



TECHNISCHE UNIVERSITEIT DELFT

MASTER OF SCIENCE THESIS IN DATA SCIENCE AND
ARTIFICIAL INTELLIGENCE TECHNOLOGY

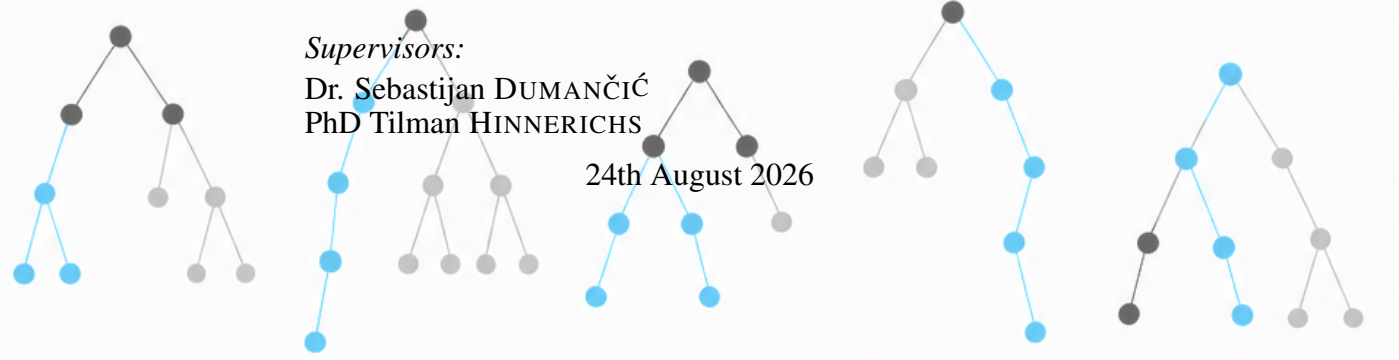
Learning from Failure in Program Synthesis: Generalizing and Minimizing Conflicts

Alperen GUNCAN
I

Supervisors:

Dr. Sebastijan DUMANČIĆ
PhD Tilman HINNERICHS

24th August 2026



Learning from Failure in Program Synthesis: Generalizing and Minimizing Conflicts

Master's Thesis in Data Science and Artificial Intelligence Technology

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

Alperen Guncan

24th August 2026

MSc presentation

2nd September, 2026

Graduation Committee

Dr. Sebastijan Dumančić (chair)	Delft University of Technology
Dr. Manuel Mazo Espinosa	Delft University of Technology
Dr. Andreea Costea	Delft University of Technology

Abstract

Program synthesis searches a grammar-defined space for a program that satisfies a user specification. Constraints can significantly reduce this search space. They can either be added manually by the user or derived through conflict analysis, which turns failed candidates into learned constraints. These constraints often retain irrelevant candidate details and accumulate costly redundancy. This thesis extends a conflict-analysis framework with core-guided MUC generalization and constraint-set cleanup. Generalization uses a semantic unsatisfiable core to wildcard irrelevant branches while preserving the candidate root and core nodes, as well as values referenced by core relations. Cleanup minimizes the active constraint set in many ways: by either removing constraints through exact deduplication or subsumption, or by merging constraints through single-differences and cross-relations. We evaluate both extensions on several benchmarks, assessing the reduced search space, constraint set size, runtime, and number of solved problems and used enumerations. At the largest program sizes, generalized MUC reduces enumerations by anywhere between 60 and 99.9%, consistently improving over the previous MUC implementation. Similarly, the strongest cleanup configuration can reduce the mean active constraint population by more than three orders of magnitude. While the new extensions manage to solve slightly more problems, their runtime improvement is negligible. Nonetheless, our extensions show that conflict analysis can learn substantially more general constraints and represent them more compactly.

Contents

1	Introduction	1
2	Background	5
2.1	Program Synthesis	5
2.1.1	The Search Space and Grammars	5
2.1.2	Problem Specification	7
2.1.3	Syntax and Semantics of Grammar Rules	8
2.1.4	Enumerating Candidate Programs	8
2.1.5	Guiding the Search with Constraints	8
2.2	Conflict Analysis in Program Synthesis	10
2.2.1	Conflict-Analysis Pipeline	10
2.2.2	Representing Learned Constraints	11
2.3	Conflict-Analysis Techniques	13
2.3.1	Unsatisfiable-Core Analysis	13
2.3.2	Semantic Analysis	14
2.3.3	Execution-Based Redundancy Analysis	14
3	Related work	15
3.1	Conflict-Driven Learning in Program Synthesis	15
3.2	Generalization from Failures in Logic Program Synthesis	16
3.3	The Herb.jl Conflict Analysis Framework	16
3.4	Clause Learning and Database Management in SAT/CP	17
4	Problem Statement	19
5	Methods	23
5.1	Extended Conflict-Analysis Pipeline	23
5.2	Core-Guided Generalization of Semantic Conflicts	25
5.2.1	Wildcarding Non-Core Branches	25
5.2.2	Preserving Semantic Support	26
5.2.3	Preserving the Candidate-Root Context	27
5.2.4	Generalization Algorithm	27
5.3	Constraint-Set Minimization	29
5.3.1	Exact Deduplication	30

5.3.2	Subsumption	31
5.3.3	Single-Difference Merging	33
5.3.4	Cross-Relation Combination	35
6	Experimental Evaluation	37
6.1	Experimental Setup	38
6.1.1	Measurements and Aggregation	39
6.2	RQ1: Core-Guided MUC Generalization	40
6.2.1	Effect on Search, Runtime, and Constraints	41
6.2.2	Generalization During Search	43
6.2.3	Usefulness of Semantic Annotations	45
6.3	RQ2: Constraint-Set Minimization	47
6.3.1	Individual Techniques	47
6.3.2	Combined Cleanup	50
6.3.3	Constraint-Set Development During Search	52
6.4	RQ3: Solution Coverage and Enumeration Effort	54
7	Conclusions, Future Work, and Reflection	57
7.1	Conclusions	57
7.2	Future Work	58
7.2.1	A Specialized Semantic Reasoner	58
7.2.2	Budgeted Unsatisfiable-Core Minimization	58
7.2.3	Pipeline Efficiency and Parallelism	59
7.2.4	More Conflict Analyses and Search Strategies	59
7.2.5	Adaptive Retention and Broader Validation	59
7.3	Reflection	60
7.3.1	Impact on the Program-Synthesis Community	60
7.3.2	Impact on Wider Society	60
7.3.3	Use of Generative AI Tools	61
A	Benchmark Grammars and Semantic Annotations	65
A.1	DeepCoder	65
A.1.1	Grammar	66
A.1.2	Semantic Annotations	66
A.2	SyGuS PBE-SLIA	67
A.2.1	Grammar	68
A.2.2	Semantic Annotations	68
A.3	Morpheus	70
A.3.1	Grammar	70
A.3.2	Semantic Annotations	70

Chapter 1

Introduction

Many useful computational tasks are easier to describe than to implement. A domain expert may know how a table should be cleaned, how a list of measurements should be transformed, or what the correct output should look like for a few representative inputs, but turning that intent into code requires programming expertise. Program synthesis aims to close this gap: given a description of *what* a program should do, a computer figures out *how* to do it. This could empower experts in fields such as data science and biology to solve computational problems without needing to become software engineers.

At its heart, program synthesis is a search problem. Given a set of building blocks defined by a grammar and a specification of the desired behavior, a synthesizer explores the space of possible programs until it finds one that satisfies the specification [Gulwani et al., 2017]. Different synthesis paradigms instantiate this idea in different ways. In sketching, the synthesizer fills holes in a partial program [Solar-Lezama et al., 2006]; in component-based CEGIS, it searches for a composition of available components [Jha et al., 2010]; and in syntax-guided synthesis, it searches within a grammar-defined space of expressions [Alur et al., 2013]. In all cases, the central challenge is that the number of possible programs grows very quickly.

This search-oriented view connects program synthesis to automated reasoning, especially SAT and constraint solving. A SAT instance represents each yes-or-no choice by a Boolean decision variable. The solver then searches for an assignment of those variables that satisfies the logical constraints [Marques-Silva et al., 2009]. Grammar-based synthesis has an analogous representation, using abstract syntax trees (ASTs). Each unresolved AST position is a decision point, and selecting a grammar rule fixes the operation at that node. In the end, a complete assignment of grammar-rule choices yields a complete program, represented as a syntax tree. Unlike the rigid logical specification in SAT, the specification can take many forms, like input-output example pairs, natural language, or other high-level descriptions. Nonetheless, both settings must avoid exploring choices that cannot lead to a solu-

tion. As such, we can often transfer techniques from one domain to another.

A basic approach of program synthesis is simple enumeration: generate a program, test it against the examples, and if it fails, discard it and move to the next one [Gulwani et al., 2017]. While simple, this method quickly becomes infeasible. The number of possible programs grows exponentially with the number of building blocks, a problem known as combinatorial explosion [Solar-Lezama et al., 2006, Balog et al., 2016]. For synthesis to be practical, the search must be intelligent.

A common way to make the search more efficient is to limit the options we are considering by placing constraints on the search space. Constraints rule out programs that cannot satisfy the specification, so the synthesizer does not construct and test them [Hinnerichs et al., 2025]. They often come from expert domain knowledge, such as "never divide by a variable that could be zero" or "the operation must preserve list length". Such human-written constraints are typically task-agnostic within a domain. Once supplied with a grammar, they can guide every synthesis task expressed in it. By pruning entire branches of the search tree upfront, constraints can offer a significant speedup.

However, pre-defined constraints are static and often insufficient. Conflict analysis, also known as conflict-driven learning [Feng et al., 2018], equips the synthesizer with a form of learning-through-failure inspired by debugging. When a human programmer writes code that does not work, they do not just discard it and start again from scratch. They analyze the error, identify the underlying mistake, and ensure it is not repeated. This concept is mirrored in clause learning in SAT. In that context, the underlying mistake is called an *explanation* of the conflict. For program synthesis, we can encapsulate this explanation as a constraint, and then prune other programs exhibiting the same kind of flaw [Feng et al., 2018, Cropper and Dumančić, 2020]. Unlike domain-wide human-written constraints, these learned constraints are task-specific, since they are derived from the current examples and specification. The narrower scope lets them target problem-specific failure patterns more effectively.

Consider the example in Figure 1.1. Suppose the user wants a program that maps $[3, 1, 4, 2]$ to $[4, 2]$ using basic list operations. A naive synthesizer would reject only `reverse(sort(x))`. With conflict analysis, the synthesizer might instead learn that the output contains numbers not present in the specification's output (3 and 1), or that its length is too large. Either observation can be turned into a constraint that guides future search.

While powerful, this uncovers an important insight: the benefit of conflict analysis depends heavily on the quality of the learned constraints. Existing conflict analysis techniques typically operate on a one-to-few basis: one failed program generates one, or a few, new constraints that are mostly focused on pruning the specific failing

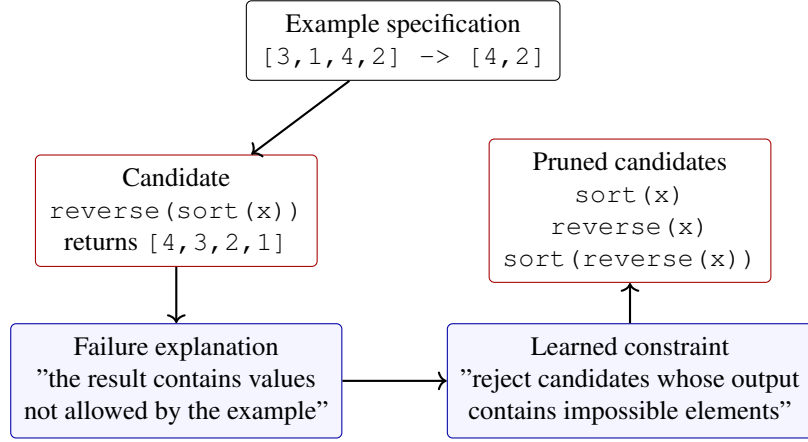


Figure 1.1: A failed candidate can be converted into a learned constraint. The constraint is useful when it rejects many other candidates that would fail for the same semantic reason.

candidate. This is inefficient and creates two significant problems:

1. This one-to-few view misses a crucial opportunity for deeper learning. Existing conflict-analysis pipelines can often explain an individual failed program, but the explanation may stay close to that particular program. The search space is large because many syntactically different candidates can have similar behavior on the examples. If the synthesizer can learn constraints at the level of that behavior, then a small number of failures can rule out a much larger part of the search space.
2. The set of constraints can grow very large, and checking every candidate program against thousands of constraints becomes a performance bottleneck in itself. In the world of SAT/CP solving, this is a well-understood problem; an excess of learned conflicts can slow down unit propagation to the point where the benefit of learning is negated [Marques-Silva et al., 2009, Audemard and Simon, 2009].

In this thesis, we study these two issues separately and jointly: how to learn constraints that are more general than individual failed programs while keeping the learned set compact. Together, these directions lead to the following question:

How can conflict analysis in program synthesis learn constraints that are both more general and more manageable, so that they prune larger parts of the search space without introducing prohibitive overhead?

We study this question by augmenting the conflict-analysis pipeline with a generalization step and a cleanup stage before learned constraints are added to the search. Generalization makes constraints more widely applicable. In the example above, a

naive framework may only discard the candidate because it retains particular unwanted values, like 1 and 3. However, we would like to capture a more general semantic reason for the failure, e.g., “you cannot retain values absent from the required output.” Minimization addresses the complementary problem of accumulation. Often, a conflict analysis framework will just keep all constraints it generates. However, unless the techniques are already very general and disjoint in their workings, the constraints can be very similar or even identical. The main methods we explore are either removing redundant constraints completely or combining constraints into more general variants that are cheaper to maintain. Intuitively, generalization makes each lesson apply to more candidates, while minimization retains the same lessons with less overhead to store and check.

We evaluate the new extensions on several benchmarks. With them, we show that generalized constraints can prune substantially more of the search space. Furthermore, it achieves this with either significantly fewer constraints or much quicker. The annotation analysis further shows that precise equalities and domain-specific relations contribute much more often to useful cores than generic range facts. We also show that minimization is effective. However, depending on whether the benchmark introduces a lot of exact or near similar-redundancy, different cleanup strategies are more effective. Under our implementation, minimization does not improve end-to-end synthesis time. Finally, together, these extensions solve slightly more problems while substantially reducing enumerations in finding them.

If these techniques can be developed further, conflict analysis could become a more practical tool for program synthesis systems. The long-term goal is a synthesizer that does not merely test candidate programs faster, but improves its own search process as it fails. Such a system would be closer to the way programmers use mistakes: not as isolated events, but as clues about the structure of the problem they are solving. More generally, this makes program synthesis less wasteful and more explainable. Failures are turned into explicit constraints rather than remaining hidden, and all generalization and minimization steps are transparent. This also has a societal dimension: if reliable synthesis tools become easier to use and reason about, they could lower the barrier to automation for people without formal programming training.

Chapter 2

Background

2.1 Program Synthesis

At its most fundamental level, program synthesis is the automated construction of an executable program from a high-level specification of its desired behavior [Gulwani et al., 2017]. It reframes the act of programming as a search problem. Instead of a human manually writing code, a synthesizer systematically explores a predefined space of possible programs. Ultimately, it has to discover a program that satisfies the user’s intent.

2.1.1 The Search Space and Grammars

As part of defining the **search space**, we first have to formalize how a program can be constructed. These rules for building syntactically valid programs are often defined with a **grammar**. A grammar specifies which combinations of components constitute valid programs [Alur et al., 2013, Gulwani et al., 2017]. This effectively encodes domain knowledge about the programming task. It usually consists of:

- **Terminals:** The atomic building blocks of the language, such as constants (`1`, `true`), variables (`x`, `y`), or specific operations (`+`, `sort`).
- **Non-terminals:** Abstract symbols that represent a class of expressions or statements, such as `Expr`, `OpSeq`, or `Int`.
- **Start symbol:** The non-terminal from which every complete program is derived. Its type determines the required type of the synthesized program.
- **Production Rules:** Rules that specify how a non-terminal can be expanded into a sequence of terminals and other non-terminals. For example, a rule `ExprNum = maximum(ExprArr)` defines taking the maximum of an array.

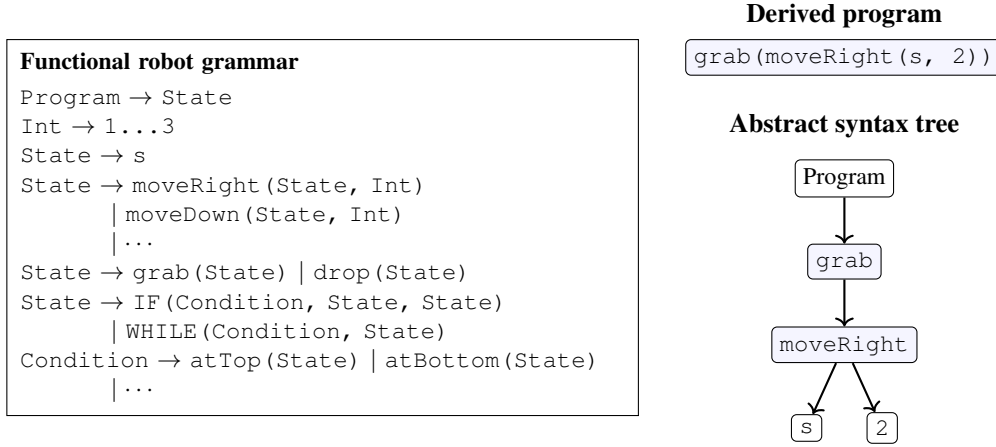


Figure 2.1: A functionalized fragment of the robot grammar and the AST for a derived program. The explicit state argument makes each action a state transformation while preserving the grammar-based tree representation.

A grammar that allows any non-terminal to be expanded without regard to its surrounding context is known as a **Context-Free Grammar (CFG)**. CFGs are popular in program synthesis due to their simplicity and expressive power. A program constructed from a grammar can be visualized as an Abstract Syntax Tree (AST), where each node records the production rule applied at that position. Rules with child non-terminals form internal nodes, while terminal rules form the leaves. The arity and child types of a rule determine which subtrees may occur beneath it. Figure 2.1 uses a functionalized fragment of the robot grammar in which every action takes a state and returns a new state.

The search space defined by a grammar, denoted $\mathcal{P}_G$, is the set of all programs that can be generated by applying the grammar’s rules from its start symbol. In principle, this space can be infinite: a CFG can generate programs of arbitrary size and complexity. In practice, synthesizers must restrict the search space to make enumeration tractable [Feng et al., 2018]. This can be done by limiting the depth of the AST, the total number of operations, or even the total number of candidates considered.

Designing a well-specified grammar is crucial. A good goal is to encode prior knowledge about which programs are likely to be useful. This way, the grammar can impose an inductive bias on the search toward promising candidates. Omitting a required operation excludes the intended solution. Conversely, many irrelevant operations explode the search space. A useful grammar must therefore balance expressiveness and tractability.

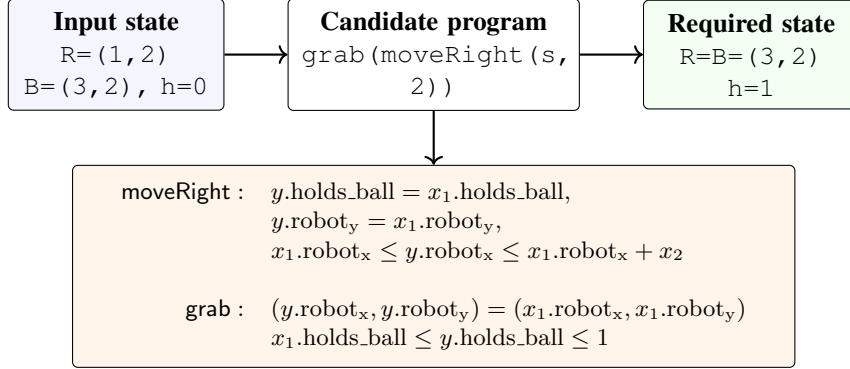


Figure 2.2: Two complementary forms of specification in the robot domain. Input-output examples describe the required behavior of a complete program, whereas semantic annotations describe formal local behavior, i.e., location and held ball are as expected (robot is assumed not to *wrap* around grid edges).

