TECHNISCHE UNIVERSITEIT DELFT

MASTER OF SCIENCE THESIS IN COMPUTER SCIENCE

Generalizing Conflict Analysis in Program Synthesis

Ole POETH

Supervisors:
Dr. Sebastijan DUMANČIĆ
Tilman HINNERICHS

30th October 2025

Delft University of Technology

Generalizing Conflict Analysis in Program Synthesis

Master's Thesis in Computer Science

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

Ole Poeth

30th October 2025

Author
Ole Poeth

Title
Generalizing Conflict Analysis in Program Synthesis

MSc presentation
7th November 2025

Graduation Committee

Dr. S. Dumančić (chair)	Delft University of Technology
Dr. A. Costea	Delft University of Technology
Dr. N. Yorke-Smith	Delft University of Technology
T. Hinnerichs, Msc.	Delft University of Technology

Abstract

Program synthesis aims to solve problems through coding by removing the need to write the programs yourself. Given the grammar and problem specification, it aims to find a program that adheres to your problem specification. This is done by iterating over many failing programs until a solution that adheres to the problem specification is found. Conflict analysis automatically takes these failing solutions and learns new constraints to make the search more efficient. Unfortunately, many conflict analysis techniques are heavily specialized. They are difficult to apply to diverse problems or when trying to use a different search algorithm. In this work, we present a modular framework in which these techniques can be implemented in a generalized way and applied independently to different problems and solvers, while having the ability to share generated constraints and parsed information. We identify two distinct conflict types and show how to use semantics in conflict analysis effectively. The framework is evaluated on two diverse domains; it prunes up to 96% of the search space, where combining techniques can further improve its average effectiveness. Domain applicability for the techniques has to be considered for optimal framework performance. Compared to an enumeration solver, the framework shows marginal improvements on a real-world benchmark. While the framework's overhead needs improvement, its modularity allows for comparison between solvers, problems, and conflict analysis techniques.

Preface

After a little more than 7 years, with this thesis, I have arrived at the final step of completing my master's degree in Computer Science. Having initially planned to become an air traffic controller, my last year in high school here in the Netherlands opened my eyes to the wonderful world of programming. Here at the TU Delft, I started my bachelor's in Computer Science and Engineering and instantly knew I wanted to continue to master the craft and pursue a career as a computer scientist.

During this thesis, I had the great opportunity to work with my supervisors Sebastijan Dumančić and Tilman Hinnerichs on this awesome project, contributing to the program synthesis framework, `Herb.jl`, and working with the PONY team.. It never felt like I was doing this thesis alone, they were always there to help, also during the difficult parts of the process, especially near the end of the thesis. I am very grateful to have been part of this team, and I wish everyone at the PONY lab an amazing future.

I also want to thank everyone I have met over all those years for bringing me to where I am now. I am very grateful for all the support I got from so many people, helping me through the lows but also celebrating the highs. Thank you, Delano and Daan, for the feedback you were still able to give so close to the deadline. This thesis is better because of you. And in particular, I want to thank my mom, my dad and Manon, for being my biggest fans. I could not have done it without you.

It has been a hell of a ride.

Ole Poeth

Delft, The Netherlands
30th October 2025

Contents

Chapter 1

Introduction

At its core, computer science is about using computers to solve our problems. When we write code and software, we teach computers how to systematically approach problems to eventually find a solution in the form of code. Writing code, however, is tedious, time-consuming, and, in many cases, difficult. Program synthesis is based on a simple idea: Let computers write their own code. At its core, program synthesis automates this process by generating and testing candidate programs until it finds a suitable one.

Naturally, humans learn by trial and error. Especially for code, we write, analyze, and refine our code snippets numerous times until we reach one that works. In this thesis, we investigate how to make the core idea of human learning from mistakes accessible to computers, so that they can write their own code.

Program synthesis [Gulwani et al., 2017] can be defined as a combinatorial optimization (CO) problem [Papadimitriou and Steiglitz, 1998], where we want to find a solution characterized by a set of options that must adhere to predefined predicates. CO originates from SAT-solving [Vizel et al., 2015], where we define a problem using logical variables and predicates, and by using different optimization techniques, we try to find a valid solution as quickly as possible. The SAT-solving field has created powerful algorithms that efficiently solve specific complex problems (e.g., scheduling problems). One of the downsides of SAT-solving is restricting the logical specification of a problem; program synthesis wants to offer more flexibility in this area.

Program synthesis aims to solve problems by finding a program within a provided search space that solves the given problem specification. The search space can be defined using a grammar, in any domain, from lists to strings to a custom robot-defined domain. Program synthesis enumerates candidate solution programs given a user specification, such as the input-output (IO) examples, until a program solves all user specifications. This specific program synthesis technique is called synthesis using examples [Gulwani, 2012]. Though powerful in theory, it is only effective on small domains or minor problems, as the number of candidate solutions exponentially grows with the grammar's size and the problem's complexity. However,

there are ways to improve the efficiency of enumerative program synthesis.

One way to improve the search is by limiting the search space using constraints [Hinnerichs et al., 2025c]. A constraint is a rule applied to a grammar rule or a (partial) candidate solution that prevents it from being enumerated by the program synthesis solver. Constraints can be created upfront or during the search, and they are usually based on expert knowledge of the final solution's (partial) functionality. Before evaluating a candidate solution, it is checked against the constraints, and if one is violated, the search ignores the candidate. This improves efficiency by only evaluating programs deemed 'valid' by the knowledge base of constraints. Propagating choices while constructing a new solution can make constraints even more potent, omitting the need to check complete programs. However, constraints do not guarantee that all evaluated candidate solutions directly solve the problem. The effectiveness of constraints heavily depends on how they are defined, and many valid candidate solutions still fail to solve the specifications.

With conflict analysis, also known as conflict-driven learning [Feng et al., 2017], we generate constraints on the fly, based on failing candidates, and automatically extract the reason for failure. Most synthesis techniques ignore failing candidates and keep enumerating new programs until they find a program that solves all specifications. However, in many cases, there is a specific reason why a program fails the specification and has not yet solved the problem. Behind this reason, a more general, encapsulating behavior could be extracted, which could also occur in future candidates. If we can automatically extract this faulty behavior and turn it into a constraint, we can iteratively reduce the search space. In most cases, some extra information or making implicit knowledge explicit is needed for conflict analysis to work effectively. This information only has to be provided upfront once or can be updated automatically during the search, minimizing overhead while significantly reducing the search space.

While several conflict analysis techniques have shown impressive results, they are difficult to generalize to other domains. Popper[Cropper and Morel, 2020] is a unique example of a program synthesis model that uses conflict analysis to improve search times, as it is one of the only approaches that combines multiple techniques. Popper does synthesis-by-example and learns from failure programs, and over the years, has been expanded with more different conflict analysis techniques[Cropper and Hocquette, 2022, Morel and Cropper, 2023] within the same model. One of Popper's drawbacks is that it can only solve problems defined in the logic programming domain[Cropper and Dumancic, 2022]. This restricts the variety of problems it can solve and eliminates the possibility of other non-logic-defined conflict analysis techniques being implemented within the same environment.

Other techniques like [Wang et al., 2017, Guria et al., 2023, So and Oh, 2017] show great results on more diverse domains, with [Guria et al., 2023] even working on language-agnostic abstract domains. However, they still lack the adaptation to different solvers while depending on specific available information. Neo [Balog et al., 2017] shows the most flexibility of the conflict analysis techniques, yet it does not show how to apply this in practice.

In this thesis, our goal is to investigate whether we can take advantage of the diversity of conflict analysis techniques through generalization by answering the following question:

How can conflict analysis be generalized to improve efficiency in program synthesis?

This thesis presents a solver-independent modular framework for integrating multiple conflict analysis techniques in program synthesis. It enables a systematic investigation of how different techniques can reduce search space across diverse domains using any solver that provides programs in the form of an abstract syntax tree. We identify two fundamental conflict types: semantic-based and execution-based. We demonstrate how we generate structural and grammar constraints from these conflicts that prune invalid programs from the search space. We show that when defining semantics for your grammar, the abstraction level, semantic diversity, and structural overlap of grammar rules are essential to the effectiveness of conflict analysis. Additionally, our framework provides a novel opportunity to share information between techniques on the same and subsequent conflicts, potentially improving efficiency and effectiveness.

We show the adaptation of three conflict analysis techniques to the framework, and how these can effectively reduce the search for different problems in the same configuration. We show how combining conflict analysis techniques can increase the pruning efficiency compared to individual performance. We test our framework on top of a naive synthesis solver on a competition-level benchmark, SyGuS. Our framework improves the number of problems that can be solved and shows a reduction in the number of enumerated programs. However, the significance of the reduction depends on the problem and how suitable it is for the applicability of a technique.

Our framework is a powerful tool for making conflict analysis more accessible for researching its applicability to any problem or solver. Due to its modularity, almost any conflict analysis technique and solver can be adapted and interchanged within the framework in a structured and intuitive way. It is a powerful tool for investigating different configurations of solvers, problems, and conflict analysis techniques. This allows for a fair comparison of these configurations and a better understanding of the behavior and workings of all individual components.

With our framework, we want to inspire more research on adapting and generalizing conflict analysis techniques and, by implementing them with existing methods, show that we can solve more difficult and varied problems. We encourage the adaptation of the framework in more fields in program synthesis, investigating the power of conflict analysis in fields like divide-and-conquer and heuristic search. Examples of future framework optimization improvements include applying parallelization to the pipeline, adding more existing conflict analysis techniques, improving implemented techniques, learning more from individual conflicts per technique, and combining constraints to increase single constraint efficiency. Additional future work can investigate whether library learning can be applied to previously generated constraints to reduce the search space of future problems.

Chapter 2

Background

2.1 Program Synthesis

Program synthesis aims to solve problems by enumerating programs from a search space that adheres to a problem specification[Gulwani et al., 2017]. This can be seen as a combinatorial satisfaction problem, as a program synthesis model enumerates over the search space, which contains programs defined by some combination of syntactic or semantic elements. In this thesis, we want to reduce the search space size using conflict analysis in program synthesis to efficiently solve more diverse and complex problems. Understanding the actual problem and where the most significant improvement can be found are the first steps to getting to this result.

2.1.1 Search Space

Program synthesis uses a set of programs to enumerate over and check whether they contain a solution satisfying the problem specification. The search space defines this set, but it cannot enumerate every existing program. This can be an infinite set and contains a lot of diversity in programs. It would enumerate many programs that do not come close to the desired solution. The first step to finding a solution efficiently is to narrow the search space and define a specific domain of programs. Intuitively, when defining a problem, you have some general idea of how it could be solved or what components you need to solve it, but not the actual solution. The program synthesis model can create a much more manageable search space by defining this knowledge in a structured way. The search space will contain programs much closer to a solution than checking every possible program.

Imagine a robot and a maze, and the goal is to write a program that makes the robot solve the maze automatically. The robot can move only a certain way, and the program is defined in terms of these steps. When defining the search space, it only needs the actual actions that the robot can perform. Restricting the search space then dramatically improves the program synthesis model's effectiveness. One popular way of restricting the search space in program synthesis is to use a grammar.

The size of the search space and the complexity of problems a model can solve with a grammar heavily depend on its content.

Grammar

A grammar provides a set of rules that specify how valid programs can be constructed from atomic components, such as operations, constants, and variables. Grammar rules are either terminal or non-terminal, where non-terminal rules can be expanded by replacing members of the rule with other terminal or non-terminal rules. If every non-terminal rule can be expanded with any other rules without any surrounding context, the grammar is called context-free (CFG). CFGs are popular due to their expressiveness and simplicity.

Figure 2.1 is an example of such a CFG of the robot domain example. A program constructed from a grammar can be visualized as a tree structure, where a program is complete and valid when all leaves are terminals. Grammars can construct programs of any size and complexity by enumerating infinite combinations of rules. To avoid running a program synthesis model indefinitely, candidate solutions are usually restricted to a specific tree depth or size, the number of candidate solutions is bounded, or the enumeration is set to a timeout.

Bigger programs usually do not result in better solutions, and the chance of finding solutions in bigger programs diminishes as the number of equally sized programs increases exponentially. By carefully designing the grammar, it can encode domain-specific knowledge and restrict the solution space to programs that are simple, interpretable, and efficient to evaluate. However, under-specifying the grammar can lead to the solution falling outside the search space. This results in a trade-off between grammar simplicity and fruitful search spaces, requiring expert knowledge on the problem and creating well-designed grammar.

Semantics

Beyond syntax, synthesis systems could also reason about programs' semantics[Gunter, 1992]; the meaning of a program or the behavior it produces when executed. Semantics reason about programs on a higher level of abstraction, making it possible to group a set of concrete programs under a single semantic behavior. This can help the synthesis process be more efficient by using these semantics to guide the search to parts of the search space that adhere to the same semantic behavior as the problem. Or, by pruning parts of the search space that do not adhere to the semantic behavior of the problem.

A Domain-Specific Language (DSL) uses semantics to be more expressive in its definition of the problem, as it can be seen as an extension of a previously described grammar. On top of the syntactic encapsulation the grammar provides, a DSL allows you to define the semantic behavior of the grammar rules and the problem specification. Combining these techniques creates a powerful basis for a

Robot Grammar

Non-terminals:

```
Program → Action
Program → If-Then
Program → Sequence

Sequence → Action ; Program

If-Then → if Condition then Program
```

Terminals:

```
Condition → obstacleAhead | atGoal

Action → moveUp | moveDown | moveLeft |
        moveRight | pickUp
```

Figure 2.1: Robot Grammar

program synthesis model, having multiple tools to expressively describe the search space you operate within.

