

Identifying and Leveraging Structural Sketches in Budgeted Program Synthesis

Online Structural Generalization from Partial Programs

Una Jacimovic

Supervisor(s): Sebastijan Dumancic

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
January 25, 2026

Name of the student: Una Jacimovic
Final project course: CSE3000 Research Project
Thesis committee: Sebastijan Dumancic, Jesper Cocx

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Budgeted program synthesis explores the program space through multiple limited search attempts, but typically discards unsuccessful attempts despite producing partially successful programs that pass some examples. We present an approach that leverages these partial solutions by integrating anti-unification into the synthesis loop to extract and reuse structural sketches. These sketches capture shared program structure while abstracting over details, and are used to guide subsequent search. Direct anti-unification over heterogeneous partial programs is often ineffective. We therefore study grouping strategies based on behavioral similarity and root syntactic structure prior to generalization. Evaluated on string-processing benchmarks from the SyGuS PBE-SLIA Track, our results show that grouping-based approaches yield more informative sketches and improve solution quality on a subset of problems, with root-based grouping being particularly effective when early structural decisions dominate the search.

1 Introduction

Programming has become a central pillar of modern society, underpinning systems in science, industry, and everyday life. As software increasingly mediates critical decisions and processes, the demand for correct and reliable programs continues to grow. Programming is challenging not because of a lack of ideas, but because translating those ideas into correct and consistent code requires careful attention to countless details. Program synthesis addresses this challenge by aiming to automatically construct programs that satisfy a user-provided specification, such as a set of input–output examples. If successful, program synthesis can reduce human effort, improve software reliability, and make programming accessible to a wider range of users.

However, program synthesis faces a fundamental challenge. Constructing a program that satisfies a specification requires searching through a space of candidate programs defined by the underlying language or grammar. Even for small languages, this space grows extremely rapidly with program size and depth, leading to a combinatorial explosion in the number of possible candidates. Exhaustively enumerating and testing all programs until a correct one is found is therefore computationally infeasible. As a result, the effectiveness of the search strategy, how candidate programs are explored, prioritized, and constrained, largely determines whether synthesis succeeds in practice.

One increasingly popular idea for addressing this challenge is budgeted search. Instead of performing a single exhaustive search, the synthesizer runs multiple limited attempts, each with a fixed time or computational budget. After each attempt, the system may restart or adjust its strategy based on what it has learned so far. This approach has two important advantages. First, it allows synthesis to make progress even when resources are limited. Second, it opens the door to learning from failure: even if an attempt does not find a correct program, it may still produce partially successful programs that might contain useful information.

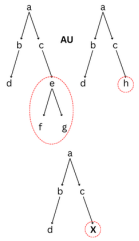


Figure 1: Illustration of anti-unification between two symbolic terms. The least general generalization preserves common structure while abstracting differing subterms into a variable.

Intuitively, partially successful programs are rarely arbitrary. Even when they fail to fully satisfy a specification, such programs often exhibit behavior that aligns with key aspects of the target solution. Satisfying even a subset of examples imposes structural constraints on the program, requiring the use of appropriate operations and compositions. As a result, partially correct programs are frequently close to a correct solution in a structural sense, though this proximity is not guaranteed: partial success may also arise from incidental correlations or spurious behaviors. Human programmers naturally exploit this observation by identifying common patterns, abstracting over irrelevant differences, and refining partial solutions iteratively. Anti-unification [1] offers a principled way to extract this shared structure. Anti-unification (AU) was introduced independently by Plotkin and Reynolds in 1970 as a formal mechanism for computing generalizations over symbolic expressions [2]. Given multiple programs, anti-unification computes their least general generalization, capturing shared structure while abstracting over points of variation. When applied to partially successful synthesis results, this process yields structural sketches that capture the overall shape of a programs that were somewhat successful. This sketch is more informative than just a single program and more flexible than concrete code fragments.

This makes anti-unification particularly well suited for budgeted synthesis. In a budgeted synthesis setting, each search attempt explores only a limited region of the program space and typically produces several partially successful programs rather than a complete solution. Treating these attempts as independent discards valuable structural information contained in these partial successes. Anti-unification makes it possible to extract this information by identifying common structural patterns shared across programs that perform well on similar parts of the specification.

The resulting sketches act as compact, high-level descriptions of promising regions of the search space. By reintroducing these sketches into subsequent synthesis attempts, the search is no longer forced to rediscover useful structure from scratch. Instead, the synthesizer can directly instantiate and refine candidate programs that already satisfy key structural constraints, reducing the number of irrelevant programs explored and accelerating convergence toward a correct solu-

tion. In this way, sketches improve synthesis efficiency not by narrowing the specification, but by reshaping the search space itself, biasing exploration toward program structures that are empirically more likely to lead to success.

In this sense, anti-unification fills a gap between two extremes. On one side, fragment-based approaches reuse concrete pieces of code, which can be too specific and noisy. On the other side, fully learned abstractions require many successful programs and significant computation. Anti-unification offers a lightweight alternative: it learns just enough structure from partial successes to meaningfully guide search, without requiring complete solutions or domain-specific knowledge.

Despite the promise of sketch-based guidance, existing work leaves open several important questions. In particular, it remains unclear how structural sketches can be learned *online* from partially successful programs within a single synthesis task, how such sketches should be selected and grouped when multiple partial solutions are available, and how they can be reused effectively under strict computational budgets. Most existing approaches either assume access to fully correct programs, rely on offline learning across many tasks, or reuse concrete fragments that are very specific. As a result, the potential of partial successes as a source of structural guidance during synthesis remains underexplored.

This thesis addresses these gaps. We investigate how anti-unification can be integrated directly into the budgeted synthesis loop to extract structural sketches from partially successful programs as they are discovered. We propose mechanisms for identifying, clustering, and reusing such sketches online, allowing the synthesizer to incrementally reshape the search space. By doing so, our approach leverages partial correctness not as an endpoint, but as a source of structural insight that guides future exploration. This enables more efficient synthesis under limited budgets, without requiring complete solutions, domain-specific heuristics, or extensive *offline* training.