2.1.2 Problem Specification

While the grammar defines what programs *can* be constructed, the **problem specification** defines what the desired program *should do*. The specification is the formal contract that the synthesizer must fulfill. It can take many forms, each with its own trade-offs between ease of use and formal precision. The ones we refer to in this thesis are:

- **Input-Output Examples:** This is a common and intuitive form of specification, where the user provides concrete examples of inputs and their corresponding correct outputs [Gulwani et al., 2017]. The synthesizer’s goal is to find a program that is consistent with all provided examples. This setting is also known as **programming by example (PBE)**.
- **Semantic Annotations:** Specifications can be embedded directly into the grammar via semantic predicates. These constraints can be specific to each production rule. They restrict the set of valid programs by enforcing properties on intermediate values (e.g., length preservation, value bounds, or relationships between inputs and outputs).

The two forms complement one another: input-output examples constrain whole programs, whereas semantic annotations record properties of individual rules. Figure 2.2 contrasts both forms for a small robot task.

The quality of the specification is paramount. A program that is correct according to a weak or incomplete specification can still not match the user’s true intent. A specification can be hardened by combining multiple paradigms. This is because different specification paradigms capture complementary aspects of program behavior, reducing spurious candidates.

2.1.3 Syntax and Semantics of Grammar Rules

A grammar describes the syntactic structure of the search space, but its rules must also have an executable meaning. Together, the available rules and their interpretations form a domain-specific language (DSL). The syntax determines which ASTs can be constructed, while the semantics determine the value produced when a rule is executed on the values of its children [Gulwani et al., 2017, Alur et al., 2013].

In this work, grammar rules may additionally carry semantic annotations that summarize properties of this behavior. Their purpose is to expose the relationships between a rule’s child values and its output. Conflict-analysis techniques later use these relationships to explain why a candidate fails.

An annotation describes a relation between a rule’s child values and its output. We denote the value of the i -th child by x_i and the output of the rule by y . For example, an annotation $y \leq \max(x_1)$ states that applying the rule cannot produce a value larger than the maximum element of its first input. Figure 2.2 illustrates the same idea for the robot domain. Multiple grammar rules may share the same annotation, allowing them to be treated as equivalent with respect to a particular semantic conflict [Feng et al., 2018, Poeth, 2025].

2.1.4 Enumerating Candidate Programs

Once the grammar and specification are defined, the synthesizer must search for a program $p \in \mathcal{P}_G$ that satisfies the specification. Enumerators may construct programs top-down or bottom-up [Gulwani et al., 2017]. Due to its simpler constraint mechanism, both for storage and application, we solely consider top-down enumeration in this thesis.

Top-down enumeration. Top-down enumeration begins with the grammar’s start symbol, represented as an unresolved position or a *hole* [Hinnerichs et al., 2025]. The enumerator repeatedly replaces a hole with a compatible production rule, introducing new holes for its children. When no holes remain, the resulting AST is a complete candidate that can be evaluated against the specification. Because partial trees are available during construction, constraints can reject invalid choices before a complete candidate is produced.

2.1.5 Guiding the Search with Constraints

The search space defined by a grammar is often too vast for naive enumeration. **Constraints** are one of the primary mechanisms for making this search intelligent. A constraint is a predicate that a partial or complete program must satisfy. If a program violates a constraint, it can be immediately discarded, which saves time

on evaluation [Hinnerichs et al., 2025, Feng et al., 2018].

Constraints in program synthesis can be broadly categorized:

- **Syntactic Constraints:** These rules forbid specific structural patterns in a program’s AST. For example, in the robot domain, a constraint could forbid `grab(grab(x))`: the second application cannot change the state further and is therefore redundant.
- **Semantic Constraints:** These rules relate to the meaning or behavior of a program. For example, a constraint may require an operation to preserve the length of its input list or ensure that its output remains within a particular range.

The inclusion of constraints has multiple benefits. By eliminating programs that are guaranteed to fail, the synthesizer reduces wasted computation and can explore deeper or more complex candidate programs. Constraints also allow the system to reason about partial solutions: if a subprogram cannot satisfy the relevant constraints, any completion that retains that subprogram can be pruned before a complete candidate is constructed.

Constraints are often expressed over the grammar rules selected at positions in a program tree. Each position has a domain containing the rules that may be placed there. When a partial program violates a constraint, the corresponding rule choices can be removed before the tree is completed. This allows the synthesizer to prune entire groups of candidate programs rather than rejecting them individually after evaluation.

Some constraints are supplied before the search begins, for example, to encode domain knowledge or rule compatibility. Other constraints can be learned during the search from failed candidate programs [Feng et al., 2018]. Through the subtree matching in Figure 2.3, one learned constraint can prune later, larger candidates. Conflict analysis, introduced in the next subsection, provides a systematic way to derive such constraints.

The benefit of a constraint depends on the search it eliminates relative to the overhead costs. These are mostly comprised of storing, propagating, and checking each constraint. Generally, a constraint that rejects many invalid candidates can substantially reduce enumeration, while a large one, or a non-disjoint set, may introduce more overhead than it saves [Hinnerichs et al., 2025, Feng et al., 2018]. Moreover, constraints must be sound: an incorrectly generalized constraint may prune a valid solution. Effective constraint-guided synthesis therefore requires a balance between pruning coverage, enforcement cost, and correctness.

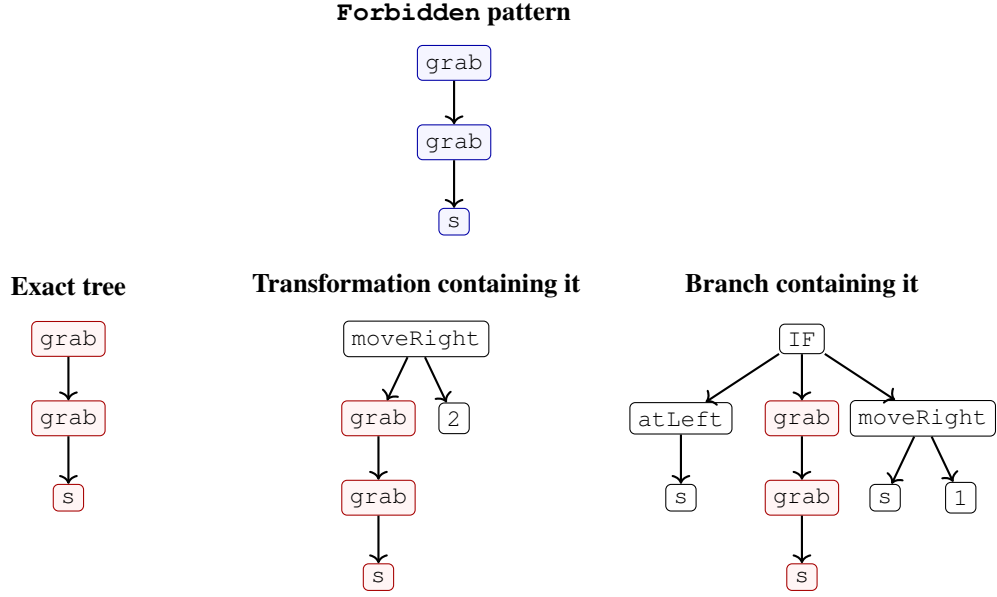


Figure 2.3: A forbidden AST pattern prunes the exact tree and every larger tree that contains it as a subprogram. Red nodes show the matching subtree; the surrounding context does not affect the match.

2.2 Conflict Analysis in Program Synthesis

Conflict analysis, also known as conflict-driven learning, treats a failed program as a *conflict* from which the synthesizer extracts an explanation [Feng et al., 2018]. The explanation is converted into a learned constraint, often called a *no-good* or *conflict clause*, that prevents the same conflicting pattern or a generalized class of it from being explored again [Marques-Silva et al., 2009].

This iterative process of failing, analyzing, learning, and sending constraints allows the synthesizer to progressively refine its search space. While it can reduce repeated exploration, its effectiveness depends on the pruning coverage of the learned constraints and the cost of managing them, which this thesis tries to improve on.

2.2.1 Conflict-Analysis Pipeline

Conflict analysis is applied after the synthesizer evaluates a complete candidate program. If the candidate satisfies every input-output example, it is returned as a solution. Otherwise, the failed candidate, its observed output, and the example it violates, also called a *counterexample*, are passed to the conflict-analysis pipeline [Poeth, 2025]. Execution exceptions can similarly be treated as failures from which constraints may be learned.

The pipeline separates conflict detection from conflict analysis. During *conflict detection*, each enabled technique examines the failed candidate and determines whether the failure contains information that it can use. A technique may, for example, detect a violated semantic property, an unsatisfiable semantic model, or redundant execution behavior. If no relevant conflict is found, that technique produces no result.

When a conflict is detected, the technique proceeds to *conflict analysis*. In this phase, it identifies the program choices associated with the failure and translates them into one or more learned constraints. Separating these phases allows several techniques to inspect the same failed candidate while using different explanations of its failure. Consequently, a single candidate may produce no learned constraints, one constraint, or several constraints. This fan-out is shown in the middle layer of Figure 2.4.

The generated constraints are classified according to how they should be applied during subsequent search. Structural constraints are supplied directly to the solver, whereas grammar-level constraints are attached to the grammar and may apply at multiple locations in future program trees [Poeth, 2025]. The complete flow, including these two destinations, is shown in Figure 2.4. The precise representation and matching behavior of these constraints are introduced in the following subsection. After the learned constraints have been installed, enumeration resumes. The solver can then use them to reject partial or complete programs that match the learned forbidden patterns.

2.2.2 Representing Learned Constraints

All methods introduced later in this thesis operate on learned constraints represented as patterns over abstract syntax trees [Poeth, 2025]. This representation is natural for grammar-based program synthesis: candidate programs are generated from grammar rules, and a learned constraint can describe a family of programs by describing a forbidden tree pattern. A constraint of the form `Forbidden(t)` rejects any program, or relevant subprogram, that matches the tree pattern *t*. If *t* is a fully concrete AST, then the constraint only forbids programs containing exactly that structure. To make constraints more general, the tree can contain more abstract nodes. The most important node types are concrete rule nodes, domain rule nodes, and wildcard nodes, shown progressively in Figure 2.5.

A concrete rule node represents a single grammar rule. For example, a node may state that the root of the forbidden pattern must be the rule `sort`. Such a constraint is precise, but it may be too specific: if several rules fail for the same reason, representing each of them separately creates multiple overlapping constraints.

A domain rule node generalizes this by allowing a set of grammar rules at the

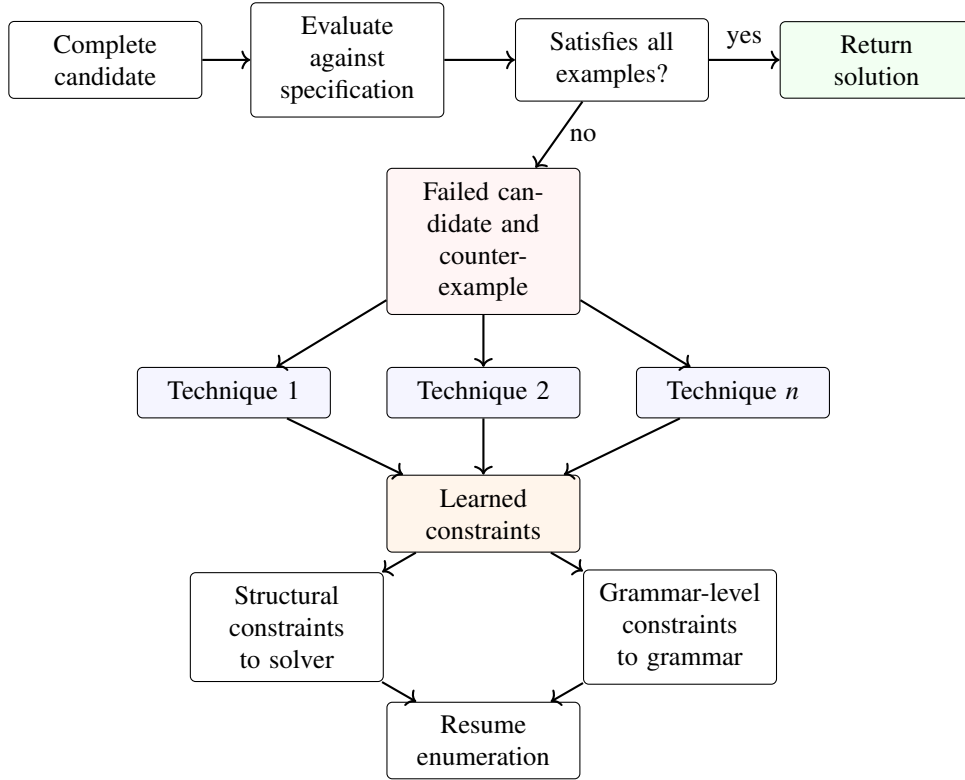


Figure 2.4: The conflict-analysis pipeline. A failed candidate can be inspected by several techniques, whose learned constraints are then installed either in the solver or in the grammar before enumeration resumes.

same position in the tree. The grammar rules must be of the same arity and children types to be compatible within the same node. For example, instead of forbidding only `sort(x)`, a constraint may forbid any program whose root is one of `{sort, reverse, filter_odds}` and whose child matches the same subpattern. Domain nodes are the main representation used when two or more rules are interchangeable with respect to a learned conflict.

A wildcard node is even more general - it can represent any subtree at that location. Wildcards are useful when the reason for failure depends only on a small part of the candidate program. Rather than preserving the entire failed candidate, the learned constraint can retain the conflict-relevant structure while replacing unrelated branches with wildcards.

These three node types create a hierarchy of specificity. As Figure 2.5 shows, each generalization retains the restrictions already expressed by the preceding pattern while relaxing one additional part of the tree. This is important for understanding the cleanup methods described later. They rely on this hierarchy for both remov-

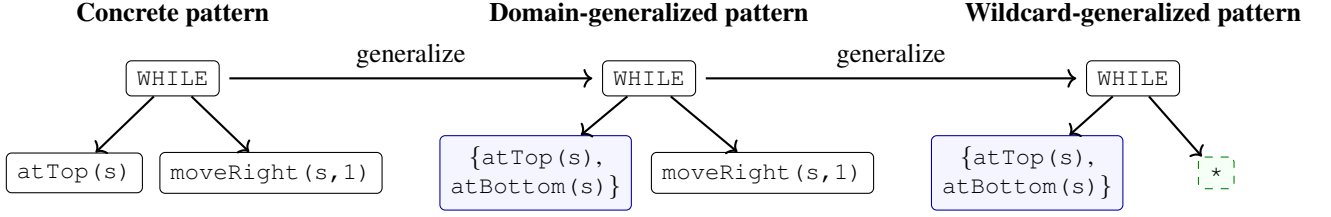


Figure 2.5: A representational example of progressively generalizing a forbidden AST pattern. The second pattern generalizes the condition with a domain node; the third assumes that the loop body is irrelevant to the conflict and replaces it with a wildcard.

ing redundant/more-specific constraints and combining multiple constraints on the same abstraction level.

2.3 Conflict-Analysis Techniques

The conflict-analysis framework provides several independent techniques with the same two-phase interface: first detect whether a failed candidate exposes a useful kind of conflict, then construct a forbidden AST pattern from that information. The techniques differ in the evidence they use and in the scope of the resulting constraint. The three techniques used in this thesis are described below.

2.3.1 Unsatisfiable-Core Analysis

The MUC technique adapts unsatisfiable-core-based conflict analysis from Feng et al. [2018]. Given a failed candidate, a grammar, and a counterexample, it constructs a satisfiability-modulo-theories (SMT) model from the semantic annotations attached to the candidate’s AST nodes and the required output. If the model is satisfiable, the technique finds no semantic conflict explanation. Otherwise, the solver returns an unsatisfiable core: a subset of the asserted semantic relations that is sufficient for the contradiction.

Each core relation is mapped back to the AST node that contributed it. For every such node, the technique computes an equivalence-modulo-conflict (EMC) class. Similar to Feng’s paper, this is the class of all compatible grammar rules with the same semantic behavior. It then preserves the candidate’s tree structure while replacing affected nodes. The resulting `Forbidden` pattern rejects the original candidate and similar ones with the same tree structure covered by the explanation.

2.3.2 Semantic Analysis

Semantic analysis is a lighter-weight alternative that checks annotations of a single relevant root-level rule rather than constructing a whole-program SMT model [Poeth, 2025]. The technique evaluates the rule’s child subprograms on the counter-example input, combines those values with the required output, and tests each annotation associated with the rule. An annotation that evaluates to false is a direct semantic explanation for the failed candidate.

For every violated annotation, the technique finds compatible rules with the same semantic property. It then constructs a `Forbidden` pattern with a domain rule node and whose children preserve the observed subprograms. Compared with MUC analysis, the technique is cheaper and easier to apply, but its explanation is limited to a local rule and cannot identify conflicts that depend on relationships between distant nodes in the AST.

2.3.3 Execution-Based Redundancy Analysis

Execution-based redundancy analysis (ERA) does not require semantic annotations or an SMT solver. It inspects the observed execution result of a small failed candidate and recognizes outcomes that are unsuitable or redundant in the chosen DSL [Poeth, 2025]. The current implementation detects execution exceptions and empty results. The latter case identifies programs that can be replaced by a simpler terminal value or are otherwise treated as unhelpful by the domain.

When ERA detects one of these cases, it returns an exact `Forbidden` pattern for the remaining subprogram. This grammar-level constraint can reject that subprogram wherever it appears during later enumeration. ERA therefore produces simple, potentially widely applicable constraints, but its usefulness depends on the chosen redundancy conditions and the imposed size limit on analyzed programs. In contrast to the semantic techniques, it does not generalize by grouping rules into semantic equivalence classes.

Chapter 3

Related work

This thesis builds on a common idea that appears in several areas of program synthesis and automated reasoning: a failed candidate should not only be discarded, but should also be used to guide the remaining search. Related work differs mainly in what kind of candidate fails, how the failure is explained, or what form of learned constraint is produced.

3.1 Conflict-Driven Learning in Program Synthesis

The work of Feng et al. [2018] introduced a CDCL-style synthesis architecture and implemented it in `Neo`. The solver alternates among decision-making, deduction, and conflict analysis. `Decision` assigns domain-specific language (DSL) components to holes in a partial AST. `Deduction` propagates implications from DSL semantics and learned lemmas. Finally, conflict analysis is triggered when symbolic constraints become inconsistent with the specification. With this pipeline, failed partial programs become reusable knowledge for future searches.

`Neo` uses the MUC and EMC machinery introduced in the previous chapter. The goal is to generalize conflicts across compatible component substitutions at selected nodes. Its learned lemmas act as no-goods over partial assignments, removing whole families of infeasible completions rather than only the observed assignment.

`Neo` demonstrates that conflict learning in synthesis can be both general and practical. It combines grammar structure, semantic annotations, and SAT/SMT-backed infeasibility proofs in one loop. When tested on list-processing and data-wrangling tasks, results show substantial reductions in the explored search space [Feng et al., 2018].

However, this generalization is still tied to variants of the same partial program structure. It can generalize substitutions at selected nodes, but it does not aim to derive constraints that apply across more structurally dissimilar candidates with

the same semantic failure mode. The approach also does not address the issue of constraint accumulation over a long search.

3.2 Generalization from Failures in Logic Program Synthesis

Failure-driven synthesis in inductive logic programming, particularly the Popper line of work, also treats negative outcomes as first-class learning signals [Cropper and Dumančić, 2020]. Instead of retaining only successful hypotheses, these methods learn constraints that rule out broad classes of future hypotheses. In effect, they generalize from failure explanations and reduce repeated exploration of equivalent dead ends. In this way, failure informs constraints over hypothesis structures and is not just used as a final accept/reject signal. The search then becomes progressively more informed: each failed hypothesis narrows future candidate generation, similar in spirit to nogood learning in constraint solving.