2.1.2 Problem Specification

Grammars and DSLs describe the search space of possible programs, but defining the problem correctly is just as important as eventually solving it. The problem specification formally describes what the desired program should do. Creating valid solutions for an ill-defined problem will not contribute anything valuable. Specifications can take various forms, including input-output examples, logical constraints, natural language descriptions, or combinations thereof. The most concrete way of defining a problem is using specific input-output examples[Gulwani, 2012]. These examples give a clear starting state and goal, which is simple to check for a synthesis model. More expressive specifications may involve universal quantification (e.g., a property must hold for all inputs) or partial specifications that under-constrain the space of solutions. The quality and completeness of the specification heavily influence the success of synthesis: a weak specification may allow many unintended programs. At the same time, an overly strict or inconsistent specification may make solving the problem impossible.

2.1.3 Constraints and Program Synthesis

When applied correctly, constraints can play a fundamental role in program synthesis by filtering out incorrect or irrelevant programs from the search space [Kadioglu and Sellmann, 2008, Ahlgren and Yuen, 2013]. Rather than exhaustively evaluating every candidate program, the problem specification, semantic correctness, or structural requirements can be encoded into constraints that candidate programs must satisfy. These constraints are typically logical formulas that express conditions over a program's inputs and outputs or intermediate expressions derived from the grammar.

Constraints enable the synthesis system to reason about program correctness symbolically and to reject entire classes of programs that violate the constraints, without evaluating them[Hinnerichs et al., 2025c]. Moreover, constraints allow partial candidate solutions to be pruned, potentially restricting enormous parts of the search space. For example, if the specification states that a program must always return a sorted list, any candidate whose structure violates this condition can be excluded based on symbolic reasoning alone.

The expressiveness of constraints allows the synthesis system to model complex properties such as functional correctness, safety, or even resource usage. For instance, constraints can encode properties like "the output list must be a permutation of the input" or "no index out-of-bounds errors may occur during execution". However, with increased expressiveness comes additional computational overhead. More complex constraints typically require more variables, nested quantifiers, or advanced theories (e.g., arithmetic, arrays, strings), significantly impacting solver performance. Consequently, the benefit of pruning the search space must be weighed against the time and memory required to solve the constraint model. A highly expressive but inefficient constraint may slow synthesis more than it helps, especially if the solver struggles to find valid solutions. Therefore, carefully choosing the level of detail and abstraction in constraint encoding is essential to balancing pruning power and solver efficiency. In constraint-based synthesis, the grammar and semantics are often encoded together with the specification into a unified constraint model. Each rule application in the grammar introduces symbolic variables and relations, while the specification constraints relate inputs and expected outputs through these variables. The solver then attempts to find solutions for the symbolic variables that satisfy all constraints and correspond to valid programs. If such a solution exists, it can be decoded into a concrete program; otherwise, the solver returns *unsatisfiable*, indicating no such program exists in the constrained space.

This thesis focuses on *syntactic constraints*, which are defined as explicit (sub)programs containing grammar rules matching (parts of) candidate programs. Hinnerichs et al. [2025c] uses this type of constraint to model program spaces, removing unwanted candidate programs while searching, applying this in their solver called `BART`. It can take any unwanted program (See Example 1) and use a syntactic constraint, like *Forbid*, which tells the solver to prune any program having the program

defined in the *Forbid* constraint as a subprogram.

Example 1. Using the robot grammar from Figure 2.1, take the unwanted program $moveUp\,;\,moveDown$. This program does nothing and would be unnecessary to check during enumeration.

How this constraint is enforced is solver-dependent; while enumerating, you can check every complete program and ignore those violating the constraint, or already ignore options of incomplete programs, which would lead to the violation of the constraint. `BART` shows that the latter results in a more efficient solving process as you prune unwanted programs earlier.

This formulation has several advantages: it avoids unnecessary enumeration, allows partial or incremental solving, and enables integration with other formal techniques such as verification or type checking. However, constraint-based synthesis also introduces new challenges: encoding the synthesis task correctly is non-trivial, and the size or complexity of the constraints can sometimes overwhelm the solver. Moreover, overly restrictive constraints may eliminate all valid programs, while weak constraints may result in too many candidates. Balancing constraint expressiveness and solver efficiency is key to practical synthesis.

2.2 Constraint Programming

Constraint programming (CP) is a field in combinatorics that focuses on solving various problems using constraints [Rossi et al., 2008]. A CP solver iterates over the search space of all possible solutions in a given domain until it finds a solution to the problem. A constraint defines properties of the solution instead of defining specific steps within the solving process. Constraints allow for detecting inconsistencies within the partial solution before constructing a final solution. These inconsistencies can then be used to backtrack within a solution, resulting in a more efficient search. *Constraint logic programming* was one of the first practical applications of CP [Jaffar and Maher, 1994]. Logic programming defines its problems and solutions in the form of formal logic. Variables and constraints are written in the form of clauses that present the problem in a structured way. Logic programming can reduce ambiguity within the problem and focus on the domain at hand. However, reducing ambiguity also restricts the expressiveness of the problem, solution, and variety of solvable issues.

2.2.1 Constraint Satisfaction Problem

The *constraint satisfaction problem* (CSP) is a specific problem that CP tries to solve. CSP aims to find a solution that satisfies all the constraints defined. Multiple solutions to the problem may exist, but in CSP, a single solution is already satisfactory. For specific problems, this can already be a challenging feat. Constraints are necessary as the problems CSP tries to solve usually become intractable when growing larger. A CSP can be formally defined as follows:

Definition 1 (Constraint Satisfaction Problem (CSP))**.** A *Constraint Satisfaction Problem (CSP)* is defined by a triple (X, D, C) where:

- $X = \{x_1, x_2, \ldots, x_n\}$ is a finite set of variables,
- $D = \{D_1, D_2, \ldots, D_n\}$ is a set of domains, where each D_i is the finite set of possible values for variable x_i,
- $C = \{c_1, c_2, \ldots, c_m\}$ is a set of constraints, where each constraint c_j specifies a relation over a subset of the variables.

A *solution* to a CSP is an assignment of values to all variables $x_i \in X$ such that every constraint $c_j \in C$ is satisfied.

An example of a CSP in logic programming is a Boolean satisfiability problem, also called SAT[Vizel et al., 2015], which only uses Boolean values to describe a problem setting. Well-defined constraints can reduce the solution space the CSP solver has to search, improving its efficiency. One should be wary of accidentally removing the final solution when adding these constraints. Well-defining these constraints is one of the challenges in CSP. Mainly doing this by hand, which can be extremely difficult.

2.2.2 Conflict Analysis

Conflict analysis wants to automate the creation of constraints in CP using conflicts that occur while solving. A conflict describes violating one or multiple constraints for a specific solution. When detecting this conflict in CP, it typically backtracks to a different solution. However, one can also learn from this conflict. One of the first techniques in the field of CP that applies conflict analysis is *conflict-driven clause learning* (CDCL) in logic programming[Marques-Silva et al., 2009]. CDCL is used in SAT-solvers to derive new clauses from a conflict. It starts with deciding and propagating Boolean assignments until a conflict arises, meaning that a constraint cannot be satisfied with the current solution. The algorithm then *explains* the conflict by creating a new clause that would prevent the conflict from happening in the first place. This clause is added to the model, and the solver backtracks non-chronologically to where the conflict occurred. However, these learned clauses can build up quickly, resulting in a limitation dependent on the available memory. Hence, one of the many improvements this algorithm has seen is forgetting clauses when they become obsolete or do not effectively prune the solution space.

Chapter 3

Related work

3.1 Solving ILP Problems using Conflict-Driven ASP

Popper [Cropper and Morel, 2020] is a system that also uses a version of conflict analysis to learn from failing programs. They use an answer set programming (ASP) solver called *clingo*[Gebser et al., 2014] to enumerate programs and predicates to define their constraint model. The programs and predicates are defined in the logic domain, allowing for sound conflict-driven learning. Popper uses multiple techniques to learn predicates from its iterated programs [Cropper and Morel, 2020, Morel and Cropper, 2023, Cropper and Hocquette, 2024], showing the effectiveness of combining analysis techniques. One of Popper's limitations is the restriction of its domain, which is limited to the family of logic programming languages. Our framework supports a broader range of domains by using a grammar to define the problem and the domain. Using a parser for the logic domain, one could integrate Popper into our framework, possibly allowing the conflict-driven learning of Popper to apply beyond its intended domain.

ILASP[Law, 2022] is another ASP solver that tries to learn definite logic programs, given some specification. It uses conflict-driven ILP to extract coverage constraints from conflicts. The iterative approach generates hypotheses and tests them against the specification. If the specification fails, it tries to explain the failure through a combination of positive and negative examples. This explanation is translated into a coverage constraint to either include positive examples that were not included before or prevent negative examples from being included. This paper describes different techniques on different levels of complexity and the size of these coverage constraints. Depending on the problem, these can improve the performance of the solver.

ILASP is a complex conflict analysis system tailored to the logic domain. Generalizing the ideas to other domains is difficult due to the nature of the explicit implementation. With a logic domain parser, we believe this can be useful to specific problems, but would remain mostly outside the framework, due to its low generalizability.

3.2 Conflict Analysis using Semantics and Statistical Guidance

Neo [Balog et al., 2017] is one of this thesis's inspirations, as it uses a similar structure for its conflict analysis to our framework. By clearly separating the steps within their search process, they allow for modularity and adaptation. In the paper, they describe a specific implementation for every one of the steps: *Decide*, *Propagate*, *CheckConflict*, *AnalyzeConflict*. The first two steps result in a valid candidate program, which, in the first step, is generated by expanding holes in an AST, chosen by a probabilistic model. *Propagate* then verifies whether the choice for expanding the hole is valid, given the current constraint model. For the conflict analysis, Neo uses semantics to check whether a program fails on a semantic level by leveraging the idea from SMT-solvers of the *Minimal Unsat Core*. This MUC identifies the nodes in the AST that are causing the conflict on a semantic level. From this information, *AnalyzeConflict* will generate constraints that avoid the iteration of programs failing for the same semantic reason.

The paper advocates for the option of changing each of the steps in the algorithm for different approaches. However, what it would look like on an implementation level is unclear, and we aim to clarify this when using our framework. In addition, we remove the need to adapt the program iteration completely by making the framework's entry point the candidate program. With other inputs like grammar and semantics, we have all the information needed for the conflict analysis, making the exchange of search techniques easier. Adapting the constraint generation might be necessary when using different search techniques, but our framework allows us to make this as universal as possible.

3.3 Automatically Applying Predicates using FTA

Blaze [Wang et al., 2017] is based on *Finite Tree Automata* to iterate programs and adds an abstraction layer (AFTA) of semantic predicates as its constraint model. These predicates are selected from a predefined universe, hand-picked by a domain expert. The predicates are semantic rules that can be applied to nodes in the AFTA search process to tighten bounds and remove options of grammar rules. Blaze analyzes a conflict by comparing the output of the candidate program to the desired output. It finds the weakest predicate explaining why the program is failing and gives that to the AFTA solver as a constraint. This will tighten the bounds on the possible programs the AFTA solver can return. The number of programs that must be checked is reduced as the semantic predicates remove more programs than the single one from the conflict, resulting in faster solve times.

Blaze is explicitly implemented, not allowing much freedom to add improvements. Additionally, the search technique is difficult to swap, but could be a powerful one to apply to other techniques and domains. With our framework, we believe we can adapt the AFTA solver as a search technique and try it with other conflict

analysis techniques. We believe the conflict analysis technique can be generalized for domains and search techniques.

3.4 Semantically-guided Synthesis

Several synthesis approaches leverage semantic information to guide the search for correct programs. Absynthe[Guria et al., 2023] uses abstract interpretation to prune the synthesis search space by computing over-approximations of program semantics. The framework maintains language-agnostic abstract domains that capture the semantic properties of partial programs, allowing it to eliminate candidate programs that cannot satisfy the specification before complete enumeration. Similarly, Synthesizing Imperative Programs from Examples Guided by Static Analysis[So and Oh, 2017] uses static analysis to extract semantic constraints from input-output examples, guiding the enumerative search by filtering out programs that violate these constraints early in the synthesis process.

While these approaches use semantic information to guide the synthesis search, our framework uses semantics for conflict analysis when synthesis fails. However, the complementary nature of these techniques could allow for a potential union. Since both Absynthe and our framework maintain semantic abstractions during synthesis, our conflict analysis capabilities could be integrated with these solvers to provide even more pruning opportunities while searching. Absynthe's language-agnostic design, in particular, aligns well with our framework's architecture, suggesting that such an integration could benefit from solver-independent conflict explanations across multiple programming languages.

Chapter 4

Problem Statement

Program synthesis aims to construct programs that satisfy a given specification automatically. Additional constraints, semantic structure, and learning from failed candidates are necessary to enhance this process for more realistic problem settings. This section formalizes these concepts through the following definitions.

Definition 2 (Program Synthesis)**.** Given a context-free grammar $\mathcal{G}$ that specifies the syntax of the programming language and program space $\mathcal{P}_\mathcal{G}$ (derived from $\mathcal{G}$), and a specification of user intent $\mathcal{S} : \mathcal{P}_\mathcal{G} \to \{0, 1\}$, **find** a program $p \in \mathcal{P}_\mathcal{G}$ such that $\mathcal{S}(p) = 1$.

Definition 3 (Constrained Program Space)**.** Given a context-sensitive grammar $\mathcal{H}$ with program space $\mathcal{P}_\mathcal{H}$ and a set of constraints $\mathcal{C} = \{\phi_1, \ldots, \phi_n\}$ where each ϕ_i is a predicate over programs in $\mathcal{P}_\mathcal{H}$, the **constrained program space** $\mathcal{P}_{\mathcal{H},\mathcal{C}}$ is defined as:

$$\mathcal{P}_{\mathcal{H},\mathcal{C}} = \{p \in \mathcal{P}_\mathcal{H} \mid \forall \phi \in \mathcal{C},\ p \models \phi\}$$

That is, $\mathcal{P}_{\mathcal{H},\mathcal{C}}$ contains all programs derivable from $\mathcal{H}$ that satisfy all constraints in $\mathcal{C}$.

