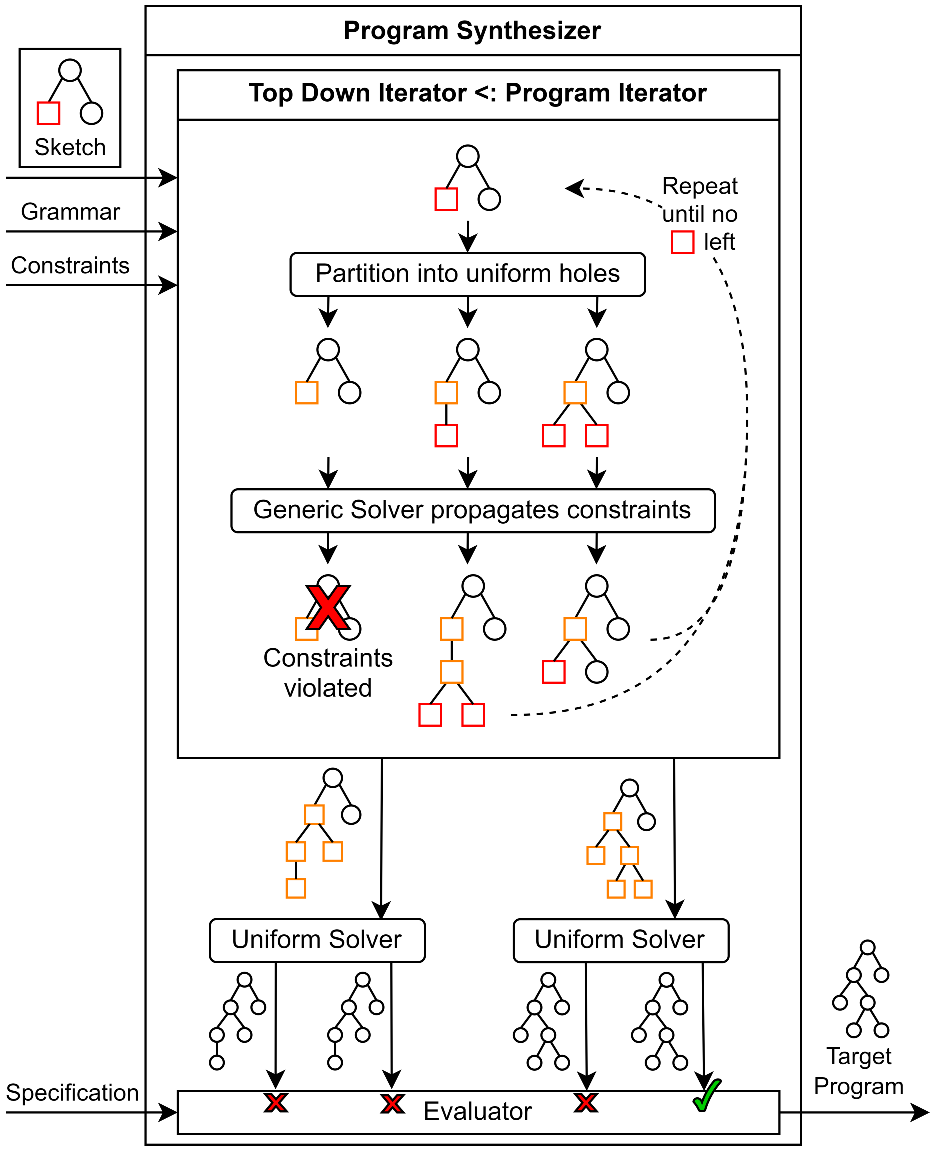


Figure 5.1: Overview of constrained program synthesis using a top-down iterator. Orange boxes represent uniform holes. Red boxes represent non-uniform holes.

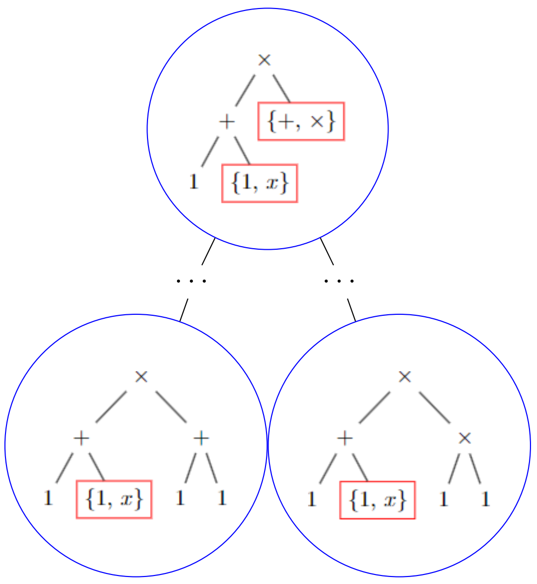
5.2 Data Structures

To implement the program synthesizer outlined in Figure 5.1, we need to define the necessary data structures.

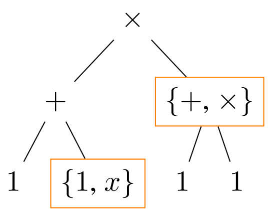
5.2.1 Uniform Holes

So far, we have only seen two types of nodes in a program tree: holes and rule nodes. In this section, we will introduce a third node type: the *uniform hole*. A uniform hole has a domain of rules of the same child types. Since the child types are known, we can already instantiate the children, without knowing the concrete rule of the parent hole. The reason why we introduce this data structure is find the shape the program tree as early as possible.

For example, in the arithmetic grammar, rules $+$ and $\times$ have the same child types. Therefore, a hole with a domain consisting of only these two rules can be converted to a uniform hole. This means we don't have to wait for the hole to be filled with either a $+$ or $\times$ before we instantiate the children. Instead, we will eagerly instantiate the two child holes, without knowing the concrete rule of the parent.



(a) Without uniform holes.



(b) With uniform holes.

Figure 5.2: Two partial programs under the "Contains(x)" constraint. In the root state of scenario (a), the hole with the x cannot be fixed, as an x might also appear underneath the right hole. After branching, the hole with the x needs to be fixed in multiple partial programs. In scenario (b), the hole with the x can be fixed and only 1 partial program is needed to represent the class of programs.

During search, rules are removed from domains. As soon as the domain of a hole becomes uniform, that hole will immediately be converted to a uniform hole and its children will be instantiated. This allows us to fix the structure of tree in a very early stage of the search and allows propagators to make stronger deductions.

The introduction of uniform holes answers the research question from Section 4.1. By expanding the tree to its definite structure in an early stage of the search, propagators can make deductions high up in the search tree. Figure 5.2 compares the propagation of a "Contains(x)" constraint with and without uniform holes. Without uniform holes, the hole with the $+$ and $\times$ needs to be expanded before the propagator can deduce anything. Uniform holes ensure that deductions can be made high up in the search tree, instead of making a similar deduction multiple times.

5.2.2 Grammar Constraints and Local Constraints

In Herb, we differentiate between two types of constraints: *grammar* constraints and *local* constraints.

Grammar Constraint. In Section 4.3, we already touched upon the definition of a grammar constraint. Recall that a grammar constraint is a kind of constraint that is imposed upon the grammar itself and does not refer to any specific hole instances. Such constraints are hard to propagate as they apply to all possible locations in the tree.

Local Constraint. Local constraints are rooted versions of grammar constraints. Each local constraint holds a `path` field that points to a location in the tree where this constraint applies.

In Section 5.3.1, we will see that the generic constraint solver is responsible for posting and propagating local constraints during search by consulting the grammar constraints. On the other hand, the uniform solver is only concerned with the propagation of local constraints, as no new holes can appear at that point.

5.2.3 States

The upcoming two sections about constraint solvers will both have a notion of a state. A *state* is a node in the search tree. It holds a partial program and any active local constraints that apply to it.

The generic solver will have many states stored in memory, while the uniform solver manipulates a single state and uses a backtracking mechanism to explore other branches of the search tree.

5.3 Generic Solver

Just like in `Mini-CP` [10], we will let a solver object take the central role in the search procedure and maintain a valid state. The solver is responsible for propagating constraints, applying tree manipulations, and managing states. In this chapter, we will define a generic solver that can be used by any program iterator.

Constraint Propagation. The primary task of the solver is to maintain a valid state. After each tree manipulation, the solver will choose which constraints to schedule for propagation to ensure that the state satisfies all constraints. The constraints that need to be propagated depend on both the location of the hole and the type of constraint. On each tree manipulation, we will only schedule constraints that can potentially make any deductions. This is managed by the constraint-specific `shouldschedule` function.

Tree manipulations. A *tree manipulation* is any event that shrinks the domain of a hole. In Section 2.1.3, we have considered top-down enumeration for program synthesis. But bottom-up enumeration, stochastic search, and genetic search are also viable search strategies. To be able to propagate constraints regardless of the type of search that is being used, we will make sure that all tree manipulations are made through the solver. Therefore, all search strategies should manipulate the current state using a combination of primitive tree manipulations:

- `remove!(solver, path, rule)`. Remove a rule from the domain of a hole at the given path. If the remaining rules in the domain of the hole have the same child types, this hole will be converted to a uniform hole and thus its children will be instantiated.

- `remove_all_but!(solver, path, rules)`. Remove all rules from the domain of the hole at the given path, except for the specified remaining rules. If possible, the hole will be converted to a uniform hole.

- `substitute!(solver, path, new_node)`. Substitute an existing node at the given path with a new node.

The solver is responsible for propagating relevant constraints after each of these tree manipulations.

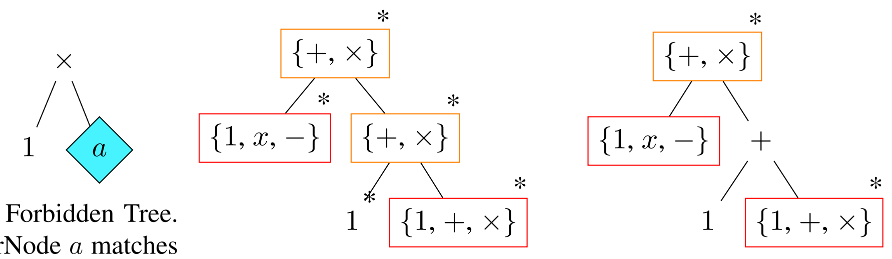
State Management. To further decouple the solver from the search strategy, we will use a generic way of dealing with states. The generic solver allows program iterators to save, load, and delete states to implement custom search strategies. To ensure absolute state independence, saving a state involves deep-copying the partial program, which is a very expensive operation. In Section 5.4, we will see how the uniform solver exploits a depth first search to overcome this issue.

5.3.1 Constraint Propagation

Recall from Section 5.2.2 that we differentiate between *grammar* and *local* constraints. As grammar constraints are hard to propagate, a design decision was made to not implement a `propagate` method for a grammar constraint. Instead, each grammar constraint implements an `on_new_node` function that gets called whenever a new node (usually a hole) appears in the program tree. The constraint will then post a local variant of the grammar constraint that is responsible for propagating the constraint at that particular location.

To illustrate how grammar constraints are split up into local constraints, consider the grammar constraint ”Forbidden($1 \times a$)” represented as a tree in Figure 5.3a. This constraint is responsible for preventing the forbidden tree from appearing anywhere in the program tree.

Now let's assume we jump-start the search with the sketch in Figure 5.3b. This means that all nodes in this tree are new, so the `on_new_node` function of the grammar constraint posts a local constraint at each of the nodes in the tree. In the figure, each * represents a local variant of the forbidden constraint at a particular node. During the fix point algorithm, all newly posted constraints are propagated at their respective location. Figure 5.3c represents the state after propagation. 3 out of the 5 local constraints are now satisfied and deleted. The other 2 constraints cannot deduce anything at this point and remain *active*. This means that if a tree manipulation occurs at or below their path, they are scheduled for (re-)propagation.



(a) Forbidden Tree. VarNode a matches any sub-tree. (b) Local constraints are posted (c) After propagation

Figure 5.3: Forbidden constraint (a) is imposed on tree (b) by posting a local constraint (*) at each location. After propagating, one hole was filled. Only 2 of the local constraints remain active. The other 3 constraints are satisfied and deleted.

Splitting up grammar constraints into local constraints has 2 main advantages:

1. Grammar constraints can be partially deactivated. By deactivating satisfied local constraints, we prevent checking these satisfied parts of a grammar constraint over and over again.

2. We can reduce the frequency of unnecessary propagation. On each tree manipulation, we can carefully choose for each active local constraint to either schedule it for propagation or ignore it. For example, in Figure 5.3c, if a tree manipulation happens on the leftmost hole, we will only schedule the local constraint at the root, as that is the only local constraint that might be affected by this manipulation. We don't have to check active constraints related to other branches in the program tree.

5.3.2 Tree manipulations

A primitive tree manipulation is any domain-decreasing event on the domain of a hole. For example, `remove` removes a rule from a domain of a hole. When a manipulation occurs, the solver propagates constraints in three steps: simplify, notify, and fix point.

Simplify. First, the solver attempts to convert a non-uniform hole to a uniform hole. If this is possible, this will trigger the creation of new nodes, which will also trigger the respective `on_new_node` functions of all grammar constraints to post new constraints.