2 Background

Program synthesis is the task of automatically constructing a program that satisfies a given specification. Specifications may take various forms, such as logical constraints, input–output examples, or partial program sketches. In this thesis, we focus on *example-based program synthesis*, where the goal is to find a program that produces the correct output for a finite set of input–output examples.

2.1 Search Based Program Synthesis

Many synthesis systems formulate program synthesis as a search problem over abstract syntax trees generated by the grammar. Programs are enumerated and evaluated against given specification, and the search continues until the satisfying program is found or resources are exhausted. Search may proceed top-down, bottom-up or through hybrid strategies and is typically guided by syntactic, semantic or cost-constraints. However, naive enumeration remains computationally expensive, motivating the use of budgets and learning mechanisms to guide search.

2.2 Budgeted and Iterative Search

Budgeted search addresses the combinatorial complexity of synthesis by dividing the search process into multiple bounded attempts. Instead of performing a single exhaustive search, the synthesizer runs repeated attempts, each limited by a time, enumeration, or cost budget. This allows synthesis to make progress even when resources are limited and provides natural checkpoints for incorporating feedback from previous attempts.

An important advantage of this setting is that unsuccessful attempts are not wasted. Even when a correct program is not found, the search may produce *partially successful programs*, programs that satisfy a subset of the examples. Such programs can provide valuable information about the structure of the desired solution and can be used to guide subsequent search attempts.

2.3 Structural Abstractions and Sketches

A recurring theme in program synthesis is the observation that correct programs often share structural regularities with partially successful ones. These regularities may take the form of recurring subtrees, common operator patterns, or high-level program shapes. Capturing such structure allows the synthesizer to constrain future search toward promising regions of the program space.

One way to represent structural abstractions is through *sketches*: abstract program shapes that fix certain syntactic structure while leaving other parts unspecified. Sketches reduce the effective search space by committing to a high-level structure while allowing flexibility in the details.

This thesis builds on this idea by learning sketches directly from partially successful programs encountered during budgeted search.

2.4 Program synthesis in Herb.jl.

Herb.jl is a modular program synthesis framework that supports grammar-based enumeration, uniform program execution, and multiple bottom-up search strategies, including size-based, depth-based, and cost-based iteration. In Herb.jl, program synthesis is formulated as a search over abstract syntax trees (ASTs) generated from a context-free grammar, subject to global resource constraints such as maximum size, depth, or cost.

Each iterator in Herb.jl defines its own notion of progress through the program space, its own internal state, and its own mechanism for advancing from simpler to more complex programs. As a result, the behavior of the search process and the points at which it can be influenced depends on the specific iterator being used.

In this thesis, program synthesis is performed using a *size-based bottom-up iterator*. Programs are represented as abstract syntax trees (ASTs) composed of `RuleNode` structures, and enumeration proceeds by constructing larger program shapes from smaller ones according to the grammar.

The iterator operates on *abstract program shapes* rather than fully instantiated concrete programs. These shapes may contain holes (`UniformHole`s) and represent families of programs that are instantiated later through uniform enumera-

tion, allowing the bottom-up phase to focus on structural exploration.

A persistent program bank indexed by return type and structural size is used to store discovered shapes. Enumeration proceeds monotonically from smaller to larger structures under a global maximum size bound. Because sketches share the same representation as grammar-derived shapes, they can be indexed and reused uniformly within this framework, making the size-based bottom-up iterator a natural foundation for sketch-based synthesis.

3 Related Work

This work lies at the intersection of budgeted program synthesis, learning from partial solutions, and structural generalization via anti-unification. Prior research has explored each of these dimensions in isolation or in limited combinations. In contrast, this thesis focuses on integrating anti-unification over partially successful programs directly into a *single-task, budgeted search loop*, where learned structural sketches are reused incrementally to guide subsequent synthesis attempts.

Budgeted program synthesis. Budgeted search has emerged as a practical strategy for controlling the combinatorial explosion inherent in program synthesis. Systems such as Probe [3] structure synthesis as a sequence of bounded attempts and adapt the search distribution between attempts by updating grammar probabilities. While effective, Probe relies on probabilistic reweighting of existing grammar rules and does not introduce new structural abstractions beyond those already present in the grammar.

DreamCoder [4] and Knorf [5] adopt a multi-task learning perspective, solving many synthesis problems simultaneously and extracting reusable subprograms that are shared across solutions. These systems demonstrate the power of abstraction learning but require multiple solved tasks and substantial computation to induce useful libraries. In contrast, this thesis targets the *single-task* setting and learns abstractions directly from partially successful programs within the same synthesis problem, without assuming access to multiple completed solutions.

Learning from partial programs and structural sketches. Several approaches exploit the observation that partially successful programs are often structurally related to the correct solution. FrAngel [6] introduces the notion of *special-case similarity*, mining code fragments from programs that satisfy subsets of examples and recombining them to form new candidates. While effective in practice, FrAngel operates on concrete fragments and relies on heuristic recombination, which can lead to noisy or overly specific search guidance.

Earlier work has shown that real programs often reuse common structural patterns. Post and Gildea [7] used Bayesian inference over Tree Substitution Grammars to model recurring subtrees in a principled way. PATOIS [8] builds on this idea by learning frequently occurring AST fragments and turning them into reusable grammar operators. Coiling [9] further demonstrates that a small number of loop patterns account for much of the structure found in real-world code. While these approaches highlight the importance of recurring structure, they typically rely on large program corpora or operate offline, and do not directly address the challenges of online, budgeted program synthesis.

Anti-unification in program synthesis. Anti-unification provides a formal mechanism for extracting shared structure between programs by computing their least general generalization [1], [2]. Cerna and Kutsia [10] provide a comprehensive survey of anti-unification techniques and applications, including its use in program analysis and transformation.