A context-free grammar (CFG) with rules restricting the program space using constraints is called a context-sensitive grammar (CSG).

Definition 4 (Semantic Annotation)**.** A grammar rule $r \in \mathcal{G}$ is a production of the form $A \to \alpha$, where A is a symbol and α is a sequence of terminals and/or nonterminals. A **semantic annotation** of r is a set of logical constraints ψ_r that expresses properties over the inputs and outputs of the program fragment generated by r.

These annotations capture semantic properties such as value preservation, structural invariants, or relational constraints. For example, a rule generating a sorting function may include the annotation:

$$\texttt{sort}(x) := \{\texttt{length}(\texttt{out}) = \texttt{length}(x)\}$$

Semantic annotations can reason about partial program behavior, enabling conflict analysis techniques to generalize and exclude invalid candidates based on violated semantic properties.

Using the notions of constrained program spaces and semantic property annotations, we define the problem of conflict analysis as follows.

Definition 5 (Conflict)**.** Given a context-sensitive grammar $\mathcal{H}$ with program space $\mathcal{P}_{\mathcal{H}}$, a set of constraints $\mathcal{C}$, and a specification $\mathcal{S} : \mathcal{P} \to \{0, 1\}$, a **conflict** is a partial or complete program $p \in \mathcal{P}_{\mathcal{H},\mathcal{C}}$ such that $\mathcal{S}(p) = 0$.

Conflict analysis identifies semantic properties of p that are responsible for the violation of $\mathcal{S}$. This may involve examining which grammar rules were applied in constructing p, and which *semantic property annotations* were violated. From this analysis, a new constraint ϕ_c is derived such that:

$$\forall p \in \mathcal{P}_{\mathcal{H}},\ p \models \phi_c \Rightarrow \mathcal{S}(p) = 0$$

That is, ϕ_c captures a generalization of the failure, covering p and all other programs with the same faulty structure or semantics.

The synthesis process then adds the negation of this constraint, $\neg\phi_c$, to the constraint set $\mathcal{C}$, thereby refining the constrained program space $\mathcal{P}_{\mathcal{H},\mathcal{C}}$ and pruning future candidates that share the identified conflict.

Chapter 5

Methods

In this chapter, we present our proposal, which consists of a conflict analysis framework and the implementation of multiple conflict analysis techniques within it. Additionally, we extended the effectiveness of semantics by parsing its contents as individual programs, enabling 'custom-defined' semantics to allow more freedom when defining semantics.

Current conflict analysis solutions lack flexibility in multiple aspects of the solution process. Most of the techniques are restricted in their usage and implementation. This can be in relation to the problem space, the program representation, or the search technique. There is no generalized way to implement these techniques, so they are usually implemented with problem-specific properties in mind. Moreover, these techniques are usually also restricted to a specific solver, making them even more inflexible. This makes applying any technique to a different problem space challenging, although the technique can be extrapolated to work with all kinds of problems.

With our framework, we want to separate the analysis from the specific problem or language by providing a general form of program representation through abstract syntax trees (ASTs). We analyze the conflicts separately from the solver, which allows us to run different search techniques. This allows the framework to analyze various problems flexibly while minimizing the effort required to switch between problems and search techniques. One does have to take into account the fact that a new technique may need to be adapted initially to fit the framework.

Multiple additional upsides of this setup lie in the centralized environment where these techniques reside. One upside is that the data can now be shared between techniques while running them simultaneously, potentially improving the solving process by filling the gaps of the other techniques. Though different techniques analyze conflicts differently, specific techniques can overlap in the data they use and the generated constraints. For example, techniques relying on semantics share the same input data, namely a DSL representation, containing the abstract behavior of a given problem. At the same time, intermediate results from smaller constraints can be used as cached data to create more impactful or larger constraints later in

the solving process. Additionally, the framework allows for a better overview of the entire solving environment, resulting in a clearer understanding of the process. Lastly, comparing different conflict analysis techniques in the same environment becomes trivial, allowing for a fair comparison of their effectiveness.

5.1 Types of Conflicts

Knowing what types of conflicts you are dealing with is essential when designing and implementing new techniques in conflict analysis. Not knowing the scope of the technique can hurt the solving process by adding constraints that remove too many candidate programs, potentially eliminating solutions. It can also harm the solving process by not removing enough candidate programs and not fully utilizing a technique's potential. In this thesis, we have identified two types of conflicts: `Semantic-based` and `Execution-based`.

5.1.1 Semantic-based

This type of conflict describes a faulty behavior using the given semantics of a problem. Semantics make implicit information about the problem or grammar explicit to the solver. This usually means information at a higher level of abstraction, e.g., the length of a list, if a number is odd, or if a string is numeric. In this thesis, we apply semantics to the grammar rules and compare them against the characteristics of (partial) results. Semantic conflicts can be a powerful tool for reasoning about programs with identical structures but different completions of grammar rules.

Take the following grammar $\mathcal{G}_1$ from Table 5.1, it resembles a list grammar together with some semantics defined per grammar rule. Here, y represents the output variable and x_n represents the nth input variable of the function defined in the production rule.

Grammar Rule	**Semantic Annotation**
1) `first(array)`	$max(y) \leq \max(x_1),\ \texttt{len}(y) \leq \texttt{len}(x_1)$
2) `sum(array)`	$\texttt{len}(y) \leq \texttt{len}(x_1)$
3) `first(array, number)`	$\max(y) \leq \max(x_1),\ \texttt{len}(y) \leq \texttt{len}(x_1)$
4) `sort(array)`	$\max(y) \leq \max(x_1),\ \texttt{len}(y) = \texttt{len}(x_1)$
5) `reverse(array)`	$\max(y) \leq \max(x_1),\ \texttt{len}(y) = \texttt{len}(x_1)$

Table 5.1: Simplified grammar $\mathcal{G}_1$ with associated semantic constraints for each production rule.

Table 5.1 shows a simplified grammar with an example of some properties.

We take the following program as a candidate solution as an example of how these annotations can be used to prune future programs.

$$y = first(reverse(sort(x)))$$

For our input/output example, we will use $x = [9, 15, 6, 7, 10, 3]$ and $y = 25$. If we run the input on this candidate program, the output will be $y = 15$, which fails to match the expected output of 25. We know the solver will enumerate all possible programs with this structure if no constraints are used. As humans, we can see that replacing any of these functions with anything but sum will fail. We know this intuitively, as some values in the list have to be combined to get to the output of 25. The only function to do this in this grammar is sum. We can automatically provide this intuitive information to the solver using semantics.

All the functions in this example have the semantic: $max(y) \leq max(x_1)$. This means that the maximum value of the array returned by these functions will never exceed the maximum value of the original input. Applying this to the input/output example, we see that it violates this semantic as the maximum value of x is 15 and $15 \leq 25$. We can save a significant amount of time by telling the solver to ignore all programs that do not contain sum in this specific structure.

The effectiveness of semantics

However, the effectiveness of this method and semantics generally depends on how you define your semantics in the grammar. The following are essential to take into account when building a grammar with semantics:

1. **Abstract semantics.** When using semantics in conflict analysis, you want to ensure your semantics are not too specific. A semantic is a property that defines a group of grammar rules that all adhere to that property. If the semantic is too specific, this group of grammar rules is small, reducing the pruning capacity.

2. **Diversity in semantics.** Another important aspect of defining your semantics is the need for diversity. If all your grammar rules have the same semantics, you can prune a significant amount if it fails, but almost nothing if it does not. You can prune more consistently when you define a diverse set of semantics that could fail for various reasons, depending on the given problem.

3. **Structural overlap between grammar rules.** If you want to prune effectively using semantics, you want to be able to replace the function(s) that fail your semantic. This means functions ideally have the same output and input signature. Otherwise, this replacement might not be correct and could even prune valid solutions.

The small example from Table 5.1 shows already a variety in semantics and overlap between the rules to prune redundant candidates while being able to solve the problem.

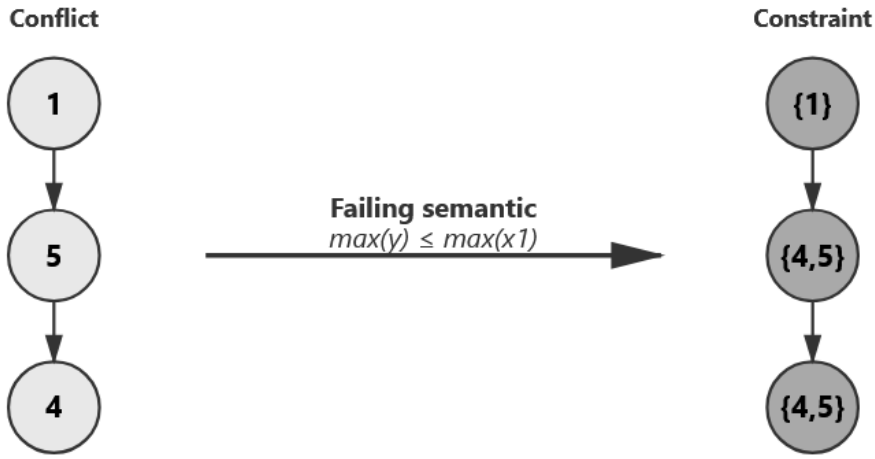


Figure 5.1: Example of a constraint being created from a semantic-based conflict. The program is failing on $\max(y) \leq \max(x_1)$, the constraint copies the conflict structure on the left, and every node gets a domain of rules representing all equal failing programs.

Structural Constraints

When we have determined the reason behind a semantic-based conflict, we have to tell the solver which specific programs it can ignore in the search process. By generating a constraint and passing this to the solver, we can offload this propagation to the solver while solving. This is more effective than applying constraints after enumeration, as shown in Hinnerichs et al. [2025b].

When generating constraints for semantic-based conflicts, we must be careful not to prune potential solutions accidentally. As semantic reasoning uses a higher level of abstraction, the structure of the current conflict matters. To ensure this correctness, we copy the structure of the candidate program and assign a list of grammar rules for every node, which corresponds to all the functions that would fail the semantic we found to be causing the conflict. Figure 5.1 shows what this kind of constraint would look like for the earlier example used in section 5.1.1. With this constraint, we tell the solver to prune all programs that you can iterate over this AST.

For this constraint, the programs defined by the constraint must match precisely with the iterated program to be pruned. Suppose it's different in any way, for example, by being a subprogram of a bigger program. In that case, we cannot ensure that this bigger program will fail under the original semantic, so we cannot prune those programs.

To underline this distinct property of these constraints, we call this type of constraint `structural constraints`.

5.1.2 Execution-based

Execution-based conflicts focus on concrete outcomes of the evaluation of candidate programs. This can be interpreted in multiple ways, as outcomes can differ significantly per example and problem. For this thesis, we focus on runtime exceptions and the evaluated outcomes of programs.

Runtime exceptions are a clear-cut conflict on the surface, as you don't even get to compare against an expected evaluation. However, depending on the exception type, the analysis can produce a completely different story of why the program is failing. Taking an arithmetic grammar as an example, this can return a 'division by 0' exception. Analyzing this conflict can seem pretty straightforward, and one of the simplest ways to use this information is to prune all programs that contain this specific program as a subprogram. Depending on the problem setting, this can already prune many programs. One can also analyze the program in more detail; if you can find the root cause of the problem, the smallest size of program that produces the same exception, the pruning capacity for that exception will be much greater than taking the original program.

The effectiveness of conflict analysis based on exceptions is problem-dependent. In a well-defined problem setting with a well-defined grammar, exceptions occur rarely or not at all. This significantly reduces the conflict analysis's effectiveness but does not increase solving time, as there is nothing to analyze. With our framework, we allow for this needed flexibility without additional overhead through the easy switching of techniques, allowing us to apply them when they are effective.

While exceptions show a direct fault in the candidate program, it becomes more difficult to infer the culprit directly if it runs but still returns the wrong result. However, we can still use this conflict to prune future programs based on the output. Again, this is problem-dependent, but this dependency also gives us the power to prune more if we have more information on the problem. Let us take another domain as an example, the robot domain. This domain defines the problem of a robot needing to solve simple tasks like solving a maze and picking up items. Its grammar is defined by simple instructions, such as *moveUp*, *moveDown*, *moveLeft*, *moveRight*, and *pickUp*. The IO examples are starting states and end states for the robot. We will likely not experience any exceptions to this grammar, so we will examine a program's outcomes. One example of things we can use to prune other programs is the idea of redundant values. In this case, a program that returns you to the starting state without any pickups is useless and a waste of our evaluation time. The simplest example of such a program is: moveRight, moveLeft. If we prune all programs containing such moves, we prune many useless programs that can be more efficient by removing this part of the program.

This is a simple example where you can imagine adding these cases upfront as constraints to avoid them already. However, this is just a simple example; as a domain expert, you might miss many more complex variations. Conflict analysis does this automatically for you. Additionally, you can define these redundant values more abstractly for a specific problem setting with our framework. This gives

the domain expert the power to prune candidate programs containing useless (parts of) programs automatically, without the need to craft all constraints by hand.

Grammar Constraints

In the previous section on semantic-based conflicts, we explained why adhering to the candidate program's structure is essential for the techniques' correctness. With execution-based conflicts, we look at the actual outcomes of (a part of) a candidate program instead of its abstract behavior. Whereas a difference in structure invalidates a constraint from semantic-based conflict analysis, this is not the case for a constraint from execution-based conflict analysis, as long as the conflict is part of the candidate solution. This distinction makes **the way** we have to treat the constraints different when it comes to pruning future programs. Structurally, the two types of conflicts could still return similar-looking constraints; the difference lies mainly in how the solver treats the constraints. To clarify this distinction in general and for the solvers, we identify these constraints as `grammar constraints`, as they can apply to any program that can be made from a grammar.