Notify. This function notifies all active local constraints that a manipulation occurred at a particular location. Local constraints that are affected by this manipulation are scheduled for propagation.

Fix point. The fix-point algorithm (Algorithm 2) propagates all scheduled constraints sorted by a heuristic for impactfulness[1]. Of course, `propagate` functions can also invoke more tree manipulations, potentially setting off a recursive chain reaction of tree manipulations[2].

5.3.3 State Management

In the context of the generic solver, a *state* is defined as a 3-tuple. It holds:

1. A partial program.

2. Active local constraints. These constraints might become violated later in the search and need to be re-propagated in the future.

3. Feasibility flag. Indicates if the program still satisfies the constraints. If a local constraint detects an inconsistency, this flag is set to false and any further propagation will be canceled.

[1]Strong inference and a fast propagate function contribute to a high priority.

[2]Even though the fix-point algorithm is part of a tree manipulation, it will not be nested and ignored when an outer fix-point algorithm is already in process.

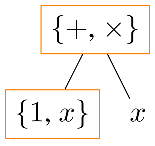
Each state can be seen as its own propagation problem. The active local constraints are active for the current state and their activation is independent of any other solver state.

By design, the generic solver should work with any kind of search strategy. Since the search strategy is unknown to the solver, it cannot partially reuse components among different states. When a state is saved, the partial tree of the state is deep-copied to ensure absolute state independence. This is a major bottleneck and can be improved by exploiting the search strategy. In Section 5.4, we will see how a depth-first search can be exploited to improve state management.

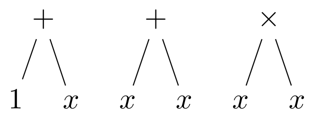
To be as flexible as possible, the generic solver supports three generic functions for state management: `new_state`, `save_state`, and `load_state`. It is up to the search strategy to store the saved states somewhere and load them when needed. During the search, the iterator is also responsible for checking if the state is still feasible. When an inconsistency is found, the iterator is responsible for loading a saved state. In Section 5.5 we will see how the top-down search strategy uses the generic solver to implement top-down program enumeration.

5.4 Uniform Solver

The uniform solver is a constraint solver for uniform trees. Uniform trees are trees with a fixed shape. That is, no new holes can appear in such trees. As an input, the solver takes a uniform tree and a list of active constraints. As an output, it is expected to yield valid candidate programs (See Figure 5.4).



(a) Input: a uniform tree (and constraints)



(b) Output: multiple candidate programs

Figure 5.4: Simple example of an input and output for the uniform solver. It uses the arithmetic grammar with a "Forbidden($1 \times x$)" constraint.

Just like the generic solver, the uniform solver is responsible for propagating constraints, applying tree manipulations, and managing states. Constraint propagation and tree manipulations are handled similarly in both solvers, so we will refrain from repeating the details. One noteworthy difference is that since no new holes can appear in the uniform solver, the grammar constraints do not have to post new local constraints on new nodes during search. The uniform solver is only concerned with propagating existing local constraints.

The most substantial difference between the solvers is in their state management. In particular, the uniform solver will exploit the fact that no new holes can appear by doing a *depth-first search*. We don't have to enumerate the candidate programs in order of size anymore, since the size of the program is always the same. Therefore, we can implement this solver in a depth-first manner. This means that there is no need to deep-copy states anymore and we can instead use a memory-efficient backtracking mechanism. The upcoming sections will focus on how this can be implemented.

5.4.1 Depth-first search

A depth-first search (Algorithm 3) is at the core of the uniform solver. In the base case, if there are no holes in the tree, we can yield the complete tree. If the tree has unfilled holes, we pick a hole from the tree using a heuristic and save the current state of the solver. Then we fill in the hole using a rule from its domain, which triggers the fix point algorithm. If the program is still valid after this tree manipulation, we recursively continue the DFS. Finally, we restore the solver state by reverting all changes made since the saved state and consider the next way to fill the hole. This process continues until all ways to fill the hole have been considered.

Algorithm 3 Depth-first search for the uniform solver with backtracking.

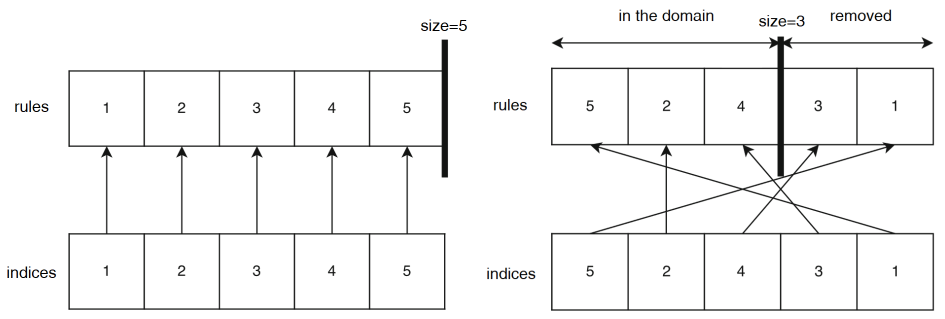
```
function dfs(solver)
hole $\leftarrow$ hole_heuristic(solver.tree)
if !hole then
    yield solver.tree
    solver.restore()
else
    for rule $\in$ hole.domain do
        solver.save_state()
        solver.fill(hole, rule)
        if solver.isfeasible() then
            dfs(solver)
        end if
        solver.restore()
    end for
end if
end function
```

The greatest improvement of this approach, compared to the generic solver, is that saving the state can be handled by tracking changes instead of copying. In Section 5.4.2, we will see exactly how we can track different kinds of changes.

5.4.2 State Management

State management for the uniform solver is handled by tracking changes since a saved state and reverting these changes to restore the saved state. Before concerning ourselves with tracking any change, we will simplify the setting and first implement *stateful integers*.

Stateful Integer. A stateful integer is an integer that can be saved and restored. Each stateful integer holds a reference to a constraint manager that keeps track of the changes to all stateful integers. When the value of a stateful integer is updated to a different value and no backup of this integer was made since the last saved state, a backup entry of this integer and its previous value will be made and stored in the state manager. On a restore call of the constraint manager, all backup entries made since the saved state will be popped and applied to revert all changes.



(a) Initial domain. (b) Domain after removing rules 1 and 3.

Figure 5.5: A stateful domain before and after the removal of rules 1 and 3

Stateful Domains. The purpose of the constraint manager is to revert any changes made to the solver state, including domain manipulations. We can achieve this by implementing a stateful domain using a stateful integer. A possible data structure for this is a state sparse set [10]. This data structure uses an array for the rules in the domain, an array of indices that point to rules, and the size of the domain (See 5.5a). When a rule is removed from the domain, the removed rule is swapped with the rule at the end of the rules array. Additionally, the array of indices is updated accordingly such that: `rules[indices[i]]=i` still holds. Lastly, the size of the domain is decreased by one. Figure 5.5b illustrates a domain after removing two rules. The key property of this data structure is that it is possible to retrieve the rules of the domain by looking at the size of the domain. Furthermore, we can revert to the previous state of the domain, just by reverting the size of the domain. This means that this domain becomes stateful if the size of the domain is implemented with a stateful integer. To have this stateful property, it must be assumed that no domain manipulation can increase the size of the domain. This assumption is justified since constraint propagation can never increase the size of a domain.

5.5 Top-Down Iterator

The purpose of the top-down iterator is to enumerate all possible programs that can be constructed with a given grammar up to a certain maximum size and/or maximum depth. Furthermore, if the grammar is equipped with grammar constraints, all the enumerated programs should satisfy these constraints.

The top-down iterator uses the generic solver to enumerate programs in increasing order of size. This is achieved by managing solver states using a priority queue, with a breadth-first priority function. Once a solver state contains a uniform tree, enumerating the programs deriving from that uniform tree is delegated to a uniform solver.

We will now look at the pseudo-code of the top-down iterator in a bit more detail (See Algorithm 4). The top-down iterator begins by constructing a hole corresponding to the starting symbol [3]. Next, the generic solver performs the initial propagation, so we need to verify feasibility. If the initial state is feasible, it is added to a priority queue, and enumeration begins. Then we repeatedly pop items from the priority queue. An item from the priority queue could either be a solver state (I) or a uniform solver (II).

Case I) We pop a solver state from the priority queue. In this case, we will load it into the generic solver. Then we will find a non-uniform hole to branch on using a hole heuristic. If there is no such hole, we are dealing with a uniform tree, so a new uniform solver can be created for this tree. This uniform solver is put back into the queue for later consideration. For now, let's assume the hole heuristic did find a non-uniform hole. This hole will be partitioned into uniform domains, that is, the domain is grouped by child types. For example, the domain $\{1, x, -. +, \times\}$ will be partitioned into $\{1, x\}$, $\{-\}$ and $\{+, \times\}$. Then for each domain in this partition, we duplicate the solver state and shrink the domain of the target hole accordingly. The generic solver will automatically propagate all relevant constraints for this tree manipulation. After the tree manipulation is completed, the iterator needs to check if the solver is still in a feasible state. If it is, we can enqueue this partial tree for further expansion.

Case II) We pop a uniform solver from the priority queue. In this case, we can get one complete program from the uniform solver and re-enqueue it for later consideration. Alternatively, the iterator could yield all the solutions from the uniform solver first, but this means it will yield all solutions of a single shape consecutively. The design decision to put solvers back into the priority queue was made to allow priority functions to potentially alternate between different shapes of trees.

[3]Alternatively, a sketch can provided to jump-start the search. But by default, we will start from the starting symbol of the grammar.

Algorithm 4 Top Down Iterator: yields all programs of a constrained grammar.

```
root ← RuleNode(starting_symbol))
solver ← GenericSolver(grammar, root, max_size, max_depth)
if !solver.isfeasible() then
    return
end if
pq ← PriorityQueue(solver.get_state())
while len(pq) $>$ 0 do
    state ← pq.dequeue()
    if state isa SolverState then
        solver.load_state(state)
        hole ← hole_heuristic(solver.get_tree())
        if !hole then
            uniform_solver ← UniformSolver(grammar, solver.get_tree())
            pq.enqueue(uniform_solver, priority_function(...))
        else
            for uniform_domain $\in$ partition(hole.domain) do
                state ← solver.save_state()
                solver.remove_all_but!(hole, uniform_domain)
                if solver.isfeasible() then
                    pq.enqueue(solver.get_state(), priority_function(...))
                end if
                load_state(solver, state)
            end for
        end if
    else if state isa UniformSolver then
        complete_tree ← next_solution(state)
        if complete_tree then
            pq.enqueue(state, priority_function(...))
            yield complete_tree
        end if
    end if
end while
```

Recall the research question from Section 4.2: *How to use memory-efficient backtracking techniques while still considering candidate programs in increasing order of size?*. The top-down iterator solves this issue by doing a BFS to find the different shapes of programs in order of size and then locally doing a DFS to find the complete programs for each shape. It is clear that a BFS is not very memory efficient, as it requires coping states to assure state independence. But a DFS can be exploited to be more efficient because a solver then only has to maintain a single state and use a backtracking mechanism. In other words, there is no more need to copy over states anymore.

5.6 Constraints

Library of Constraints. This thesis serves as a framework that can be used to propagate any kind of grammar constraint. Other developers are encouraged to add more constraints for specific purposes. A couple of constraints have been implemented for this thesis and will be discussed in the upcoming subsections:

`Forbidden(tree)`. Ensures the program tree does not contain the specified tree as a sub-tree. The tree may also contain variable symbols. For example, Forbidden($a \times 1$) contains a variable a that can match anything. This constraint will forbid any multiplication with 1 (e.g. $(x + 1) \times 1$).

`Ordered(tree)`. Ensures that if the program tree contains the specified tree, the variable symbols in the tree are in lexicographical order. This constraint aims to reduce the program space by breaking commutative properties. For example, Ordered($a \times b$) ensures that only $x \times (1 + 1)$ is a valid program, and $(1 + 1) \times x$ is not.

`Contains(rule)`. Ensures the specified production rule appears somewhere in the program tree. A possible use-case of this constraint is to enforce that the input parameters of a program are contained in the program.

`Unique(rule)`. Ensures the specified production rule appears at most once in the program tree. A possible use-case of this constraint is to ensure a particular function call occurs at most once.

`ContainsSubtree(tree)`. Ensures a sub-tree matching the specified template tree appears somewhere in the program tree. This is a more general version of the regular Contains constraint.

`ForbiddenSequence(sequence)`. Ensures the specified sequence of rules does not appear in some vertical path in the program tree. In Section 5.6.5, we will see a concrete use-case of this constraint.

Template Trees. Some of the constraints are defined with a template `tree`. Such trees can consist of two special types of nodes: `DomainRuleNodes` and `VarNodes`. These node types can be used to express an entire class of trees using a single template. This allows us bundle grounded constraints into first-order constraints, which reduces the amount of constraints and increases the inference strength of propagators. Pattern matching (See Appendix B.1) is a core functionality of propagators. It compares two nodes and returns a flag that indicates whether they match. This procedure can pairwise compare the following four types of nodes:

`RuleNode`. This node holds a single rule that should be exactly matched. The children of a RuleNode should also be exactly matched.