The main strength of this line of work is the explicit feedback loop between failed candidates and future pruning. A key open issue appears when moving to richer grammar-based synthesis spaces where conflicts can be local (single node), compositional (interaction between subtrees), or global (specification-level semantic mismatch). In those settings, independently learned constraints can overlap heavily and require consolidation. The goal is to avoid trading search cost for constraint-management overhead. Yet, consolidation of such constraints in grammar-based synthesis remains unclear. Unlike this line of logic-program synthesis, this thesis explicitly studies how overlapping constraints can be generalized and consolidated after they are learned.

3.3 The Herb.jl Conflict Analysis Framework

A predecessor of this thesis is the work of Poeth [2025] in implementing a flexible and general conflict analysis framework tailored to `Herb.jl`. It separates conflict analysis from solver internals through a shared AST-oriented interface. The framework also supports multiple analysis techniques in a single pipeline. Finally, it distinguishes between semantic-based and execution-based conflicts and shows how each yields different pruning behavior.

This modular design has practical consequences. It enables side-by-side evaluation of techniques under the same solver conditions. Extending the pipeline with new techniques is also straightforward. Most importantly, it reuses parsed conflict information across analyses and makes domain adaptation more systematic. This design makes explicit how the usefulness of conflicts depends on available

semantic annotations, their abstraction level, and overlap among grammar rules.

The framework still leaves open the algorithmic question of compressing many related learned constraints into a smaller, stronger set with less overlap, particularly when multiple techniques are similar in nature and are active. Ideally, constraints produced by different techniques could reinforce one another, yielding more general constraints than any individual technique would produce alone. This thesis extends the framework by introducing a new stage in the pipeline that can inspect constraints to combine and clean up across many techniques.

3.4 Clause Learning and Database Management in SAT/CP

Constraint-solving literature provides a complementary perspective on the same challenge: learning from conflicts is only effective when the learned database remains informative and manageable. CDCL-style SAT solving established conflict analysis and clause learning as core mechanisms for pruning search [Marques-Silva et al., 2009]. In those systems, conflict analysis identifies a cause of failure in the implication graph and converts it into a learned clause that prevents recurrence of the same reasoning pattern.

Two technical lessons are particularly relevant. First, learned clauses differ in quality. Some clauses explain a conflict compactly and prune many future assignments, while others are valid but weak. Moreover, even individually sound clauses can collectively slow the solver down if the learned database becomes too large or too redundant. Work on learned clause quality and database management, such as the work of Audemard and Simon [2009], therefore studies which clauses should be kept and which should be removed.

The same conclusion is drawn, regardless of whether it is studied in the domain of constraint programming or nogood-learning. That is, high-quality explanations can dramatically reduce search and be worthwhile, so long as their upkeep cost is not too extreme. Therefore, effective systems must balance soundness, pruning power, and compactness. This thesis adopts that database-management motivation but differs in its objective. Usually, constraint solvers use heuristics, like how often a clause is used, how long it is, or when it was found. Our proposal is fully focused on preserving the pruning and only getting rid of redundancy.

Chapter 4

Problem Statement

In this section, we formally define the problem of program synthesis and then introduce the specific problem of **Conflict Generalization**, which aims to learn more effective constraints from multiple failed synthesis attempts.

Definition 1 (Program Synthesis). Given a context-free grammar $\mathcal{G}$ that defines a program space $\mathcal{P}_{\mathcal{G}}$, and a specification of user intent $\mathcal{S} : \mathcal{P}_{\mathcal{G}} \rightarrow \{\top, \perp\}$, **find** a program $p \in \mathcal{P}_{\mathcal{G}}$ such that $\mathcal{S}(p) = \top$.

To guide the search, the grammar can be augmented with constraints.

Definition 2 (Program Constraint). Given a program space $\mathcal{P}_{\mathcal{G}}$, a **constraint** is a predicate $\phi : \mathcal{P}_{\mathcal{G}} \rightarrow \{\top, \perp\}$. A program p satisfies the constraint, written $p \models \phi$, if $\phi(p) = \top$; otherwise, the constraint excludes p from the search space.

Definition 3 (Constrained Program Synthesis). Given a grammar $\mathcal{G}$ with an initial set of constraints $\mathcal{C}$, and a specification $\mathcal{S}$, **find** a program $p \in \mathcal{P}_{\mathcal{G}, \mathcal{C}}$ that is derivable from $\mathcal{G}$, satisfies all constraints in $\mathcal{C}$, and for which $\mathcal{S}(p) = \top$. More formally,

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

During the search, the synthesizer may generate complete programs that satisfy the current constraints but fail the specification. We call these **conflicts**.

Definition 4 (Conflict). Given a constrained program space $\mathcal{P}_{\mathcal{G}, \mathcal{C}}$ with a specification $\mathcal{S}$, a **conflict** is a program $p \in \mathcal{P}_{\mathcal{G}, \mathcal{C}}$ for which $\mathcal{S}(p) = \perp$.

We distinguish a conflict from the learned constraint derived from it. The conflict is the failed program observed during search. Throughout this thesis, we call a learned constraint that closely preserves the failed program’s structure a **direct constraint** (as it *directly* takes the conflict’s structure). A naive conflict-analysis process generates one direct constraint ϕ_p for each conflict p , with little to no extrapolation, and then extends $\mathcal{C}$ by adding ϕ_p .

Definition 5 (Conflict and Constraint Sets). A **conflict set** $\mathcal{K} = \{p_1, p_2, \dots, p_n\}$ is a collection of n distinct conflicts generated during a synthesis task. The corresponding *naive* constraint set $\mathcal{C}_{\mathcal{K}}$ is the set of all derived *direct constraints*, i.e., $\mathcal{C}_{\mathcal{K}} = \{\phi_{p_1}, \phi_{p_2}, \dots, \phi_{p_n}\}$.

However, this one-to-one mapping of conflicts to constraints is nearly always sub-optimal. A single underlying flaw in the search space can manifest as numerous, structurally similar conflicts. We now define the general operation for soundly transforming one constraint set into another more useful set.

Definition 6 (Sound Replacement). Given two constraint sets $\mathcal{C}_A$ and $\mathcal{C}_B$ over the same program space $\mathcal{P}_{\mathcal{G}}$ and specification $\mathcal{S}$, $\mathcal{C}_A$ is a **sound replacement** for $\mathcal{C}_B$ if it preserves the valid programs that pass the specification. Colloquially, all satisfying programs in $\mathcal{P}_{\mathcal{G}, \mathcal{C}_B}$ must also be present in $\mathcal{P}_{\mathcal{G}, \mathcal{C}_A}$,

$$\forall p \in \mathcal{P}_{\mathcal{G}}, \quad \mathcal{S}(p) = \top \wedge p \in \mathcal{P}_{\mathcal{G}, \mathcal{C}_B} \Rightarrow p \in \mathcal{P}_{\mathcal{G}, \mathcal{C}_A}$$

The definition above is relative: it does not say that $\mathcal{C}_A$ is sound in isolation, but that it is safe to use in place of $\mathcal{C}_B$. Yet, so long as we start with a provably sound constraint set, any successive sound replacement will also result in a sound constraint set. We formalize this notion below.

Proposition 7 (Soundness invariant). Let $\mathcal{C}_0, \mathcal{C}_1, \dots, \mathcal{C}_n$ be constraint sets over the same *unconstrained* program space $\mathcal{P}_{\mathcal{G}}$ and specification $\mathcal{S}$. Let $\mathcal{C}_0$ be the starting constraint set. If for all $i \in \{1, \dots, n\}$, $\mathcal{C}_i$ is a *sound replacement* for $\mathcal{C}_{i-1}$, then $\mathcal{C}_n$ is sound with respect to $\mathcal{S}$.

Proof. Let $p \in \mathcal{P}_{\mathcal{G}}$ be any satisfying program, so $\mathcal{S}(p) = \top$. We show by induction that p remains allowed after every replacement step, i.e.,

$$\forall i \in \{0, \dots, n\}, \quad p \in \mathcal{P}_{\mathcal{G}, \mathcal{C}_i}$$

Base Case. As per the definition of an unconstrained program space, $\mathcal{C}_0 = \emptyset$. An empty constraint set excludes no programs, therefore $p \in \mathcal{P}_{\mathcal{G}, \mathcal{C}_0}$.

Induction Step. Assume that $p \in \mathcal{P}_{\mathcal{G}, \mathcal{C}_{i-1}}$ for some $i \in \{1, \dots, n\}$. Since $\mathcal{C}_i$ is a *sound replacement* for $\mathcal{C}_{i-1}$, every satisfying program allowed by $\mathcal{C}_{i-1}$ is also allowed by $\mathcal{C}_i$. Because $\mathcal{S}(p) = \top$ and $p \in \mathcal{P}_{\mathcal{G}, \mathcal{C}_{i-1}}$, it follows that $p \in \mathcal{P}_{\mathcal{G}, \mathcal{C}_i}$.

By induction, every satisfying program remains allowed by $\mathcal{C}_n$. Therefore, $\mathcal{C}_n$ is sound with respect to $\mathcal{S}$. $\square$

We can also start from a search space that is constrained by a naive constraint set. By definition, each constraint ϕ_p within that set is a direct constraint derived from a conflict p , which cannot satisfy the specification. This makes the naive constraint set sound; the proof of the invariant is then analogous to the one above, with the

base case swapped.

More informally, running conflict analysis without any generalization or minimization attempts can be viewed as applying sound replacements to an unconstrained program space. Toggling generalization and/or minimization can then be viewed as applying sound replacements to the naive constraint set.

Definition 8 (Constraint Generalization). Given a program space $\mathcal{P}_G$, a *sound replacement* $\mathcal{C}_A$ for $\mathcal{C}_B$ is a **generalizing replacement** if it prunes strictly more programs, that is

$$\mathcal{P}_{G,\mathcal{C}_A} \subset \mathcal{P}_{G,\mathcal{C}_B}$$

Definition 9 (Constraint Minimization). Given a program space $\mathcal{P}_G$, a *sound replacement* $\mathcal{C}_A$ for $\mathcal{C}_B$ is a **minimizing replacement** if it uses fewer constraints, that is

$$|\mathcal{C}_A| < |\mathcal{C}_B|$$

We alternatively call the sound replacement $\mathcal{C}_A$ for $\mathcal{C}_B$ a **sound generalization** or **sound minimization** if it generalizes/minimizes $\mathcal{C}_B$ throughout the thesis.

Definition 10 (Pruning-preserving Minimization). Given a program space $\mathcal{P}_G$, a *minimizing replacement* $\mathcal{C}_A$ for $\mathcal{C}_B$ is **pruning-preserving** if it does not weaken pruning,

$$\mathcal{P}_{G,\mathcal{C}_A} \subseteq \mathcal{P}_{G,\mathcal{C}_B}$$

In this thesis, we mainly consider minimizations that are *pruning-preserving*. The main goal of minimization is to reduce the overhead of constraint management without giving up pruning benefits. As such, minimizations that have larger program spaces are of little interest. All minimizing replacements referred to later in the thesis are pruning-preserving, unless otherwise stated.

Definition 11 (The Conflict Generalization Problem). Given a conflict set $\mathcal{K}$ and its corresponding naive constraint set $\mathcal{C}_\mathcal{K}$, **find** a constraint set $\mathcal{C}_G$ such that $\mathcal{C}_G$ is a *sound generalization and/or pruning-preserving minimization* of $\mathcal{C}_\mathcal{K}$.

We assume the synthesis instance is *realizable*, i.e., there exists some $p \in \mathcal{P}_G$ such that $\mathcal{S}(p) = \top$. Otherwise, an all-encompassing forbid is a vacuously sound generalization and maximal minimization ($|\mathcal{C}| = 1 \wedge \mathcal{P}_{G,\mathcal{C}} = \emptyset$).

Chapter 5

Methods

Conflict analysis improves program synthesis by converting failed candidate programs into constraints that prune related candidates from the search space. Its effectiveness, however, depends on both the coverage of each learned constraint and the size of the resulting constraint set. A constraint that reproduces most of the failed candidate’s exact AST matches few structurally different candidates and therefore provides limited pruning. Repeatedly learning equivalent or overlapping constraints also increases the cost of storing, comparing, and enforcing them.

This chapter introduces two complementary extensions to the conflict analysis process. The first extends the information learned from an individual failure. Core-guided semantic constraints are generalized beyond the concrete candidate by identifying the parts of the program required to preserve the conflict. The second extension manages the constraints produced by conflict analysis before incorporating them into the search.

Together, these extensions address the two aspects of effectiveness: generalization increases the pruning coverage of the information learned from each failure, while minimization limits redundancy among constraints within each cleanup scope. We first describe how we augment the conflict analysis pipeline and then introduce these extensions in detail.

5.1 Extended Conflict-Analysis Pipeline

The original pipeline evaluates each enabled technique independently and routes its results by enforcement scope [Poeth, 2025]. Structural constraints are installed in the current solver, whereas grammar-level constraints are installed in the shared grammar. It does not compare constraints produced by different techniques or grammar-level constraints learned after different failures.

Distributing and cleaning analysis results. We insert a **cleanup stage** between analysis and installation, as shown in Figure 5.1. The same configurable procedure processes both scopes, but with different state lifetimes. This placement keeps cleanup independent of the technique that produced a constraint and allows results from different techniques to be compared.

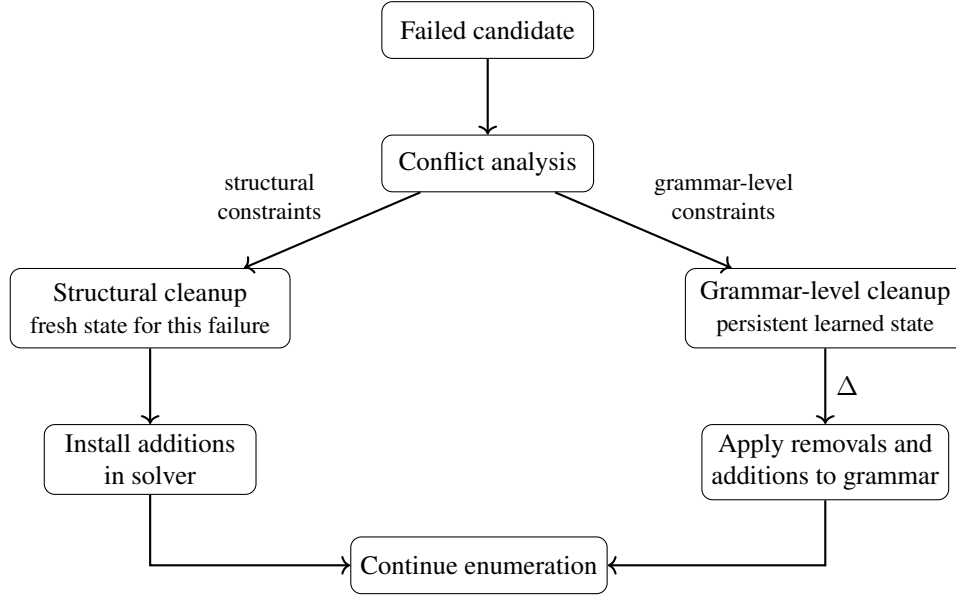


Figure 5.1: Extended conflict-analysis pipeline. Structural cleanup is local to one failure; grammar-level cleanup persists across failures and returns additions and removals. The persistent state excludes constraints supplied with the original grammar.

Structural constraints are learned for the particular solver and structural search context in which a failed candidate was produced. Their meaning depends on the represented AST shape. Persisting them in a shared cleanup state could therefore cause constraints from distinct structural contexts to be compared or combined. This will often result in unsound constraints for the given context. Moreover, structural constraints are installed through an addition-only solver interface, so constraints removed from a persistent cleanup state could not be removed from the solver. Structural constraints are consequently minimized in a fresh state for each pipeline invocation.

Grammar-level constraints are independent of a particular solver state and are interpreted over the shared grammar. Their cleanup state can persist across the whole synthesis run. Cleanup returns changes to this state as a $\Delta = (\Delta^+, \Delta^-)$: new constraints in Δ^+ are installed, while learned constraints in Δ^- are removed. Constraints supplied with the original grammar do not enter the cleanup state and thus cannot be removed.

Integrating the pipeline into enumeration. The synthesis loop creates the enabled techniques and persistent grammar state once per synthesis run. Algorithm 1 summarizes the failure-processing path that follows. The core pipeline of analysis and routing remains the same. We run a cleanup step on the two categories of constraints and apply the returned delta to the grammar constraints. Structural additions are posted to every active solver representation, including the uniform solver when used by the enumerator. Then the search continues with the updated constraints. Enumeration and program evaluation are omitted from the pseudocode for brevity.

Algorithm 1 Failure processing in the extended conflict-analysis pipeline.

Require: failed candidate p , enabled techniques T , persistent grammar state S_G

- 1: $C \leftarrow \text{ANALYZEALL}(T, p)$
 - 2: $(C_S, C_G) \leftarrow \text{PARTITIONBYScope}(C)$
 - 3: $\Delta_S \leftarrow \text{MINIMIZE}(\emptyset, C_S)$
 - 4: $\text{ADDCONSTRAINTSTOSOLVER}(\Delta_S^+)$
 - 5: $\Delta_G \leftarrow \text{MINIMIZE}(S_G, C_G)$
 - 6: $\text{REMOVEGRAMMAR}(\Delta_G^-)$
 - 7: $\text{ADDGRAMMAR}(\Delta_G^+)$
-

Generalization can either be introduced as a secondary stage that precedes cleanup, or be tightly integrated into a specific technique. This ordering lets cleanup treat generalized and regular constraints uniformly. Given that our generalization is specific to MUC, it is constructed during analysis. Ultimately, it is important that generalization is independent of enumeration and constraint-set management.

5.2 Core-Guided Generalization of Semantic Conflicts

The existing MUC technique constructs a learned constraint using the failed candidate as its structural basis. Rules associated with the unsatisfiable core may be replaced by their equivalent-modulo-conflict classes, but branches outside the core retain the shape of the candidate [Poeth, 2025]. The resulting constraint can therefore be substantially more specific than the semantic conflict it represents.

5.2.1 Wildcarding Non-Core Branches

Consider a schematic list-processing grammar with $\text{mapPlus} : \text{Int} \times \text{List} \rightarrow \text{List}$ that takes an integer offset followed by a list, and the integer constant 10 is available. Consider the input $x = [5, 15, 10, 0, 4]$ and desired output $y = 20$, together with the failed candidate $\text{sum}(\text{mapPlus}(10, \text{sort}(x)))$. Suppose the unsatisfiable

core identifies a conflict in the outer semantic relations, in this case `sum` and `mapPlus`, and the selected relation at `mapPlus` does not refer to its list-valued second input x_2 . The existing MUC constraint nevertheless retains the complete inner expression `sort( $x$ )`. It therefore does not match candidates such as

$$\begin{aligned} &\text{sum}(\text{mapPlus}(10, \text{reverse}(x))), \\ &\text{sum}(\text{mapPlus}(10, \text{take}(x, \dots))), \\ &\text{sum}(\text{mapPlus}(10, x)), \end{aligned}$$

even when they reproduce the same semantic conflict. The learned constraint remains tied to the expansion selected in the failed candidate rather than expressing only the structure needed for the conflict.

The central question is therefore which parts of a failed candidate must remain fixed for its semantic conflict to remain valid. A first intuition is that branches outside the unsatisfiable core can be replaced by wildcards. Assuming that the selected core relations do not refer to the value produced by the `sort( $x$ )` branch, this would produce a pattern such as `sum(mapPlus(10,  $w_1$ ))`, where w_1 may match different list-producing subtrees. Because this abstraction requires no additional SMT queries, both the original local constraint and an additional generalized constraint can be constructed from the same core.

The original local constraint is retained and installed in the current solver. The generalized pattern preserves the candidate root, replaces the irrelevant branches with wildcards, and is installed in the grammar. Each abstracted branch receives a distinct wildcard so that different branches may match unrelated subtrees without imposing an unintended equality requirement.

5.2.2 Preserving Semantic Support

Core membership alone does not determine whether a branch can be abstracted. A semantic relation in the core may refer to the concrete value produced by a child even when that child contains no core relation itself. Consider `isZero(sub(10, len( $x$ )))`, with semantic annotations

$$\text{sub} : y = x_1 - x_2, \quad \text{isZero} : y = (x_1 = 0).$$

If the relation for `sub` occurs in the core, then the value of its second child is required through x_2 . Replacing `len( $x$ )` by a wildcard would yield `isZero(sub(10,  $w_1$ ))`. For a specification requiring the result to be true, this pattern would also forbid the valid program `isZero(sub(10, 10))`.