5.2 Enabling Custom Semantics

During this thesis, we concluded multiple times that semantics are a powerful way for extracting conflicts when analyzing faulty solutions. This meant we can explain why a solution is faulty on a higher level of abstraction. Using this knowledge, we can prune a significant portion of the solution space where other faulty solutions have the exact reason for failure. However, these semantics must be explicitly defined in the grammar by a domain expert and given to the solver.

Semantics can be a powerful tool for describing the behavior and properties of grammar rules and, therefore, your program. Depending on the context, this can range from relations between the rule's in- and output(s) to pre- and post-requisites that need to hold for the rule itself.

To accomplish this, it is essential to compare the semantics effectively and efficiently. That is why we use arbitrary variables instead of concrete names from the rule to denote input and output variables. If the rule's structure and input and output types matter, this can be considered when implementing conflict analysis techniques in the solver. This way, we maintain flexibility and the possibility to reason between rules of different structures and types when necessary. Suitable semantics are key to pruning large amounts of programs. This implementation gives domain experts great power to enhance a grammar or DSL and improve the results while keeping it understandable.

5.3 Conflict Analysis Framework

This thesis's main contribution is a framework designed to compare and combine conflict analysis techniques in a shared environment. The framework gives a skeleton of functions to implement, which is inspired by the four-step algorithm of Neo [Feng et al., 2017]. Neo uses the following steps in its algorithm: *Decide*, *Propagate*, *CheckConflict*, and *AnalyzeConflict*. The methods for these steps are described explicitly in the paper, but with the caveat that per-step changes can be made to alter the search process. We take this idea a step further by splitting the algorithm and generalizing the conflict part of it. The *Decide* and *Propagate* steps have their own task, but still overlap in the idea of generating a (partial) solution using a top-down iterator and a probabilistic model to generate these solutions. In our framework, we want to omit any specific implementation for generating (partial) solutions. The framework starts by receiving a solution, checking whether a conflict can be extracted, and then analyzing the conflict to create constraints. However, we need to use some standardized data structure to allow for this generalization of techniques to work. As our goal is to generate valid programs, we opted for the Abstract Syntax Tree(AST) to represent the solutions the solver gave. An AST is a widely adopted way of effectively and structurally representing (partial) programs, allowing various solvers and conflict analysis techniques to fit into this framework.

While this presents us with great flexibility, something else might become somewhat more challenging to adapt to different solvers. When analyzing conflicts to learn a more general reason why a program has failed, we need to tell the solver how to use this information in future iterations. In Constraint Programming(CP), constraints define solutions' boundaries and prune them when violated. These have been proven effective ways of communicating to a solver how to prune more faulty solutions. The compromise we must make when using constraints is that they can also be solver or search technique-specific. As we decide to separate the solver from the framework, we must consider the possibility that either different types of constraints have to be used within the framework or a conversion has to be fit between the framework and the solver to use the framework effectively.

5.3.1 Framework Overview

With this framework, we want to make conflict analysis more accessible, understandable, and easy to work with. That is why we split the framework into clear parts, each with its own purpose. Figure 5.2 shows a visual overview of the components and data flows of the framework. The process starts outside the actual framework, at the solver. Different types of solvers differ in their form of solution and the types of constraints they can manage. When a solver is chosen, the techniques that could be applied can be selected and implemented. The solver always outputs a candidate solution that does not solve the problem, the so-called *conflict*. In addition, the solver incorporates additional information about the do-

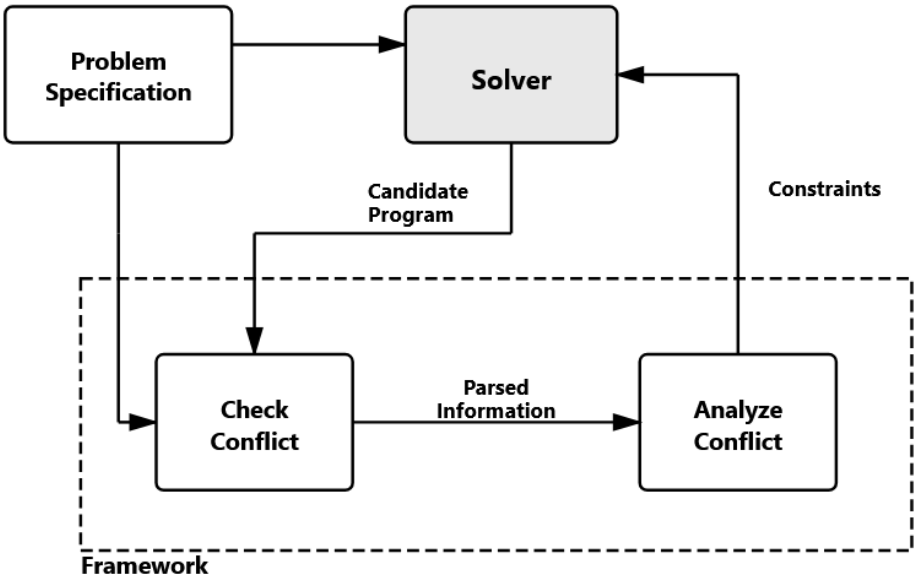


Figure 5.2: Framework Overview. This figure shows the main data flows when working with the framework. The framework is designed to be extended with additional input information or workflows, like new techniques requiring more input data.

main or solution (e.g., grammar, evaluation results, semantics, etc.), which is sent to the framework. The framework consists of two distinct steps: *CheckConflict* and *AnalyzeConflict*. This distinction is inspired by [Balog et al., 2017], where we first check whether the failing candidate is a conflict in the domain of every implemented technique. If this is the case, additional data manipulation on the given input may occur, which is then forwarded to the next step. Here, we analyze the actual conflict and generate constraints for every technique based on the learned information.

The framework's modularity allows us to do more than run all techniques in sequence. Firstly, data of learned constraints can be shared between techniques between every iteration, allowing future iterations to create more powerful and complex constraints. Secondly, data sharing is possible between the *CheckConflict* and *AnalyzeConflict* steps, again allowing for more advanced constraints. The main improvement of this data sharing is the idea of 'caching' previously generated constraints, which can be used to create new constraints more efficiently. Thirdly, suppose you decide to keep the conflict analysis technique separate between iterations. In that case, they can be run in parallel as the input data or solver is not directly impacted, only indirectly through constraints. This can reduce the framework's overhead and further improve its effectiveness.

Check Conflict

When the solver has iterated the next candidate solution and evaluated it on the specification, resulting in a failing program, the conflict analysis starts at the *CheckConflict* phase. Based on the information given by the solver, we check for every conflict analysis technique if there is an actual conflict we can analyze. There are many reasons why a candidate program fails, but not all reasons apply to every candidate program. That is why we first check for the conflict. When we detect a conflict, the function will prepare the data for the *AnalyzeConflict* phase to extract the constraints from the actual conflict.

Analyze Conflict

When the *CheckConflict* phase has detected a conflict and parsed the data, it arrives at the *AnalyzeConflict* phase. Here, we analyze the given data and extract constraints to learn as much as possible from this conflict. The more general we can make the constraints, the more programs we can prune during solving. Every technique has its unique way of analyzing these conflicts and creating unique constraints. However, there can also be a noticeable overlap between techniques. This can sometimes only be observed when the techniques are implemented next to each other.

5.3.2 Pipeline of the Framework

We need a robust and easy-to-use structure to generalize conflict analysis techniques intuitively and make them work smoothly in a single environment. In the previous section, we showed what this structure looks like for the flow of the whole synthesis process and the adaptation of individual techniques inside our framework. However, these techniques require various inputs for their analysis, which can differ per problem. It would be cumbersome to call each technique individually from the framework. With the framework, we added a layer that serves as a pipeline taking all the necessary inputs as a context, distributing the inputs over the enabled techniques, running all the techniques, and returning two lists of constraints separated by the type of technique that created the constraints. This pipeline is run every time the solver iterates a candidate program that fails to solve the problem description. We will now go into more detail by looking at the pseudocode of this pipeline (See Algorithm 1).

We start with an important distinction: *structural constraints* and *grammar constraints*. This distinction is necessary to tell the solver what programs it can prune with these constraints. In section 5.1, we explained the difference between *semantic-based* and *execution-based* conflicts and the pruning capacity of their constraints. In the pipeline, we create separate lists for the two types of constraints to return to the solver at the end. Then, we loop through all the enabled techniques and run the following steps: collect the necessary input from the context, run *CheckConflict*, if

Algorithm 1 Distributing pipeline of the framework

$struct\_constraints \leftarrow []$
$grammar\_constraints \leftarrow []$
for $tech \in input.techniques$ **do**
    AUTOINPUT$(tech, input.context)$
    $tech.data \leftarrow$ CHECKCONFLICT$(tech)$
    if $tech.data =$ NULL **then**
        continue
    end if
    $analyze\_result \leftarrow$ ANALYZECONFLICT$(tech)$
    if $analyze\_result =$ NULL **then**
        continue
    end if
    if $analyze\_result \in$ ABSTRACTCONFLICTCONSTRAINT **then**
        if $analyze\_result.add\_to\_grammar$ **then**
            $grammar\_constraints$.APPEND$(analyze_result)$
        else
            $constraints$.APPEND$(analyze_result)$
        end if
    else if $analyze\_result \in$ ABSTRACTVECTOR **then**
        for $c \in analyze\_result$ **do**
            if $c.add\_to\_grammar$ **then**
                $grammar\_constraints$.APPEND(c)
            else
                $constraints$.APPEND(c)
            end if
        end for
    end if
end for
return $(constraints, grammar\_constraints)$

applicable, run *AnalyzeConflict*, and append created constraints to the correct list depending on the technique.

At the solver level, we bundle all the necessary information defined by the *context*. This context is defined and updated when initially adapting the conflict analysis technique to the framework. At runtime, each iteration, the context receives all the updated information from the solver, the problem description, and cached data from earlier iterations or runs. In the pipeline, the *autoinput* method distributes the correct information given the technique. Then, the *CheckConflict* method is run for that technique, followed by the *AnalyseConflict* method if there is actually data to be analyzed. If *AnalyzeConflict* returns constraints, the *add_to_grammar* tag is checked to determine the correct type of constraint list to add the constraints to. These steps are run for every available and enabled technique, whereafter the two types of constraint lists are returned to the solver to be used in the search process to prune the search space.

5.3.3 MUC-Based Conflict Analysis

As a first implementation, we provide a technique based on finding a *Minimal Unsatisfiable Core* (MUC) using an SMT solver. This is an adapted version of the conflict analysis technique described in [Feng et al., 2017]. This technique analyzes semantic-based conflicts and prunes programs based on the semantic equality of programs. We can use the semantics to group grammar rules by their semantic behavior. These groups of equal grammar rules under a particular semantic behavior are called Equivalent Modulo Conflicts (EMC's). When we can determine which semantics are failing a candidate program, and specifically for which nodes in the program, we can use these EMC's to define a set of candidate solutions that would fail for the same reason.

Defining these in a constraint and returning it to the solver will prune them from the list of programs the solver can iterate over.

Algorithm 2 *CheckConflict* for MUC technique

```
1: input ← technique.input
2: smt_model ← INFERSPEC(input)
3: core ← SMTSOLVE(smt_model, smt_solver)
4: if core = NULL then
5:     return
6: end if
7: constraint_map ← CORETOCONSTRAINTS(core)
8: semantic_map ← CONSTRAINTSTOSEMANTICS(constraint_map, input)
9: return semantic_map
```

Check Conflict (Algorithm 2) We start by receiving the input from the solver: the candidate program, the grammar, and a counterexample. The candidate pro-

gram is an AST, with each node consisting of a single grammar rule, and the root node determining the output type. Then, we create an SMT model by substituting grammar rules at every node with their semantics. This represents the program in its semantic form, which we use as an SMT model. We also define the inputs and outputs as constants in the SMT model, which completes it for checking. With the addition of custom semantics from section 5.2, we do not have to rely on the SMT solver's supported functions for the semantics. We run the inputs and outputs on the semantics of the nodes corresponding to the ones in the program before adding them to the SMT model. The semantics are encoded in unique variables to ensure a consistent model.

After this, we run the SMT solver first to check whether the semantic representation still fails on the provided problem. If the SMT solver returns *SAT*, we learn that the program might be close to a solution, as the outputs seem consistent on a *semantic* level with the inputs. When this happens, we return $nothing$ and cut the technique sort, avoiding unnecessary computing.

If the solver returns *UNSAT* on the given model, we know we have a semantic conflict that we can leverage. However, we do not know yet which semantics are failing the program and which nodes are responsible for the conflict. For this, we use the idea of the MUC, which is part of the SMT solver implementation. The MUC defines the minimum set of constraints that still makes the model unsatisfiable. Removing one of these constraints from the model will make the model satisfiable again. Because of our semantic representations in the SMT model, we can call *get-unsat-core* and get this set of specific semantics that is the reason for our conflict. Before moving on to the analysis step, we decode the SMT representation back to our node and semantics representation.

Analyze Conflict (Algorithm 3) In this step, we use the following information to analyze the conflict: the mapping nodes in the program to their conflicting semantics for this instance, the candidate program, and the grammar. The mapping only consists of the nodes corresponding to the semantics defined in the MUC from the previous step. Depending on the conflict, this can be one or multiple nodes and one or multiple semantics per node. Each extra node or semantic improves the pruning power further, as the number of programs that can be pruned is defined by all combinations of the EMC classes. The more these classes there are and the greater their size, the more combinations there are, and the more programs are to be pruned.