`DomainRuleNode`. This node holds a domain of rules and matches to any 1 of the nodes in its domain. The children of a DomainRuleNode should also be exactly matched.

`VarNode`. This node holds a variable name. The first match will bind the subtree to this name. Any subsequent matches of VarNodes of the same name should exactly match this binding. Note that bindings can also include Holes, DomainRuleNode, and nested VarNodes.

`Hole`. Holes are the only node type that may only appear in the program tree and not in a template tree. Nevertheless, pattern matching supports hole-hole comparisons as they can appear when a VarNode is bound to a hole. During pattern matching, we keep track of all holes that need to be filled in a specific way to complete the match. In case of a forbidden constraint, if only 1 such hole exists, it can prevent a match by eliminating the possibility of filling that hole in that way.

API for Propagators. We have already seen how grammar constraints are split into local constraints, so in this chapter, we are only concerned with the propagation of local constraints. Propagators can use the following solver functions to make deductions:

`set_infeasible!(solver)`. If a propagator detects an inconsistency, the solver should be notified and cancel any other scheduled propagators.

`deactivate!(solver, constraint)`. If a constraint is satisfied, it should deactivate itself to prevent re-propagation. In the generic solver, this is handled by removing it from the set of active constraints. In the uniform solver, this is handled by toggling off a stateful boolean.

`post!(solver, constraint)`. A constraint is allowed to post new local constraints. This might be helpful if a constraint can be reduced to a smaller constraint. In that case, a constraint will deactivate itself and then post a smaller constraint that captures the problematic part of the constraint.

`remove!(solver, path, rule)`. A constraint can remove rules from any hole's domain. This tree manipulation removes the specified rule from the hole located at the provided path. Besides the primitive 'remove' tree manipulation, propagators can also use similar tree manipulations: `remove_all_but!`, `remove_all_above!` and `remove_all_below!`.

5.6.1 Forbidden

The forbidden constraint can be used to forbid syntactically valid, yet semantically redundant sub-programs from the grammar. For example, the arithmetic grammar (Example 2) has a rule "$Int \rightarrow -Int$". A forbidden constraint could be used to prevent $-(-(a))$ from appearing anywhere in the program tree. We do not lose the target program by eliminating such sub-programs, because $-(-(a))$ is represented by a in another program.

Algorithm 5 Propagation of a local forbidden constraint.

```
node ← get_node_at_location(solver, constraint.path)
match ← pattern_match(node, constraint.tree)
**if** match isa HardFail **then**
    deactivate!(solver, constraint)
**else if** match isa Success **then**
    set_infeasible!(solver, constraint)
**else if** match isa SuccessWhenHoleAssignedTo **then**
    remove!(solver, match.hole, match.rule)
    deactivate!(solver, constraint)
**end if**
```

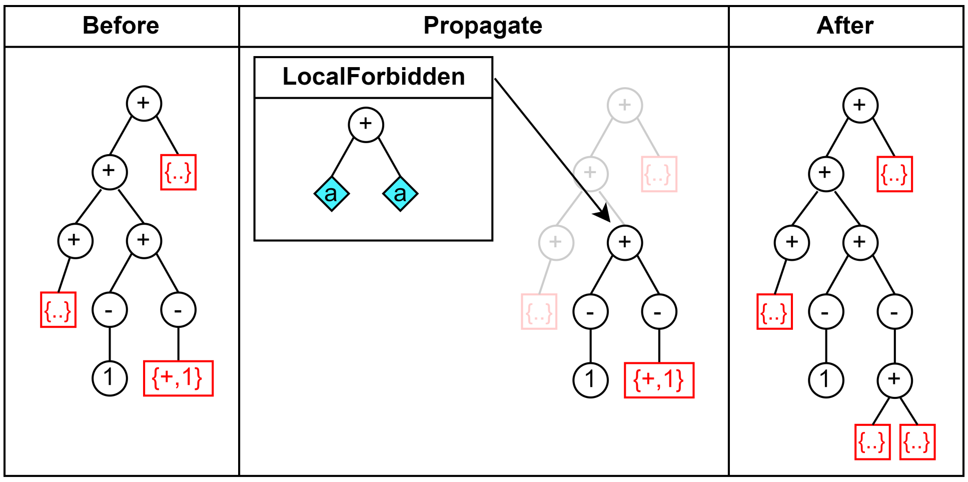


Figure 5.6: Propagation of a LocalForbidden constraint at path $[2, 1]$. The constraint contains two VarNodes, represented by a blue diamond, holding variable a. In this example, the pattern match returned "SuccessWhenHoleAssignedTo".

A local forbidden constraint has two components: a `path` and a forbidden template `tree`. Algorithm 5 describes how a local forbidden constraint is propagated. It uses the pattern match function to match the forbidden tree with the node located at the path. If the match fails, the constraint is already satisfied and can be deactivated. If the match is successful, the forbidden tree is present in the program, so the state must be set to infeasible. If a match can be prevented by removing a rule from a hole, the constraint does so and then deactivates itself. In Figure 5.6, a local forbidden constraint will remove rule 1 from the bottom right hole. After propagation, this specific local forbidden constraint is always satisfied and can be deactivated. It is also possible that multiple holes are involved and no deduction can be made. In that case, the constraint remains active and will be re-propagated whenever one of the holes involved is updated.

5.6.2 Ordered

The ordered constraint ensures that if the program tree contains the specified template tree, the matched `VarNodes` in the tree are in lexicographical order. This constraint is particularly useful for breaking commutative properties. For example, in the arithmetic grammar, an ordered constraint can be used to ensure that only one of $a \times b$ and $b \times a$ is valid. Since they are semantically equivalent, we won't lose the target program by eliminating either one the two programs. The ordering we use is not important, as long as it is consistent throughout the search. We will use an ascending ordering in the rule index and tie break in a depth-first manner in case of equality.

Algorithm 6 Propagation of a local ordered constraint.

```
node ← get_node_at_location(solver, constraint.path)
vars ← Dict()
match ← pattern_match(node, constraint.tree, vars)
if match isa Fail then
    deactivate!(solver, constraint)
else if match isa Success then
    should_deactivate ← true
    $n$ ← length(constraint.order)
    for name1, name2 ∈ zip(constraint.order[1:$n$-1], constraint.order[2:$n$]) do
        result ← make_less_than_or_equal!(solver, vars[name1], vars[name2])
        if result isa HardFail then
            set_infeasible!(solver)
        else if result isa SoftFail then
            should_deactivate ← false
        end if
    end for
    if should_deactivate then
        deactivate!(solver, constraint)
    end if
end if
```

A local ordered constraint has three components: a `path`, a template `tree`, and a required `order` of VarNodes. Algorithm 6 describes how a local ordered constraint is propagated. First, the template tree is matched with the node located at the path. This binds all VarNodes to node instances in the tree. Then the `make_less_than_or_equal!` tree manipulation attempts to enforce the ordering upon the bound nodes. The goal is to make rule a node less than or equal rule node b. During this process, impossible rules will be removed from holes. After the deductions have been made, a `result` flag is returned that describes the current state of the $\leq$ inequality, There are three possible results:

- Success. $a \leq b$ is guaranteed under all possible assignments of the holes involved. This means that the constraint is always satisfied and can be deactivated.

- Hard Fail. $a > b$ is guaranteed under all possible assignments of the holes involved. In this case, the constraint is violated, so the solver state must be set to infeasible.

- Soft Fail. In this case, $a \leq b$ and $a > b$ are still possible depending on how the holes involved are filled. We cannot make a deduction at this point, and the constraint needs to be re-propagated if one of the holes involved is updated.

Making node a less than or equal to node b is done by comparing the rule indices of the nodes. Only when the nodes are equal, the tie is broken by comparing the children in a depth-first matter. Figure 5.7 holds four examples scenarios of the propagation of a LocalOrdered constraint.

In scenario (i), we can remove rule 5 to eliminate the possibility that $a > b$. Then, since the roots of a and b are equal, we break the tie by comparing the children in a depth-first manner. The left child of b is fixed to rule 1, this means we can remove all rules higher than 1 from the left child of a. This, again, leads to a tie, so we will continue by comparing the right children. Similarly, the right child of b is fixed to value 2, so we can remove all rules higher than 2 from the right child of a. Now $\leq$ is assured, so we will return a "Success" result.

In scenario (ii), $a = 4$ and $b = \{4, 5\}$, so $a \leq b$ always holds. However, if b gets assigned to 4, we need to break the tie by comparing the children. Let's assume this is needed and compare the children. We compare two holes: $\{3, 4, 5\} \leq \{1, 2\}$. This inequality never holds for any assignment of the two holes. This is considered a hard fail. However, we can prevent the hard fail by assuring that tie break is never needed by removing the 4 from hole b. Now the $\leq$ inequality always holds, regardless of the rules of the children. We can return a "Success" result to indicate that the constraint can be deactivated.

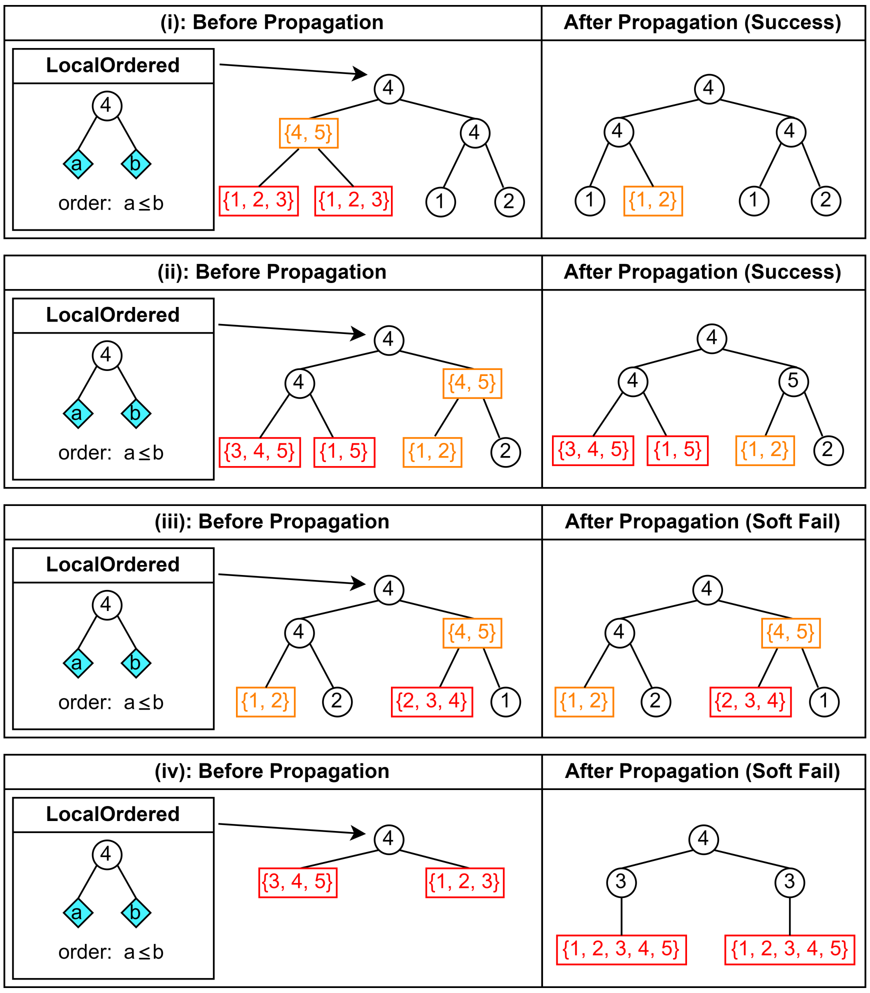


Figure 5.7: Four scenarios of the propagation of a LocalOrdered constraint. In each scenario, VarNodes a and b are matched to sub-trees. Then '$\leq$' is enforced by comparing rule indices of the nodes. In the case of equality, the tie is broken by comparing the children in a depth-first manner. In the last 2 cases, $\leq$ cannot be guaranteed and the propagator needs to be reconsidered. This is called a "Soft Fail".

In scenario (iii), $a = 4$ and $b = \{4, 5\}$, so $a \leq b$ always holds. However, if b gets assigned to 4, we need to break the tie by comparing the children. Let's assume this is needed and compare the children. We compare two holes: $\{1, 2\} \leq \{2, 3, 4\}$. Again, this inequality always holds, but if both holes are assigned to a 2, we would need to tie break. Let's assume this is needed and compare the right children: $2 > 1$. This is a hard fail, so we conclude that tie-breaking is not allowed. So either b needs to be set to 4, or the left children need to be set to 2. Since both are valid options, we have to return a "Soft Fail" and re-propagate later.

In scenario (iv), $a = \{3, 4, 5\} \leq \{1, 2, 3\} = b$. The lowest rule in a is 3, so we must remove all rules lower than 3 from b. Similarly, the highest rule of b is 3, so we must remove all rules higher than 3 from a. After removing these rules, the children of a and b can be instantiated. This example uses the arithmetic grammar of Figure 4.1, where rule 3 corresponds to a negation, so we instantiate only one child under rule 3. Since $a = b$, we need to tie break. We cannot make any further deductions here and therefore return a "Soft Fail".