More recently, BABBLE [11] integrates anti-unification into DreamCoder's library learning phase by performing generalization over equivalence classes represented as e-graphs. This *library learning modulo theory* (LLMT) approach discovers powerful abstractions but operates in a multi-task setting and relies on equational reasoning to identify equivalent programs. As such, it is computationally heavier and conceptually distinct from the lightweight, single-task setting considered in this thesis.

Positioning of this work. In contrast to prior approaches, this thesis integrates anti-unification *directly into a budgeted, bottom-up enumeration process* for a single synthesis task. Learned sketches are represented as abstract program shapes with explicit holes and are treated uniformly with grammar-derived shapes by a size-based bottom-up iterator. This design allows structural generalizations to be introduced incrementally, reused immediately, and explored under the same size, depth, and type constraints as the original search space.

Unlike fragment-based approaches, the proposed method generalizes structure rather than reusing concrete code. Unlike probabilistic grammar adaptation, it introduces new structural shapes rather than merely reweighting existing ones. Finally, unlike multi-task library learning systems, it operates effectively in a single-task setting and learns from partial successes rather than completed solutions. Together, these distinctions position this work as a lightweight yet principled approach to integrating structural learning into budgeted program synthesis.

4 Problem Statement

Program synthesis aims to automatically construct a program that satisfies a user-provided specification. Given a grammar G that defines the program space P_G and a specification function $S : P_G \to \{0, 1\}$, the synthesis task is to find a program $(p \in P_G)$ such that $(S(p) = 1)$. In iterative or budgeted synthesis, programs are generated over multiple attempts under limited search time or computational resources, making it essential to incorporate feedback from earlier attempts to guide later ones.

This thesis investigates how anti-unification can be leveraged to make iterative synthesis attempts progressively more effective at discovering the correct program. The key challenge is tuning the granularity of the resulting generalizations. Applying anti-unification across a wide range of programs tends to produce patterns that are overly general and therefore uninformative, whereas applying it to a narrowly selected programs sets risks introducing many overly specific patterns into the DSL, ultimately hindering subsequent search.

For anti-unification to be effective in this setting, we must determine which synthesized programs should serve as input,

how these programs should be grouped prior to generalization, and under what criteria the extracted patterns should be accepted into the grammar. In this thesis, a program's quality is assessed by the number of specification tests it satisfies, with full satisfaction indicating correctness. Programs that satisfy at least one test case are treated as partially successful and provide the structural material from which reusable generalizations may be extracted.

The central problem addressed in this thesis is therefore the following:

> **Research Question.** Given a set of partially successful programs $\mathcal{P}^+$ generated during iterative synthesis, determine a method for extracting structural generalizations g and integrating them into the DSL such that subsequent synthesis becomes more efficient and more effective.

To answer this question, the thesis focuses on three subproblems:

- **Program Selection:** Identifying which partially successful programs should be considered for anti-unification under limited search budgets.
- **Program Grouping:** Determining how to cluster or otherwise organize programs so that anti-unification produces generalizations that are neither too broad to be meaningful nor too specific to be reusable.
- **Rule Admission:** Establishing criteria for deciding when a generalized pattern should be added to the DSL, balancing the potential benefits of search-space compression against the risk of introducing overly specialized rules or expanding the grammar to the point where it hinders efficient synthesis.

5 Methodology

The project will be conducted using the Herb.jl program synthesis framework, implemented in Julia. Herb.jl provides a modular infrastructure for defining grammars, search strategies, and synthesis tasks, making it suitable for implementing and evaluating multiple budgeted search approaches within a unified environment.

5.1 Implementing budgeted search

A core component of this project is the extension of Herb.jl with a budgeted search controller that structures program synthesis as a sequence of bounded search phases. A key motivation for introducing Budgeted Search Controller is that the original iterator abstractions in Herb.jl were designed primarily for uninterrupted enumeration of candidate programs. Iterators expose a stream of programs but do not provide a structured interface for pausing execution, inspecting intermediate results, or injecting new search biases during enumeration. As a result, integrating learning mechanisms directly into the synthesis loop was previously ad hoc and tightly coupled to specific iterator implementations.

The budgeted search controller introduces a clear separation between *enumeration* and *control*. By treating synthesis as a sequence of bounded attempts with explicit state handoff between them, the controller establishes a uniform interface through which intermediate results can be observed, transformed, and reintegrated. Between successive attempts, the controller exposes two key hooks that define the learning interface. First, a *selector* chooses which partially successful programs produced during the attempt should be retained for learning. Second, an *updater* operates on the selected programs: it applies grouping and anti-unification, injects any learned sketches into the iterator state, and returns control to the iterator for the next attempt.

This separation ensures that learning operates strictly *between* enumeration phases and remains agnostic to the iterator's internal traversal strategy. In particular, the controller does not assume a specific notion of search order or progress; instead, it treats the iterator as a black box whose state can be inspected and augmented at well-defined boundaries.

Beyond enforcing budgets, the controller explicitly captures the outcome of each attempt, including the best program found, its score, and the set of partially successful programs produced during enumeration. These programs are retained together with lightweight behavioral metadata describing which specification examples they satisfy. Importantly, the enumerator state itself is preserved and can be modified between attempts.

Algorithm 1 summarizes the control flow of the budgeted search controller, highlighting the points at which partially successful programs are selected, generalized, and reintegrated into subsequent synthesis attempts.