One additional optimization step we can apply concerns the grouping of terminal rules used by functions in mapping the nodes. Some functions might use parameters as constants to alter how the function works on a syntactic level. For example, take the function $counEq(arr, int)$ where arr is any array and int is the integer the function counts in the input array. If this parameter is a terminal rule in the grammar, we can group all terminal rules that could replace this parameter and prune all additional programs containing those values. We can do this due to

Algorithm 3 *AnalyzeConflict* for MUC technique

```
1: input ← technique.input
2: arity ← MAXARITY(input.grammar)
3: muc_map ← technique.data.semantic_map
4:
5: if root_id ∈ muc_map then
6:     rules ← EXTRACTEMCCLASS(muc_map[root_id], input.grammar)
7:     if |rules| > 1 then
8:         cons_root ← DOMAINRULENODE(rules)
9:     else
10:        cons_root ← RULENODE(input.root.rule)
11:    end if
12: else
13:    cons_root ← RULENODE(input.root.rule)
14: end if
15:
16: original_q ← QUEUE()
17: rebuilt_q ← QUEUE()
18: ENQUEUE(original_q, input.root)
19: ENQUEUE(rebuilt_q, cons_root)
20: while original_q ≠ ∅ do
21:    curr_orig ← DEQUEUE(original_q)
22:    curr_new ← DEQUEUE(rebuilt_q)
23:    for (i, child_orig) ∈ ENUMERATE(GETCHILDREN(curr_orig)) do
24:        child_id ← GETINDEX(child_orig)
25:        if child_id ∈ muc_map then
26:            rules ← EXTRACTEMCCLASS(muc_map[child_id], input.grammar)
27:            if |rules| > 1 then
28:                child_new ← DOMAINRULENODE(rules)
29:            else
30:                child_new ← RULENODE(child_orig.rule)
31:            end if
32:        else if ISTERMINAL(child_orig.rule)∧
33:            curr_new isaDOMAINRULENODE then
34:            child_new ← DOMAINRULENODE(GETTYPEDTERMINALSLIST(child_orig.rule))
35:        else
36:            child_new ← RULENODE(child_orig.rule)
37:        end if
38:        curr_new.children[i] ← child_new
39:        ENQUEUE(original_q, child_orig)
40:        ENQUEUE(rebuilt_q, child_new)
41:    end for
42: end while
43: return FORBIDDEN(cons_root)
```

the nature of semantics, where the value of the inputs does not change the function's semantic workings. This additionally improves the pruning effectiveness per analyzed conflict when applicable.

When building the constraint for this technique, we take the original candidate program as the base and define a domain of grammar rules equal to its EMC class for every node in the mapping. Then, we wrap this constraint representation in a *Forbidden* constraint to return to the solver.

5.3.4 Semantic Analysis

The second technique we have implemented also uses semantics to define and analyze its conflict. The difference lies in the technique's information, where more conflicts can be inferred at the program's root. The idea comes from the paper [Wang et al., 2017], where they check the root for any conflict regarding the given semantics. If they cannot find a conflict, the candidate is passed, and they move on to the next candidate. Using this idea, we came up with our own technique, which combines this idea with the concept of the Equivalent Modulo Conflict classes (EMC's) from the MUC-based technique of section 5.3.3.

Algorithm 4 *CheckConflict* for Semantic Analysis technique

```
1:  input ← technique.input
2:  if grammar.semantics[GetRule(input.root)] = ∅ then
3:      return
4:  end if
5:
6:  eval_dict ← {}
7:  eval_dict[y] ← input.counter_example.out
8:  for (i, child) ∈ Enumerate(GetChildren(input.root)) do
9:      expr ← RuleNodeToExpr(child, input.grammar)
10:     eval_dict[x_i] ← ExecuteOnInput(symboltable, expr, input.counter_example.in)
11: end for
12:
13: semantics ← []
14: for semantic ∈ grammar.semantics[GetRule(input.root)] do
15:     if ¬EvaluateExprNaive(semantic, eval_dict) then
16:         semantics.Append(semantic)
17:     end if
18: end for
19:
20: if isEmpty(semantics) then
21:     return
22: end if
23: return semantics
```

Check Conflict (Algorithm 4) We receive the following information to check this conflict: the candidate program, the grammar, and a counterexample. First, we check whether our root function has any defined semantics in the grammar. If not, we can stop the checking early, as we will not find a semantic-based fault in the conflict without semantics. If the program's root function has defined semantics, we take the root node's children as subprogram(s) and evaluate their intermediate outputs. We do this simply by running them as separate programs and filling in the inputs where applicable. Then, we use these intermediate results as inputs to the semantics of the root function, together with the desired output of the problem. For every semantic, we evaluate whether it holds with the provided inputs and outputs and keep track of the semantics that do not hold. These semantics then describe our semantic-based conflict and represent the reason(s) why our candidate program is failing the problem specification.

Algorithm 5 *AnalyzeConflict* for Semantic Analysis technique

```
1:  input ← technique.input
2:  semantics ← technique.data.semantics
3:
4:  constraints ← []
5:  for semantic ∈ technique.data.semantics do
6:      emc_class ← EXTRACTEMC(semantic, input.grammar)
7:      if |sec_class| < 2 then
8:          continue
9:      end if
10:     tree ← DOMAINRULENODE(emc_class)
11:     for (i, child) ∈ ENUMERATE(GETCHILDREN(tree)) do
12:         if ISTERMINAL(child.rule) then
13:             child_new ← DOMAINRULENODE(GETTYPEDTERMINALSLIST(child.rule))
14:         else
15:             child_new ← child
16:         end if
17:         tree.children[i] ← child_new
18:     end for
19:     constraints.APPEND(FORBIDDEN(tree))
20: end for
21: return constraints
```

Analyze Conflict (Algorithm 5) To analyze the conflict, we continue with the following information: the list of failing semantics, the candidate program, and the grammar. Then, we extract the EMC class, as from the MUC technique defined in section 5.3.3, using the provided list of semantics to determine which other grammar rules will still violate the problem if swapped with the current root function.

We take the original program and interchange the root node with a list of grammar rules equal to the EMC class. We can also apply the same optimization step as 5.3.3, where we can group the terminal rules for further pruning, if the root function uses constants as input parameters. Finally, we wrap this constraint representation using the domains to represent a set of programs in a *Forbidden* constraint. We do this for every semantic that failed to hold in *CheckConflict* and return a list of constraints to be used by the solver.

5.3.5 Error and Redundant Values Analysis

This technique is based on the *execution-based* conflicts described in section 5.1.2 The core idea is exploiting the nature of faulty and redundant programs. The technique is simple but versatile and practical, depending on the domain and the provided information. As redundant values are subjective, this technique becomes more powerful when given more domain-specific information that can be used to analyze more conflicts that apply to this technique. We do recognize some general redundant values that are problem-agnostic.

In a grammar, there are terminal rules that are defined like constants. This means they cannot be expanded and, more importantly, do not change during the search. If a candidate solution's return value is equal to any of the terminal rules defined in the grammar, we can replace the whole program by having the terminal as the program. Depending on how your grammar is defined, this can prune valid solutions.

Algorithm 6 *CheckConflict* for ERA technique

```
1: input ← technique.input
2: if |input.root| > technique.max_constraint_size then
3:     return
4: end if
5:
6: if input.evaluation = NULL then
7:     return input.root
8: else if |program| > 1 ∧ input.evaluation ∈
9:         GETTERMINALS(input.grammar) then
10:    return input.root
11: else if input.evaluation ∈ ARRAY ∧ input.evaluation = [] then
12:    return input.root
13: else
14:    return
15: end if
16: return
```

Check Conflict (Algorithm 6) Again, the information we use with this technique depends on the problem, what information is available, and what is considered

redundant. For now, we use the candidate program, the evaluation of the counter-example, and a maximum constraint size value for the technique's base implementation.

Depending on your budget, you might want to include the maximum constraint size and ignore programs larger than this value. The constraints generated by this technique are carried throughout the search process; they are not bound to a program structure, as with the semantic-based techniques. This means that the more constraints you add to the solver, the more time it will spend checking them. We use program size as a boundary value, as smaller programs occur in more programs, providing more pruning power.

The first thing we can always check is whether the evaluation result is an exception. If this is the case, we have a program that we know will throw the same error if incorporated in a bigger program, so we can forward it to be used for pruning candidate programs. If the evaluation is an actual output value, we can check it against the redundant values. As mentioned, these are problem-specific, but examples could be the starting state for the robot domain, empty strings for the string domain, etc. Suppose the evaluation result is equal to any of these given values. In that case, we can treat the candidate program as equal to the program that throws an exception, namely, as a program we do not want to use in any other larger program. The current redundant values are evaluations equal to terminals in the grammar, and for arrays specifically, when they are empty. For this technique, we do not have to parse the inputs any further, and we return the candidate program to be analyzed and made into a constraint.

Algorithm 7 *AnalyzeConflict* for ERA technique

1: $root \leftarrow technique.data.root$
2:
3: **return** $Forbidden(root)$

Analyze Conflict (Algorithm 7) In the initial version of this technique, this step takes the candidate program and wraps it in a *Forbidden* constraint to return to the solver.

As the constraints returned by this technique prune larger programs containing these conflict programs as subprograms, ensure that any 'start' symbol is not included in the constraint. Some grammars use a 'start' symbol in the grammar, which is always the root node where the program is expanded from. If this symbol is included in the constraints, the constraints will not prune anything, as the 'start' symbol is never contained inside a larger program, only again as the root.

This step only wraps the candidate program under a constraint to be returned to the solver. One option to enhance this analysis step is to find the root cause of a given conflict and find the smallest subprogram that still results in the same conflict. This technique's efficiency will improve, as the constraint can be a subprogram of

more programs than its original candidate program.

Chapter 6

Experimental Evaluation

In this chapter, we evaluate the effectiveness of our framework by answering the following research questions:

RQ1. Does combining conflict analysis techniques reduce the program space?

RQ2. Does the conflict analysis framework improve solvers on real-world problems?

6.1 Experimental Setup

For the creation of the results, we implemented the framework in Julia using the program synthesis framework Herb.jl [Hinnerichs et al., 2025a]. All experiments have been run on an Intel Core i7-10700F CPU @ 2.90GHz with 16GB of RAM.

Tasks: We evaluate our framework on two diverse program synthesis domains:

- **List Domain**: The List benchmark [Balog et al., 2017] is made for synthesizing functional programs using list operations and a core benchmark for the evaluation of *Neo* [Feng et al., 2017]. We use the same semantics for conflicts as Neo for a fair comparison. The solutions have a size of at least 5 to ensure a baseline difficulty. There are 5 I/O examples per problem and 100 problems to solve.

- **String Domain**: This is a difficult string-manipulation (SLIA) benchmark from the SyGuS challenge 2019 [Padhi et al., 2019]. As it is not available by default, we added our own (abstract) semantics to this grammar to be able to run all conflict analysis techniques on the benchmark.

For both domains, a concrete example of the grammar, together with the semantics can be found in Appendix A

Techniques: Our framework implements the following three conflict analysis techniques:

(1) **MUC-based Analysis (MUC).** Converts the candidate solution into a semantic representation in SMT and the specification. This is run using an off-the-shelf SMT-solver[1], CVC5, which returns an MUC of faulty nodes when available. An *Equivalent Modulo Conflict* (EMC) class is created per faulty node and encapsulated into a single constraint. This, in turn, will prune equal structural solutions resembling the same semantic failure.

(2) **Semantic-Analysis (SeAn).** Runs the input specification on the candidate solution up to the root function. This intermediate result is checked with the output specification on the semantics of the root function. For every failing semantic, a constraint is generated, pruning the programs of the same structure where the root is swapped by any of the functions containing the same semantic.

(3) **Error-Redundant Value Analysis (ERA).** Checks whether running the candidate solution results in an error or returns a domain-dependent redundant result. If this is the case, it creates a single constraint that prunes all candidate solutions containing this solution as a sub-program.

6.2 RQ1: Combining Conflict Analysis Techniques

To answer our first research question, we demonstrate the framework's effectiveness and modularity by running individual techniques and various combinations on two domains.

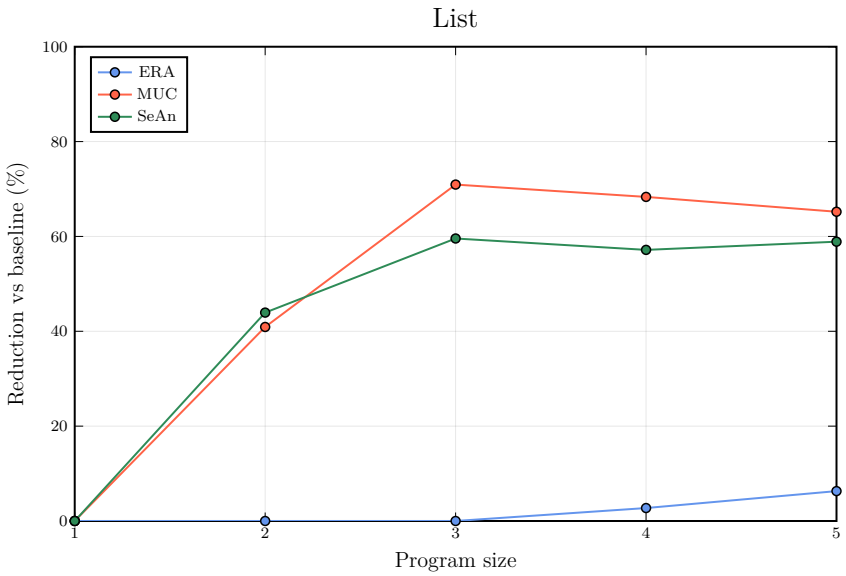
Experiment 1: Individual Techniques on Two Domains

This experiment evaluates each conflict analysis technique's effectiveness individually, demonstrating that our framework successfully reduces the search space across different domains.