5.6.3 Contains and Unique

The contains constraint can be used to enforce that a certain grammar rule has to appear somewhere in the program tree. For example, in the arithmetic grammar (Example 2), a user might want to enforce that an x (the input symbol) has to appear in the program. Otherwise, the program will be a constant value.

Another use-case for the contains constraint is when a complex constraint can simplify itself into a contains constraint by deactivating and replacing itself with one or more contains constraints. Such a complex constraint does not exist yet, but could easily be added to the framework.

The unique constraint is very similar to the contains constraint, but enforces uniqueness instead of existence of a rule. It can be used to enforce that a certain grammar rule cannot appear more than once. In the robot environment (Example 1), this can be used to enforce that a 'grab' operation appears at most once in the entire program.

5.6.4 Contains Sub-tree

The 'contains sub-tree' constraint is a more general version of the 'contains' constraint. It enforces that a provided template tree appears in the program tree at least once, instead of being restricted to a single rule index.

Only a single local constraint will be posted (at the root). This local constraint enforces that the provided template `tree` appears somewhere at or below the root. Propagating this constraint is very expensive, as it requires pattern matching all nodes in the program tree with the template. To improve the efficiency of the propagation, we will keep track of all nodes that can potentially match the template. These nodes are called *candidates*. On re-propagation, we only need to update the list of candidates. If a candidate matches the template, the constraint is satisfied and deactivated. If a candidate fails to match the template, it will be removed as a candidate. Finally, if there is only a single candidate remaining, it will be enforced to equal the template. Then, the constraint can be deactivated.

Algorithm 7 describes how a local 'contains sub-tree' constraint is propagated. It can be assumed that the `constraint` has the following fields:

- `tree`, the to be matched template tree
- `candidates`, an initial list of nodes in the current partial program tree that potentially match the template tree. This list is created on initial propagation.
- `indices`, a state sparse set of indices pointing to candidates that are still candidates.

Algorithm 7 Re-propagation of a 'contains sub-tree' constraint.

```
for $i \in$ constraint.indices do
    candidate $\leftarrow$ constraint.candidates[$i$]
    match $\leftarrow$ pattern_match(candidate, constraint.tree)
    if match isa HardFail then
        remove!(constraint.indices, $i$)
    else if match isa Success then
        deactivate!(solver, constraint)
    end if
end for
$n \leftarrow$ length(constraint.indices)
if $n == 0$ then
    set_infeasible!(solver, constraint)
else if $n == 1$ then
    $i \leftarrow$ minimum(constraint.indices)
    candidate $\leftarrow$ constraint.candidates[$i$]
    result $\leftarrow$ make_equal!(solver, candidate, constraint.tree)
    if result isa HardFail then
        set_infeasible!(solver, constraint)
    else if result isa Success then
        deactivate!(solver, constraint)
    end if
end if
```

Since this constraint is stateful, it is not possible to propagate this constraint in the generic solver. It would be possible to propagate a non-stateful version of this constraint, but it will likely be very inefficient because of two reasons: (1) candidates are not tracked, so all nodes in the tree need to be checked in each propagate call, and (2), non-uniform holes can almost always expand into the template tree, so no actual deductions can be made until all the holes are uniform. This is why the design decision was made to only enforce this constraint in uniform trees.

Figure 5.8 contains a small example of the propagation of `ContainsSubtree`. In this scenario, the constraint initially has two candidates. Then, on re-propagation, one of these candidates becomes invalid and the remaining candidate is enforced to equal the template tree.

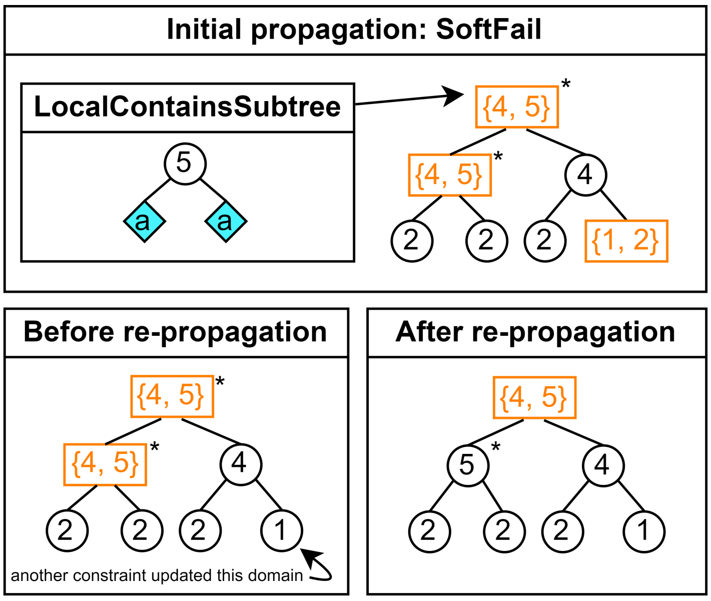


Figure 5.8: Propagation of a LocalContainsSubtree constraint at the root (path = []). In the initial propagation, all 7 nodes were pattern-matched against the template tree $5(a, a)$. Only two of the nodes potentially match the template and are considered 'candidates' (represented with a *). After another constraint updated a domain, the candidate at the root no longer matched the template and was removed as a candidate. Since only 1 candidate remains, it is enforced to match the template. The constraint is now satisfied and deactivated.

5.6.5 Forbidden Sequence

The forbidden sequence constraint forbids a sequence of rules from the root to any path in the tree. When specifying a forbidden sequence, all sequences that contain the forbidden sequence as a sub-sequence are also forbidden (e.g. the forbidden sequence [1, 2, 3] also forbids the sequence [1, 2, 1, 2, 4, 2, 3]).

To propagate the forbidden sequence constraint, a local constraint is posted at each `path` in the tree. The local constraint assumes that the sequence will end at exactly its `path` and makes the smallest possible match from this path back to the root. If a match was found, and only a single hole is involved, the rule that completes the forbidden sequence will be removed from its domain. Then, the constraint is immediately re-propagated and tries to find the next match. If no match was found, the constraint is satisfied and can be deactivated. In Figure 5.9, only one match existed and was prevented. If multiple holes exist in a match, the constraint soft-fails and will be re-propagated if any domain on the path is updated.

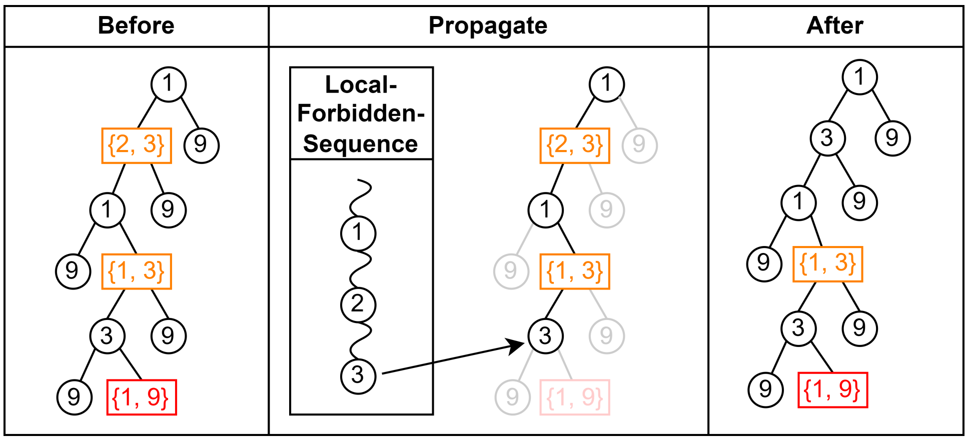
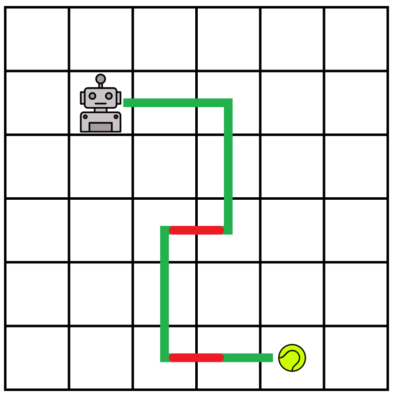


Figure 5.9: Propagation of a LocalForbiddenSequence constraint at path [1,1,2,1].

The forbidden sequence constraint can also be equipped with an `ignore_if` list of rules. If any of the rules in that list is encountered within the sequence, the constraint is ignored. This is particularly useful for the robot environment (Example 1), as the constraints we will define for it should only be applied when the robot does not grab or drop a ball in between. The following forbidden sequence constraints will be used to constrain the robot environment:

- Shortest path constraints (Figure 5.10a). We can ensure the robot takes the shortest path:
  - ForbiddenSequence([left, right], ignore_if = [drop, grab]);
  - ForbiddenSequence([right, left], ignore_if = [drop, grab]);
  - ForbiddenSequence([up, down], ignore_if = [drop, grab]);
  - ForbiddenSequence([down, up], ignore_if = [drop, grab]);

- Symmetry breaking constraints (Figure 5.10b). We can break symmetries in the grammar by taking horizontal steps before vertical steps:
  - ForbiddenSequence([down, right], ignore_if = [drop, grab]);
  - ForbiddenSequence([down, left], ignore_if = [drop, grab]);
  - ForbiddenSequence([up, right], ignore_if = [drop, grab]);
  - ForbiddenSequence([up, left], ignore_if = [drop, grab]);


(a) Shortest path constraint.

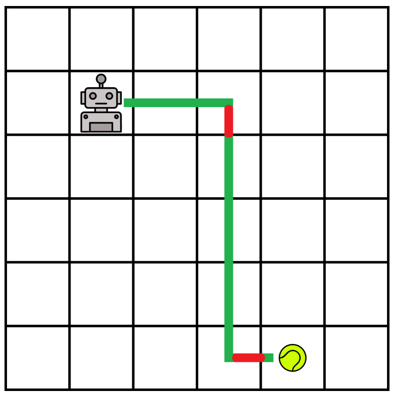

(b) Symmetry breaking constraint.

Figure 5.10: Two violations of ForbiddenSequence constraints in the Robot Environment (Example 1).

Chapter 6

Evaluation

In this chapter, we will evaluate the proposed methods from Chapter 5 by answering the following questions:

1. How much of the program space needs to be pruned to justify the overhead of constraint propagation?
2. Does combining multiple grounded constraints into a single first-order constraint improve the performance of the search?
3. How much do the `Generic-` and `UniformSolver` solver components contribute to the overall search procedure?
4. What are the bottlenecks in the proposed methods?

6.1 Setup

All the experiments have been executed on an Intel i7-10750H CPU @ 2.60GHz with 16 GB of RAM. The results were collected using `Herb.jl v0.3.0`[1], with the exception of the `HerbSearch`[2] and `HerbConstraints`[3] repositories. They were set to their respective `dev` branches as of May 2024.

The experiments will be executed on 4 different grammars, each with their own production rules and constraints. Two of the grammars will be based on the running examples of this thesis, but slightly modified to better utilize the constraints.

6.1.1 Robot Grammar

The robot grammar describes programs in the Robot Environment (Example 1). This environment consists of an $n \times n$ grid, a robot and a ball. A state describes

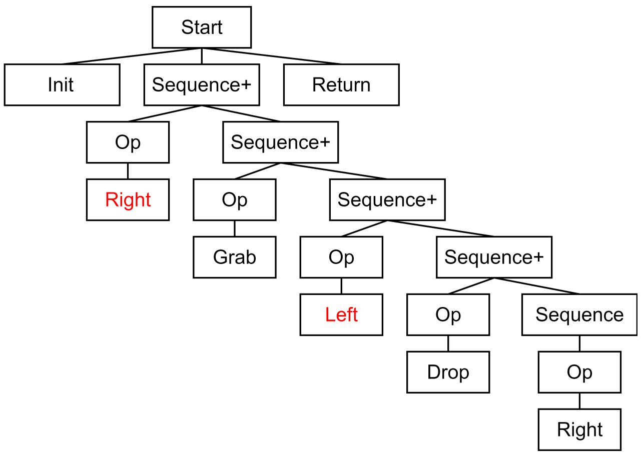
[1]https://github.com/Herb-AI/Herb.jl/releases/tag/v0.3.0

[2]https://github.com/Herb-AI/HerbSearch.jl/commit/a43eb05963ba4825b271ea45d7117a899c9efd65

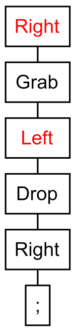
[3]https://github.com/Herb-AI/HerbConstraints.jl/commit/c91af15a251f25a452d0a655bf40b23a4a1cf500

the position of the robot and the ball. The task is to learn to transform the initial state to the final state by moving the robot and letting it grab and drop the ball.

We will simplify the earlier defined robot grammar (Example 2.3) by omitting conditional statements, as the environment is too simple for them to be needed. Furthermore, to use the forbidden sequence constraint, we will consider a variant of the grammar where operations lie on a vertical path in the program tree (See Figure 6.1). This modification is made solely for convenience. Alternatively, the semantics of `ForbiddenSequence` could be changed to consider a sequence of (terminal) nodes in left sub-trees instead of on a vertical path.