The procedure must therefore preserve two kinds of structure: nodes represented directly in the unsatisfiable core and branches whose values support core relations through references to x_i . In other words, if a selected relation at node j mentions

x_i , then the value produced by its i th child participates in the core’s semantics indirectly. The branch rooted at that child must consequently be retained. A reference to y identifies the value produced by the current core node and creates no additional child dependency. The path connecting that node to the candidate root is retained as core context.

5.2.3 Preserving the Candidate-Root Context

Wildcarding unrelated branches determines which parts of the candidate may vary, but it does not determine the context in which the resulting pattern should be applied. A pattern containing only the core nodes would discard the enclosing structure that made those nodes relevant. When reused elsewhere, such an unanchored pattern could match the same rules beneath a different sequence of enclosing operations. The construction therefore starts at the failed candidate’s root and retains the concrete connector nodes on every path from that root to the core. Sibling branches that provide neither core nodes nor semantic support may still be replaced by wildcards.

Two integration strategies can make the generalized pattern available beyond the solver in which it was discovered:

1. Construct a candidate-root-preserving grammar constraint. The grammar then makes the pattern available to subsequent search states without requiring conflict analysis to manage those states individually.
2. Propagate the pattern to every queued search state whose root structure is compatible with the anchor. This keeps installation solver-local but requires access to the enumeration queue and explicit bookkeeping for existing and future states.

Both strategies were prototyped and produced little practical difference. We retain the first strategy because it integrates with the existing grammar-constraint mechanism, automatically applies to future search states, and avoids coupling conflict analysis to the enumeration strategy.

5.2.4 Generalization Algorithm

The preceding design decisions give the generalization procedure four preservation rules. It retains the original local MUC constraint; anchors the generalized pattern at the root and preserves paths to core nodes; preserves non-core branches referenced by core semantics; and applies the existing EMC replacement at core nodes. Every remaining branch is replaced by a uniquely identified wildcard.

In the running example, the procedure starts at the outer sum node and follows the branch containing the core mapPlus node. At mapPlus, the constant 10 is

retained because its value is referenced by the relation. The $\text{sort}(x)$ branch is not referenced, so it becomes a wildcard. These recursive decisions produce the generalized pattern $\text{sum}(\text{mapPlus}(10, w_1))$ described above. Algorithm 2 formalizes this traversal.

Algorithm 2 Core-guided generalization of a local MUC constraint.

Require: candidate tree p , grammar G , core map M
Ensure: a generalized pattern or NULL

```

1: procedure GENERALIZEMUC( $p, G, M$ )
2:   if  $M = \emptyset$  then
3:     return NULL
4:   end if
5:    $N_M \leftarrow \text{RESOLVECORENODES}(p, G, \text{Keys}(M))$ 
6:   if  $N_M = \text{NULL}$  then
7:     return NULL
8:   end if
9:    $N_S \leftarrow \emptyset$ 
10:  for  $v \in N_M$  do
11:    for  $\rho \in \text{RELATIONSAT}(M, v)$  do
12:       $V_\rho \leftarrow \text{SEMANTICVARIABLES}(\rho)$ 
13:      for each child variable  $x_i \in V_\rho$  do
14:         $N_S \leftarrow N_S \cup \{\text{CHILD}(v, i)\}$ 
15:      end for
16:    end for
17:  end for
18:  return  $\text{BUILDPATTERN}(\text{Root}(p), G, M, N_M, N_S)$ 
19: end procedure
20: procedure BUILDPATTERN( $v, G, M, N_M, N_S$ )
21:   $K \leftarrow []$ 
22:  for  $c \in \text{CHILDREN}(v)$  do
23:    if  $\text{SUBTREE}(c)$  contains a node in  $N_M$  then
24:       $c' \leftarrow \text{BUILDPATTERN}(c, G, M, N_M, N_S)$ 
25:    else if  $c \in N_S$  then
26:       $c' \leftarrow \text{BUILDLOCALMUC}(c, G, M)$ 
27:    else
28:       $c' \leftarrow \text{FRESHWILDCARD}$ 
29:    end if
30:     $K \leftarrow K \parallel [c']$ 
31:  end for
32:  if  $v \in N_M$  then
33:     $D \leftarrow \text{EQUIVALENTRULES}(v, G, \text{RelationsAt}(M, v))$ 
34:    return  $\text{NODEFROMRULES}(v, D, K)$ 
35:  end if
36:  return  $\text{NODEFROMRULES}(v, \text{Rule}(v), K)$ 
37: end procedure

```

Implementation. The input to generalization is the core map M produced by the existing MUC analysis. Each key identifies a position in the failed candidate, and its value contains the semantic relations selected for the core at that position. RESOLVECORENODES translates these keys into AST nodes N_M . The implementation performs this translation through maximum-arity AST indices. No generaliz-

ation is done if M ends up empty. The procedure next derives the semantic-support set N_S . For every selected relation ρ at a core node v , SEMANTICVARIABLES collects the variables encountered into V_ρ . A variable x_i denotes the value produced by the i th child of v . The variable y denotes the output of v itself. It creates no additional child dependency and is already in N_M . Algorithm 2 then uses N_M and N_S to construct the generalized pattern.

At a core node, EQUIVALENTRULES applies the existing EMC extraction done in the local version of MUC. A non-core node reached by the recursion is a connector on a root-to-core path and therefore retains its concrete rule. Each call to FRESH-WILDCARD creates a distinct identifier, preventing two abstracted branches from being required to match the same subtree.

The result is wrapped in a grammar-level `Forbidden` constraint and returned alongside the original solver-local MUC constraint. An empty or unmapped core yields only the local constraint. The method can increase pruning by removing irrelevant structure, but its precision is limited by the core and semantic annotations. This generalized constraint is then sent to the cleanup stage.

5.3 Constraint-Set Minimization

Conflict analysis can repeatedly encounter the same or closely related failures. Installing every resulting constraint preserves pruning power, but it also leaves the solver or grammar with constraints that may be identical, covered by a more general constraint, or representable more compactly. Constraint-set minimization processes each newly learned constraint relative to the applicable cleanup state before installation. The objective is not to infer new semantic conflicts, but to retain the pruning represented by the learned constraints while reducing redundant matching work and storage.

Cleanup driver. Algorithm 3 gives the complete cleanup control flow. It is presented first as a roadmap for the individual techniques. Exact deduplication rejects a repeated constraint. Subsumption rejects a constraint whose matches are already covered. Single-difference merging factors two patterns that differ at one rule position. Finally, cross-relation combination groups complete child-pattern alternatives. Candidates are processed sequentially. Transformations that may construct a replacement constraint run before duplicate detection and subsumption. The procedure returns a delta consisting of two collections. One collection is the constraints to add, the other is the previously learned ones to remove. In the pseudocode, CONSTRAINTSETDELTA is the data structure that stores this pair. The delta reports the net difference between the initial and final state. Thus, a constraint added and then replaced during the same call is not reported.

Algorithm 3 Constraint-cleanup driver.

Require: mutable state S , incoming constraints C , cleanup options

Ensure: the net additions and removals relative to the initial state

```
1:  $added \leftarrow []$ ;  $removed \leftarrow []$ 
2: for  $c \in C$  do
3:   if single-difference merging is enabled then
4:      $(c, R) \leftarrow \text{MERGEINTOSTATE}(S, c)$ 
5:      $\text{RECORDREMOVALS}(R, added, removed)$ 
6:   end if
7:   if cross-relation combination is enabled then
8:      $(c, R) \leftarrow \text{COMBINEINTOSTATE}(S, c)$ 
9:      $\text{RECORDREMOVALS}(R, added, removed)$ 
10:  end if
11:  if single-difference merging is enabled and  $c$  is a combination then
12:     $(c, changed) \leftarrow \text{MERGECOMPATIBLETUPLES TOFIXEDPOINT}(c)$ 
13:    if  $changed$  then
14:       $(c, R) \leftarrow \text{MERGEINTOSTATE}(S, c)$ 
15:       $\text{RECORDREMOVALS}(R, added, removed)$ 
16:    end if
17:  end if
18:  if deduplication is enabled and  $\text{ISDUPLICATE}(S, c)$  then
19:    continue
20:  end if
21:  if subsumption is enabled then
22:    if  $\exists e \in S : \text{CONSTRAINTSUBSUMES}(e, c)$  then
23:      continue
24:    end if
25:     $R \leftarrow \{e \in S \mid \text{CONSTRAINTSUBSUMES}(c, e)\}$ 
26:     $\text{REMOVE}(S, R)$ 
27:     $\text{RECORDREMOVALS}(R, added, removed)$ 
28:  end if
29:   $\text{INSERT}(S, c)$ 
30:   $added \leftarrow added \parallel [c]$ 
31: end for
32: return  $\text{CONSTRAINTSETDELTA}(added, removed)$ 
```

The order is intentional. Merging and cross-relation combination can change the candidate and remove the constraints from which it was formed. When both transformations are enabled, we further iteratively simplify the combination if possible. Ordinary merging is retried at the end, in case the combination collapses to a singular `Forbidden`. Duplicate detection inspects the final transformed candidate. Subsumption runs last because it may reject that candidate or remove additional constraints covered by it.

5.3.1 Exact Deduplication

The simplest redundancy is a constraint structurally identical to one already retained. It adds no additional pruning and can be discarded. The implementation supports both a linear scan, and hash-indexed lookup. We mostly focus on the latter, as it is more efficient because it avoids scanning the full state.

Hash-based lookup stores constraints in buckets indexed by a structural hash. Concrete nodes contribute their rule and ordered child hashes. Domain nodes additionally contribute their rule domain. Finally, wildcards contribute their identifiers. A matching hash is not proof of equality: we compare exhaustively against all constraints within the selected bucket. The method-level lookup is therefore:

Algorithm 4 Hash-based exact duplicate detection.

Require: grammar state S , candidate constraint c

Ensure: whether c is a duplicate of a retained constraint

```

1: procedure ISDUPLICATE( $S, c$ )
2:    $h \leftarrow \text{STRUCTURALHASH}(c)$ 
3:   if  $h \notin S$  then
4:     return false
5:   end if
6:   for  $e \in S[h]$  do
7:     if  $\text{STRUCTURALLYEQUAL}(e, c)$  then
8:       return true
9:     end if
10:  end for
11:  return false
12: end procedure

```

5.3.2 Subsumption

Exact deduplication cannot remove a structurally different constraint already covered by a more general one. Colloquially, a general constraint is preferred over each of its specific interpretations, as it *subsumes* each one: if every program forbidden by constraint c_s is also forbidden by c_g , retaining c_s provides no additional pruning.

Let $\mathcal{D}_R$ denote a domain node containing the rules in R . Figure 5.2 illustrates progressively more general versions of the same pattern. Each constraint subsumes every constraint to its right. A wildcard subsumes any subtree at its position, while a domain subsumes concrete rules or narrower domains contained within it. For example, the leftmost pattern subsumes the next one because its wildcard accepts the complete `sort/reverse` subtree. Also, its integer domain contains the narrower domain. Moving further right replaces domains with smaller domains or concrete rules.

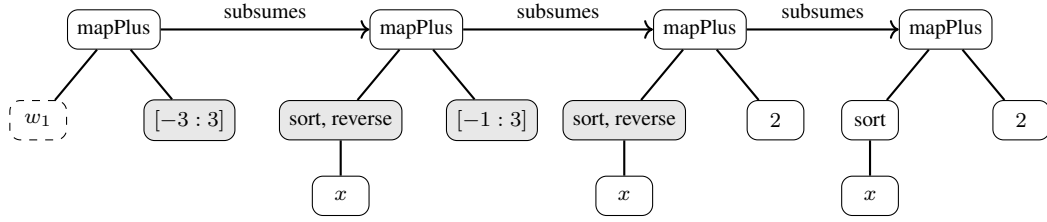


Figure 5.2: Specificity ordering between four forbidden AST patterns. Domain nodes are shaded and the wildcard is dashed. Moving left replaces concrete choices with larger domains or a wildcard and therefore matches a superset of the programs to its right. Each displayed AST is wrapped in a `Forbidden` constraint.

The recursive comparison starts at the roots of the original pattern a (potentially more general) and a candidate b (potentially more specific):

- If the two subtrees are structurally identical, then a subsumes b at this position.
- If a is a wildcard, it subsumes any subtree rooted at b . If b is a wildcard, then a cannot cover it.
- Otherwise, both nodes carry grammar rules. The rules allowed by a must contain all rules allowed by b . This covers equal nodes, and a wider domain containing a narrower one or a concrete rule.
- Finally, the nodes must have the same number of children, and the procedure recursively applies these cases to every pair of corresponding children. Pattern a subsumes b only if every comparison succeeds.

The wildcard case assumes independent wildcards. Generalized MUC satisfies this assumption by assigning a fresh identifier to every abstracted branch.

For ordinary `Forbidden` constraints, this comparison is applied to their AST patterns. The cleanup decision is bidirectional: an incoming constraint is rejected if an existing constraint subsumes it, while existing constraints subsumed by an accepted incoming constraint are removed. The extension to tuple-preserving constraints is described in section 5.3.4.

Algorithm 5 Recursive tree subsumption.

```
1: procedure CONSTRAINTSUBSUMES( $a, b$ )
2:   if STRUCTURALLYEQUAL( $a, b$ ) then
3:     return true
4:   end if
5:   if  $a$  is a wildcard then
6:     return true
7:   end if
8:   if  $b$  is a wildcard then
9:     return false
10:  end if
11:  if ALLOWEDRULES( $b$ )  $\not\subseteq$  ALLOWEDRULES( $a$ ) then
12:    return false
13:  end if
14:  if  $|children(a)| \neq |children(b)|$  then
15:    return false
16:  end if
17:  for  $(a_i, b_i)$  in corresponding child pairs do
18:    if  $\neg$ CONSTRAINTSUBSUMES( $a_i, b_i$ ) then
19:      return false
20:    end if
21:  end for
22:  return true
23: end procedure
```

Subsumption removes only constraints whose forbidden program set is already covered. It does not construct a new pattern and therefore cannot introduce combinations that were not represented by an existing learned constraint.

5.3.3 Single-Difference Merging

Subsumption cannot simplify two incomparable constraints even when they differ at only one AST position. Repeated failures may, for example, produce one constraint containing sort and another containing reverse in the same position. Neither constraint covers the other, but retaining both repeats the surrounding structure and requires two pattern checks.

Single-difference merging replaces the two alternatives at that position with a domain node, as shown in fig. 5.3. The merged constraint represents exactly the union of the two original patterns because every other AST position is identical.

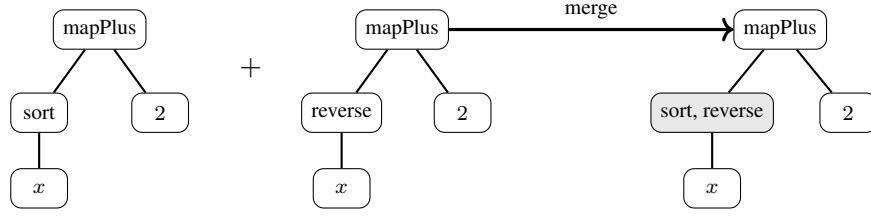


Figure 5.3: Single-difference merging of two forbidden AST patterns. The only differing position contains `sort` or `reverse`; the merged AST replaces that position with a domain node. Each AST is wrapped in a `Forbidden` constraint.

Algorithm 6 Merging trees that differ at exactly one AST position.

Require: forbidden constraints c_1 and c_2

Ensure: an exact-union replacement or FAILURE

```

1: procedure MERGEONEDIFFERENCE( $c_1, c_2$ )
2:   if either constraint is not an ordinary Forbidden then
3:     return FAILURE
4:   end if
5:    $(a, b) \leftarrow (\text{PATTERN}(c_1), \text{PATTERN}(c_2))$ 
6:   if  $a$  and  $b$  do not have compatible shapes and wildcards then
7:     return FAILURE
8:   end if
9:    $D \leftarrow \{q \mid q \text{ is rule-bearing and } \text{ALLOWEDRULES}(a, q) \neq \text{ALLOWEDRULES}(b, q)\}$ 
10:  if  $|D| \neq 1$  then
11:    return FAILURE
12:  end if
13:   $q \leftarrow$  the sole position in  $D$ 
14:   $t \leftarrow \text{COPY}(a)$ 
15:   $\text{SETALLOWEDRULES}(t, q, \text{AllowedRules}(a, q) \cup \text{AllowedRules}(b, q))$ 
16:  return FORBIDDEN( $t$ )
17: end procedure

```

Allowing more than one differing position would be unsafe. Merging $r(A, B)$ and $r(C, D)$ into $r(\{A, C\}, \{B, D\})$ would also forbid the unseen combinations $r(A, D)$ and $r(C, B)$. Single-difference merging therefore recursively counts differing AST positions and succeeds only when the final count is exactly one. At the differing position, concrete rules are converted to a domain and existing domains are combined by set union. Trees with different arities or non-identical wildcards are incompatible. When comparing two retained constraints, their conflict-constraint type and installation scope must also agree.

The candidate is repeatedly compared with retained constraints. Whenever a merge succeeds, the matched retained constraint is removed and the merged result becomes the new candidate. This permits a domain to accumulate several alternatives at the same position while preserving the one-difference condition for every merge step.

5.3.4 Cross-Relation Combination

The restriction to one differing position prevents unjustified Cartesian products, but it also leaves constraints with the same root and several differing children separate. Suppose the observed forbidden trees are $r(A, B)$ and $r(C, D)$. Generalizing their children independently would also forbid $r(A, D)$ and $r(C, B)$. Figure 5.4 contrasts this Cartesian product with the tuple-preserving representation.

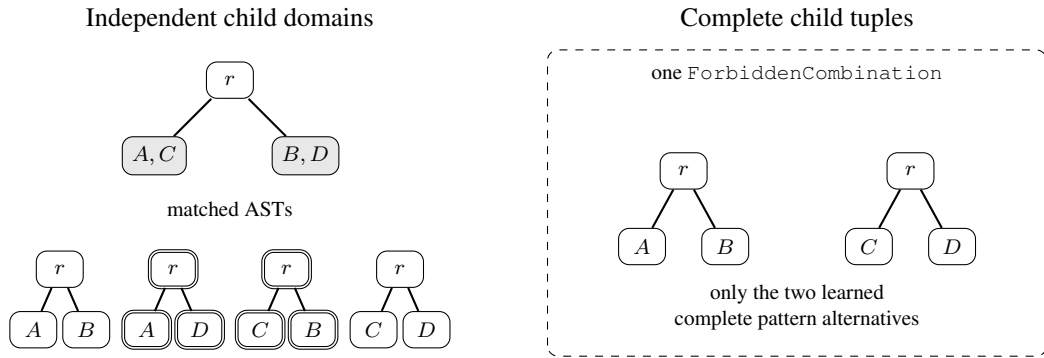


Figure 5.4: Independent child domains match four ASTs, including the unlearned cross combinations shown with double borders. `ForbiddenCombination` instead groups the two complete learned child-pattern tuples without introducing their cross-product.

To represent this disjunction, we introduce `ForbiddenCombination`. It stores a shared concrete or domain root and a set of complete child-pattern tuples. A subtree is forbidden only when its root and all children match one stored tuple. Inputs must be ordinary forbidden patterns or existing combinations with the same conflict-constraint type, installation scope, nonleaf root domain, and tuple arity. Combination itself takes the set union of complete tuples, never independent domains across positions.

Algorithm 7 Tuple-preserving cross-relation combination.