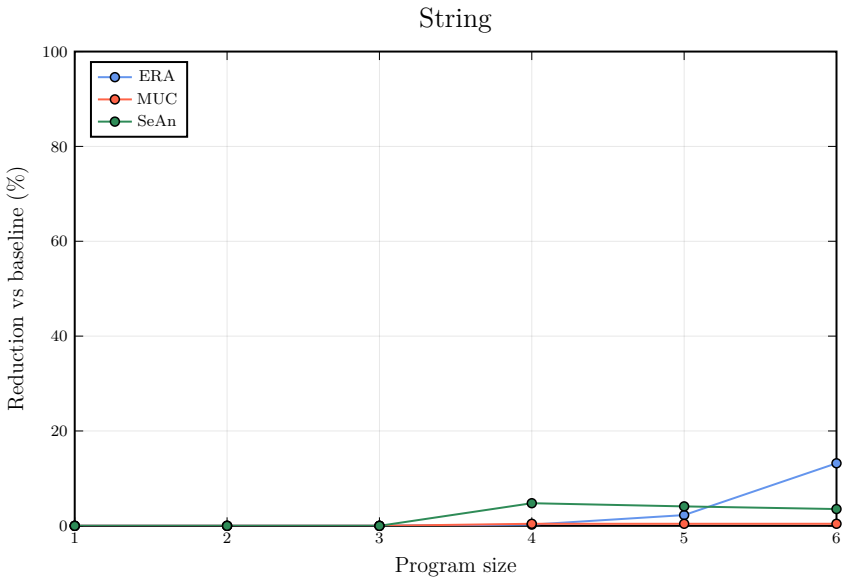
We run all three implemented techniques (MUC, SeAn, and ERA) independently on the list and string domains. Using a breadth-first search as the search algorithm, we compare against plain enumeration as the baseline.

We measure the reduction of program space across program sizes: $n \in [1, 5]$ for the list domain and $n \in [1, 6]$ for the string domain. The reduction is calculated as the difference between the number of enumerated programs of the baseline and the technique divided by the number of enumerated programs of the baseline times 100We report the average reduction of the program space with each analysis technique for each program size. The results appear in Figure 6.1a (list domain) and Figure 6.1b (string domain), with average reduction percentages summarized in Table 6.1.

[1] https://github.com/elsoroka/Satisfiability.jl



(a) List Domain



(b) String Domain

Figure 6.1: The average reduction per program size for the list (a) and string (b) domains in comparison to plain enumeration for each technique individually. The effectiveness differs heavily between domains, where semantic-based techniques (MUC, SeAn) perform well on the list domain and the execution-based technique (ERA) performs better on the string domain.

Technique	String			List		
	Avg	Std	Max	Avg	Std	Max
`MUC`	0.2	1.4	11.8	**49.1**	**35.9**	**94.9**
`SeAn`	2.1	**9.9**	**87.2**	43.9	32.8	90.5
`ERA`	**2.6**	6.4	45.3	1.8	4.2	14.4

Table 6.1: Reduction statistics for single techniques (average, standard deviation, maximum, in percentages).

The two domains reveal contrasting patterns in technique effectiveness. For the list domain, the semantic-based techniques (MUC and SeAn) almost instantly reduce search space substantially, with MUC averaging 49.1% reduction and SeAn reaching 43.9%. The ERA technique shows a limited impact, with a 1.8% average reduction and a maximum reduction of 14.4%. The string domain exhibits a different pattern. Overall, the space reduction is significantly lower in its averages, showing the framework's weaker applicability in this domain and for these program sizes. While the MUC is clearly ineffective for the string domain, the other semantic-based technique, SeAn, seems to perform well, though from the high contrast between the average reduction and the maximum reduction, on specific problems.

An interesting observation from both domains is the steady increment in the effectiveness of the ERA technique with increasing program size. Though the average reduction is low due to the nature of the ERA technique, which prunes larger programs using smaller ones, we still reach a maximum of 45.3% reduction on a relatively small program size. Especially in the string domain, there is a significant jump in effectiveness between program sizes 5 and 6, potentially increasing its effectiveness when run on bigger problems.

These domain-specific results reflect fundamental differences in grammar structure. The list grammar's high overlap between structures of rules and semantics makes it ideal for semantic-based techniques. This alignment enables efficient pruning within the same program structure, as many 'rewrites' failing on the same semantic can be eliminated at multiple points in the program. Conversely, the string grammar exhibits little overlap between structure and semantics, limiting the average effectiveness of semantic-based techniques, although they show potential in specific cases. Instead, this domain contains more errors and redundant values, playing to ERA's strengths. ERA's delayed impact, using smaller programs to prune bigger programs, explains why its effectiveness increases with program size. Increasing the program size will increase the effectiveness further; however, it will also increase the enumeration time.

Experiment 2: Different Configurations of the Framework

Building on the individual technique results, this experiment investigates whether combining techniques produces compounding effects. Our framework's modularity makes testing different combinations straightforward.

We test four combinations: all three techniques and three pairwise combinations (ERA+MUC, MUC+SeAn, and SeAn+ERA). We maintain breadth-first search as the search algorithm for our baseline.

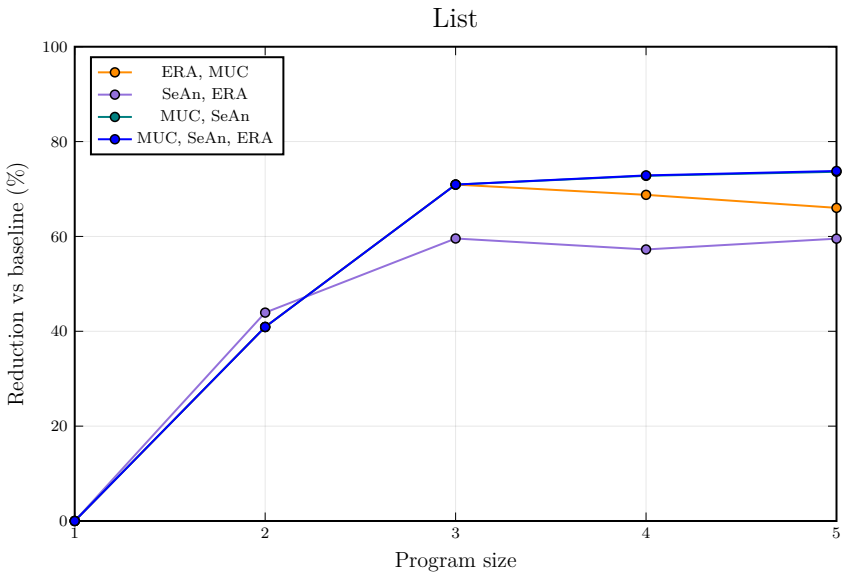
Following Experiment 1's methodology, we measure program space reduction across program sizes $n \in [1, 5]$ and $n \in [1, 6]$ for the list and string domains, respectively. The reduction is calculated as the difference between the number of enumerated programs of the baseline and the technique divided by the number of enumerated programs of the baseline times 100Results are shown in Figure 6.2a (list domain) and Figure 6.2b (string domain), with detailed statistics in Table 6.2.

Technique	String			List		
	Avg	Std	Max	Avg	Std	Max
`MUC, SeAn`	2.1	9.9	87.2	**51.7**	37.2	95.5
`ERA, MUC`	2.8	6.5	45.3	49.3	36.0	94.9
`SeAn, ERA`	**4.7**	**11.3**	**87.2**	44.1	32.9	90.5
`MUC, SeAn, ERA`	**4.7**	**11.3**	**87.2**	**51.7**	**37.3**	**95.6**

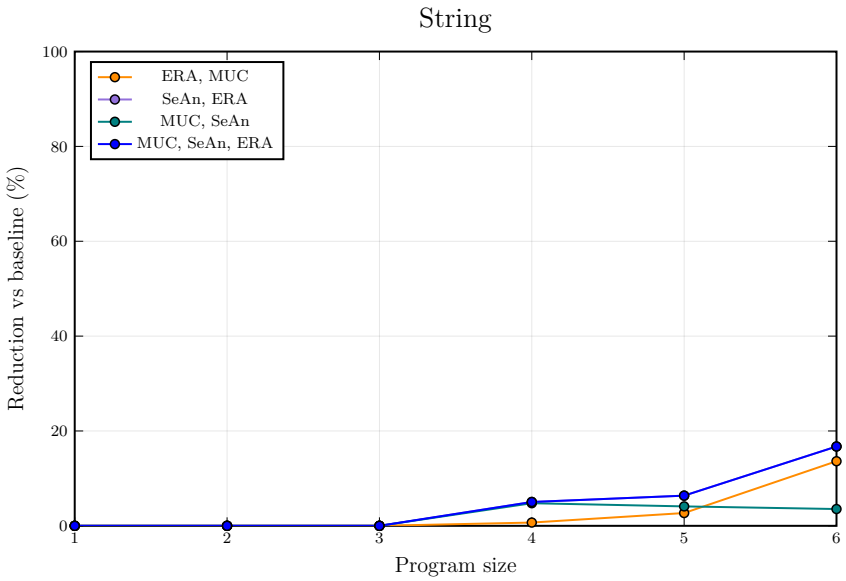
Table 6.2: Reduction statistics for technique combinations (average, standard deviation, maximum, in percentages).

Combining techniques yields further search space reduction, though effectiveness varies by domain and combination. The semantic-based techniques (MUC, SeAn) complement each other well on the list domain. While individually promising, they prune slightly more when combined, achieving an average reduction of 51.7%. The effectiveness increases more with larger program sizes, with the semantic-based technique pruning almost 10% more than running individually. The full combination (MUC, SeAn, ERA) maintains this 51.7% reduction, showing that adding ERA provides minimal additional benefit. Other pairings with ERA similarly show little improvement over the semantic techniques alone. The combinations also show modest improvements on the string domain. The SeAn+ERA pairing achieves an average reduction of 4.7%, equal to the combination of their individual results. A promising result, though still maintaining the relatively low reduction on the domain compared to the list domain.

The similarities between MUC and SeAn on the list domain reveal an essential insight: closely related conflict analysis techniques can target different parts of the search space despite using similar information. This complementary pruning explains why their combination exceeds either technique's individual performance. The limited benefit of adding ERA to semantic techniques on the list domain aligns



(a) List Domain



(b) String Domain

Figure 6.2: The average reduction per program size for the list (a) and string (b) domains in comparison to plain enumeration for the different combinations of techniques. We observe an increase in effectiveness compared to the individual techniques, but the semantic-based techniques seem to overlap in the search space they are pruning, as the increase is up to 10%.

with ERA's weak individual performance on this domain.

Conversely, the combination of SeAn+ERA on the string domain shows more relative improvements when the combined techniques are used together. This demonstrates that techniques based on different conflicts prune other parts of the search space and complement each other very well.

Key Insights

1. **Domain-technique matching is essential**. Applying all available techniques without any problem-specific considerations provides no guarantee of optimal results. Understanding domain characteristics, such as syntactic and semantic behavior concerning the implemented techniques, enables effective selection of correctly applicable techniques.

2. **Framework as comparison tool**. The framework can also function as a tool to compare conflict analysis techniques regarding effectiveness across domains and other conflict analysis techniques. With the generalizing nature of the framework, new conflict analysis techniques could easily be adapted into the framework. As the domains are already implemented, we can directly investigate and compare the newly adapted techniques by running the same experiments without any significant changes to the framework. The same can be done for new domains; these have to be defined once, and the impact of the conflict analysis can be directly investigated by running the experiments on the new domain.

3. **Combination effects are not strictly additive**. Related techniques have a higher chance of pruning overlapping search space regions (as semantic techniques do on lists). In contrast, less related techniques may still provide complementary pruning (as with SeAn+ERA on strings). Evaluation of different combinations is necessary to identify effective ones.

These results positively answer our first research question: individual conflict analysis techniques *significantly* reduce search space across multiple domains, and strategic combinations can further enhance this reduction. The key insight is that effectiveness depends on matching technique characteristics to domain properties; semantic techniques excel where structure aligns with semantics, while syntactic techniques like ERA prove more robust across diverse domains. Combining all available techniques does not guarantee optimal results and adds unnecessary overhead.

6.3 RQ2: Improving Solvers on Real-World Benchmarks

To answer the second research question, we run our framework on the string manipulation tasks of the SyGuS benchmark. We show that the framework can be applied to enumeration solvers and improves the results on the benchmark. The previous

section shows the internal versatility and flexibility of the framework concerning multiple conflict analysis techniques. This section shows the framework's external versatility and applicability by adapting it to multiple solvers.

Experiment 3: Running the Framework on SyGuS

This experiment evaluates the framework's effectiveness on real-world program synthesis problems from the SyGuS benchmark. We investigate whether conflict analysis reduces the number of enumerated programs needed to find solutions and whether this enables solving previously unsolvable problems within resource constraints.