(a) Original AST



(b) Simplified AST

Figure 6.1: Comparing semantically equivalent ASTs for the robot environment. (a) and (b) derive from the grammars in Figures 2.3 and 6.2 respectively.

In the upcoming experiments, we will use the robot grammar as defined in Figure 6.2, equipped with the grammar constraints in Figure 6.3. To replicate the results, add each constraint using `addconstraint!(grammar, constraint).`

```
grammar = @csgrammar begin
    Sequence = (moveRight(); Sequence)
    Sequence = (moveDown(); Sequence)
    Sequence = (moveLeft(); Sequence)
    Sequence = (moveUp(); Sequence)
    Sequence = (drop(); Sequence)
    Sequence = (grab(); Sequence)
    Sequence = return
end

```

Figure 6.2: A simplified grammar for the robot environment in Herb.jl.

```
# the robot can drop and grab at most once
Unique(r_drop)
Unique(r_grab)

# shortest path constraints
ForbiddenSequence([r_left, r_right], ignore_if = [r_drop, r_grab])
ForbiddenSequence([r_right, r_left], ignore_if = [r_drop, r_grab])
ForbiddenSequence([r_up, r_down], ignore_if = [r_drop, r_grab])
ForbiddenSequence([r_down, r_up], ignore_if = [r_drop, r_grab])

# symmetry breaking constraints
ForbiddenSequence([r_down, r_right], ignore_if = [r_drop, r_grab])
ForbiddenSequence([r_down, r_left], ignore_if = [r_drop, r_grab])
ForbiddenSequence([r_up, r_right], ignore_if = [r_drop, r_grab])
ForbiddenSequence([r_up, r_left], ignore_if = [r_drop, r_grab])

```

Figure 6.3: 10 constraints on the robot grammar defined in Figure 6.2

6.1.2 Arithmetic Grammar

The arithmetic grammar (Example 2) can be used to synthesize simple arithmetic expressions with an input symbol x. For example, given IO examples $1 \rightarrow 2$ and $5 \rightarrow 6$, the synthesizer will return $x + 1$ as the target program.

We will slightly modify the earlier defined grammar to exactly match the grammar of the previous master thesis on constraints in Herb [5] (See Figure 6.4). This variant of the grammar includes 10 terminal rules for constants, instead of 1.

```
grammar = @csgrammar begin
    Int = Int + Int
    Int = Int * Int
    Int = Int - Int
    Int = |(0:9)
    Int = x
end

```

Figure 6.4: An arithmetic grammar in Herb.jl, borrowed from [5].

The constraints for the arithmetic grammar (See Figure 6.5) all break semantic symmetries in integer arithmetic. They forbid trivial cases like adding 0, and multiplying by 1. The ordered constraints will be used to break the commutativity of $+$ and $\times$. This is not an exhaustive list of symmetry breaking constraints, and could be extended to break more symmetries.

```
Forbidden(RuleNode(times, [VarNode(:a), RuleNode(zero)]))
Forbidden(RuleNode(minus, [VarNode(:a), VarNode(:a)]))
Forbidden(RuleNode(minus, [VarNode(:a), RuleNode(zero)]))
Forbidden(RuleNode(plus, [VarNode(:a), RuleNode(zero)]))
Forbidden(RuleNode(times, [VarNode(:a), RuleNode(one)]))
Forbidden(RuleNode(minus, [
    RuleNode(times, [VarNode(:a), RuleNode(two)])
    VarNode(:a)
]))
Forbidden(RuleNode(plus, [VarNode(:a), VarNode(:a)]))
Forbidden(RuleNode(minus, [
    RuleNode(times, [VarNode(:a), RuleNode(three)])
    VarNode(:a)
]))
Forbidden(RuleNode(plus, [RuleNode(zero), VarNode(:a)]))
Forbidden(RuleNode(times, [RuleNode(zero), VarNode(:a)]))
Forbidden(RuleNode(times, [RuleNode(one), VarNode(:a)]))
Ordered(RuleNode(plus, [VarNode(:a), VarNode(:b)]), [:a, :b])
Ordered(RuleNode(times, [VarNode(:a), VarNode(:b)]), [:a, :b])
```

Figure 6.5: 13 constraints on the arithmetic grammar defined in Figure 6.4

6.1.3 Symbolic Grammar

The symbolic grammar (see Figure 6.6), and its 21 constraints (see Appendix B.2) have no semantics and exists solely to push the constraint solver to its limits.

Semantic grammars can be constrained to prune a large amount of the program space, but a significant amount of it remains valid. In traditional CP, often only a few, if any at all, solutions exist. In such cases, most time is spent in finding even a single solution. To mimic this setting and push constraint propagation to its limit, we will consider the highly constrained symbolic grammar, in which only a handful of programs satisfy the constraints.

```
grammar = @csgrammar begin
    S = t1 #terminals             # rule 1
    S = t2                        # rule 2
    S = t3                        # rule 3
    S = u1(S) #unary functions    # rule 4
    S = u2(S)                     # rule 5
    S = u3(S)                     # rule 6
    S = b1(S, S) #binary functions  # rule 7
    S = b2(S, S)                  # rule 8
    S = b3(S, S)                  # rule 9
    S = unique                    # rule 10
end
```

Figure 6.6: A symbolic grammar without any semantics.

6.1.4 List Grammar

The list grammar (see Figure 6.7) is a semantic grammar that can be used for list manipulations. It supports basic list operations and can be used to construct a program that takes two input integers x and y and returns a list.

To illustrate program synthesis using this grammar, consider the following two IO examples:
$(x = 0, y = 1) \rightarrow [1, 3]$.
$(x = 5, y = 4) \rightarrow [3, 9]$.
The intended behaviour of the program is to return a sorted list of the sum of the input values and the constant 3. Our program synthesizer is successfully able to find a satisfying program:
`sort!(push!(push!([], sum(push!(push!([], y), x))), 3))`

The constraints for the list grammar (See Figure 6.8) eliminate semantically redundant programs. For example, by forbidding sorting a list twice in a row. We also define a domain rule node representing all unary functions. Two of the constraints use this node to forbid unary functions on an empty list (line 14) and a singleton list (line 15) respectively. The constraint on line 17 aims to forbid reversing a constant list. For example, `reverse!(push!(push!([],2),x)))` can be forbidden, as it is already represented by `push!(push!([],x),2)`.

Just like for the arithmetic grammar, the provided list of constraints is not exhaustive, and could be further extended. However, they do prune enough of the program space to evaluate the performance of constraint propagation in the upcoming sections.

```julia
grammar = @csgrammar begin
    List = []
    List = push!(List, Int)
    List = reverse!(List)
    List = sort!(List)
    List = append!(List, List)
    Int = maximum(List)
    Int = minimum(List)
    Int = sum(List)
    Int = prod(List)
    Int = |(1:3)
    Int = x
    Int = y
end

```

Figure 6.7: A grammar with basic list operations in Herb.jl.

```
A = VarNode(:a)
B = VarNode(:b)
V = VarNode(:v)
unaryfunction(node::AbstractRuleNode) = DomainRuleNode(grammar,
    [_reverse, _sort, _max, _min, _sum, _prod], [node])

Forbidden(reverse(reverse(A)))
Forbidden(sort(reverse(A)))
Forbidden(sort(sort(A)))
Forbidden(sort(append(A, reverse(B))))
Forbidden(sort(append(A, sort(B))))
Forbidden(sort(push(sort(A), V)))

Forbidden(unaryfunction(empty))
Forbidden(unaryfunction(push(empty, V)))

ForbiddenSequence([_reverse, _empty], ignore_if=[_sort, _append])
ForbiddenSequence([_append, _empty], ignore_if=[_reverse, _sort])

```

Figure 6.8: 10 constraints on the list grammar defined in Figure 6.7

6.2 Reducing the Program Space

In this section, we will using the top-down iterator to enumerate all programs of the previously defined grammars, up to a certain maximum program size. To measure the correctness and effectiveness of the constraint propagation, we will compare three variations of enumeration.

1. **Plain Enumeration (+checking)**. Enumerates all programs, ignoring the constraints. Then, retrospectively checks the constraints and eliminates all programs that violate any constraint.

2. **Plain Enumeration**. Enumerates all programs, ignoring the constraints.

3. **Constrained Enumeration**. Enumerates all programs that satisfy the constraints using the proposed constraint solvers.

For all experiments, the programs obtained from methods 1 and 3 are exactly the same. This means constraint propagation does not eliminate valid programs, nor keep any invalid programs. Although this is not a hard proof of correctness, it does increase the confidence that the propagators are implemented correctly and working as intended.

In Figure 6.9, we compare the program space with and without constraints by dividing the amount of valid programs by the total amount of programs. We see that the constraints significantly reduce the program space[4]. With a maximum of 11

[4]The symbolic grammar is omitted from this graph, since its smallest valid program has 12 nodes.

nodes, the imposed constraints can already eliminate roughly 99% of the total program space (The exact results can be found in Appendix C).

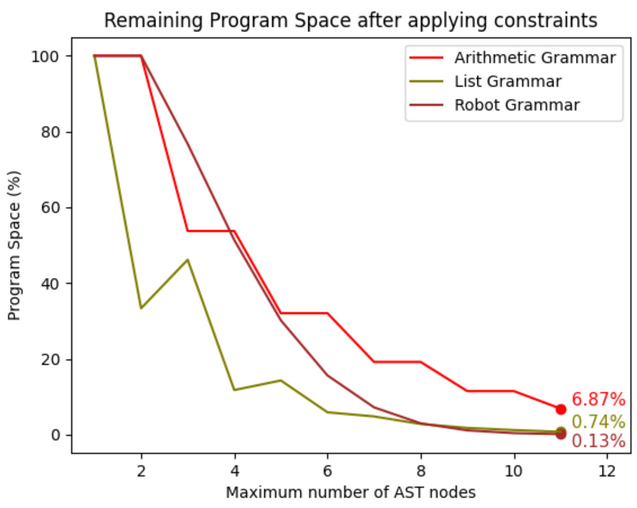


Figure 6.9: The remaining portion of the program space after applying the constraints, represented by the ratio of valid/total programs

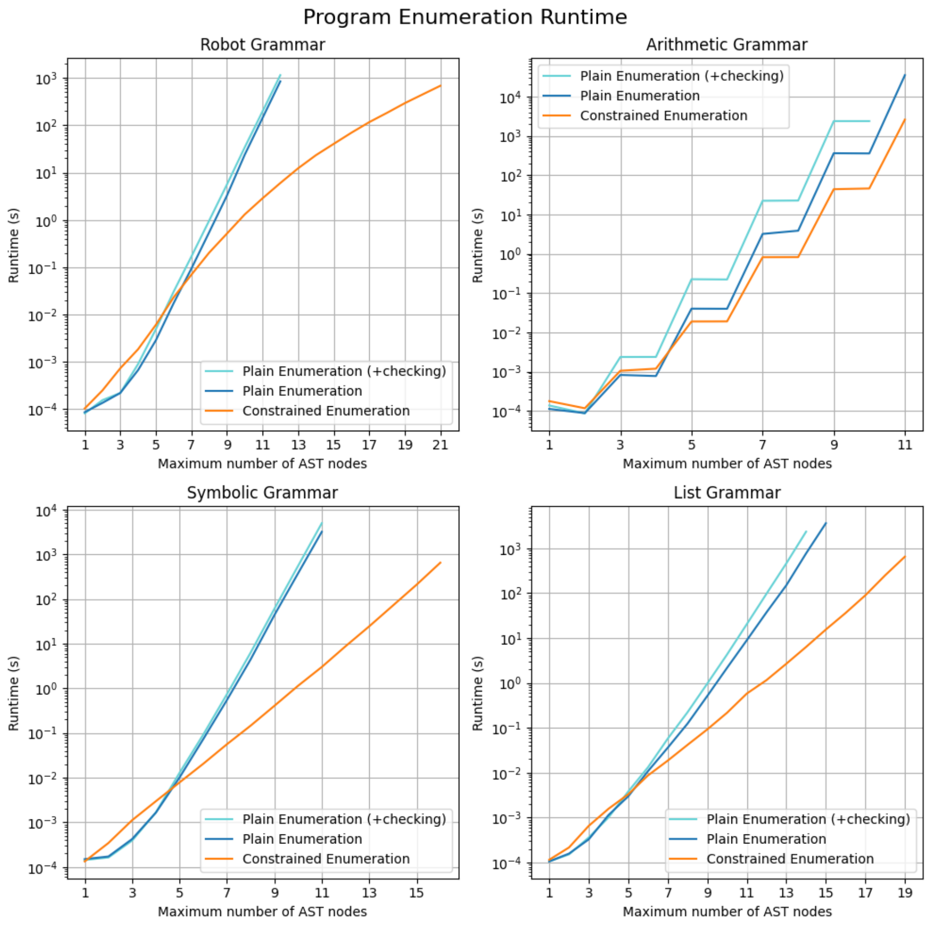
Figure 6.10 compares the runtime of the three types of enumeration. We observe that the runtime of plain enumeration without retrospectively checking the constraint is strictly higher than that of plain enumeration alone. This is because the verification process adds time on top of the enumeration.