Between successive attempts, the controller exposes well-defined intervention points that allow learning mechanisms to operate on the accumulated partial results. At these points, a user-defined *selection module* chooses which partially successful programs to retain, a *grouping module* organizes them into clusters, and a *generalization module* (anti-unification) extracts structural sketches from each group. Accepted sketches are then injected into the iterator state before

Algorithm 1 Budgeted synthesis with inter-attempt learning

Require: Problem specification $\mathcal{S}$, grammar G, attempt budget B, learning modules $(\mathcal{S}el, \mathcal{G}roup, \mathcal{L}earn)$
Ensure: Best program found (if any)

```
Initialize iterator state σ
𝒦 ← ∅                                   ▷ Accumulated sketches
for i = 1 to B do
    Run bounded synthesis attempt with state σ
    Collect partially successful programs 𝒫_i^+
    if 𝒫_i^+ ≠ ∅ then
        𝒫_i ← Sel(𝒫_i^+)
        𝒞_i ← Group(𝒫_i)
        𝒦_i ← Learn(𝒞_i)
        Integrate 𝒦_i into iterator state σ
    end if
    if solution found then
        return solution
    end if
end for
return best program found
```

the next attempt begins, influencing future enumeration without restarting the search from scratch.

Concretely, each budgeted iteration proceeds as follows: (1) run bounded enumeration using the current iterator state; (2) collect partially successful programs and their behavioral signatures; (3) apply selection, grouping, and anti-unification to produce candidate sketches; and (4) integrate these sketches into the iterator so that subsequent enumeration considers both grammar-derived and sketch-derived program shapes.

The extended controller therefore separates *search* from *learning*, while tightly coupling them through a controlled feedback loop. This design ensures that learning can bias future search directions while keeping the total computational effort fixed and comparable across different experimental settings.

The controller supports:

- explicit enforcement of per-attempt resource constraints (e.g., depth, size, cost, and number of enumerations);
- systematic collection of partially successful programs and their test outcomes;
- modular integration of selection, grouping, and rule-admission strategies between attempts;
- reproducible execution across grammars, benchmarks, and learning configurations.

5.2 Extending the Size-Based Bottom-Up Iterator with Sketch Integration

This thesis extends Herb.jl's size-based bottom-up iterator to support the incremental reuse of learned structural sketches during budgeted synthesis. In the original iterator, enumeration proceeds exclusively through grammar-derived program shapes stored in a persistent program bank and combined bottom-up under size and depth constraints. While this design enables systematic structural exploration, it provides no mechanism for injecting externally learned structure into the search.

To address this limitation, we augment the iterator with three tightly coupled mechanisms: (i) explicit storage of learned sketches in the iterator state, (ii) bounded expansion of sketches using the existing program bank, and (iii) priority-based interleaving of sketch-derived and grammar-derived enumeration.

Sketch representation and admission. Sketches produced by anti-unification are represented as abstract program shapes containing `UniformHole`s. Because these sketches share the same AST-based representation as grammar-derived shapes, they can be stored in the iterator state and indexed by return type and size without modifying the grammar. Upon admission, sketches are treated as first-class program shapes and are subject to the same global size and depth constraints as standard enumeration.

Sketch-based shape instantiation. Sketches are consumed by the iterator in the same way as grammar-derived shapes. Each `UniformHole` in a sketch is filled with compatible program shapes retrieved from the program bank, respecting return types and size constraints. This produces a concrete abstract syntax tree that fixes the high-level structure of the sketch while instantiating its unspecified components.

The resulting shape is then passed to Herb's uniform iterator, which enumerates fully concrete programs consistent with the instantiated structure. In this way, sketches do not introduce a separate execution mechanism but reuse the existing uniform enumeration pipeline, ensuring semantic consistency with grammar-derived programs.

Bounded sketch expansion. To preserve the budgeted nature of search, sketch expansion is explicitly bounded. For each synthesis attempt, only a fixed number of sketch-derived programs may be generated, and each sketch is expanded only a limited number of times. These bounds prevent sketch-based enumeration from overwhelming grammar exploration and ensure that learned structure complements rather than replaces standard search.

Priority-based enumeration. The iterator maintains a separate priority queue for sketch-derived candidates. Whenever available, sketch-generated programs are explored before grammar-derived combinations. Once sketch expansion is exhausted, enumeration resumes normal bottom-up exploration. This prioritization enables rapid instantiation of promising structural patterns while preserving the original enumeration order and internal invariants of the iterator.

Together, these extensions allow learned sketches to be reused immediately and systematically within the existing bottom-up framework. Structural abstractions learned from partial solutions are instantiated using the same mechanisms as grammar-defined shapes, enabling sketch-guided synthesis without altering the grammar or compromising budget constraints.

5.3 Integrating Program Selection and Grouping Mechanisms

Overview of selection strategies. This thesis evaluates four selection strategies that differ in how candidate programs are chosen and grouped prior to anti-unification.

(1) No selection (baseline). The selector from budgeted controller selects no programs and no learning occurs.

(2) All programs. All partially successful programs from an attempt are passed directly to anti-unification. In practice, this often fails to produce a sketch due to structural incompatibility among programs or yields overly abstract results.

(3) Behavioral clustering. Programs are grouped by behavioral similarity before anti-unification. Each program is associated with a binary mask $m \in \{0,1\}^n$, where $m_i = 1$ if the program satisfies the i-th example. Similarity between two programs is defined as normalized mask overlap:

$$\text{sim}(m_1, m_2) = \frac{|m_1 \cap m_2|}{\sqrt{|m_1| \cdot |m_2|}}, \tag{1}$$

Programs whose similarity exceeds a fixed threshold are clustered together. This grouping restricts anti-unification to behaviorally aligned programs, increasing the likelihood of extracting meaningful shared structure (Figure 2).

(4) Root-based grouping. Programs are grouped by the grammar rule applied at the root of their AST. Anti-unification requires syntactic alignment at the root; when root

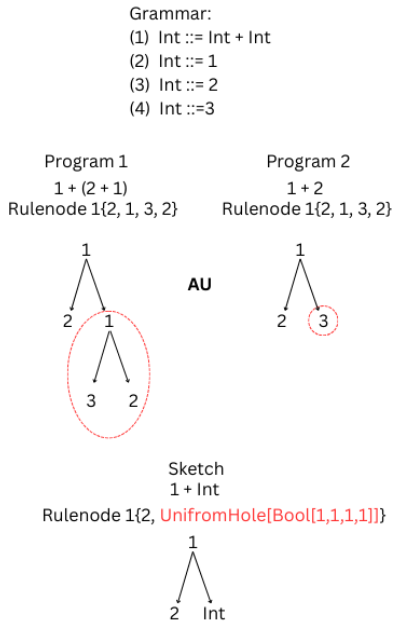


Figure 2: Anti-unification over program shapes for a simple integer grammar.

rules differ, generalization often fails or degenerates to trivial types. Root-based grouping ensures structural compatibility and enables generalization even when programs exhibit similar behavior but differ syntactically at lower levels.