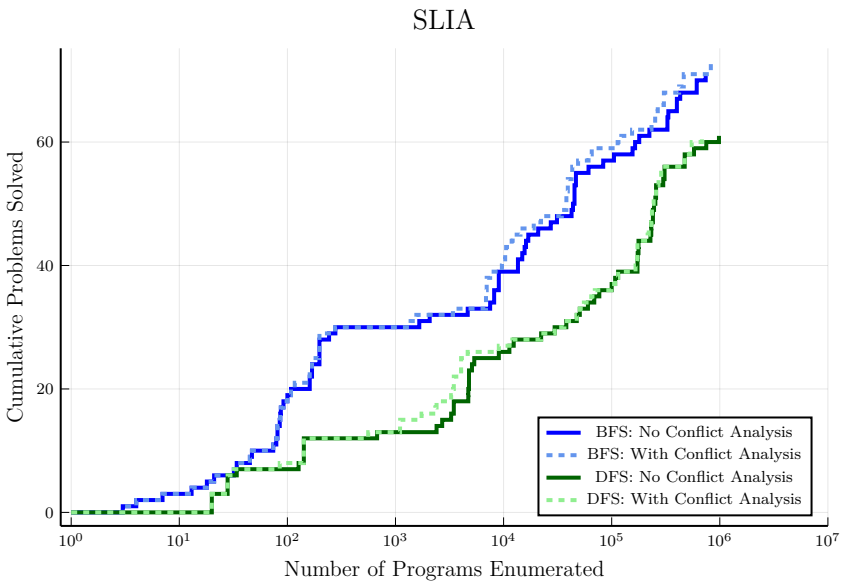
We test the framework on the SyGuS SLIA benchmark, which includes problems from sources such as Stack Overflow. We compare two search strategies: breadth-first search (BFS) and depth-first search (DFS), each run with and without the conflict analysis framework enabled. Based on Experiment 1's findings that semantic-based techniques prove ineffective, we configure the framework to use only the ERA technique. This represents the optimal configuration for this domain, maximizing reduction while minimizing computational cost. We impose a limit of one million enumerated programs and a 120-second timeout per problem to ensure termination.

We evaluate solver performance across three dimensions: the number of problems solved, the number of programs enumerated before finding solutions, and the runtime required. Results appear in Figure 6.3, where subfigure (a) shows problems solved versus programs enumerated, and subfigure (b) shows problems solved versus runtime.

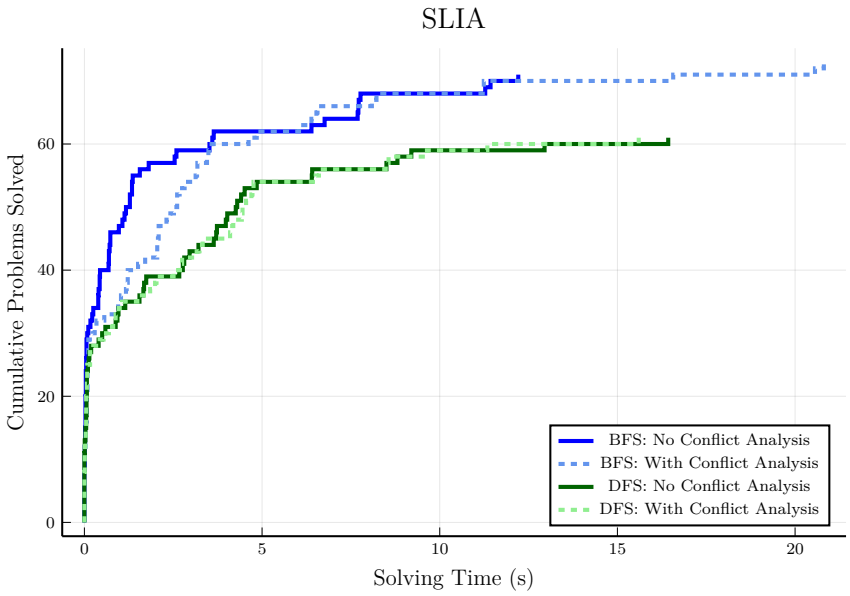
The framework demonstrates benefits in search efficiency while introducing a runtime tradeoff. Regarding search space reduction, the conflict analysis framework consistently reduces the number of programs that must be enumerated to find solutions. More significantly, this reduction solves additional problems that exceeded the one-million program enumeration limit without the framework. The BFS solver outperforms the DFS solver in terms of the number of problems solved and the solving time for these same problems.

Looking at the framework's impact on the number of enumerated programs, we can see it is more effective on the BFS solver but also solves an additional problem on the DFS solver. On the solving time, we observe a different result; where the BFS solver experiences a significant amount of overhead from the framework, the DFS solver actually performs better with the framework time-wise.

These results validate the framework's core capability to reduce search space on real-world problems; however, they also highlight an important implementation consideration. The framework successfully achieves its primary goal: pruning the search space to make previously unsolvable problems tractable within resource constraints. The ability to solve additional problems by staying under the enumeration limit demonstrates practical value, particularly for complex problems where exhaustive search would fail. However, the reduction in the string domain is not



(a) Enumerated Programs



(b) Runtime

Figure 6.3: Results of BFS and DFS iterators with and without the conflict analysis framework enabled on the SLIA benchmark. The graphs show the number of solved problems against (a) the enumerated programs and (b) the runtime.

as significant as in the list domain, as shown in Experiment 6.2. The framework's effectiveness in this setup can be improved to make it more viable for use out of

the box.

Key Insights

1. **Implementation efficiency matters**. A framework that reduces the search space may lack overall performance if conflict analysis overhead exceeds enumeration savings. The suitability of techniques must be carefully considered, especially when benchmarks consider solving time as the main scoring mechanism. However, the framework's modularity makes optimization easier; techniques can be enabled or disabled based on cost-benefit analysis for specific domains.

2. **Applicability to benchmarks**. The framework can be applied to existing benchmarks, solving more problems with fewer enumerated programs, but with increased solving time. Although the improvements are marginal when applying the framework to a benchmark test, using more effective and runtime-efficient techniques with an SOTA solver can significantly improve performance.

3. **Comparison tool for solvers** As discussed in the results of this experiment, performance on a benchmark can differ significantly based on the solver used. Comparing the performance of different solvers on benchmarks in equivalent environments can give powerful insights into their applicability and the problem's solution space. With its modular setup, the framework provides an environment where solvers and benchmarks can be evaluated in different combinations without added complexity.

In this section, we demonstrated how the framework improves existing solvers on real-world problems by consistently reducing the search space and enabling solutions to previously unsolvable problems within enumeration constraints. As the key insights above discuss, the framework's current configuration improves the solving performance marginally, depending on the solver. However, there is room for improvement regarding the effectiveness and implementation performance of conflict analysis techniques for future work. Additionally, the framework showed its potential as a comparison tool for testing different benchmarks on different solvers, together with conflict analysis.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This thesis introduces a framework for generalizing multiple conflict analysis techniques in a shared environment, running them simultaneously and independently of any specific solving technique or problem specification. We identify two types of conflicts, `Semantic-based` and `Execution-based`, from which we derive `Structural` and `Grammar` constraints, offering a variety of ways to prune programs. We demonstrate the potential of semantics in conflict analysis and how the abstraction level, semantic diversity, and structural overlap of grammar rules are essential to the effectiveness of conflict analysis when defining semantics for your grammar.

We show the adaptation of one existing [Feng et al., 2017] and two handcrafted conflict analysis techniques to the framework based on different types of conflicts. The framework prunes up to 96% of the search space on a list operations domain and up to 45% of the search space on a string transformation domain. This validates the potential of the techniques in improving the solving process by reducing the search space. Additionally, we show how combining conflict analysis techniques can increase the pruning efficiency and the ability to prune programs in different problems with the same configuration. However, the individual performance of a technique does not guarantee strictly additive performance when combining them. Matching techniques to the domain is key to the framework's good performance. Moreover, not every technique is as effective in every domain, and it only adds unwanted overhead when used.

We test the framework on top of a naive synthesis solver on a competition-level benchmark, SyGuS, trying to solve real-life examples of string transformation problems. The framework improves the enumeration in the number of problems it can solve and shows a reduction in the number of enumerated programs. However, the significance of the reduction depends on the problem and how suitable it is for the applicability of a technique. While the current state of the framework shows marginal improvements on the selected benchmark and varying overhead depend-

ing on the selected techniques, the foundation is strong, and there is much potential for improvement in future works.

Our framework is a powerful tool for making conflict analysis more accessible for researching its applicability to any problem or solver. Due to its modularity, almost any conflict analysis technique and solver can be adapted and interchanged within the framework in a structured and intuitive way. It is a powerful tool for investigating different configurations of solvers, problems, and conflict analysis techniques. This allows for a fair comparison of these configurations and a better understanding of the behavior and workings of all individual components.

7.2 Future Work

7.2.1 Implementing Existing Solvers

In this thesis, we did not test the power of the framework's independence regarding existing solvers. Chapter 6.3 already showed an improvement in the effectiveness of two plain enumeration solvers, and we believe that this improvement can be at least as impactful on state-of-the-art synthesis solvers. The framework could benefit from running on more fruitful programs, and the solvers could evaluate more diverse, interesting, and larger problems.

Our framework seamlessly integrates with synthesizers from different fields of program synthesis, e.g., divide-and-conquer [Si et al., 2019] and heuristic learning [Barke et al., 2020]. The performance of our framework depends on the enumeration order. In future work, we want to investigate how state-of-the-art solvers impact the performance of our framework. Another more challenging adaptation would be to the logic domain. Logic programs are difficult to represent using an *Abstract Syntax Tree*; therefore, it would be interesting to investigate whether we can find a conversion that fits the framework or adapt the logic domain conflict analysis techniques [Cropper and Morel, 2020, Cropper and Hocquette, 2022, Morel and Cropper, 2023] to our framework.

7.2.2 Introducing More Techniques

With the setup of our framework, we aim to make it as intuitive as possible to introduce existing conflict analysis techniques into the framework. Adapting other techniques, like we did with the ones in this thesis, can further increase the versatility and reach of the problems we might be able to solve more efficiently.

Blaze [Wang et al., 2017] In section 3.3, we covered Blaze, a conflict analysis technique based on *Finite Tree Automata* (FTA). Starting with their abstraction FTA solver, adapting their conflict analysis technique into the framework should allow for reproducing its results. With the shared environment, other techniques can be applied to improve results or make Blaze work on a different solver and various problems.

Absynthe [Guria et al., 2023] Section 3.4 covered two techniques using the same semantic information we used for our semantic-based conflicts. Due to their higher abstraction layer, semantics can be very powerful in pruning the search space. Absynthe uses abstract interpretation to compute over-approximations and prune the search space. This is another way of analyzing semantic-based conflicts, which could prune another part of the search space, adding to the framework's effectiveness. Absynthe uses language-agnostic domains, fitting well with our generalized framework.

SIMPL [So and Oh, 2017] This is the second semantic-based technique mentioned in section 3.4. When adapted to the framework, it adds another layer of value by using semantic constraints based on static analysis. SIMPL could fit nicely, expanding its applicability to other domains.

7.2.3 Library Learning

In [Ellis et al., 2021], the authors present a library learning method, DreamCoder, where they use functions learned from earlier solutions in subsequent problems to solve them more efficiently. It would be interesting to see if we can apply the idea of library learning to the framework's learned constraints. As we generate syntactic constraints, which can be seen as programs, this idea of library learning seems to fit nicely. When using the framework in this thesis, we learn many constraints, some of which could be applied to multiple problems. Using previously learned constraints, we can already start a problem with a smaller search space. In that case, it can improve the efficiency of solving similar problems over time, as we learn more general constraints we can apply to multiple issues. The challenge would be in selecting the right constraints while guaranteeing that we do not prune away solutions in other problems and that not too many constraints overload the solver from the start.

7.2.4 Individual Technique Improvements

In this thesis, we implemented three conflict analysis techniques to demonstrate the framework's potential; however, the effectiveness of these techniques can still be improved. One limitation of semantic-based techniques is that they are restricted to the program's structure; they cannot generalize to bigger programs containing the same faults. One idea could be to cache constraints that prune many problems and check if we can use them in other conflicts that fail on the same semantics and where they match as subprograms. Another idea is to create more constraints from a single conflict, extending the generated constraint with additional functions that would fail for the same reason. This can already prune larger programs without first being analyzed as a conflict.

For the *Semantic Analysis* technique described in section 5.3.4, we can also improve the pruning capacity by identifying more nodes that fail with the same se-

mantic reason as the root. If the children of the root have the same semantics, they can be included in the reason of the conflict, making the resulting constraint more powerful. We can apply this logic to the rest of the program until we find the functions without these failing semantics. We will generate constraints that, depending on the problem, will only allow the solver to evaluate a handful of programs.

7.2.5 Improving Efficiency of the Framework

Although this initial version of the framework is promising in terms of results, it still has more to offer regarding its overhead and implementation efficiency. One direct performance improvement to the pipeline could be its parallelization. The results generated in chapter 6 show the techniques run sequentially, which could be run in parallel between the solver's iterations. This is possible while maintaining a shared data environment, as the techniques run independently on the data between pipeline iterations. When combining techniques, running the pipeline in parallel can reduce the framework's overhead.

The framework also generates many constraints, depending on the number of techniques used and the problem specification. A more innovative way of handling those constraints could be to combine them into higher-order constraints and check for duplicate or redundant constraints, further reducing the overhead. The program synthesis library Herb.jl [Hinnerichs et al., 2025a], used for the implementation of the framework, already provides these higher-order constraints in their constraint model [Hinnerichs et al., 2025c]. The *MUC-based* and the *Semantic Analysis* techniques generate the same kind of constraint; thus, they could be merged into a single constraint. This further improves individual constraints' effectiveness and the cached constraints' potential described in Section 7.2.4.

Bibliography

J. Ahlgren and S. Y. Yuen. Efficient program synthesis using constraint satisfaction in inductive logic programming. *J. Mach. Learn. Res.*, 14(1):3649–3682, Dec. 2013. ISSN 1532-4435.

M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs, 2017. URL https://arxiv.org/abs/1611.01989.

S. Barke, H. Peleg, and N. Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. *Proc. ACM Program. Lang.*, 4(OOPSLA), Nov. 2020. doi: 10.1145/3428295. URL https://doi.org/10.1145/3428295.

A. Cropper and S. Dumancic. Inductive logic programming at 30: A new introduction. *J. Artif. Intell. Res.*, 74:765–850, 2022.

A. Cropper and C. Hocquette. Learning logic programs by discovering where not to search, 02 2022.

A. Cropper and C. Hocquette. Learning logic programs by finding minimal unsatisfiable subprograms, 2024. URL https://arxiv.org/abs/2401.16383.