Propagating the constraints during search outperforms the plain enumeration. This is an expected result, as only a small fraction of the program space needs to be enumerated in a constrained search. The ratio plot reveals that for all four grammars the constrained search performs significantly better as the program space grows larger, but the exact improvement highly depends on the grammar.

An interesting observation is that the plain enumeration outperforms constrained enumeration for small program spaces. In these cases, the reduced program space does not justify the overhead of propagating constraints.

We conclude that when a relatively large portion[5] of the program space can be eliminated, constraint propagation outweighs its overhead.

[5]Roughly speaking, 75% or more. This depends on many other factors, such as the grammar and the type and amount of constraints.



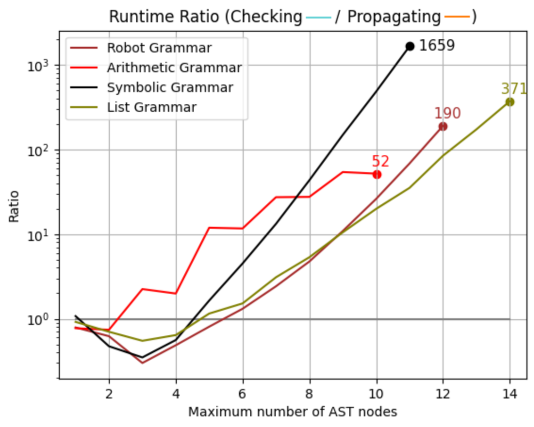


Figure 6.10: Runtime of program enumeration with and without constraints. The ratio plot illustrates the difference in runtime between two approaches: checking constraints after enumeration and propagating constraints during enumeration.

6.3 First-order Constraints

In Section 5.6, we have seen two kinds of first-order rule nodes that can be used to forbid a class of sub-trees: `DomainRuleNode` and `VarNode`. In the following experiment, we will measure the effectiveness of the domain rule nodes using the Symbolic Grammar (6.1.3) with a new constraint:

```
Forbidden(DomainRuleNode({7, 8, 9}),[
    DomainRuleNode({1, 2, 3, 10}),
    DomainRuleNode({1, 2, 3, 10})
]
```

This constraint eliminates all trees that contain any of the $3 \cdot 4 \cdot 4 = 48$ forbidden sub-trees. Alternatively, we can break down this first-order constraint into 48 grounded constraints, each forbidding a particular sub-tree.

We will run the constrained program enumeration of the Symbolic Grammar maximum of 8 AST nodes, and vary the amount of constraints we use. In each run, we will use 1 first-order constraint, and 1 grounded constraint for each missing case. For example, suppose we remove one of the rules from the first-order constraint:

```
Forbidden(DomainRuleNode({7, 8}),[
    DomainRuleNode({1, 2, 3, 10}),
    DomainRuleNode({1, 2, 3, 10})
]
```

Then 16 grounded constraints must be constructed to cover the missing cases. Note that the amount of valid programs remains the same. The difference lies in the amount of propagators and inference strength. Regardless of which combination of constraints is used, we will always enumerate 1.358.656 out of the 2.355.328 total programs.

In Figure 6.11, we see a positive correlation between the number of constraints and `propagate` calls. This is unsurprising, as having more constraints, also means having more propagators. The fact that this also increases the runtime, means a higher quantity of grounded propagators is more computationally expensive than a lower quantity of first-order propagators.

We also see a positive correlation between the number of constraints and search nodes. This can be explained by inference strength. When a first-order constraint is split up into grounded constraints, they can no longer exploit constraint interaction. For example: $forbid(\{7, 8\})$ is able to deduce that domain $\{7, 8\}$ is inconsistent. But $forbid(7)$ and $forbid(8)$ separately cannot make any deductions, and require the search node to branch before they can spot the same inconsistency.

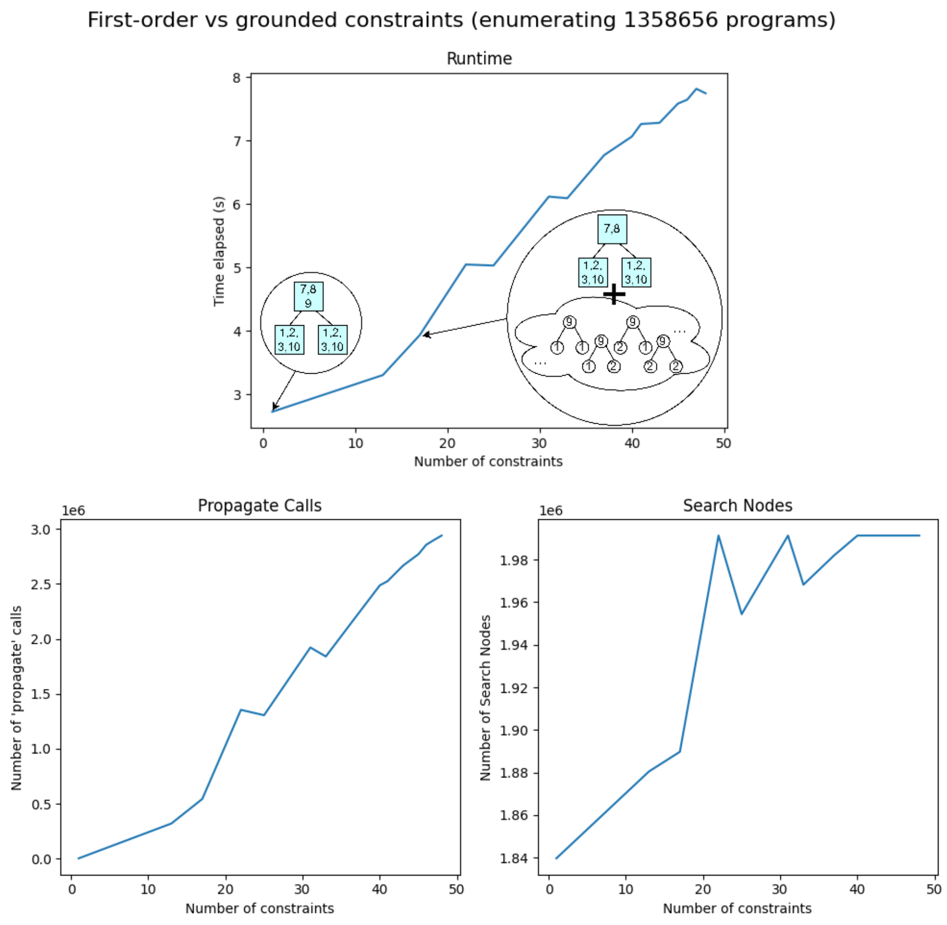


Figure 6.11: Enumerating programs of the Symbolic Grammar using different combinations of first-order and grounded forbidden constraints. The plots depict runtime, propagate calls and search nodes respectively.

We conclude that bundling grounded constraints into a first-order constraint does increase the performance of the search.

6.4 Ablation Study

So far, all experiments have been executed with the top-down iterator that uses both the `Generic-` and `UniformSolver`. The generic solver is used to propagate constraints on non-uniform trees, which has the potential to eliminate an entire search branch of uniform trees. It also uses memory-intensive state management to ensure programs are enumerated in increasing order of size. The uniform solver has efficient state management but is restricted to solving uniform trees and a DFS.

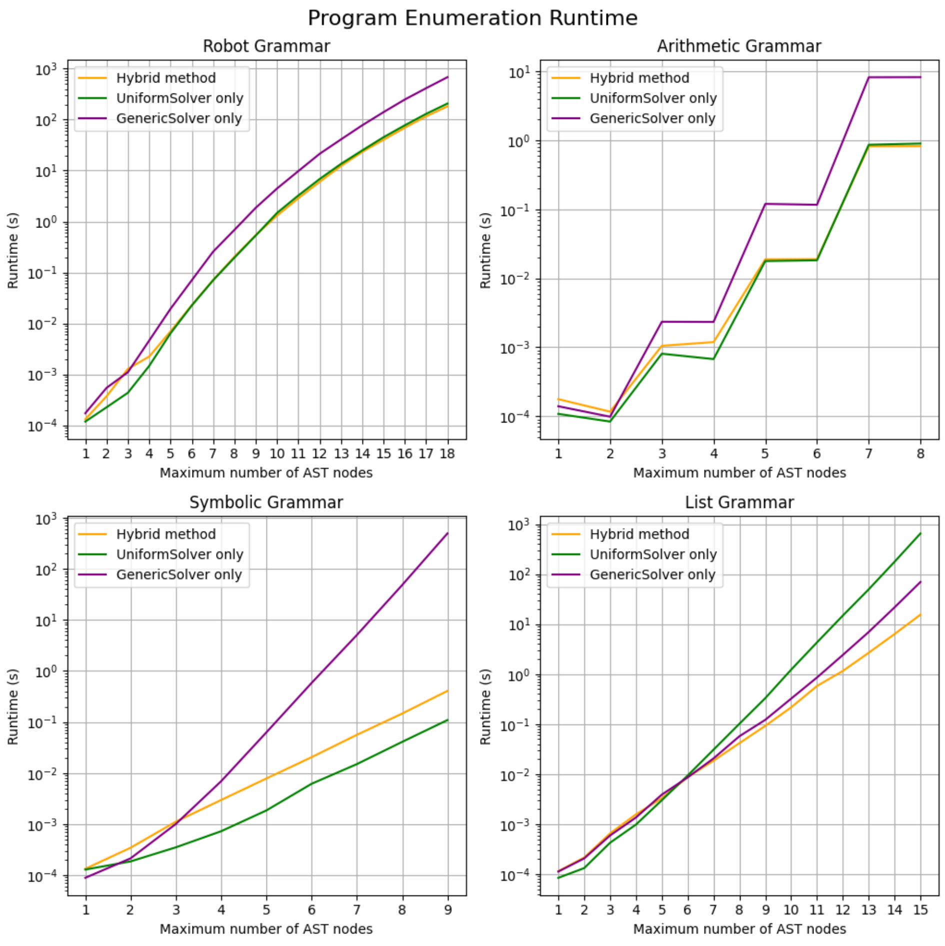


Figure 6.12: An ablation study. Comparing the runtime of the overall search procedure using both or only 1 of the 2 built-in solvers.

In this section, we will conduct an ablation study. Instead of just using the proposed hybrid method, we will also measure the performance of the search using only one of the implemented solvers. More precisely, we will compare the following three methods:

1. Hybrid method: The generic solver is used to enumerate uniform trees, and uniform solvers are used to enumerate all complete programs of each uniform tree, as described in Algorithm 4.

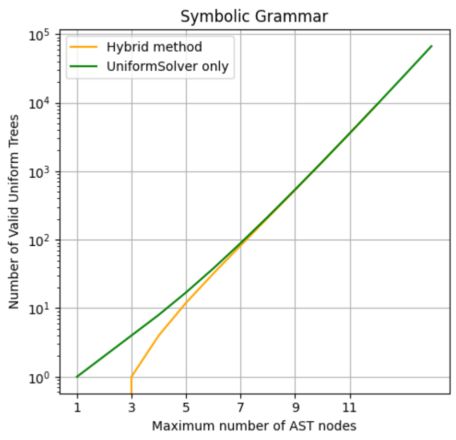
2. `UniformSolver` only: We ignore constraints in the `GenericSolver`, and only start propagating constraints once a uniform tree is reached.

3. `GenericSolver` only: We never dispatch to the `UniformSolver`. Even uniform trees will be expanded by the `GenericSolver`.

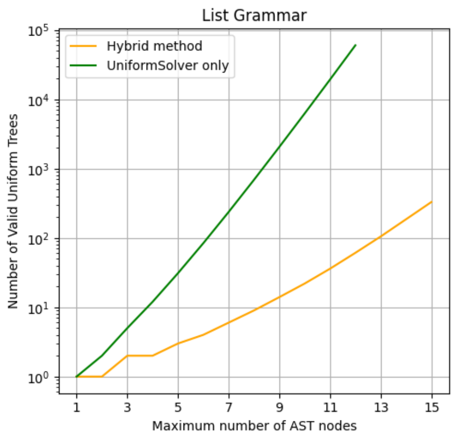
Figure 6.12 holds the result of the ablation study. We see that the hybrid method always outperforms using only the generic solver. In both the hybrid and generic solver methods, the number of uniform trees is exactly the same. In other words, the results indicate that solving a uniform tree with the uniform solver, optimized for solving uniform trees, outperforms the generic solver.

For the symbolic grammar, only using the uniform solver outperforms the hybrid method (see Figure 6.12). This indicates that the overhead of propagating constraints in the generic solver does not outweigh the gained inference. This suspicion is confirmed by the follow-up experiment in Figure 6.13a. We see that the generic solver (orange line) only eliminates a negligible small amount of uniform trees, which means the inference is weak. For this particular grammar, using the generic solver is not worth its overhead.

On the contrary, for the list grammar (see Figure 6.12), the hybrid method outperforms only using the uniform solver. This indicates that the generic solver has strong inference for this grammar. Again, this suspicion is confirmed by the follow-up experiment in Figure 6.13b. We see that the generic solver (orange line) eliminates a significant portion of uniform trees. A reduced number of uniform trees means that less uniform solvers have to be instantiated. Hence, the hybrid method will outperform only using the uniform solver.