Require: compatible conflict constraints c_1 and c_2

Ensure: a combined constraint or FAILURE

```
1: procedure COMBINECROSSRELATIONS( $c_1, c_2$ )
2:   if  $c_1$  and  $c_2$  have incompatible types or scopes then
3:     return FAILURE
4:   end if
5:    $(r_1, T_1) \leftarrow \text{ROOTANDTUPLES}(c_1)$ 
6:    $(r_2, T_2) \leftarrow \text{ROOTANDTUPLES}(c_2)$ 
7:   if either decomposition fails then
8:     return FAILURE
9:   end if
10:  if  $\text{ROOTDOMAIN}(r_1) \neq \text{ROOTDOMAIN}(r_2)$  then
11:    return FAILURE
12:  end if
13:  if the tuples do not have one shared arity then
14:    return FAILURE
15:  end if
16:   $T \leftarrow T_1 \cup T_2$ 
17:  if  $T = T_1$  then
18:    return FAILURE
19:  end if
20:  return FORBIDDENCOMBINATION( $r_1, T$ )
21: end procedure
```

The combined result is repeatedly compared with the state so all compatible tuples under one root can be collected. Equality treats the tuple set as unordered. For subsumption, every complete pattern on the more specific side must be covered by some complete pattern on the more general side.

Interaction with single-difference merging. This new constraint can be further simplified if single-difference merging is also enabled. For one, we can merge some children pairs. For example,

$$\{(A, B), (A, D)\} \longrightarrow \{(A, \{B, D\})\}.$$

This is done repeatedly up to a fixed point. If after cleanup only one tuple remains, it can be converted back to an ordinary `Forbidden`. Conversely, a complete rectangle such as $(A, B), (A, D), (C, B), (C, D)$ can become one pattern with domains $\{A, C\}$ and $\{B, D\}$. An incomplete rectangle, on the other hand, remains a tuple-preserving combination.

Together, the four techniques address different kinds of redundancy. Deduplication removes exact repetitions, subsumption removes constraints whose pruning is already covered, single-difference merging factors alternatives, and cross-relation combines groups that share a root. Their sequential integration in Algorithm 3 ensures that each later comparison observes the representation produced by the earlier transformation.

Chapter 6

Experimental Evaluation

This chapter evaluates the two extensions introduced in Chapter 5: core-guided generalization of MUC conflicts and minimization of the learned constraint set. We assess their effectiveness based on search-space reduction for generalization, constraint-set behavior for cleanup, and whether either improvement translates into lower end-to-end synthesis time. This is formalized in the research questions below,

- *RQ1: Can discovered conflicts be soundly generalized to prune a larger part of the search space?*

The primary comparison is between the original local MUC constraint and the core-guided grammar-level generalization constructed from the same unsatisfiable core. We investigate whether the additional generalized constraint reduces the number of enumerated programs and the time required for synthesis, and how often an MUC check yields a usable core from which a generalized constraint can be constructed. In terms of soundness, the experiments provide an empirical check over some of the problems for each benchmark, by recording whether learned constraints exclude a known solution (called a *witness*), when available.

- *RQ2: Do conflict-set minimization techniques improve synthesis performance compared with retaining all learned constraints?*

We evaluate exact deduplication, subsumption, single-difference merging, and cross-relation combination separately and in combination. Constraint count explains whether the techniques produce a more compact representation. Runtime determines whether that representation benefits synthesis. A trace-based analysis also records when most rejections and removals occur as the search progresses.

- *RQ3: Does generalization, combined with minimization, allow for solving more problems in fewer enumerations?*

We compare the number of solved problems and the enumerations required for each solution. Plain enumeration uses no conflict analysis, base conflict analysis uses local MUC, and improved conflict analysis combines generalized MUC with the strongest measured cleanup configuration.

6.1 Experimental Setup

The implementation is written in Julia and extends the Herb synthesis framework. Conflict analysis is implemented in `ConflictAnalysis.jl`; the experiment drivers, instrumentation, and aggregation code are contained in `ConflictAnalysisExperiments.jl`. Programs are enumerated in increasing AST size with Herb’s breadth-first iterator. For each candidate, the synthesizer executes the available conflict-analysis techniques after an unsuccessful evaluation, minimizes the newly produced constraints according to the selected configuration, and installs the returned additions in the grammar or solvers. MUC extraction uses CVC5 as the SMT solver in all primary comparisons.

All experiments are run on the DelftBlue supercomputer. `memory` nodes were used to accommodate the high memory requirements for recording constraint traces in long runs. Each such node has 2 Intel Xeon Gold 6248Rs, with 24 cores @ 3.00GHz each, and a total node RAM of 750GB. Each experiment is split across many array workers - each gets 1 core and sufficient amount of RAM.

The experiments use three programming-by-example benchmarks with different value domains. The complete grammars and their semantic annotations will be included in Appendix A.

- **DeepCoder** contains functional programs over integer lists and scalar values [Feng et al., 2018]. The benchmark contains 100 tasks with five input-output examples each. Task inputs are added as typed terminals to a shared DSL. Its rules include operations such as mapping, filtering, scanning, and aggregation. The relatively rich semantic annotations over list length, extrema, and boundary values make this benchmark particularly useful for studying MUC-based generalization.
- **SyGuS PBE-SLIA** consists of string-manipulation tasks collected from the 2019 SyGuS PBE-SLIA benchmark set [Padhi et al., 2019]. Tasks have individual grammars with string, integer, and Boolean operations such as concatenation, substring extraction, replacement, and index search. Because these grammars do not originally contain semantic annotations required by MUC and SeAn, we also supply our own function-level semantics.
- **Morpheus** contains example-driven table-transformation tasks derived from the PLDI 2017 benchmark suite [Feng et al., 2017]. The local implementation contains 72 tasks, each with one example over an input table and a target

table. Programs compose operations such as gathering, spreading, filtering, grouping, and mutating columns. Each task specializes in a shared grammar with its specific input-output parameters, including available, required, or temporary columns, as well as literals observed in the example. This benchmark is especially interesting for us - while the previous two benchmarks are propagation/constraint-heavy, this one is evaluation-heavy (by far the most expensive benchmark to evaluate a program in).

All paired comparisons use the same benchmark task, maximum AST size, resource limits, conflict-analysis techniques, and enumeration strategy. The resource limits are defined in Table 6.1. Only the feature named by the ablation is changed. A run terminates when a satisfying program is found, the search space up to the maximum size is exhausted, or the time- or enumeration limit is reached.

Runs that find a solution or exhaust the bounded search are treated as *completed*. Completion filtering is used for the RQ1 enumeration-reduction statistic as described below.

Table 6.1: Experimental parameters and executed workloads.

Parameter	DeepCoder	SyGuS	Morpheus
Maximum AST size	7	8	8
Timeout per run	$300\text{ (rq1)} / 120\text{ (rq2)}$		
Enumeration limit	10^8		
Tasks per benchmark	100	200	72

6.1.1 Measurements and Aggregation

The primary search-space measure is the number of evaluated candidate programs. For RQ1, let I_c denote the tasks completed by configuration c at a given benchmark and size bound. Enumeration reduction relative to plain enumeration is computed as a ratio of sums,

$$R_{\text{enum}}(c) = 1 - \frac{\sum_{i \in I_c} N_{c,i}}{\sum_{i \in I_c} N_{\text{plain},i}}, \quad (6.1)$$

where $N_{c,i}$ and $N_{\text{plain},i}$ are the enumeration counts for task i . Completion filtering is performed independently for each configuration. Local and generalized MUC curves are therefore not restricted to a common completed-task intersection.

The field recorded by the experiment as `ConstraintsAdded` is a hybrid quantity. Let Q denote the cumulative number of accepted solver-local structural additions during the run plus the final active grammar-level cleanup population. This is due to the distinct installation tactics for structural and grammar constraints. It

is not the total number of add operations or the complete number of constraints stored by the grammar and solvers. We therefore refer to Q as the *reported constraint measure*. Reducing Q indicates a more compact reported representation, but its two components have different lifetimes.

For RQ2, the plots use the largest measured AST size for each benchmark. Cleanup has the greatest opportunity to matter at these bounds, as learned constraints have had the longest time to accumulate redundancy. We retain all raw runs at the selected bound, including capped runs, and compare each cleanup configuration with no cleanup over the same task. The individual ablation reports the hybrid measure Q , as we are interested to see how cleanup interacts with solver-level structural constraints. Afterwards, we only look at the grammar constraints for the combined trade-off. Let $A_{c,i}$ denote the mean active grammar-level population in the final observed progress interval of task i . The plotted measures are then,

$$R_Q(c) = 1 - \frac{\sum_i Q_{c,i}}{\sum_i Q_{0,i}}, \quad R_A(c) = 1 - \frac{\sum_i A_{c,i}}{\sum_i A_{0,i}}, \quad \Delta_T(c) = \frac{\sum_i T_{c,i}}{\sum_i T_{0,i}} - 1. \quad (6.2)$$

The sums range over the same raw tasks at the selected size for each conflict-analysis configuration. The principal timer begins after construction of the grammar, iterator, interpreter, and symbol table. Technique construction, solver startup and shutdown, and cleanup are included in conflict-analysis time. We measure witness checks separately and subtract them from total synthesis time.

For the behavioral analysis, we divide each run into 20 equal intervals based on observed enumeration progress. RQ1 dynamics pool observations across all measured AST sizes. RQ2 only considers the largest size bound. Within each interval, we record incoming and accepted constraints, and rejections/installations from each cleanup strategy.

6.2 RQ1: Core-Guided MUC Generalization

Local MUC analysis retains much of the failed candidate’s AST structure. Core-guided generalization wildcards those unrelated branches while preserving the candidate root, paths to core nodes, semantic support, and equivalent-modulo-conflict domains. The central question is whether this additional coverage reduces the evaluated search space enough to offset the cost of propagating another grammar-level constraint.

The primary ablation compares MUC with generalization disabled (*local MUC*) and enabled (*generalized MUC*). Both configurations retain the original local constraint and use the same MUC backend. We run both individually and with additions like ERA or SeAn. The latter tests whether the effect remains visible when

several conflict sources prune overlapping regions of the search space.

6.2.1 Effect on Search, Runtime, and Constraints

Results. Figure 6.1 color codes all technique configurations, while comparing local to generalized MUC. On DeepCoder, generalized MUC reaches approximately 99.86% enumeration reduction at size 7. This holds for all technique sets, indicating that SeAn and ERA do not provide additional pruning for generalized MUC on this benchmark. The corresponding local reductions are 23.17% for MUC, 31.09% for ERA+MUC, and 58.02% for SeAn+MUC.

The DeepCoder separation is also visible in the raw-run runtime means: generalized configurations take about 167 seconds at size 7, compared with 265 seconds for their local counterparts. The reported constraint measure is less straightforward. It increases slightly for MUC and ERA+MUC, but decreases slightly for SeAn+MUC. Generalization changes which candidates fail. More importantly, it affects which solver-local additions are emitted and which grammar-level constraints remain active. Stronger pruning can therefore accompany either a larger or a smaller value of the hybrid measure. Nonetheless, the generalized MUC’s hybrid measure does not deviate significantly from the local one across all technique sets.

On SyGuS at size 8, generalized MUC shows a similarly large improvement. Enumerations are reduced by 47.30%, 58.97%, and 45.47% for MUC, ERA+MUC, and SeAn+MUC, respectively. More importantly, local MUC is ineffective in this benchmark without ERA. The local counterparts reduce the search space by 0.01%, 28.40%, and 3.25%. The raw runtime means are around 210 seconds. The constraint measure is substantially lower for all generalized MUC runs, being around 4 times lower.

On Morpheus, generalized MUC reduces enumeration by 63.93% at size 7 for every technique set and lowers mean runtime from approximately 38 seconds to 13 seconds. At size 8, the reductions go to 69.32% for MUC, 74.54% for ERA+MUC, and 69.79% for SeAn+MUC. Local MUC equivalents are 32.93%, 35.86%, and 33.20%, respectively - about half of the reduction. This benchmark proves to be more difficult, as even the generalized MUC configurations often time out at size 8.

Discussion. These results answer the effectiveness part of RQ1 overwhelmingly positively. For all benchmarks, generalized MUC is immensely more powerful at pruning than its local equivalent. In the worst case (Morpheus) it only provides double the search space reduction, while in other cases it can make MUC finally effective (SyGuS), or even reduce the search space to almost a thousandth of its original size (DeepCoder).

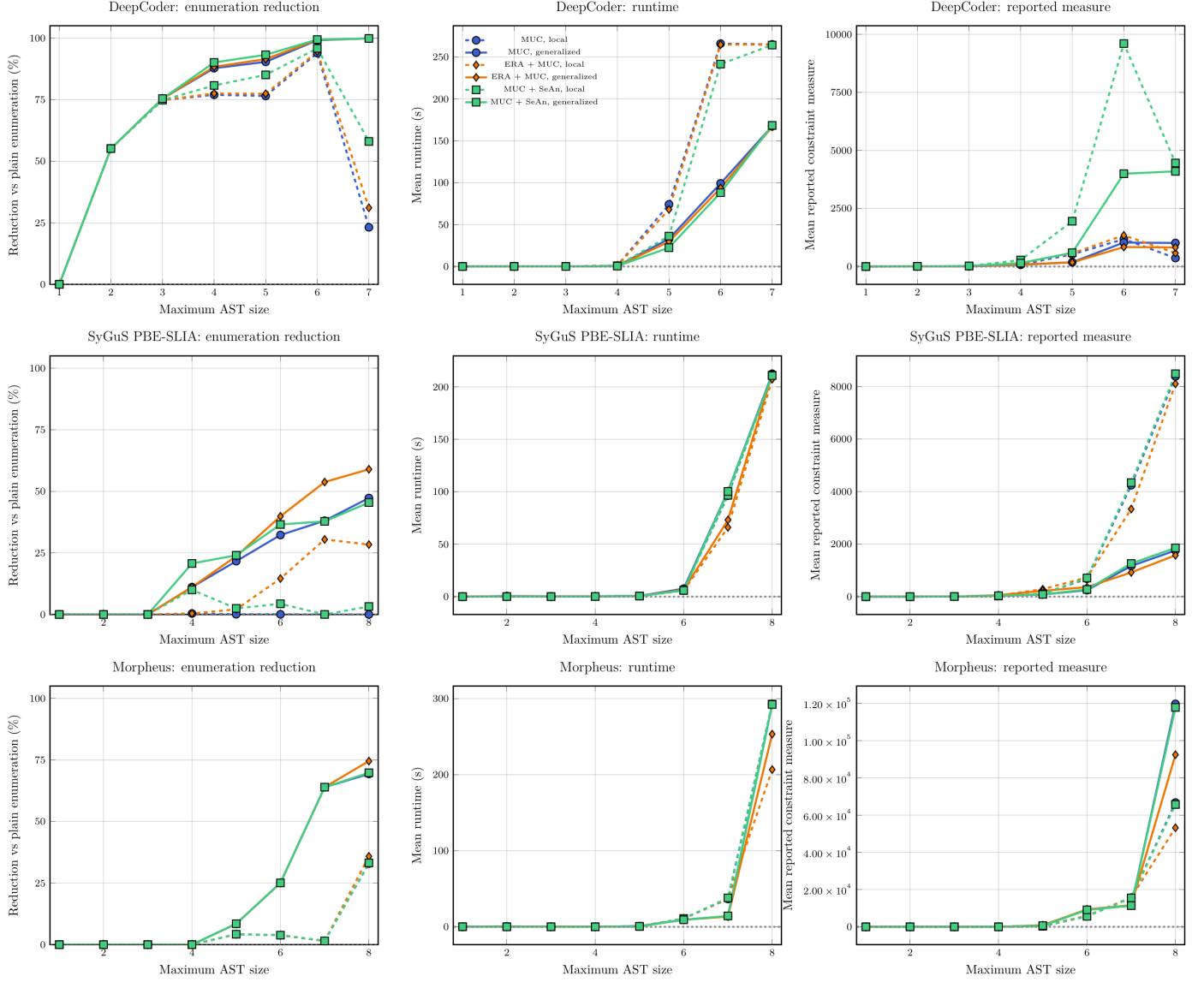


Figure 6.1: Local MUC (dashed) and generalized MUC (solid) by benchmark, size, and conflict-analysis technique sets. Blue circles denote MUC, orange diamonds ERA+MUC, and green squares SeAn+MUC. Enumeration reduction uses independently completed workloads. Runtime and the reported constraint measure use all raw runs.

Furthermore, generalized MUC may theoretically be even more powerful than the experiments indicate. Runs that reach the timeout are excluded from the local enumeration-reduction statistic. Extending the timeout could admit difficult tasks on which local MUC provides weaker pruning. Given local MUC's weaker pruning, this affects local MUC much more than the general one. Therefore, the reported local reductions are much more optimistic than the generalized variant's.

As to the constraint measure, one could naively hypothesize that generalized MUC should always store more constraints. After all, the generalization retains the local MUC as well; it is an additive construct. However, generalized MUC also exhausts the potential search space much quicker, especially if the general constraints are much more descriptive. All three benchmarks investigate the balance of these two factors: DeepCoder achieves its reductions in much less time, with approximately the same number of added constraints; SyGuS does it with much fewer constraints in about the same time. Morpheus demonstrates that its evaluation-heavy task complexity favors MUC generalization much less - the increase in pruning is somewhat counteracted by the additional runtime and constraint measure. In summary, runtime improves when the avoided work changes whether a run can progress before the limit; otherwise, solver work and constraint management can dominate the wall-clock result.

Regarding the technique sets specifically, you may have noticed that the blue MUC-only line is mostly absent. That is because it almost always overlaps completely with the green SeAn+MUC line. This indicates that once MUC is generalized, it rarely misses programs that SeAn would usually prune. On the other hand, ERA complements MUC nicely for general forbidden sub-patterns, especially for SyGuS.

6.2.2 Generalization During Search

Results. Figure 6.2 separates the availability of generalizable conflicts from the accumulated constraints. On DeepCoder, all three stable-cohort curves begin with a mean per-run core yield near 15% and finish at around 40%. The reconstructed mean populations grow monotonically from approximately 7/8 to 130-160 constraints.

SyGuS begins near 23% for all three technique sets, peaks around 30% early in the search, and finishes near 12%. The reconstructed populations grow monotonically from approximately 10 to approximately 200 constraints. Useful explanations are more frequent early, but generalized constraints continue to accumulate throughout the run, at a progressively lower pace.