A. Cropper and R. Morel. Learning programs by learning from failures, 2020. URL https://arxiv.org/abs/2005.02259.

K. Ellis, C. Wong, M. Nye, M. Sablé-Meyer, L. Morales, L. Hewitt, L. Cary, A. Solar-Lezama, and J. B. Tenenbaum. Dreamcoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 835–850, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454080. URL https://doi.org/10.1145/3453483.3454080.

Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program synthesis using conflict-driven learning, 2017. URL https://arxiv.org/abs/1711.08029.

M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Clingo = asp + control: Preliminary report, 2014. URL https://arxiv.org/abs/1405.3694.

S. Gulwani. Synthesis from examples: Interaction models and algorithms. In *2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 8–14, 2012. doi: 10.1109/SYNASC.2012.69.

S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017. ISSN 2325-1107. doi: 10.1561/2500000010. URL http://dx.doi.org/10.1561/2500000010.

C. A. Gunter. *Semantics of programming languages: structures and techniques*. MIT press, 1992.

S. N. Guria, J. S. Foster, and D. Van Horn. Absynthe: Abstract interpretation-guided synthesis. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi: 10.1145/3591285. URL https://doi.org/10.1145/3591285.

T. Hinnerichs, R. G. Reid, J. de Jong, B. Swinkels, P. Wochner, N. Filat, T. Magurescu, I. Hanou, and S. Dumancic. Herb.jl: A unifying program synthesis library, 2025a. URL https://arxiv.org/abs/2510.09726.

T. Hinnerichs, B. Swinkels, J. de Jong, R. G. Reid, T. Magirescu, N. Yorke-Smith, and S. Dumancic. Modelling program spaces in program synthesis with constraints, 2025b. URL https://arxiv.org/abs/2508.00005.

T. Hinnerichs, B. Swinkels, J. Jong, R. Reid, T. Magirescu, N. Yorke-Smith, and S. Dumancic. Modelling program spaces in program synthesis with constraints, 07 2025c.

J. Jaffar and M. J. Maher. Constraint logic programming: a survey. *The Journal of Logic Programming*, 19-20:503–581, 1994. ISSN 0743-1066. doi: https://doi.org/10.1016/0743-1066(94)90033-7. URL https://www.sciencedirect.com/science/article/pii/0743106694900337. Special Issue: Ten Years of Logic Programming.

S. Kadioglu and M. Sellmann. Efficient context-free grammar constraints. volume 1, pages 310–316, 01 2008.

M. Law. Conflict-driven inductive logic programming, 2022. URL https://arxiv.org/abs/2101.00058.

J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning sat solvers. *Handbook of satisfiability*, pages 131–153, 2009.

R. Morel and A. Cropper. Learning logic programs by explaining their failures, 2023. URL https://arxiv.org/abs/2102.12551.

S. Padhi, A. Udupa, A. Fu, E. Polgreen, and A. Reynolds. Benchmarks for SyGuS competition. https://github.com/SyGuS-Org/benchmarks, 2019. Accessed: 2025-10-13.

C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

F. Rossi, P. van Beek, and T. Walsh. Chapter 4 constraint programming. In F. van Harmelen, V. Lifschitz, and B. Porter, editors, *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 181–211. Elsevier, 2008. doi: https://doi.org/10.1016/S1574-6526(07)03004-0. URL https://www.sciencedirect.com/science/article/pii/S1574652607030040.

X. Si, Y. Yang, H. Dai, M. Naik, and L. Song. Learning a meta-solver for syntax-guided program synthesis. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Syl8Sn0cK7.

S. So and H. Oh. Synthesizing imperative programs from examples guided by static analysis. In F. Ranzato, editor, *Static Analysis*, pages 364–381, Cham, 2017. Springer International Publishing. ISBN 978-3-319-66706-5.

Y. Vizel, G. Weissenbacher, and S. Malik. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE*, 103(11):2021–2035, 2015. doi: 10.1109/JPROC.2015.2455034.

X. Wang, I. Dillig, and R. Singh. Program synthesis using abstraction refinement. *Proc. ACM Program. Lang.*, 2(POPL), Dec. 2017. doi: 10.1145/3158151. URL https://doi.org/10.1145/3158151.

Appendix A

Example Grammars with Semantics

A.1 List Domain

Non-terminal	Production Rule	Semantic Constraints
`Int`	`-3, -2, -1, 0, 1, 2, 3`	
`ExprNum`	`Int`	
`ExprNum`	`maximum(ExprArr)`	$\text{len}(x_1) > 1,\ \max(y) = \max(x_1),\ \min(y) > \min(x_1)$
`ExprNum`	`minimum(ExprArr)`	$\text{len}(x_1) > 1,\ \max(y) < \max(x_1),\ \min(y) = \min(x_1)$
`ExprNum`	`sum(ExprArr)`	$\text{len}(x_1) > 1$
`ExprNum`	`first(ExprArr)`	$\text{len}(x_1) > 1,\ \max(y) \leq \max(x_1),\ \min(y) \geq \min(x_1),$ $\text{first}(y) = \text{first}(x_1),\ \text{last}(y) = \text{first}(x_1)$
`ExprNum`	`last(ExprArr)`	$\text{len}(x_1) > 1,\ \max(y) \leq \max(x_1),\ \min(y) \geq \min(x_1),$ $\text{first}(y) = \text{last}(x_1),\ \text{last}(y) = \text{last}(x_1)$

`ExprNum`	`getindex(ExprArr, ExprNum)`	$\text{len}(x_1) > 1,\ \max(y) \leq \max(x_1),\ \min(y) \geq \min(x_1),\ \text{first}(x_2) > 0,\ \text{len}(x_1) > \text{first}(x_2)$
`ExprNum`	`countSt(ExprArr, Int)`	$\text{len}(x_1) > 1,\ \text{last}(y) \leq \text{len}(x_1),\ \text{last}(y) \geq 0$
`ExprNum`	`countGt(ExprArr, Int)`	$\text{len}(x_1) > 1,\ \text{last}(y) \leq \text{len}(x_1),\ \text{last}(y) \geq 0$
`ExprNum`	`countEq(ExprArr, Int)`	$\text{len}(x_1) > 1,\ \text{last}(y) \leq \text{len}(x_1),\ \text{last}(y) \geq 0$
`ExprNum`	`countNeq(ExprArr, Int)`	$\text{len}(x_1) > 1,\ \text{last}(y) \leq \text{len}(x_1),\ \text{last}(y) \geq 0$
`ExprNum`	`countMod(ExprArr, Int)`	$\text{len}(x_1) > 1,\ \text{last}(y) \leq \text{len}(x_1),\ \text{last}(y) \geq 0$
`ExprNum`	`countNmod(ExprArr, Int)`	$\text{len}(x_1) > 1,\ \text{last}(y) \leq \text{len}(x_1),\ \text{last}(y) \geq 0$
`ExprArr`	`drop(ExprArr, ExprNum)`	$\text{len}(y) < \text{len}(x_1),\ \max(y) \leq \max(x_1),\ \min(y) \geq \min(x_1),\ \text{last}(y) = \text{last}(x_1),\ \text{first}(x_2) > 0,\ \text{len}(x_1) > \max(x_2)$
`ExprArr`	`take(ExprArr, ExprNum)`	$\text{len}(y) < \text{len}(x_1),\ \max(y) \leq \max(x_1),\ \min(y) \geq \min(x_1),\ \text{first}(y) = \text{first}(x_1),\ \text{first}(x_2) > 0,\ \text{len}(x_1) > \max(x_2)$
`ExprArr`	`sort(ExprArr)`	$\text{len}(y) = \text{len}(x_1),\ \max(y) = \max(x_1),\ \min(y) = \min(x_1),\ \text{first}(y) = \min(x_1),\ \text{last}(y) = \max(x_1)$
`ExprArr`	`reverse(ExprArr)`	$\text{len}(y) = \text{len}(x_1),\ \max(y) = \max(x_1),\ \min(y) = \min(x_1),\ \text{first}(y) = \text{last}(x_1),\ \text{last}(y) = \text{first}(x_1)$
`ExprArr`	`filterSt(ExprArr, Int)`	$\text{len}(y) < \text{len}(x_1),\ \max(y) \leq \max(x_1),\ \min(y) \geq \min(x_1)$
`ExprArr`	`filterGt(ExprArr, Int)`	$\text{len}(y) < \text{len}(x_1),\ \max(y) \leq \max(x_1),\ \min(y) \geq \min(x_1)$
`ExprArr`	`filterEq(ExprArr, Int)`	$\text{len}(y) < \text{len}(x_1),\ \max(y) \leq \max(x_1),\ \min(y) \geq \min(x_1)$
`ExprArr`	`filterNeq(ExprArr, Int)`	$\text{len}(y) < \text{len}(x_1),\ \max(y) \leq \max(x_1),\ \min(y) \geq \min(x_1)$
`ExprArr`	`filterMod(ExprArr, Int)`	$\text{len}(y) < \text{len}(x_1),\ \max(y) \leq \max(x_1),\ \min(y) \geq \min(x_1)$
`ExprArr`	`filterNmod(ExprArr, Int)`	$\text{len}(y) < \text{len}(x_1),\ \max(y) \leq \max(x_1),\ \min(y) \geq \min(x_1)$

`ExprArr`	`mapPlus(ExprArr, Int)`	$\text{len}(y) = \text{len}(x_1)$
`ExprArr`	`mapMult(ExprArr, Int)`	$\text{len}(y) = \text{len}(x_1)$
`ExprArr`	`mapDiv(ExprArr, Int)`	$\text{len}(y) = \text{len}(x_1)$
`ExprArr`	`mapPow(ExprArr, Int)`	$\text{len}(y) = \text{len}(x_1)$
`ExprArr`	`zipwithMax(ExprArr, ExprArr)`	$\text{len}(y) = \text{len}(x_1)$, $\text{len}(y) = \text{len}(x_2)$
`ExprArr`	`zipwithMin(ExprArr, ExprArr)`	$\text{len}(y) = \text{len}(x_1)$, $\text{len}(y) = \text{len}(x_2)$
`ExprArr`	`zipwithPlus(ExprArr, ExprArr)`	$\text{len}(y) = \text{len}(x_1)$, $\text{len}(y) = \text{len}(x_2)$
`ExprArr`	`zipwithMinus(ExprArr, ExprArr)`	$\text{len}(y) = \text{len}(x_1)$, $\text{len}(y) = \text{len}(x_2)$
`ExprArr`	`zipwithMult(ExprArr, ExprArr)`	$\text{len}(y) = \text{len}(x_1)$, $\text{len}(y) = \text{len}(x_2)$
`ExprArr`	`scanl1Plus(ExprArr)`	$\text{len}(y) = \text{len}(x_1)$, $\text{first}(y) = \text{first}(x_1)$
`ExprArr`	`scanl1Minus(ExprArr)`	$\text{len}(y) = \text{len}(x_1)$, $\text{first}(y) = \text{first}(x_1)$
`ExprArr`	`scanl1Mult(ExprArr)`	$\text{len}(y) = \text{len}(x_1)$, $\text{first}(y) = \text{first}(x_1)$
`ExprArr`	`scanl1Max(ExprArr)`	$\text{len}(y) = \text{len}(x_1)$, $\text{first}(y) = \text{first}(x_1)$, $\max(y) = \max(x_1)$, $\min(y) \geq \min(x_1)$, $\text{last}(y) = \max(x_1)$
`ExprArr`	`scanl1Min(ExprArr)`	$\text{len}(y) = \text{len}(x_1)$, $\text{first}(y) = \text{first}(x_1)$, $\max(y) \leq \max(x_1)$, $\min(y) = \min(x_1)$, $\text{last}(y) = \min(x_1)$
`ExprArr`	`_arg_1`	
`ExprArr`	`_arg_2`	

A.2 String Domain

Symbol	Production Rule	Semantics
Start	ntString	
ntStr	_arg_1	
ntStr	""	
ntStr	" "	
ntStr	"."	
ntStr	concat_cvc(ntStr, ntStr)	$\text{len}(y) = \text{len}(x_1) + \text{len}(x_2)$
ntStr	replace_cvc(ntStr, ntStr, ntStr)	$\text{len}(y) = \text{len}(x_1),\ \text{len}(y) \geq 0$
ntStr	at_cvc(ntStr, ntInt)	$x_2 \geq 1,\ \text{len}(x_1) \geq x_2,\ \text{len}(y) = 1$
ntStr	int_to_str_cvc(ntInt)	$\text{len}(y) \geq 1$
ntStr	ntBool ? ntStr : ntStr	
ntStr	substr_cvc(ntStr, ntInt, ntInt)	$x_2 \geq 1,\ x_2 \leq x_3,\ \text{len}(x_1) \geq x_3,\ \text{len}(y) = (x_3 - x_2 + 1)$
ntInt	1	
ntInt	0	
ntInt	-1	
ntInt	ntInt + ntInt	
ntInt	ntInt - ntInt	
ntInt	len_cvc(ntStr)	$\text{len}(x_1) = y,\ y \geq 0$
ntInt	str_to_int_cvc(ntStr)	$\text{len}(x_1) \geq 1$
ntInt	ntBool ? ntInt : ntInt	
ntInt	indexof_cvc(ntStr, ntStr, ntInt)	$x_3 \leq y,\ \text{len}(x_1) \geq y$
ntBool	true	
ntBool	false	
ntBool	ntInt == ntInt	
ntBool	prefixof_cvc(ntStr, ntStr)	$\text{len}(x_1) \leq \text{len}(x_2)$
ntBool	suffixof_cvc(ntStr, ntStr)	$\text{len}(x_1) \leq \text{len}(x_2)$
ntBool	contains_cvc(ntStr, ntStr)	$\text{len}(x_1) \leq \text{len}(x_2)$