Algorithm 2 summarizes the generic sketch-learning procedure used by the selection strategies that perform grouping, showing how partially successful programs are clustered and generalized via anti-unification between budgeted synthesis attempts.

Sketch-derived programs are explored with higher priority than grammar-derived combinations, but their expansion is explicitly bounded (Algorithm 3). This prevents sketches from dominating the search and ensures that each budgeted attempt continues to explore new grammar-defined structures alongside learned ones.

By decoupling program selection, grouping, generalization, and reuse, this design enables systematic evaluation of learning strategies within a single budgeted search framework. In particular, it allows us to assess whether behavioral or syntactic grouping leads to more informative sketches and improved synthesis performance compared to unstructured generalization.

6 Experimental Evaluation

This chapter presents an experimental evaluation of the proposed sketch-learning framework based on budgeted program synthesis and anti-unification. The goal of the experiments is to assess whether learning structural sketches from partial solutions can improve synthesis efficiency under fixed computational budgets.

Experimental configuration. The experimental evaluation in this thesis is conducted using the string-processing bench-

Algorithm 2 Sketch learning between budgeted synthesis attempts

Require: Set of partially successful programs P
Require: Grammar G, similarity threshold τ
Ensure: Set of learned sketches S
$C \leftarrow \emptyset$
$S \leftarrow \emptyset$
for each program $p \in P$ **do**
  Assign p to a cluster $c \in C$ if $\text{sim}(p, c) \geq \tau$
  if no such cluster exists **then**
    Create new cluster containing p
  end if
end for
for each cluster $c \in C$ **do**
  if $|c| < 2$ **then**
    continue
  end if
  $s \leftarrow \text{AntiUnify}(c, G)$
  if $s \neq \emptyset$ **then**
    $S \leftarrow S \cup \{s\}$
  end if
end forreturn S

Algorithm 3 Bounded sketch-prioritized enumeration

for each sketch s **do**
  Enqueue a bounded number of sketch-derived candidates
end for
while budget remains **do**
  Prefer sketch-derived candidates when available
end while

marks from the SyGuS PBE-SLIA Track 2019. These bencmarks consist of program-sythesis tasks with string-manipulation specifications. Each benchmark is defined with a set of input-output examples and a grammar specifying permissible space of string programs.

Unless stated otherwise, all experiments use the following fixed configuration. Synthesis is performed using the size-based bottom-up iterator with a maximum program depth of 10 and a maximum program size of 15. Each budgeted attempt is limited to 10,000 program evaluations, and synthesis is run for a fixed number of attempts per benchmark. Evaluation errors during execution are allowed, and synthesis terminates early if a fully satisfying program is found.

Programs are considered *partially successful* if they satisfy at least one example, i.e., if their score is strictly greater than zero. These programs form the candidate pool for selection, grouping, and anti-unification between attempts.

For clustering-based learning, the behavioral similarity threshold is fixed to $\tau = 0.8$ across all benchmarks. This value was chosen to ensure that only programs with substantial agreement on satisfied examples are grouped together.

To evaluate the effect of learning from partial solutions, the controller applies selection and grouping mechanisms after each attempt. Anti-unification is then applied within clusters to extract structural sketches, which are integrated into the it-

erator state and reused in subsequent attempts. Clusters containing fewer than two programs are discarded, and a fixed similarity threshold is used across all benchmarks.

The experimental evaluation reports the following metrics:

- synthesis success rate under fixed enumeration budgets;
- number of budgeted attempts required to discover a correct program;
- effect of sketch learning on search efficiency across attempts;

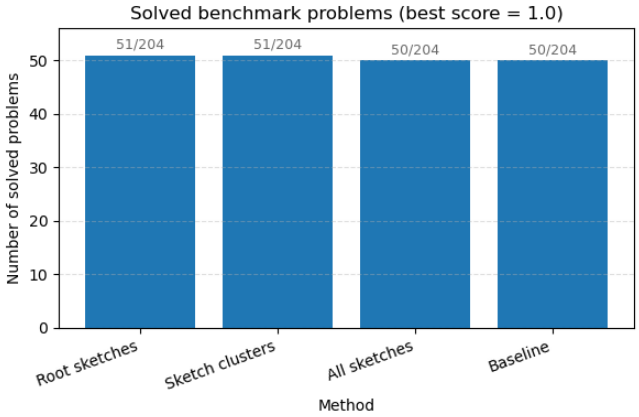


Figure 3: Number of solved benchmark problems under a fixed enumeration budget. Results are shown for the baseline (no learning), direct anti-unification (`all_sketches`), clustering-based sketch learning (`sketch_clusters`), and root-based grouping (`root_sketches`).

Figure 3 provides an overview of the number of solved benchmark problems under a fixed budget across the evaluated strategies. While learning-based approaches improve performance, no single strategy dominates across all benchmarks, highlighting the importance of how partial solutions are selected and grouped prior to anti-unification.

6.1 Experimental Research Questions

The experimental evaluation is structured around the following research questions:

RQ1: Does directly applying anti-unification to all partially successful programs produce useful structural sketches?

RQ2: Does grouping partially successful programs by behavioral similarity before anti-unification lead to more informative sketches?

RQ3: Does sketch learning influence synthesis efficiency under fixed computational budgets?

These questions are designed to isolate the contribution of sketch learning and to evaluate whether additional structure in the learning process is necessary to obtain meaningful generalizations.