(a) Symbolic Grammar



(b) List Grammar

Figure 6.13: Comparing the total number of uniform trees with and without the generic solver.

6.5 Bottlenecks

In this section, we will look at the bottlenecks in the proposed methods to aid developers in optimizing the solvers. To find the bottlenecks, we will use Julia's built-in profiler tool to measure the activity distribution. We can make several observations from the results in Figure 6.14.

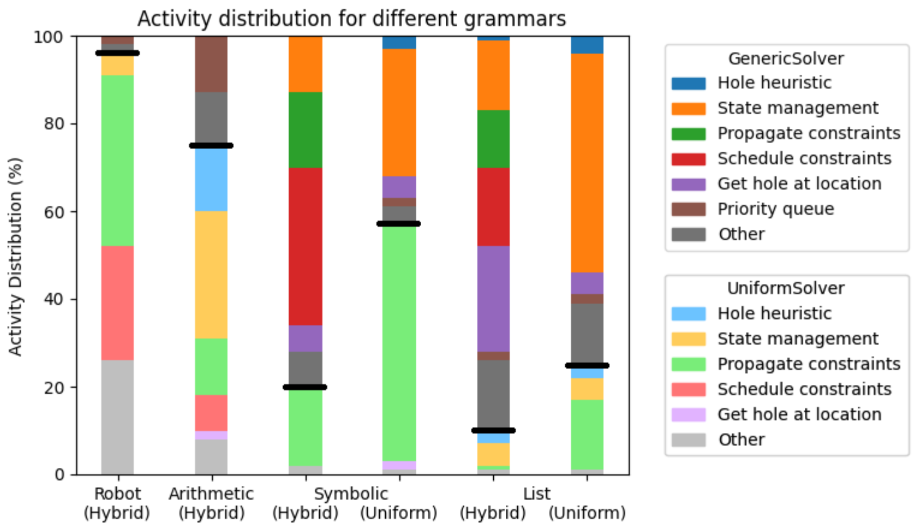


Figure 6.14: The activity distribution of the generic- and uniform solver in the top-down iterator for several grammars. For the Symbolic- and List grammars, an additional experiment without constraint propagation in the generic solver is included (labeled 'Uniform'). The activity was measured using a profiler tool and grouped into high-level categories.

For the Robot- and Arithmetic grammars, we see that a significant amount of time is spent in scheduling constraints, almost as much as propagation itself. This is because propagators are scheduled unnecessarily often. For the robot grammar, a total of 2.315.425 propagators were scheduled, but only 16% of them were able to make any kind of deduction[6].

For the Symbolic grammar, the issue is even more pronounced. We see that more time is spent scheduling than propagating. This indicates that many constraints are scheduled, but unable to make any deductions. And indeed, the follow-up experiment in Figure 6.15 shows that in the generic solver only 52 deductions were made. This is an expected result, as the generic solver deals with non-uniform trees, and

[6]A propagator makes a *deduction* iff it prunes 1 or more rules from 1 or more domains.

most constraints can only make deductions if the tree's structure is known. In the ablation study (See Figure 6.12), we have seen that if we refrain from propagating constraints in the generic solver, we can improve the total runtime by a rough factor of 3. However, a more sustainable approach would be to limit the conditions for constraint scheduling, thereby reducing the number of unnecessary propagators.

	Constraint	Propagation	Deduction	Satisfied
Generic Solver	Contains	329210	24	5
	ContainsSubtree	0	0	0
	Forbidden	801961	0	221893
	ForbiddenSequence	342677	28	42651
	Ordered	538524	0	103054
	Unique	109727	0	0
	All	2122099	52	367603
Uniform Solver	Contains	30208	339	133
	ContainsSubtree	22099	15269	14
	Forbidden	131145	5	106106
	ForbiddenSequence	202483	1258	46555
	Ordered	74252	1	46326
	Unique	1469	822	188
	All	461656	17694	199322

Figure 6.15: Statistics for constraint propagation in the enumeration of the Symbolic Grammar with the maximum program size set to 12.

For the List Grammar, we also see a high amount of scheduling in the generic solver. However, unlike for the Symbolic Grammar, the constraints in the generic solver can make enough deductions to overcome the overhead of propagation. This is reflected by the relative increase in state management activity in the generic solver (See Figure 6.14, right orange bar). If we refrain from propagating constraints, we increase the number of search nodes, and state management (deep-copying) becomes the bottleneck of the overall search procedure.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This thesis introduced constraint propagation in program synthesis using a hybrid method of two solvers. The generic solver can be used to enumerate tree shapes in increasing order of size. The uniform solver is restricted to one shape but has memory-efficient backtracking techniques inspired by MiniCP [10]. The ablation study revealed that employing the hybrid method yields a notable enhancement compared to solely relying on the generic solver.

We have compared the implemented constrained program enumeration against plain program enumeration. The results show that constraint propagation significantly outperforms retrospectively checking the constraints. Although it must be noted that the effectiveness highly depends on the grammar and its constraints.

Furthermore, we have seen that combining grounded constraints into a single first-order constraint can further reduce the number of search nodes. This is an expected result as a combined constraint is able to make deductions based on the interaction of the grounded constraints and therefore has stronger inference.

7.2 Future Work

7.2.1 Example-based constraint generation

In Chapter 6, we have seen that for semantic grammars we can prune a large amount of programs of the program space. However, a large amount ($> 1\%$) of candidate programs remains to be evaluated. Automatically discovering new constraints based on the specification of the problem at hand, has the potential to prune a significantly larger amount of the program space.

In Chapter 3, we have seen three related works that learn new constraints based on failed IO examples. In this thesis, we have only considered constraining the grammar itself. We did not take into account the problem that is to be solved. A constraint extractor could take the IO examples of the problem and generate grammar constraints for the specific problem instance. Or, like in the related works, generate new constraints based on failed IO examples during search.

7.2.2 More First-Order Constraints

For the forbidden constraint, we added a `DomainRuleNode` to allow users to forbid an entire class of program trees, instead of having to define multiple similar constraints. For example,
`RuleNode(4, VarNode(:a), VarNode(:a))` and
`RuleNode(5, VarNode(:a), VarNode(:a))`
can be represented using a single constraint:
`DomainRuleNode({4, 5}, VarNode(:a), VarNode(:a))`

This principle can be applied to other kinds of constraints too. For example, the `Contains(1)`,`Contains(2)` and `Contains(3)` constraints could be combined to a single constraint that enforces that all three rules are contained in the tree. Such a first-order constraint can have stronger inference and reach an infeasible state earlier than three individual constraints.

7.2.3 Full DFS Solver

In this thesis, I presented a trade-off between a BFS and DFS that allows problems to be enumerated in order of size, while being memory-efficient once a uniform tree has been reached. In some cases, however, enumeration in order of program size is not important. For example, the stochastic search strategy uses a top-down iterator as a sub-process to sample a new tree. In that case, only a single solution is requested of the top-down iterator, so a DFS would suffice.

In such cases, a full DFS would be more memory-efficient than the presented hybrid method. To achieve this, we can create a new solver that is very similar to Herb's UniformSolver, but without the restriction of solving for non-uniform trees. Additionally, since no trees will be deep-copied, a handful of other optimizations can be made for the new solver. The following ideas can be used to implement this in Herb.

Only use a single hole type for the entire search. Right now, program iterating works with 3 node types. `Hole`, `UniformHole`, and `RuleNode`. Each of these types adds a restriction upon the previous type. As soon as a domain satisfies the next restriction, the object will be replaced with a new object type of that new restriction. This approach has two issues. (1) Replacing the object in the tree requires an $O(n)$ traversal to the node location in the tree. (2) References to the

replaced object are invalidated, requiring references to the replaced object to be replaced with the new object too.
Instead, the new solver should only use a single hole type. Trait functions can be used to check if that hole's domain is uniform or is filled with a single rule.

Notify lists. Since the holes are never replaced nor copied, each hole can have an `on_domain_change` notify list of constraints. Whenever a domain change happens on that hole, all constraints in that list will be scheduled for propagation. During the post method and on a soft-fail, a constraint can add itself to notify lists of related holes.
In the current implementation, re-propagation is based on the path of the tree manipulation and not on hole instances. I have attempted to re-propagate using notify lists, but since states of the generic solver get deep-copied on each tree manipulation, the notify lists had to be deep-copied too. This was not worth the reduced number of propagations.

Use a StateStack to store the children of nodes. In addition to revertible domains, newly appeared nodes should also be uninstantiated on backtrack. This can be achieved using a state stack. A `StateStack` is a strictly increasing immutable list. Users can only push items on the state stack. Backtracking is done by decreasing the size of the stack to the original size.

Constraints can make use of stateful properties. In the current implementation, constraints are shared among different solver states. The generic solver requires absolute independence between states, so a design decision was made for constraints to be stateless. In a new solver, there will only be a single active state, so stateful constraints are a valid option again. When a constraint is propagated, certain properties can be cached and reused during re-propagation. For example, the matches of a `VarNode` will be the same.

7.2.4 Contributions to Herb.jl

This thesis serves as a basic framework for constraint propagation in Herb.jl and is by no means a fully optimized project. In Section 6.5, we have seen the bottlenecks of the implementation, such as constraint scheduling and state management. Implementation-level details on how to improve the presented methods can be found on the GitHub page of Herb[1]. As Herb is an open-source project, readers are warmly invited to address one of these issues, implement new types of constraints, or simply experiment with the framework.

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

Bibliography

[1] Noam Chomsky. Three models for the description of language. *IRE Trans. Inf. Theory*, 2:113–124, 1956.

[2] Andrew Cropper and Sebastijan Dumančić. Inductive logic programming at 30: A new introduction. *Journal of Artificial Intelligence Research*, 74:765–850, 2022.

[3] Andrew Cropper and Rolf Morel. Learning programs by learning from failures. *CoRR*, abs/2005.02259, 2020.

[4] Allen Cypher, editor. *Watch What I Do – Programming by Demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[5] Jaap de Jong. Speeding up program synthesis using specification discovery. Master's thesis, Delft University of Technology, 07 2023.

[6] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. *SIGPLAN Not.*, 53(4):420–435, jun 2018.

[7] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330. ACM, 2011.

[8] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[9] Herb-AI. Herb.jl: A Julia framework for program synthesis. `https://github.com/Herb-AI/Herb.jl`, 2023.

[10] L. Michel, P. Schaus, and P. Van Hentenryck. Minicp: a lightweight solver for constraint programming. *Mathematical Programming Computation*, 13(1):133–184, 2021.

[11] OpenAI. Gpt-4. `https://openai.com/product/gpt-4`, 2023. Visited on 23/05/2023.

[12] Laurent Perron. Search procedures and parallelism in constraint programming. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming – CP'99*, pages 346–360, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[13] Alexander Schiendorfer. Constraint programming for hierarchical resource allocation. pages 57–68, 01 2014.

[14] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming — CP98*, pages 417–431, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[15] Armando Solar-Lezama. *Program synthesis by sketching*. PhD thesis, Massachusetts Institute of Technology, 2008.

[16] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT Press, Cambridge, 1989.

Appendix A

Encodings

A.1 Logic program system of the robots grammar

```
head_pred(f,2).
body_pred(at_top,1).
body_pred(at_bottom,1).
body_pred(at_left,1).
body_pred(at_right,1).
body_pred(move_left,2).
body_pred(move_right,2).
body_pred(move_up,2).
body_pred(move_down,2).

direction(f,(in,out)).
direction(move_left,(in,out)).
direction(move_right,(in,out)).
direction(move_up,(in,out)).
direction(move_down,(in,out)).
direction(at_top,(in,)).
direction(at_bottom,(in,)).
direction(at_left,(in,)).
direction(at_right,(in,)).
```

```
size(10).

at_left(w(1,_)).

at_bottom(w(_,1)).

at_top(w(_,Y)):-
    size(Y).
at_right(w(X,_)):-
    size(X).

move_right(w(X1,Y),w(X2,Y)):-
    size(Size),
    X1 #< Size,
    X2 #= X1 + 1.

move_left(w(X1,Y),w(X2,Y)):-
    X1 #> 1,
    X2 #= X1 - 1.

move_up(w(X,Y1),w(X,Y2)):-
    size(Size),
    Y1 #< Size,
    Y2 #= Y1 + 1.

move_down(w(X,Y1),w(X,Y2)):-
    Y1 #> 1,
    Y2 #= Y1 - 1.
```