Morpheus differs sharply from the other domains. Core yield starts near 96%,

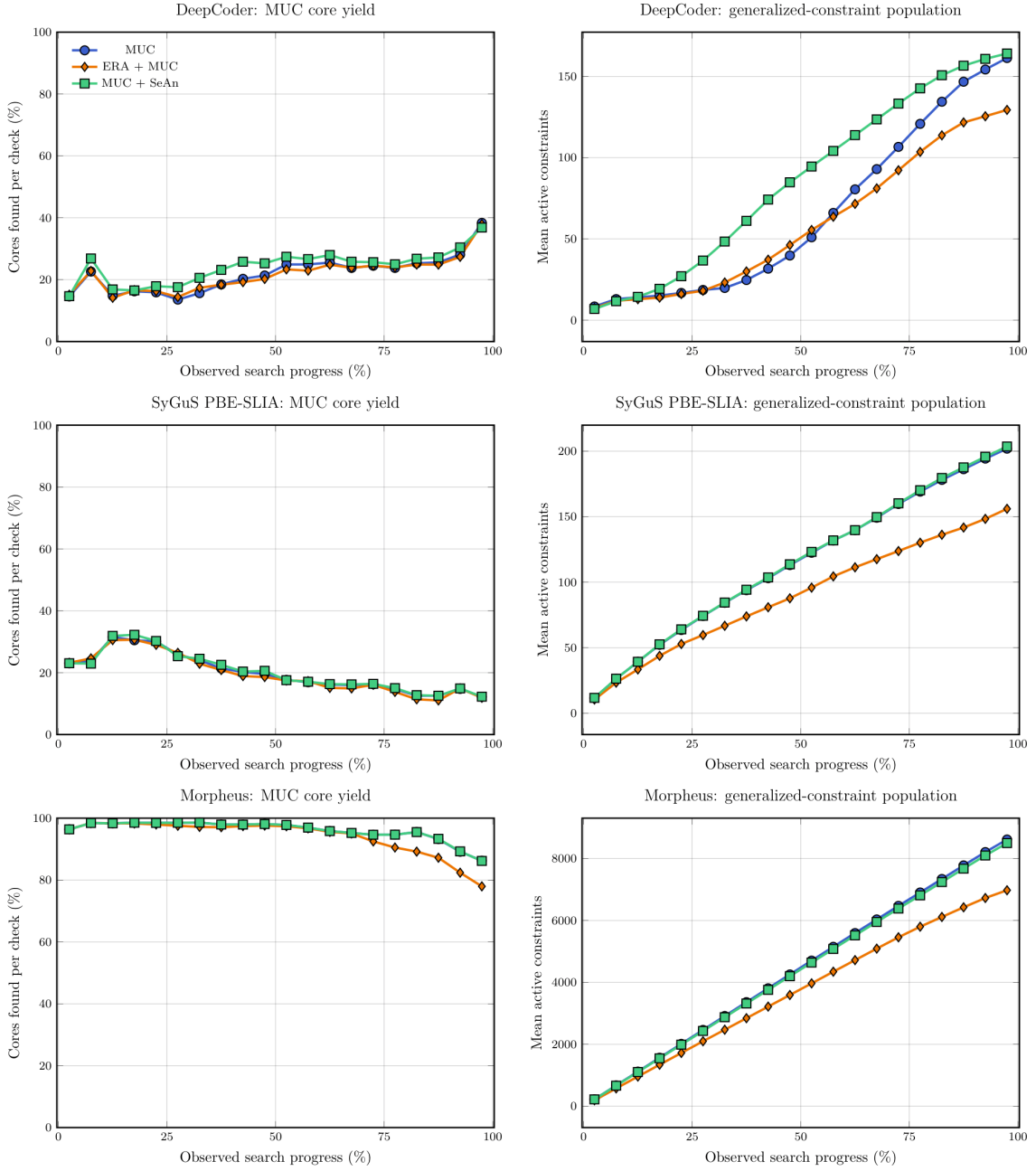


Figure 6.2: RQ1 MUC core yield and active generalized-constraint population over normalized search progress, pooled across all measured AST sizes. Core yield is the percentage of MUC checks returning a usable core and uses a stable cohort observed in all 20 intervals. Population uses all raw runs with no-cleanup state carried forward and counts active generalized grammar constraints; it is distinct from the hybrid reported constraint measure Q . Blue circles denote MUC, orange diamonds ERA+MUC, and green squares SeAn+MUC.

reaches approximately 99% in earlier and middle intervals, and finishes at 86.27% for MUC, 77.96% for ERA+MUC, and 86.18% for SeAn+MUC. Correspondingly, the reconstructed mean population grows monotonically from around 200 to 7000-8600. This explains the significantly higher constraint count for this benchmark: most MUC checks yield a generalized constraint, so the unminimized population grows by orders of magnitude.

Discussion. One would expect that generalization is head-heavy - generalizing constraints at the beginning of the search should be most useful. However, we observe all three distinct behaviors, depending on the benchmark. Morpheus obtains a core from almost every MUC check through most of the search, SyGuS obtains its highest yield early, and DeepCoder continues finding cores later. A high core yield is therefore neither necessary nor sufficient for a proportional runtime improvement.

If anything, the results suggest an inverse correlation between core yield and extra pruning from generalization. Morpheus yields almost all of the MUC cores it checks, yet it showed the smallest improvement out of the three benchmarks (both in pruning power and constraint count). Generalization attempts are simply not as beneficial, and thus the active population continues to grow monotonically and linearly throughout the search. Meanwhile, DeepCoder’s active constraint population is dictated by the core yield. Notice how the active population grows the slowest at the end of the search, especially for SeAn+MUC and ERA+MUC, when the core-yield is at its highest.

6.2.3 Usefulness of Semantic Annotations

Our semantic instrumentation tracks annotation occurrences asserted in MUC checks and whether those occurrences are selected in returned cores. For annotation a , the *core-inclusion rate* is

$$I(a) = \frac{\text{occurrences of } a \text{ selected in returned cores}}{\text{occurrences of } a \text{ asserted in MUC checks}}. \quad (6.3)$$

The instrumentation field named `Constraints` equals the `Core` field for every recorded annotation row, so it does not measure survival through cleanup and is not interpreted as such. To clarify, we only track the *occurrence* of an annotation within a core - in the case it is not minimal, it does not imply necessity.

For a descriptive comparison, we divide the annotations into exact equalities, coarse domain or range bounds, and other predicates or relations. Coarse bounds include generic facts such as non-negative length or a scalar lower bound near zero. This post-hoc grouping distinguishes annotations that precisely connect values or structures from annotations that primarily admit values. Table 6.2 reports MUC-only

results over all sizes.

For example, $\text{first}(y) = \text{minimum}(x_1)$ (for `sort`) is an exact equality because it fixes a particular output property to an input property. The annotation $\text{length}(x_1) \geq 0$ is a domain bound. Finally, $\text{length}(x_1) \geq \text{length}(y)$ (for `filter`) belongs to the other-predicate-or-relation family - it relates input and output properties without fixing them.

Table 6.2: Core inclusion by semantic-annotation family. The last column is each family’s share of all annotation occurrences selected in generalized-MUC cores.

Benchmark	Annotation family	Local	Generalized	Gen. core share
DeepCoder	Exact equality	6.22%	5.09%	99.03%
DeepCoder	Domain/range bound	0.03%	0.05%	0.06%
DeepCoder	Other predicate/relation	0.13%	0.07%	0.91%
SyGuS	Exact equality	20.27%	5.49%	69.66%
SyGuS	Domain/range bound	0.01%	0.02%	0.25%
SyGuS	Other predicate/relation	2.12%	1.52%	30.09%
Morpheus	Exact equality	97.64%	96.94%	65.06%
Morpheus	Domain/range bound	0.00%	0.00%	0.00%
Morpheus	Other predicate/relation	20.11%	21.36%	34.94%

The pattern is clear: exact equalities dominate the generalized cores in all three domains. They account for 99.03% of selected DeepCoder annotation occurrences, 69.66% on SyGuS, and 65.06% on Morpheus. In contrast, the coarse-bound family contributes only 0.25% of SyGuS core occurrences, almost none in DeepCoder, and none on Morpheus. Other predicates and relations remain important when they express domain-specific structural information: they account for 30.09% of SyGuS and 34.94% of Morpheus core occurrences. We list some of the strongest individual generalized-MUC annotations in Table 6.3.

The strongest DeepCoder relations constrain extrema, while the Morpheus relations constrain table dimensions and grouping structure. These annotations rule out concrete undesirable structural or content relations. Generic admissibility facts - like enforcing a non-negative length - rarely help. The reason is twofold: many grammars implicitly handle these cases, and then they would fail due to another, more powerful semantic; also, ERA can capture such errors, if used. The SyGuS `isdigit_cvc( $x_1$ )` result also shows that annotation expressiveness is not a decisive factor - unary annotations can do just as well. Overall, annotation specificity is the most important factor.

Table 6.3: Annotations with the highest generalized-MUC core-inclusion rate in each benchmark. Counts are core occurrences over asserted occurrences.

Benchmark	Semantic relation	Referenced operations	Core/asserted	Inclusion
DeepCoder	$\text{first}(y) = \text{minimum}(x_1)$	sort, scanl_min, min	90 421/211 282	42.80%
DeepCoder	$\text{last}(y) = \text{maximum}(x_1)$	sort, scanl_max, max	90 426/294 372	30.72%
DeepCoder	$\text{minimum}(y) = \text{minimum}(x_1)$	minimum, sort, reverse	92 755/504 106	18.40%
SyGuS	$\text{length}(y) = 1$	at_cvc; char processing	45 347/309 430	14.66%
SyGuS	$\text{isdigit_cvc}(x_1)/\text{isdigit_cvc}(y)$	int_to_str, str_to_int	30 914/291 942	10.59%
SyGuS	$\text{length}(y) = (x_3 - x_2) + 1$	substr_cvc	37 866/480 807	7.88%
Morpheus	$\text{colCount}(y) = \text{colCount}(x_1) - 1$	unite (combine columns)	422 400/422 400	100.00%
Morpheus	$\text{rowCount}(y) = \text{rowCount}(x_1)$	unite/separate (combine/split columns)	1 520 468/1 556 520	97.68%

6.3 RQ2: Constraint-Set Minimization

Now that we have established generalized MUC’s effectiveness, RQ2 fixes it as enabled, and we start looking into the cleanup options. We use the same three conflict-analysis technique sets. In this section, we abbreviate hash-based exact deduplication as D , subsumption as S , single-difference merging as M , and cross-relation combination as C . The baseline NONE retains all learned constraints.

The cleanup driver generated subsets containing at most three of the four options, so we did not measure the full $DSMC$ pipeline. After all, subsumption fully contains deduplication, so having both enabled is redundant. We collapse repeated records of the same effective execution by task, size, conflict-analysis technique sets, and effective configuration. For verbosity, the remaining combined configurations are DC , DM , DMC , MC , SC , SM , and SMC .

6.3.1 Individual Techniques

Results. Figure 6.3 shows that every individual technique reduces the reported constraint measure at the largest bound, but at differing magnitudes. For MUC and ERA+MUC on DeepCoder, D and S reduce the measure by 3.72%/5.11% and 6.1%/8.2%, while M reaches 42-43.74% and C - about 50%. The ordering changes significantly for SeAn+MUC. There, deduplication is more effective (20.53%) than cross-relation (16.47%), and subsumption dominates (40.64%). Merging reaches 32.62%. In summary, cleanup effectiveness is mostly determined by the benchmark, set of techniques and their overlap/synergy, rather than by the techniques in isolation.

On SyGuS, reductions range from 27.63% to 57.02%. Cross-relation combination is strongest for all three technique sets, reaching 49.02% for MUC, 57.02% for

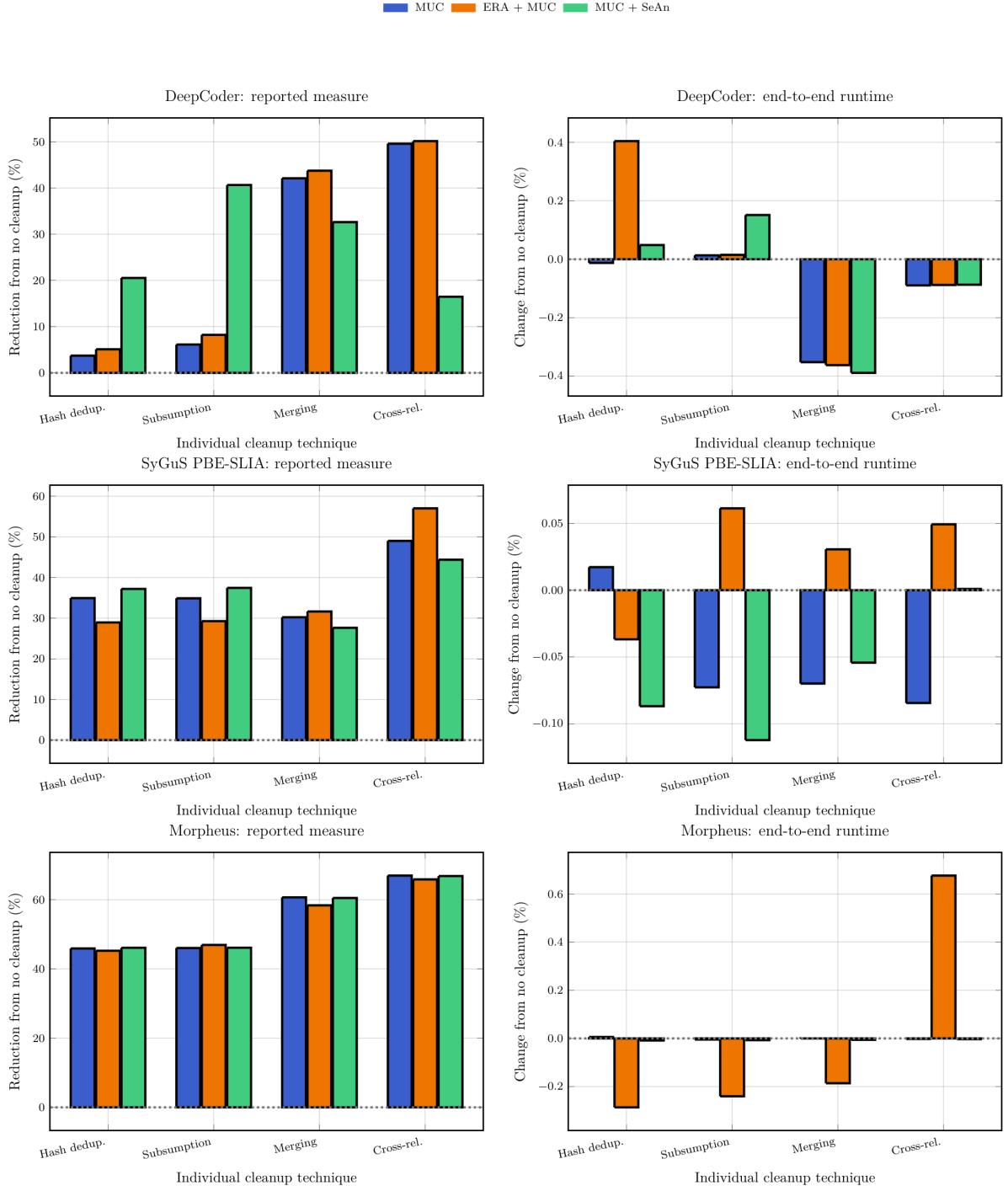


Figure 6.3: Individual cleanup ablation at the largest measured size. Bars show the reduction in the reported constraint measure and the end-to-end runtime change relative to generalized MUC with no cleanup. The shared legend identifies the conflict-analysis technique sets.

ERA+MUC, and 44.38% for SeAn+MUC. Deduplication and subsumption each remove approximately 30-38%, while merging removes 28-32%. The narrower spread suggests that several forms of redundancy coexist in SyGuS.

Morpheus produces the largest and most consistent individual reductions. Deduplication removes around 46%, subsumption 47%, merging 60%, and cross-relation combination 67%. These values stay relatively consistent across technique sets. This agrees with the RQ1 trace. Morpheus had the largest active constraint population, and minimization works best with a larger base of related constraints.

The runtime panels show a neutral result. Across every individual technique set and benchmark, the largest observed change lies between a 0.39% decrease and a 0.68% increase. These sub-percent differences do not establish either a speedup or a meaningful overhead.

Discussion. The ordering of the cleanup techniques follows the amount of structural freedom each one can exploit. Deduplication is the narrowest operation that can only remove identicals. Subsumption is strictly broader since it can also remove non-identical subsumed constraints. As an empirical soundness check, we can see that S is performing strictly better to D . The experiments thus show that learned states contain meaningful containment relations beyond exact equality.

Cross-relation combinations prove to be the most effective form of minimization, essentially halving the constraint count in our experiments. This is not surprising given its workings. The experiments consolidate its generality and effectiveness across diverse benchmarks. Less intuitively, single-difference merging also performs well. Its strong DeepCoder and Morpheus results indicate that failed candidates often recur with the same surrounding structure and only a few alternatives changed. If they share a semantic similarity, they would already be collapsed by the generalized MUC, but if not, then they are minimized by merging.

The chain of effectiveness is consistent for all benchmarks and technique sets, except for MUC+SeAn on DeepCoder. Given our results in RQ1, we already know that SeAn provides little- to no extra pruning over generalized MUC. Especially for DeepCoder, it seems to be the case that SeAn simply finds structurally exact alternatives to generalized MUC, which deduplication and subsumption can already get rid of. This leaves fewer same-root multi-child families for cross-relations and merging to minimize.

The experiment shows no clear end-to-end advantage for running single-feature cleanup, regardless of the benchmark or technique set. It shows that, under the current budget and implementation, the time saved from checking a smaller set

of constraints is balanced by the cleanup overhead. This is especially plausible for Morpheus, where candidate evaluation is much more expensive than grammar-level constraint management. Lastly, notice that minimization’s benefit lands inversely to generalization: DeepCoder, the benchmark that benefited the most from MUC generalization, benefits the least from minimization, and vice-versa for Morpheus.

6.3.2 Combined Cleanup

Results. Figure 6.4 replaces the hybrid reported measure with the retained grammar-level state. This makes the combined configurations directly comparable in terms of what cleanup leaves active. We can also assess how enabling multiple features improves upon single features on the persisted grammar-level constraints.

Across DeepCoder and SyGuS, configurations containing cross-relation combination are performing visibly better than those without it. Adding merging on top of cross-relation combination, as in *MC*, *DMC*, and *SMC*, further reduces the retained population. When referring to the raw data, without cleanup, the active population reaches approximately 317 constraints on DeepCoder, 473 on SyGuS, and 23,085 on Morpheus. The strongest configuration *SMC* reduces these averages to below 40 for DeepCoder, and below 20 for the other two benchmarks. Thus, the strongest combined cleanup retains only a small fraction of the unminimized grammar state; on Morpheus, the reduction is several orders of magnitude.

Runtime still provides no reliable ranking. While most configurations now point to a decrease in runtime, the change is still not significant. The useful result is therefore the separation between state size and wall-clock time: combinations can almost eliminate the retained grammar population without producing a measurable end-to-end speedup in this experiment.

Discussion. The difference between *DM/SM* and the *C*-containing combinations further demonstrates the broad applicability and effectiveness of cross-relation combination. Merging nevertheless remains an important complement. After cross-relation combination collects several tuples, we can continuously merge within each child tuple in the combination. Compatible tuples can collapse into domains, and a complete tuple rectangle can eventually become a single ordinary forbidden pattern. The strongest configurations that contain both are thus not just additively better when looking at *M* and *C* in isolation.

Once *M* and *C* have transformed the incoming candidate, *D* or *S* decides whether the resulting constraint still needs to be retained. This justifies their placement in the cleanup driver shown in Chapter 5. Its marginal contribution can nevertheless become small in *SMC*. If merging and cross-relation combination have already

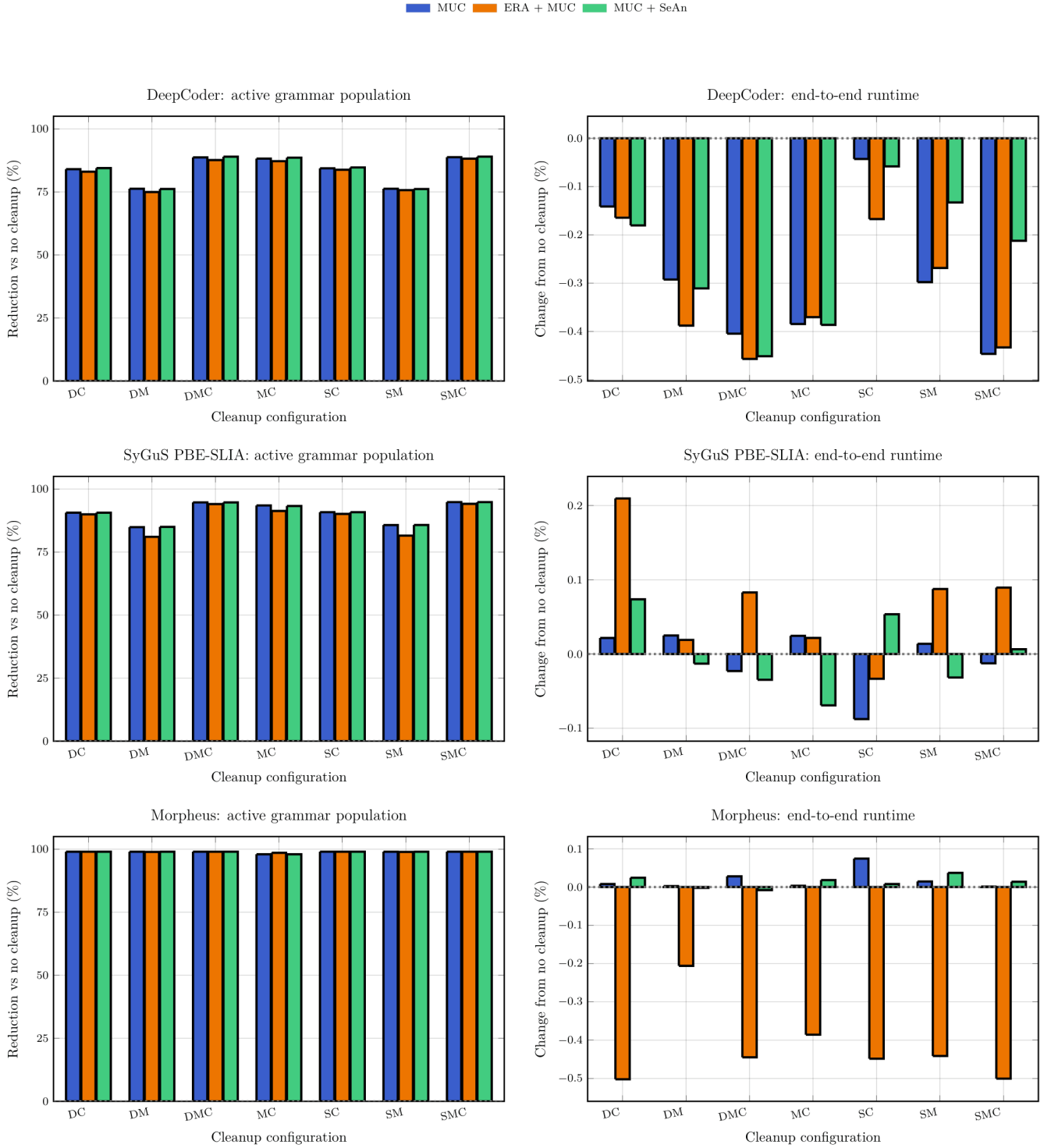


Figure 6.4: Combined cleanup configurations at the largest measured size. D , S , M , and C denote hash deduplication, subsumption, merging, and cross-relation combination. The shared legend identifies the conflict-analysis technique sets.

compressed a family into one representative, little redundant state remains to remove by subsumption. The combined cleanups already minimize the constraint base significantly, which is why deduplication and subsumption seem to have little to no effect here, compared with the last experiment.