6.2 Direct Anti-Unification Without Grouping

Why this question matters. Why this question matters. Anti-unification is commonly used to extract shared structure from programs. However, when applied to diverse programs, it can produce patterns that are too general to effectively guide search. Evaluating direct anti-unification therefore provides a necessary baseline against which more structured approaches can be compared.

Experimental design. In the first experimental configuration, all partially successful programs produced during a budgeted attempt are passed directly to the anti-unification procedure. No selection or grouping is performed. If anti-unification returns a generalized program, the resulting sketch is added to the iterator state and made available during subsequent enumeration.

Observed results. Across benchmarks, direct anti-unification did not perform better than the baseline as observed on figure 4. When sketches are produced, they tend to be either excessively abstract or structurally trivial, offering little guidance beyond what is already encoded in the grammar. In cases with large budget for each iteration, the anti-unification procedure returns no pattern due to structural incompatibilities among the selected programs.

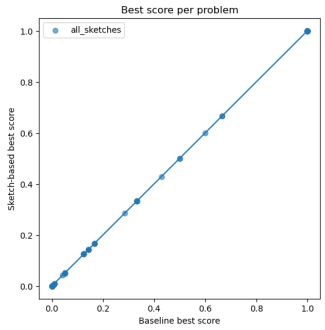


Figure 4: Best score per problem achieved by sketch-based synthesis versus the baseline. Each point corresponds to a benchmark problem. Points lying on the diagonal indicate no improvement over the baseline.

We additionally examined whether direct anti-unification improves the speed at which high-quality solutions are discovered. Figure 5 compares the enumeration step at which equally scoring solutions are found by the baseline and by sketch-based synthesis using direct anti-unification. Points lie almost entirely along the diagonal, indicating that direct anti-unification does not consistently reduce the number of enumeration steps required to reach the same solution quality.

This further supports the conclusion that sketches produced without grouping provide little effective guidance. Even when such sketches are generated, they do not meaningfully bias the search toward better candidates earlier in the enumeration process.

Interpretation. These results suggest that partially successful programs are very diverse. Even when programs solve the same example, their structures can be quite different. Because of this, directly applying anti-unification across all such programs rarely produces a useful shared pattern. This indicates that some form of additional structure is needed to guide the generalization process.

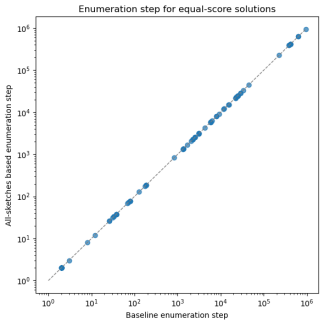
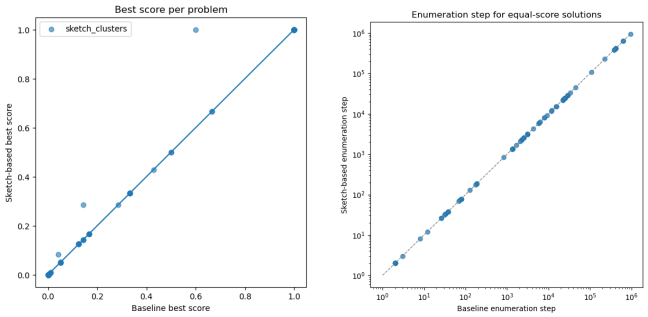


Figure 5: Enumeration step at which equally scoring solutions are found by the baseline and sketch-based synthesis using direct anti-unification. Points near the diagonal indicate no consistent speedup over the baseline.



(a) Best score per problem achieved by cluster sketch-based synthesis versus the baseline.

(b) Enumeration step at which equally scoring solutions are found by the baseline and cluster sketch-based synthesis.

Figure 6: Performance of clustering-based sketch learning. Left: solution quality compared to the baseline. Right: enumeration step for equally scoring solutions. Points near the diagonal indicate no improvement over the baseline.

6.3 Clustering-Based Anti-Unification

Why this question matters. The negative results show that partial solutions are too diverse to be generalized without additional structure. However, programs that satisfy similar subsets of examples may reflect shared reasoning about the specification and are therefore more likely to share meaningful structural components. Clustering programs by behavioral similarity prior to anti-unification aims to reduce heterogeneity while preserving diversity across different solution strategies.

Experimental design. In this configuration, partially successful programs are grouped according to their behavioral similarity before anti-unification is applied.

Observed results. Clustering yields smaller and more coherent program sets for generalization, often enabling the extraction of meaningful sketches. Figure 6a compares the best score achieved per problem by cluster sketch-based synthesis against the baseline. While most points lie close to the diagonal, indicating similar performance on many benchmarks, there are notable exceptions. In particular, clustering successfully solves one benchmark problem that the baseline fails to solve, and achieves a strictly higher best score on two other problems. These cases correspond to points above the diagonal and demonstrate that clustering can lead to qualitatively different search behavior when partial solutions admit meaningful behavioral structure. These improvements, while modest in number, suggest that behavioral clustering can meaningfully constrain the generalization process when sufficient partial structure is available.

We also evaluated whether clustering-based sketches improve the speed at which high-quality solutions are discovered. Figure 6b compares the enumeration step at which equally scoring solutions are found by the baseline and the cluster sketch-based method. Points lie closely along the diagonal, indicating that clustering does not consistently reduce the number of enumeration steps required to reach the same solution quality.

Interpretation. These results support the hypothesis that behavioral alignment is a useful proxy for structural similarity in program synthesis. By restricting anti-unification to programs that agree on a substantial portion of the specification, clustering filters out structurally incompatible candidates and increases the likelihood of extracting informative generalizations. This suggests that the effectiveness of sketch learning depends critically on how partial solutions are grouped, rather than on anti-unification alone.

6.4 Root-Based Anti-Unification

Why this question matters. While behavioral clustering reduces heterogeneity by grouping programs that satisfy similar subsets of examples, it does not account for syntactic compatibility. Anti-unification operates on abstract syntax trees and requires structural alignment at the root. When partial solutions differ in their top-level grammar rule, anti-unification often fails or degenerates to trivial generalizations, even if the programs exhibit similar behavior. Root-based grouping directly addresses this limitation by enforcing structural compatibility prior to generalization.