Appendix B

Herb Code Snippets

B.1 Pattern match

```julia
"""
    abstract type PatternMatchResult end

print("Hello")
A result of the `pattern_match` function. Can be one of 4 cases:
- `PatternMatchSuccess`
- `PatternMatchSuccessWhenHoleAssignedTo`
- `PatternMatchHardFail`
- `PatternMatchSoftFail`
"""
abstract type PatternMatchResult end

"""
The pattern is exactly matched and does not involve any holes at
    all
"""
struct PatternMatchSuccess <: PatternMatchResult end

"""
The pattern can be matched when the `hole` is filled with any of
    the given `ind`(s).
"""
struct PatternMatchSuccessWhenHoleAssignedTo <: PatternMatchResult
    hole::AbstractHole
    ind::Union{Int, Vector{Int}}
end

"""
The pattern is not matched and can never be matched by filling in
    holes
"""
struct PatternMatchHardFail <: PatternMatchResult end

"""
The pattern can still be matched in a non-trivial way. Includes
```

```
    two cases:
- multiple holes are involved. this result stores a reference to
    one of them
- a single hole is involved, but needs to be filled with a node of
     size >= 2
"""
struct PatternMatchSoftFail <: PatternMatchResult
    hole::AbstractHole
end

"""
Recursively tries to match `AbstractRuleNode` `rn` with `
    AbstractRuleNode` `mn`.
Returns a `PatternMatchResult` that describes if the pattern was
    matched.
"""
function pattern_match(rn::AbstractRuleNode, mn::AbstractRuleNode)
    ::PatternMatchResult
    pattern_match(rn, mn, Dict{Symbol, AbstractRuleNode}())
end

"""
Generic fallback function for commutativity. Swaps arguments 1 and
     2, then dispatches to a more specific signature.
If this gets stuck in an infinite loop, the implementation of an
    AbstractRuleNode type pair is missing.
"""
function pattern_match(mn::AbstractRuleNode, rn::AbstractRuleNode,
     vars::Dict{Symbol, AbstractRuleNode})
    pattern_match(rn, mn, vars)
end

"""
Pairwise tries to match two ordered lists of AbstractRuleNodes.
Typically, this function is used to pattern match the children two
     AbstractRuleNodes.
"""
function pattern_match(rns::Vector{AbstractRuleNode}, mns::Vector{
    AbstractRuleNode}, vars::Dict{Symbol, AbstractRuleNode})::
    PatternMatchResult
    @assert length(rns) == length(mns) "Unable to pattern match
    rulenodes with different arities"
    match_result = PatternMatchSuccess()
    for child_match_result in map(tup -> pattern_match(tup[2][1],
    tup[2][2], vars), enumerate(zip(rns, mns)))
        @match child_match_result begin
            ::PatternMatchHardFail => return child_match_result;
            ::PatternMatchSoftFail => (match_result =
    child_match_result); #continue searching for a hardfail
            ::PatternMatchSuccess => (); #continue searching for a
     hardfail
            ::PatternMatchSuccessWhenHoleAssignedTo => begin
                if !(match_result isa PatternMatchSuccess)
                    return PatternMatchSoftFail(child_match_result
    .hole)
                end
                match_result = child_match_result;
            end
        end
    end
    return match_result
end

"""
Comparing any `AbstractRuleNode` with a named `VarNode`
"""
function pattern_match(rn::AbstractRuleNode, var::VarNode, vars::Dict{Symbol, AbstractRuleNode})::PatternMatchResult
    if var.name in keys(vars)
        return pattern_match(rn, vars[var.name])
    end
    vars[var.name] = rn
    return PatternMatchSuccess()
end

"""
Comparing any `AbstractRuleNode1 with a `DomainRuleNode`
"""
function pattern_match(node::AbstractRuleNode, domainrulenode::DomainRuleNode, vars::Dict{Symbol, AbstractRuleNode})::PatternMatchResult
    if isfilled(node)
        #(RuleNode, DomainRuleNode)
        if !domainrulenode.domain[get_rule(node)]
            return PatternMatchHardFail()
        end
        return pattern_match(get_children(node), get_children(domainrulenode), vars)
    else
        #(AbstractHole, DomainRuleNode)
        if are_disjoint(node.domain, domainrulenode.domain)
            return PatternMatchHardFail()
        end
        if length(get_children(domainrulenode)) != length(get_children(node))
            return PatternMatchSoftFail(node)
        end
        children_match_result = pattern_match(get_children(node), get_children(domainrulenode), vars)
        @match children_match_result begin
            ::PatternMatchHardFail => return children_match_result;
            ::PatternMatchSoftFail => return children_match_result;
            ::PatternMatchSuccess => begin
                if is_subdomain(node.domain, domainrulenode.domain)
                    return children_match_result
```

```julia
                end
                intersection = get_intersection(node.domain, domainrulenode.domain)
                if length(intersection) == 1
                    return PatternMatchSuccessWhenHoleAssignedTo(node, intersection[1]) #exactly this value
                end
                return PatternMatchSuccessWhenHoleAssignedTo(node, intersection) #one of multiple values
            end
            ::PatternMatchSuccessWhenHoleAssignedTo => begin
                if is_subdomain(node.domain, domainrulenode.domain)
                    return children_match_result
                end
                return PatternMatchSoftFail(children_match_result.hole)
            end
        end
    end
end

"""
Comparing any pair of `Rulenode` and/or `AbstractHole`.
It is important to note that some `AbstractHole`s are already filled and should be treated as `RuleNode`.
This is why this function is dispatched on `(isfilled(h1), isfilled(h2))`.
The '(RuleNode, AbstractHole)' case could still include two nodes of type `AbstractHole`, but one of them should be treated as a rulenode.
"""
function pattern_match(h1::Union{RuleNode, AbstractHole}, h2::Union{RuleNode, AbstractHole}, vars::Dict{Symbol, AbstractRuleNode})::PatternMatchResult
    @match (isfilled(h1), isfilled(h2)) begin
        #(RuleNode, RuleNode)
        (true, true) => begin
            if get_rule(h1) != get_rule(h2)
                return PatternMatchHardFail()
            end
            return pattern_match(get_children(h1), get_children(h2), vars)
        end

        #(RuleNode, AbstractHole)
        (true, false) => begin
            if !h2.domain[get_rule(h1)]
                return PatternMatchHardFail()
            end
            if isuniform(h2)
                children_match_result = pattern_match(get_children(h1), get_children(h2), vars)
                @match children_match_result begin
```

```
                    ::PatternMatchHardFail => return children_match_result;
                    ::PatternMatchSoftFail => return children_match_result;
                    ::PatternMatchSuccess => return PatternMatchSuccessWhenHoleAssignedTo(h2, get_rule(h1));
                    ::PatternMatchSuccessWhenHoleAssignedTo => return PatternMatchSoftFail(children_match_result.hole);
                end
            end
            if !h2.domain[get_rule(h1)]
                return PatternMatchHardFail()
            end
            if isempty(h1.children)
                return PatternMatchSuccessWhenHoleAssignedTo(h2, get_rule(h1))
            end
            return PatternMatchSoftFail(h2)
        end

        #(AbstractHole, RuleNode)
        (false, true) => pattern_match(h2, h1, vars)

        #(AbstractHole, AbstractHole)
        (false, false) => begin
            if are_disjoint(h1.domain, h2.domain)
                return PatternMatchHardFail()
            end
            if isuniform(h1) && isuniform(h2)
                children_match_result = pattern_match(get_children(h1), get_children(h2), vars)
                @match children_match_result begin
                    ::PatternMatchHardFail => return children_match_result;
                    ::PatternMatchSoftFail => return children_match_result;
                    ::PatternMatchSuccess => return PatternMatchSoftFail(h1);
                    ::PatternMatchSuccessWhenHoleAssignedTo => return PatternMatchSoftFail(children_match_result.hole);
                end
            end
            return PatternMatchSoftFail(isuniform(h1) ? h2 : h1)
        end
    end
end
```

B.2 Symbolic Grammar Constraints

```
_t1,_t2,_t3,_u1,_u2,_u3,_b1,_b2,_b3,_unique = 1,2,3,4,5,6,7,8,9,10

a = VarNode(:a)
b = VarNode(:b)
```

```julia
c = VarNode(:c)

t1 = RuleNode(_t1)
t2 = RuleNode(_t2)
t3 = RuleNode(_t3)

u1(c1) = RuleNode(_u1, [c1])
u2(c1) = RuleNode(_u2, [c1])
u3(c1) = RuleNode(_u3, [c1])

b1(c1, c2) = RuleNode(_b1, [c1, c2])
b2(c1, c2) = RuleNode(_b2, [c1, c2])
b3(c1, c2) = RuleNode(_b3, [c1, c2])

unique = RuleNode(10)

constraints = Vector{AbstractGrammarConstraint}()

# contains constraints
push!(constraints, Contains(_t1))
push!(constraints, Contains(_t2))
push!(constraints, Contains(_t3))

# contains subtree constraints
push!(constraints, ContainsSubtree(b3(unique, a)))
push!(constraints, ContainsSubtree(b3(a, u1(a))))
push!(constraints, ContainsSubtree(b2(a, u1(b))))
push!(constraints, ContainsSubtree(b1(u2(b), a)))

# unique constraint
push!(constraints, Unique(_unique))

# forbidden sequence constraints
push!(constraints, ForbiddenSequence([_b1, _b2, _t2]))
push!(constraints, ForbiddenSequence([_b1, _b3, _t2]))
push!(constraints, ForbiddenSequence([_u1, _t1]))
push!(constraints, ForbiddenSequence([_u2, _t1]))
push!(constraints, ForbiddenSequence([_u3, _t3]))

# forbidden constraints
push!(constraints, Forbidden(b1(a, a)))
push!(constraints, Forbidden(b2(a, a)))
push!(constraints, Forbidden(b2(a, b1(b, b))))
push!(constraints, Forbidden(b2(u1(a), b)))
push!(constraints, Forbidden(u1(u2(u3(a)))))

# ordered constraints
push!(constraints, Ordered(b1(a, b), [:a, :b]))
push!(constraints, Ordered(b1(b1(a, b), u1(c)), [:a, :b, :c]))
push!(constraints, Ordered(b1(u1(c), b1(a, b)), [:a, :b, :c]))
```

Appendix C

Exact Results

C.1 Robot Grammar

	Without Constraints		**With Constraints**	
Size	Program Space	Runtime (s)	Program Space	Runtime (s)
1	1	0.000	1	0.000
2	7	0.000	7	0.000
3	43	0.000	33	0.001
4	259	0.000	133	0.002
5	1 555	0.002	469	0.006
6	9 331	0.017	1 457	0.022
7	55 987	0.093	4 025	0.071
8	335 923	0.548	9 997	0.202
9	2 015 539	3.270	22 605	0.538
10	12 093 235	23.848	47 129	1.306
11	72 559 411	141.965	91 665	2.855
12	435 356 467	844.853	168 021	6.010
13	2 612 138 803	(*)	292 741	12.288
14	15 672 832 819	(*)	488 257	23.258
15	94 036 996 915	(*)	784 169	40.322
16	564 221 981 491	(*)	1 218 653	69.516
17	3 385 331 888 947	(*)	1 839 997	116.150
18	20 311 991 333 683	(*)	2 708 265	182.691
19	121 871 948 002 099	(*)	3 897 089	292.376
20	731 231 688 012 595	(*)	5 495 589	447.164
21	4 387 390 128 075 571	(*)	7 610 421	681.843

(*) Instead of actual enumeration, the theoretical number of programs was calculated with $\sum_{k=0}^{n-1} 6^k$.

C.2 Arithmetic Grammar

	Without Constraints		With Constraints	
Size	Program Space	Runtime (s)	Program Space	Runtime (s)
1	11	0.000	11	0.000
2	11	0.000	11	0.000
3	374	0.000	201	0.001
4	374	0.000	201	0.001
5	24 332	0.039	7 798	0.018
6	24 332	0.039	7 798	0.018
7	2 000 867	3.190	383 688	0.814
8	2 000 867	3.852	383 688	0.819
9	184 632 701	361.538	21 192 628	43.842
10	184 632 701	357.280	21 192 628	45.778
11	18 265 184 267	35048.885	1 254 647 849	2592.975

C.3 Symbolic Grammar

	Without Constraints		With Constraints	
Size	Program Space	Runtime (s)	Program Space	Runtime (s)
1,	4	0.000	0	0.000
2,	16	0.000	0	0.000
3,	100	0.000	0	0.001
4	640	0.001	0	0.002
5	4 708	0.010	0	0.007
6	35 920	0.072	0	0.020
7	287 236	0.550	0	0.056
8	2 355 328	4.469	0	0.146
9	19 763 524	43.766	0	0.406
10	168 628 240	377.454	0	1.143
11	1 459 357 732	3213.797	0	3.011
12	-	-	11	8.752
13	-	-	108	24.491
14	-	-	2597	71.632
15	-	-	27667	210.122
16	-	-	345428	653.034

C.4 List Grammar

	Without Constraints		With Constraints	
Size	Program Space	Runtime (s)	Program Space	Runtime (s)
1	1	0.000	1	0.000
2	3	0.000	1	0.000
3	13	0.000	6	0.001
4	51	0.001	6	0.001
5	217	0.002	31	0.003
6	951	0.010	56	0.008
7	4 297	0.036	206	0.018
8	19 887	0.124	556	0.041
9	93 757	0.513	1 656	0.092
10	448 875	2.172	5 381	0.216
11	2 176 261	8.963	16 006	0.580
12	10 663 563	37.675	53 631	1.159
13	52 724 209	150.302	166 881	2.681
14	262 718 895	771.796	548 256	6.346
15	1 317 979 105	3629.086	1 779 631	15.498
16	-	-	5 817 631	36.277
17	-	-	19 329 756	89.897
18	-	-	63 692 256	250.183
19	-	-	213 391 631	652.133