Yet, the benchmarks tell a lot about the cleanup behavior. In the case of Morpheus, the benchmark with the most redundancy, cleanup is not only the most effective but also uniform across all cleanup configurations. Furthermore, across all benchmarks, there is little to no difference between technique sets.

6.3.3 Constraint-Set Development During Search

Results. Without cleanup, the mean active grammar-level population in Figure 6.5 grows from approximately 24 to 317 on DeepCoder, from 46 to 473 on SyGuS, and from 600 to 23,085 on Morpheus. The left color scale is logarithmic so that everything is visible. The right heatmaps normalize cleanup actions by conflict events rather than plotting absolute counts, as this better reflects how often a cleanup feature is triggered. We calculate means by 100 events, but keep in mind that it can exceed 100 because one conflict may reject or remove several constraints.

On DeepCoder, individual deduplication and subsumption finish with populations of 294 and 285, respectively. Merging and cross-relation finish closer to 100, while *SMC* finishes below 40. Deduplication and subsumption act most frequently near the beginning. Merging and cross-relation configurations remain active through the middle and later search intervals.

SyGuS cleanup is more front-loaded. The no-cleanup population grows to approximately 473, while *D* and *S* finish near 179 and 177, *M* near 199, and *C* near 80. The *SMC* population finishes near 20. Its activity peaks at 26.67 actions per 100 conflicts in the first interval and falls to 5.29 at the end.

Morpheus has the most intense cleanup activity. Without cleanup, its active population reaches approximately 23,085. Deduplication and subsumption finish near 100, merging near 4212, and cross-relation combination near 3303, while *SMC* finishes below 20 as well. Given how often deduplication and subsumption trigger, they perform close to 1 action per conflict throughout the run.

Discussion. The traces show that cleanup is not a one-time correction. SyGuS exposes much of its reusable redundancy early, after which the cleanup rate declines. DeepCoder continues to generate mergeable and cross-relatable structures well into the middle and later search. Particularly, around 60-ish% into the run,

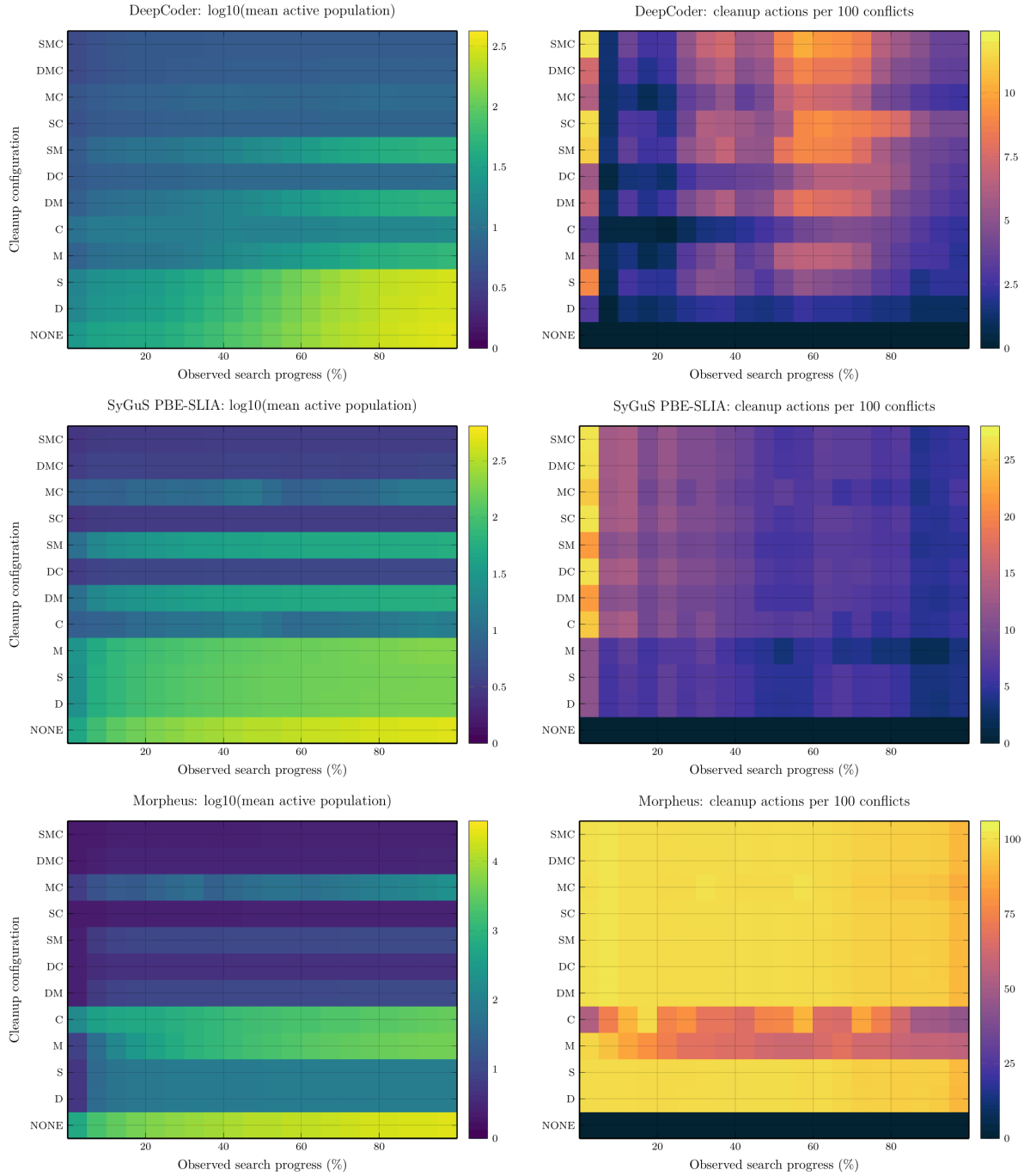


Figure 6.5: Cleanup dynamics at the largest measured size, aggregated across conflict-analysis technique sets. The left heatmaps show $\log_{10}(\text{mean active grammar population})$; the right heatmaps show cleanup actions per 100 conflict events.

DeepCoder experiences a hefty cleanup. Morpheus requires sustained cleanup almost throughout the whole run, as it keeps generating similar conflicts.

The Morpheus trace clarifies the distinction between the individual Q ablation and the active-population measurements. Deduplication and subsumption leave far fewer grammar constraints active than one would expect from the first experiment, especially for Morpheus. Yet, M and C reduce Q more strongly. In short, D and S are extremely effective at preventing repeated grammar entries, whereas M and C also change how structurally related constraints are represented and installed. The combined pipeline benefits from both effects.

RQ2 gives a positive answer for learned-state minimization but not for end-to-end speed. The experiments consistently show that the benchmarks generate significant redundancy. Cross-relation combination is the most broadly useful individual transformation, and its interaction with merging makes MC particularly strong. This is especially true when structural constraints are also taken into account. The exact ranking still depends on the benchmark, because each can generate different conflict sources. If the goal is a minimal constraint representation over pipeline simplicity, running costlier configurations is therefore worthwhile, as the overhead is minimal.

6.4 RQ3: Solution Coverage and Enumeration Effort

Figure 6.6 and table 6.4 present the solution coverage, and the enumerations for each scenario. The solution coverage is complete - all problems that the baseline solves are also solved by both plain- and improved conflict analysis. Morpheus is omitted as no problems were solved by any scenario. For problems solved by the baseline, the figure is normalized based on the plain’s enumeration count. A value below one therefore indicates fewer enumerations. Green-only bars represent the additional problems solved only by improved conflict analysis.

Table 6.4: Scenario enumeration totals. The common cohort contains exactly the problems solved by plain enumeration and base CA. The all-solved total additionally includes the improved-only solutions.

Benchmark	Method	Solved	Common-cohort enums.	All-solved enums.
DeepCoder	Plain enumeration	11	10,473,600	10,473,600
DeepCoder	Base CA (local MUC)	11	5,614,870	5,614,870
DeepCoder	Improved CA (generalized MUC + SMC)	15	4,314,869	9,797,928
SyGuS PBE-SLIA	Plain enumeration	24	312,361	312,361
SyGuS PBE-SLIA	Base CA (local MUC)	24	274,567	274,567
SyGuS PBE-SLIA	Improved CA (generalized MUC + SMC)	26	167,112	353,475
Morpheus	All configurations	0	–	–

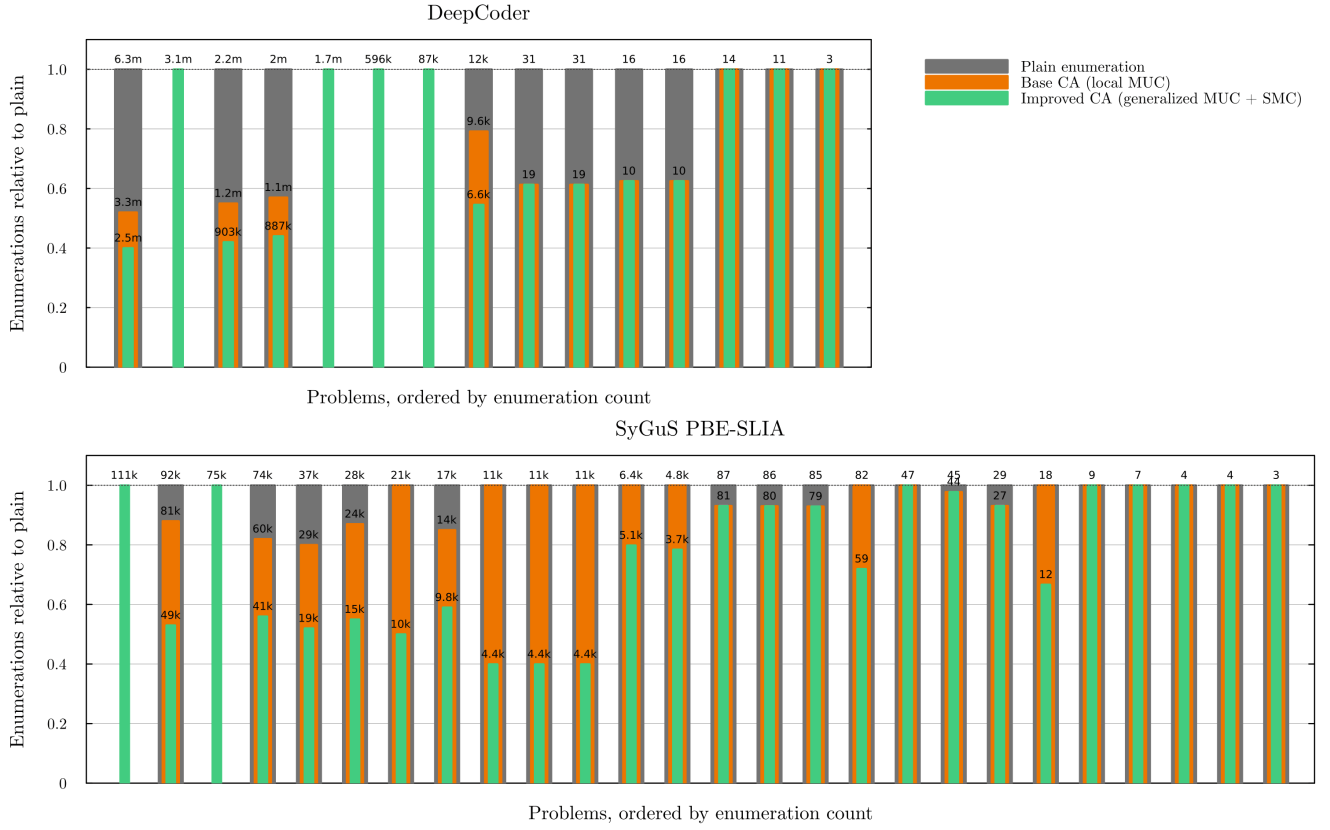


Figure 6.6: Gray bars show plain enumeration, narrower orange bars show base conflict analysis with local MUC, and narrow green bars show generalized MUC with *SMC*. On shared problems, bar height is the number of enumerations relative to plain enumeration - lower is better. Labels give absolute enumeration counts using *k* and *m* abbreviations. Problem names are omitted, and all tasks are ordered by enumeration count.

Results. On DeepCoder’s common 11-task cohort, base conflict analysis reduces enumerations from 10.47 million to 5.61 million. Improved conflict analysis reduces them further to 4.31 million. Improved CA also solves four additional tasks, for a total of 15. Its all-solved total is 9.80 million.

SyGuS has the same ordering. On its common 24-task cohort, plain enumeration needs 312,361 evaluations. Base and improved CA need 274,567 and 167,112 respectively. Improved CA then solves two additional tasks, reaching 26. Its all-solved total is 353,475. Morpheus provides neither coverage nor enumeration totals because every compared configuration remains unsolved.

Discussion. The overall ordering is consistent. Improved CA outperforms base CA, which outperforms plain enumeration. On the common cohorts, each successive configuration requires fewer enumerations. Improved CA also expands solution coverage on DeepCoder and SyGuS. We can conclude that generalization and minimization are effective, reducing enumerations and increasing the attainable problem set modestly.

A big consideration here is that not all problems are even theoretically solvable given our strict max size limits. For instance, Morpheus has a fixed solution format of three sequential table operations. Given that most of these take 2-3 parameters, sometimes even more, a solution cannot be represented by fewer than 12 or so nodes. As such, it is fully expected that no instances were solved. Although improved CA reduces enumeration enough to increase the attainable size by a few steps, that margin is insufficient (for most problems in DeepCoder and SyGuS, and *for all* problems in Morpheus).

A natural follow-up is to consider higher max sizes, but then the runtime profile becomes an issue. The timing audit in Table 6.5 shows that the MUC conflict-analysis pipeline dominates the wall-clock cost of MUC runs. Across the three benchmarks, it accounts for 95+% of recorded runtime. Most of that time is the SMT model construction, encoding, `check-sat` and unsatisfiable-core retrieval. The results therefore support the conclusion that the MUC/SMT pipeline dominates runtime.

Table 6.5: Aggregate timing of MUC-enabled RQ2 runs. Conflict-analysis time is the share of the recorded end-to-end runtime accumulated in `ConflictAnalysisNS`.

Benchmark	CA time (s)	Total time (s)	CA share
DeepCoder	1,390,278	1,429,701	97.2%
SyGuS PBE-SLIA	1,814,970	1,833,114	99.0%
Morpheus	1,412,824	1,594,557	88.6%

RQ3 gives a qualified positive answer. The improved configuration preserves lower enumeration effort on shared tasks and extends solution coverage a few problems for DeepCoder and SyGuS. Most of the solutions, however, remain untenable given our size constraints. Increasing the time budget, or optimizing the technique, is an essential next step to recover more solutions.

Chapter 7

Conclusions, Future Work, and Reflection

7.1 Conclusions

This thesis investigated how conflict analysis in program synthesis can learn constraints that are both more general and more manageable. Enumeration creates an enormous search space, so our goal was to learn general constraints, with the least redundancy possible. Learning only narrow constraints can replace repeated candidate evaluation with a large collection of similar lessons. To address this goal, the conflict-analysis pipeline was extended in two complementary ways. Core-guided MUC generalization uses the semantic explanation of a failure to preserve only the relevant structure of a candidate. Constraint-set cleanup then removes or combines redundant learned constraints.

The evaluation shows that core-guided generalization makes learned conflicts substantially more effective. Its effectiveness depends on the semantics available to explain a failure - precise equalities and domain-specific relationships are more informative than broad admissibility properties. This suggests that semantic annotations should be selected for the specific distinctions we aim to have, rather than greater numbers. The empirical witness checks did not reveal incorrect pruning, although a formal soundness argument remains outside the scope of this thesis.

The cleanup experiments further show that learned constraint sets contain considerable redundancy. No single cleanup operation is best in every setting - different options target different kinds of redundancy across benchmarks. More expressive operations that combine related constraint patterns are generally valuable. Simpler deduplication and subsumption remain useful when techniques repeatedly discover very similar or the same knowledge. However, a smaller learned state did not reliably lead to faster synthesis under the current implementation and budgets.

These findings provide a qualified positive answer to the main research question. Conflict analysis can learn broader constraints by generalizing from semantic explanations, while cleanup can keep the resulting knowledge manageable. The combined evaluation suggests lower search effort on already solved tasks and may extend the solution horizon a bit more. However, the overhead of the broader conflict-analysis pipeline remains the main practical limitation.

7.2 Future Work

7.2.1 A Specialized Semantic Reasoner

Profiling shows that the conflict-analysis pipeline dominates the measured runtime of MUC. Most of that time is spent on building and checking the SMT model. This overhead is the main limitation for not trying larger program sizes, and thus retrieving more solutions. A first obvious step is to then improve the reasoning model used. The semantic annotation language used in the benchmarks is much narrower than general SMT. It consists mostly of equalities, inequalities, Boolean implications, and applications of a fixed set of abstract properties. A specialized incremental reasoner could exploit this structure and reuse state between related candidates. Furthermore, a big runtime burden is the whole parsing done to and from the SMT. A custom reasoner that works directly over AST nodes and annotations could avoid a significant amount of overhead, and thus be much more efficient.

7.2.2 Budgeted Unsatisfiable-Core Minimization

The current CVC5 configuration retrieves an unsatisfiable core, but does not guarantee that this core is inclusion-minimal. A preliminary implementation attempted to minimize every returned core through additional solver arguments on calls. It proved to be quite computationally expensive and did not visibly improve the resulting generalized constraints. Consequently, MUC does not always provide the smallest possible explanation before generalization begins.

Future work should compare raw cores with deletion-based or budgeted minimization. In theory, a smaller core may allow more branches to be wildcarded and can therefore produce a more general learned constraint. It could also reduce cleanup work if smaller cores produce more compact conflict explanations. On the other hand, a more general constraint may overlap with more existing patterns. Measuring core size, generalized-pattern size, cleanup actions, pruning, and solver time together would reveal this trade-off. Furthermore, referring back to the custom reasoner, there is room for optimization to find minimal cores more efficiently.

7.2.3 Pipeline Efficiency and Parallelism

The conflict-analysis techniques inspect the same failed candidate independently and execute sequentially. Their checks could be run in parallel between enumeration steps, after which the produced constraints would be reconciled through a cleanup stage. This requires careful ownership of solver state and deterministic merging of results, but could reduce latency when several expensive techniques are enabled.

There are also sequential improvements available. Parsed semantics and SMT declarations can be cached more aggressively. Cleanup can index constraints by root rule, arity, and shape instead of scanning the whole population. Cleanup can also be triggered adaptively or in batches rather than after every failure. The search traces already record when each cleanup operation is useful. They could drive a policy that enables expensive operations only when justified.