Experimental design. In this configuration, partially successful programs are grouped according to the grammar rule applied at the root of their abstract syntax tree. Anti-unification is then applied independently within each root group. Programs with different root rules are never generalized together, ensuring that all inputs to anti-unification share the same high-level syntactic structure.

Observed results. In several benchmarks, this approach leads to higher best scores than behavioral clustering, particularly when uncertainty in the search is dominated by early structural choices rather than by lower-level details.

Figure 7 compares the best score achieved per problem by root-based sketch learning against the baseline. As in the clustering-based setting, most points lie close to the diagonal, indicating comparable performance on many benchmarks. However, several problems appear above the diagonal, showing cases where root-based sketches achieve strictly higher best scores than the baseline. In these instances, root-based sketch learning is able to guide the search more effectively by committing early to high-level structural decisions, reducing uncertainty before lower-level details are explored.

These improvements are particularly evident in benchmarks where the primary challenge lies in selecting the correct overall program structure rather than refining local operations.

We additionally examined whether root-based sketches improve the speed at which high-quality solutions are discovered. Figure 8 compares the enumeration step at which equally scoring solutions are found by the baseline and the root-based sketch method. As in the clustering-based setting, points lie closely along the diagonal, indicating that root-based sketch learning does not consistently reduce the number of enumeration steps required to reach the same solution quality.

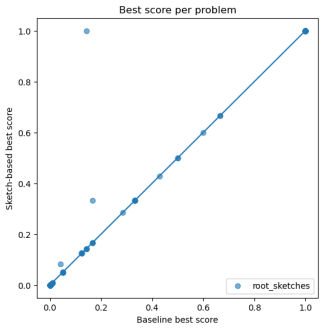


Figure 7: Best score per problem achieved by root-based sketch learning versus the baseline. Each point corresponds to a benchmark problem. Points lying on the diagonal indicate no improvement over the baseline.

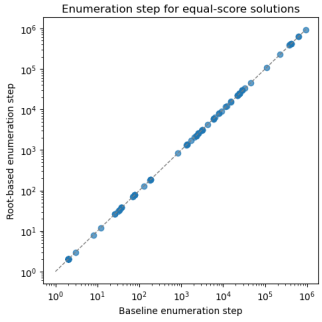


Figure 8: Enumeration step at which equally scoring solutions are found by the baseline and root-based sketch learning. Points near the diagonal indicate no consistent speedup.

Interpretation. These results indicate that enforcing syntactic alignment at the root is a strong and effective constraint for sketch learning. Root-based grouping enables anti-unification to extract meaningful high-level program shapes by eliminating structural incompatibilities upfront. At the same time, its restrictive nature highlights a trade-off: while structural compatibility is necessary for generalization, overly rigid grouping can prevent learning when the correct top-level structure has not yet been explored. This suggests that combining behavioral and structural criteria may offer a more balanced approach to sketch learning.

6.5 Conclusion

Figure 3 highlights that sketch learning can improve synthesis performance under fixed budgets, but that its effectiveness depends strongly on how partial solutions are selected and grouped prior to anti-unification. Direct anti-unification over all partially successful programs (`all_sketches`) does not improve over the baseline, confirming that unstructured generalization is insufficient in heterogeneous search spaces.

In contrast, grouping-based strategies yield more consistent gains. Behavioral clustering (`sketch_clusters`) improves performance on a subset of benchmarks by isolating programs that agree on similar portions of the specification, leading to more informative sketches. Root-based grouping (`root_sketches`) performs best on problems where the primary uncertainty lies in early structural decisions, indicating that enforcing syntactic compatibility is often critical for successful generalization.

At the same time, no single strategy dominates across all benchmarks. Several problems remain unsolved by all methods, suggesting that sketch learning is most effective once the search has already discovered programs with approximately correct high-level structure. When early structural choices are still missing, neither behavioral nor syntactic grouping alone is sufficient to guide the search.

Overall, these results demonstrate that sketch learning can meaningfully reshape the search space and accelerate convergence, but only when combined with careful selection and grouping of partial solutions. This reinforces the central claim of this thesis: the benefit of anti-unification in budgeted synthesis arises not from generalization alone, but from how and when that generalization is applied within the search process.

7 Responsible Research

Ensuring the reproducibility and transparency of scientific research is a central consideration in this thesis. All experimental results are obtained using a clearly specified synthesis setup, including fixed computational budgets, benchmark selection, and algorithmic parameters. These details are explicitly documented in the experimental evaluation section to enable fair comparison and independent replication.

The proposed methods are described at an algorithmic level and motivated step by step, clarifying how program selection, grouping, and sketch integration influence the synthesis process. To further support reproducibility, the implementation will be integrated into the Herb.jl framework, an open-source system for grammar-based program synthesis. By building on an existing, modular research framework, this work facilitates reuse, verification, and extension by other researchers, without requiring re implementation from scratch.

More broadly, responsible research in program synthesis requires careful consideration of how automated systems are evaluated and applied. Program synthesis tools can influence software development workflows, potentially affecting code quality, maintainability, and correctness. Overstating the generality or effectiveness of synthesis techniques may lead to inappropriate deployment or misplaced trust in automatically generated programs. For this reason, this thesis emphasizes

controlled experimental evaluation, explicit reporting of negative results, and clear discussion of the limitations of sketch learning. By presenting both the strengths and the boundaries of the proposed approach, this work aims to support responsible adoption of synthesis techniques and to encourage further research that builds on realistic assumptions about their capabilities.

8 Conclusions and Future Work

This thesis investigated whether structural sketches learned from partially successful programs can improve the efficiency of program synthesis under fixed computational budgets. By integrating anti-unification into a budgeted, bottom-up enumeration framework, the proposed approach enables synthesis to learn from intermediate failures and reuse structural information across search attempts.

The experimental evaluation demonstrates that anti-unification alone is not sufficient to produce useful generalizations when applied to heterogeneous sets of partial solutions. Directly generalizing all partially successful programs does not yields informative sketches and does not outperform a baseline without learning. This result highlights the importance of constraining the generalization process.

Introducing program grouping prior to anti-unification improves the quality of learned sketches. Behavioral clustering, based on agreement over specification examples, reduces heterogeneity and increases the likelihood of extracting meaningful shared structure. Root-based grouping further enforces syntactic compatibility and proves particularly effective on benchmarks where high-level program shape is critical. Across the evaluated benchmarks, grouping strategies lead to more consistent improvements in synthesis performance than unstructured generalization.

A key advantage of the proposed approach lies in the immediate reuse of learned sketches during enumeration. In a standard size-based bottom-up iterator, programs with complex structure can only be explored once their full size is reached through gradual composition. By contrast, learned sketches introduce high-level structure early in the search process. Once a sketch is injected into the iterator state, its concrete instantiations can be explored immediately by filling holes with existing subprograms from the bank, rather than waiting for the iterator to reconstruct the same structure bottom-up. This effectively allows the search to jump ahead to structurally promising regions of the program space, substantially accelerating exploration when the sketch captures relevant structure. As a result, sketch learning improves synthesis efficiency not by expanding the budget or relaxing constraints, but by reordering exploration toward programs that would otherwise only be reached much later.

Overall, the results indicate that the effectiveness of sketch learning depends primarily on how partial solutions are selected and grouped. When properly constrained, sketch learning provides a lightweight and principled way to bias search toward promising regions of the program space within a single-task, budgeted synthesis setting.

Several promising directions remain for future research.

First, richer strategies for forming anti-unification groups could be explored. This thesis considers behavioral clustering and root-based grouping as independent mechanisms, each enforcing a single notion of similarity prior to generalization. However, these criteria could be combined hierarchically to form more meaningful groups. For example, programs could first be separated by their top-level grammar rule to ensure syntactic compatibility, and subsequently clustered by behavioral similarity within each root group. More generally, multi-stage or adaptive grouping strategies may allow anti-unification to operate on program sets that are both structurally aligned and behaviorally coherent, increasing the likelihood of extracting informative sketches while avoiding overly coarse generalizations.

Second, selective sketch admission offers an opportunity to further refine the learning process. In this thesis, the number of sketch-derived programs is bounded to preserve fairness between baseline and learning-based configurations. However, all admitted sketches are treated uniformly once introduced, regardless of their subsequent usefulness. In practice, sketches may vary substantially in how effectively they guide search, and their utility may change over time. Incorporating such utility-aware or diversity-aware criteria could help ensure that sketch learning remains effective throughout the synthesis process, rather than accumulating sketches that no longer contribute meaningful guidance.

Third, extending the approach beyond size-based bottom-up enumeration would be valuable. This thesis focuses on a size-based iterator, where progress is measured in terms of program size and enumeration proceeds in a uniform, breadth-first manner. While this setting provides a clear and controlled environment for studying sketch learning, other iterators in `Herb.jl` define fundamentally different notions of progress and state.

Cost-based or probabilistic iterators, for example, prioritize programs according to estimated cost, likelihood, or learned probabilities rather than structural size alone. Integrating sketch learning into such iterators would require rethinking how sketches influence search, including when sketch-derived programs should be introduced and how they should be ranked relative to non-sketch programs. Exploring these interactions may reveal new trade-offs between exploration and exploitation, as sketches could either concentrate probability mass on promising structures or prematurely bias the search away from alternative solutions. Studying sketch learning across different iterator designs would therefore help clarify which aspects of the proposed framework are iterator-agnostic and which depend on the underlying search strategy.

Together, these directions indicate that structural sketch learning offers a promising path toward improving the scalability and effectiveness of program synthesis under realistic resource constraints, while highlighting the importance of carefully integrating generalization into the search process.

References

[1] T. Kutsia, "Anti-unification in computer science," *Artificial Intelligence*, vol. 215, pp. 1–52, 2012.

[2] G. D. Plotkin, "A note on inductive generalization," *Machine Intelligence*, vol. 5, pp. 153–163, 1970.

[3] R. Shin, O. Polozov, and S. Gulwani, "Program synthesis by probing," in *Neural Information Processing Systems (NeurIPS)*, 2019.

[4] K. Ellis, L. Morales, A. Solar-Lezama, and J. Tenenbaum, "Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2021.

[5] J. Curto, K. Ellis, and J. B. Tenenbaum, "Knorf: Learning to reuse subroutines in program synthesis," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[6] K. Shi and N. Polikarpova, "Frangel: Component-based synthesis with control structures," in *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, 2019.

[7] M. Post and D. Gildea, "Bayesian learning of a tree substitution grammar," in *Proceedings of the 47th Annual Meeting of the Association for Computational Linguistics (ACL)*, Suntec, Singapore: Association for Computational Linguistics, 2009, pp. 45–53. [Online]. Available: https://aclanthology.org/P09-1006/.

[8] R. Shin et al., "Patois: Learning program idioms for synthesizing programs," in *Advances in Neural Information Processing Systems (NeurIPS)*, Curran Associates, Inc., 2019, pp. 9991–10 002. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[9] R. Rolim et al., "Mining idioms from code corpora," in *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2017, pp. 1–27.

[10] D. M. Cerna and T. Kutsia, *Anti-unification and generalization: A survey*, arXiv preprint (v5), last revised 3 Jun 2023, 2023. arXiv: 2302.00456 [cs.LO].

[11] D. Cao, R. Kunkel, C. Nandi, M. Willsey, Z. Tatlock, and N. Polikarpova, *BABBLE: Learning better abstractions with e-graphs and anti-unification*, Submitted on 8 Dec 2022, 2022. arXiv: 2212.04055 [cs.PL].