7.2.4 More Conflict Analyses and Search Strategies

The evaluation uses MUC alone or in combination with ERA or SeAn. Adding techniques based on abstract interpretation, static analysis, or domain-specific execution properties would test whether cleanup remains effective with more diverse explanations. The existing modular interface makes such comparisons possible. Cross-technique cleanup may expose redundancies that cannot be observed currently.

The experiments also use one breadth-first enumerator. Applying the same generalized constraints and cleanup pipeline to depth-first or stochastic enumerators, or even a bottom-up one, would test whether the observed gains depend on the enumeration strategy. More importantly, different base solvers may encounter a different distribution of failures, and thus different learned knowledge.

7.2.5 Adaptive Retention and Broader Validation

The present cleanup operations preserve the represented forbidden set while making it more compact. A complementary direction is to apply heuristics similar to clause learning in constraint solvers. We can record how often each constraint prunes a state, how expensive it is to propagate, and how recently it was useful, and then deactivate some constraints based on a budget. Deactivation must be restricted to learned constraints, so that user-supplied grammar constraints are always active. This requires a carefully crafted retention heuristic to balance memory, propagation cost, and pruning power.

Finally, the methods should be evaluated on more domains, larger AST bounds, and longer time limits. Broader experiments would show whether compact learned state eventually translates into runtime improvements once searches are long enough.

7.3 Reflection

7.3.1 Impact on the Program-Synthesis Community

The most direct impact of this work is on conflict-driven program synthesis. It demonstrates that the value of a learned conflict depends on two distinct properties that should be measured separately (pruning coverage of individual constraints and compactness of accumulated constraint population). Treating these as separate pipeline stages makes it possible to compare future explanation and constraint-management techniques without tying them to one solver implementation.

The experiments also provide a practical lesson for grammar and benchmark design. Semantic annotations are not equally informative. For researchers building synthesis DSLs, investing in precise relational semantics may therefore be more valuable than adding a large number of weak ones. This redirects attention toward explanation algorithms and end-to-end profiling.

The work remains limited to one framework, one enumeration strategy, and three programming-by-example domains. Its immediate impact is consequently methodological rather than universal. It contributes concrete mechanisms and implementation points, accompanied by experimental measurements, that the program-synthesis community can reuse or build on top of.

7.3.2 Impact on Wider Society

Program synthesis can make building software accessible to people who can describe desired behavior but cannot implement it themselves. Better use of failed candidates more closely mirrors how engineers build software in practice. Most importantly, it allows synthesizers to handle more complex tasks and do so transparently, with explicit reasoning for failures. This can benefit scientists and other domain experts who currently depend on scarce programming expertise. More efficient search could also reduce the computational resources and energy required for synthesis, although this requires further improvements to the whole pipeline.

Greater accessibility also creates risks. A synthesized program can satisfy the supplied examples while still being wrong on unseen inputs. The users who benefit the most from program synthesis are also the least equipped to detect such gaps. Learned constraints improve traceability, but they do not replace thorough validation. Synthesis tools should therefore expose assumptions and encourage testing beyond the examples used for synthesis. Their deployment should support human judgment. The appropriate goal is not automation at any cost. Instead, it is to provide reliable assistance whose limitations are visible. Within that goal, this thesis contributes a modest step towards making a synthesizer’s learned knowledge explicit and measurable.

7.3.3 Use of Generative AI Tools

Generative AI tools were used during this thesis to brainstorm research and presentation ideas. In terms of code generation, the general rule is to use agentic tools based on code criticality. The algorithms we implemented to tackle the main research questions were mostly manually written by the author. Experimental boilerplate and visualizations were mostly AI-assisted, but reviewed and revised by the author. Grammarly was used for grammar, spelling, and style suggestions while writing the report. These tools did not run the experiments or determine the reported results. Finally, the front cover illustration is AI-generated, as a curiosity endeavor to explore the agent’s capabilities.

Bibliography

- R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–17, October 2013.
- G. Audemard and L. Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, pages 399–404, 2009.
- M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs, 2016. URL <https://arxiv.org/abs/1611.01989>.
- A. Cropper and S. Dumančić. Learning large logic programs by going beyond entailment, 2020. URL <https://arxiv.org/abs/2004.09855>.
- Y. Feng, R. Martins, J. V. Geffen, I. Dillig, and S. Chaudhuri. Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*, pages 422–436. ACM, 2017. doi: 10.1145/3062341.3062351. URL <https://doi.org/10.1145/3062341.3062351>.
- Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program Synthesis using Conflict-Driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018*, pages 420–435. ACM, 2018. doi: 10.1145/3192366.3192382. URL <https://doi.org/10.1145/3192366.3192382>.
- S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Found. Trends Program. Lang.*, 4(1-2):1–119, 2017. doi: 10.1561/2500000010. URL <https://doi.org/10.1561/2500000010>.
- 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, 2025. URL <https://arxiv.org/abs/2508.00005>.

- S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 215–224. ACM, 2010. doi: 10.1145/1806799.1806833. URL <https://doi.org/10.1145/1806799.1806833>.
- J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning sat solvers. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185, pages 131–153. IOS Press, 2009. doi: 10.3233/978-1-58603-929-5-131.
- S. Padhi, U. Abhishek, A. Fu, E. Polgreen, and A. Reynolds. Benchmarks for sygus competition. <https://github.com/SyGuS-Org/benchmarks>, 2019.
- O. A. R. Poeth. Generalizing conflict analysis in program synthesis. Master’s thesis, Delft University of Technology, 2025. URL <https://resolver.tudelft.nl/uuid:93bd1efb-7e03-41ed-8d3e-21927c4f22ac>.
- A. Solar-Lezama, L. Tancau, R. Bodik, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415. ACM, 2006. doi: 10.1145/1168857.1168907. URL <https://doi.org/10.1145/1168857.1168907>.

Appendix A

Benchmark Grammars and Semantic Annotations

This appendix records the grammars and semantic annotations used in the experiments. The notation follows the implementation. y is the value produced by a grammar rule. x_i is the value of its i th child. Rules without a displayed annotation have no semantic predicate attached.

The benchmark loaders specialize the grammars per task. DeepCoder adds each typed task input to its shared grammar. SyGuS supplies a task-specific grammar with its own start type, inputs, literals, and production subset. Morpheus adds its input table, columns, new-column names, temporary columns, and observed literal values.

A.1 DeepCoder

DeepCoder uses the nonterminals *Int*, *ExprNum*, and *ExprArr*. The start nonterminal is chosen as *ExprNum* or *ExprArr* according to the task output. Each scalar input is added as an *ExprNum* terminal and each list input as an *ExprArr* terminal.

A.1.1 Grammar

The shared production schema is:

$$\begin{aligned}
Int &::= -3 \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid 3, \\
ExprNum &::= Int \mid NumInput \\
&\mid \text{maximum}(ExprArr) \mid \text{minimum}(ExprArr) \mid \text{sum}(ExprArr) \\
&\mid \text{first}(ExprArr) \mid \text{last}(ExprArr) \mid \text{getIndex}(ExprArr, ExprNum) \\
&\mid \text{countSt}(ExprArr, Int) \mid \text{countGt}(ExprArr, Int) \\
&\mid \text{countEq}(ExprArr, Int) \mid \text{countNeq}(ExprArr, Int) \\
&\mid \text{countMod}(ExprArr, Int) \mid \text{countNmod}(ExprArr, Int), \\
ExprArr &::= ArrInput \mid \text{drop}(ExprArr, ExprNum) \mid \text{take}(ExprArr, ExprNum) \\
&\mid \text{sort}(ExprArr) \mid \text{reverse}(ExprArr) \\
&\mid \text{filterSt}(ExprArr, Int) \mid \text{filterGt}(ExprArr, Int) \\
&\mid \text{filterEq}(ExprArr, Int) \mid \text{filterNeq}(ExprArr, Int) \\
&\mid \text{filterMod}(ExprArr, Int) \mid \text{filterNmod}(ExprArr, Int) \\
&\mid \text{mapPlus}(ExprArr, Int) \mid \text{mapMult}(ExprArr, Int) \\
&\mid \text{mapDiv}(ExprArr, Int) \mid \text{mapPow}(ExprArr, Int) \\
&\mid \text{zipwithMax}(ExprArr, ExprArr) \mid \text{zipwithMin}(ExprArr, ExprArr) \\
&\mid \text{zipwithPlus}(ExprArr, ExprArr) \mid \text{zipwithMinus}(ExprArr, ExprArr) \\
&\mid \text{zipwithMult}(ExprArr, ExprArr) \\
&\mid \text{scanl1Plus}(ExprArr) \mid \text{scanl1Minus}(ExprArr) \\
&\mid \text{scanl1Mult}(ExprArr) \mid \text{scanl1Max}(ExprArr) \mid \text{scanl1Min}(ExprArr).
\end{aligned}$$

A.1.2 Semantic Annotations

For brevity, we write $L(v) = \text{length}(v)$, $H(v) = \text{maximum}(v)$, $M(v) = \text{minimum}(v)$, $F(v) = \text{first}(v)$, and $T(v) = \text{last}(v)$.

$$\text{maximum}(x_1) : H(y) = H(x_1) \wedge M(y) \geq M(x_1)$$

$$\text{minimum}(x_1) : H(y) \leq H(x_1) \wedge M(y) = M(x_1)$$

$$\text{sum}(x_1) : H(y) \leq L(x_1)H(x_1) \wedge M(y) \geq L(x_1)M(x_1)$$

$$\text{first}(x_1) : H(y) \leq H(x_1) \wedge M(y) \geq M(x_1) \wedge F(y) = F(x_1) \wedge T(y) = F(x_1)$$

$$\text{last}(x_1) : H(y) \leq H(x_1) \wedge M(y) \geq M(x_1) \wedge F(y) = T(x_1) \wedge T(y) = T(x_1)$$

$$\text{getIndex}(x_1, x_2) : H(y) \leq H(x_1) \wedge M(y) \geq M(x_1)$$

Count rules Each of the six count rules listed in the grammar has $y \leq L(x_1)$.

$$\text{drop}(x_1, x_2) : L(y) < L(x_1) \wedge H(y) \leq H(x_1) \wedge M(y) \geq M(x_1) \wedge T(y) = T(x_1)$$

$$\text{take}(x_1, x_2) : L(y) < L(x_1) \wedge H(y) \leq H(x_1) \wedge M(y) \geq M(x_1) \wedge F(y) = F(x_1)$$

$$\text{sort}(x_1) : L(y) = L(x_1) \wedge H(y) = H(x_1) \wedge M(y) = M(x_1) \wedge F(y) = M(x_1) \wedge T(y) = H(x_1)$$

$$\text{reverse}(x_1) : L(y) = L(x_1) \wedge H(y) = H(x_1) \wedge M(y) = M(x_1) \wedge F(y) = T(x_1) \wedge T(y) = F(x_1)$$

Filter rules Each of the six filter rules listed in the grammar has $L(y) < L(x_1) \wedge H(y) \leq H(x_1) \wedge M(y) \geq M(x_1)$.

Map rules Each of mapPlus, mapMult, mapDiv, and mapPow has $L(y) = L(x_1)$.

Zip rules Each of zipwithMax, zipwithMin, zipwithPlus, zipwithMinus, and zipwithMult has $L(y) = L(x_1) \wedge L(y) = L(x_2)$.

Basic scan rules Each of scanl1Plus, scanl1Minus, and scanl1Mult has $L(y) = L(x_1) \wedge F(y) = F(x_1)$.

$$\text{scanl1Max}(x_1) : L(y) = L(x_1) \wedge F(y) = F(x_1) \wedge H(y) = H(x_1) \wedge M(y) \geq M(x_1) \wedge T(y) = H(x_1)$$

$$\text{scanl1Min}(x_1) : L(y) = L(x_1) \wedge F(y) = F(x_1) \wedge H(y) \leq H(x_1) \wedge M(y) = M(x_1) \wedge T(y) = M(x_1)$$

A.2 SyGuS PBE-SLIA

Each SyGuS task supplies its own finite set of input symbols and string or integer literals. Its start symbol selects the required return type. The schema below is the union of the task-specific grammars: each task contains only its declared subset of these productions.

A.2.1 Grammar

$Start ::= String \mid Int \mid Bool$ (exactly the task's declared result type),
 $String ::= StringInput \mid StringLiteral$
 $\quad \mid concat(String, String) \mid replace(String, String, String)$
 $\quad \mid at(String, Int) \mid intToStr(Int)$
 $\quad \mid substr(String, Int, Int) \mid ite(Bool, String, String),$
 $Int ::= IntInput \mid IntLiteral$
 $\quad \mid Int + Int \mid Int - Int$
 $\quad \mid len(String) \mid strToInt(String)$
 $\quad \mid indexOf(String, String, Int) \mid ite(Bool, Int, Int),$
 $Bool ::= true \mid false \mid Int = Int$
 $\quad \mid prefixOf(String, String) \mid suffixOf(String, String)$
 $\quad \mid contains(String, String).$

A.2.2 Semantic Annotations

Again for brevity, we write $L(s) = \text{length}(s)$, and $A(s)$, $S(s)$, $D(s)$, $P(s)$ for a string containing an alphabetic character, space, digit, or a special character respectively. Lastly, $C(s, t)$ represents the number of occurrences of t in s . String indices are one-based.

$contains(x_1, x_2) :$

$$\begin{aligned}
&L(x_1) \geq 0, \quad L(x_2) \geq 0, \\
&y = (\text{indexOf}(x_1, x_2, 1) \neq -1), \quad y = (C(x_1, x_2) \geq 1), \\
&\text{Int}(y) \leq \text{Int}(C(x_1, x_2) \geq 1), \\
&\neg y \vee L(x_1) \geq L(x_2), \\
&\neg y \vee (\neg A(x_2) \vee A(x_1)), \quad \neg y \vee (\neg S(x_2) \vee S(x_1)), \\
&\neg y \vee (\neg D(x_2) \vee D(x_1)), \quad \neg y \vee (\neg P(x_2) \vee P(x_1)).
\end{aligned}$$

$concat(x_1, x_2) :$

$$\begin{aligned}
&L(y) = L(x_1) + L(x_2), \quad L(x_1) \leq L(y), \quad L(x_2) \leq L(y), \\
&\text{contains}(y, x_1), \quad \text{contains}(y, x_2), \\
&\text{prefixOf}(x_1, y), \quad \text{suffixOf}(x_2, y), \\
&L(x_1) \geq 0, \quad L(x_2) \geq 0, \quad L(y) \geq 0, \\
&\text{substr}(y, 1, L(x_1)) = x_1, \\
&\text{substr}(y, L(x_1) + 1, L(y)) = x_2, \quad \text{indexOf}(y, x_1, 1) = 1.
\end{aligned}$$

For each $Q \in \{A, S, D, P\}$, $\neg Q(x_1) \vee Q(y)$ and $\neg Q(x_2) \vee Q(y)$.

replace(x_1, x_2, x_3) :

$$L(y) \geq 0, \quad L(x_1) \geq 0, \quad L(x_2) \geq 0, \quad L(x_3) \geq 0.$$

For each $Q \in \{A, S, D, P\}$, $\neg Q(y) \vee (Q(x_1) \vee Q(x_3))$.

at(x_1, x_2) :

$$x_2 \geq 1 \wedge L(x_1) \geq x_2 \wedge L(y) = 1 \wedge L(x_1) \geq 0 \wedge L(x_1) \geq 1.$$

intToStr(x_1) :

$$L(y) \geq 0 \wedge L(y) \geq 1 \wedge \text{isDigit}(y) \wedge \text{strToInt}(y) = x_1 \wedge D(y).$$

substr(x_1, x_2, x_3) :

$$\begin{aligned} x_2 \geq 1, \quad x_3 \geq 1, \quad x_2 \leq x_3, \quad L(x_1) \geq x_3, \\ L(x_1) \geq x_2, \quad L(y) = x_3 - x_2 + 1, \quad L(x_1) \geq L(y), \\ L(x_1) \geq 0, \quad L(y) \geq 0. \end{aligned}$$

For each $Q \in \{A, S, D, P\}$, $\neg Q(y) \vee Q(x_1)$.

len(x_1) :

$$L(x_1) = y \wedge y \geq 0 \wedge L(x_1) \geq 0.$$

strToInt(x_1) :

$$L(x_1) \geq 1 \wedge L(x_1) \geq 0 \wedge \text{isDigit}(x_1) \wedge \text{intToStr}(y) = x_1 \wedge D(x_1).$$

indexOf(x_1, x_2, x_3) :

$$L(x_1) \geq y \wedge L(x_1) \geq 0 \wedge L(x_2) \geq 0 \wedge x_3 \geq 1 \wedge x_3 \leq L(x_1) \wedge y \geq -1.$$

prefixOf(x_1, x_2) :

$$\begin{aligned} L(x_1) \geq 0, \quad L(x_2) \geq 0, \\ (\text{substr}(x_2, 1, L(x_1)) = x_1) = y, \\ (\text{indexOf}(x_2, x_1, 1) = 1) = y, \\ \text{Int}(y) \leq \text{Int}(C(x_2, x_1) \geq 1), \\ \neg y \vee L(x_1) \leq L(x_2). \end{aligned}$$

For each $Q \in \{A, S, D, P\}$, $\neg y \vee (\neg Q(x_1) \vee Q(x_2))$.

suffixOf(x_1, x_2) :

$$\begin{aligned} L(x_1) \geq 0, \quad L(x_2) \geq 0, \\ (\text{substr}(x_2, L(x_2) - L(x_1) + 1, L(x_2)) = x_1) = y, \\ \text{Int}(y) \leq \text{Int}(C(x_2, x_1) \geq 1), \\ \neg y \vee L(x_1) \leq L(x_2). \end{aligned}$$

For each $Q \in \{A, S, D, P\}$, $\neg y \vee (\neg Q(x_1) \vee Q(x_2))$.

The arithmetic, equality, conditional, Boolean-terminal, input, and literal productions carry no function-level annotation.

A.3 Morpheus

Morpheus uses *Start*, *Table*, *ColSet*, *Col*, *NewCol*, *Agg*, *Pred*, *ValExpr*, and *Value*. For each task, *Table* receives the input terminal. *Col* contains the union of input columns, output columns, valid spread-generated names, and four temporary names. *NewCol* contains output and temporary names. *Value* contains the values observed in the input or output in addition to 0 and 1.

A.3.1 Grammar

```
Start ::= Table,  
Table ::= TableInput | gather(Table, NewCol, NewCol, ColSet)  
          | spread(Table, Col, Col) | unite(Table, NewCol, Col, Col)  
          | separate(Table, Col, NewCol, NewCol) | select(Table, ColSet)  
          | group_by(Table, ColSet) | arrange(Table, ColSet)  
          | filter(Table, Pred) | summarise(Table, NewCol, Agg)  
          | mutate(Table, NewCol, ValExpr),  
ColSet ::= cols(Col) | cols2(Col, Col) | not_cols(Col)  
          | not_cols2(Col, Col) | union_cols(ColSet, ColSet),  
Col ::= col(TaskColumn),  
NewCol ::= newcol(TaskNewColumn),  
Agg ::= mean_agg(Col) | sum_agg(Col) | count_agg(),  
Pred ::= positive(Col) | negative(Col) | eq_value(Col, Value)  
        | neq_value(Col, Value) | lt_value(Col, Value) | gt_value(Col, Value),  
ValExpr ::= copy_expr(Col) | div_expr(Col, Col) | div_sum_expr(Col)  
           | add_expr(Col, Col) | sub_expr(Col, Col) | mul_expr(Col, Col),  
Value ::= lit(0) | lit(1) | lit(ObservedLiteral).
```

Before enumeration, six task-independent forbidden patterns are also installed. They disallow equal names in the two *NewCol* positions of *gather*, equal columns in *spread*, equal source columns in *unite*, equal new-column names in *separate*, and equal columns in *cols2* and *not_cols2*.

A.3.2 Semantic Annotations

The Morpheus table and value-expression annotations are used unchanged from Feng et al. [Feng et al., 2017]. The remaining rules have no attached semantic annotations.