TECHNISCHE UNIVERSITEIT DELFT

MASTER OF SCIENCE THESIS IN COMPUTER SCIENCE

One Task at a Time: Task-Level Adaptation for Inductive Program Synthesis

Stef RASING

Supervisors:
MSc. Tilman HINNERICHS
Dr. Sebastijan DUMANČIĆ

31st August 2026



Delft University of Technology

One Task at a Time: Task-Level Adaptation for Inductive Program Synthesis

Master's Thesis in Computer Science

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

Stef Rasing

31st August 2026

Author
Stef Rasing

Title
One Task at a Time: Task-Level Adaptation for Inductive Program Synthesis

MSc presentation
9th September 2026

Graduation Committee

Dr. S. Dumančić (chair)	Delft University of Technology
Dr. S. Proksch	Delft University of Technology

Abstract

Existing Inductive Program Synthesis (IPS) techniques generate programs from examples and are restricted to specific task classes to exploit domain knowledge. To compete with general-purpose LLM-based code generation, a single IPS system must handle tasks with different types of inputs, outputs, and instructions without relying on domain-specific knowledge. This thesis introduces **task-level adaptation**, a framework in which a synthesizer must learn how to search only from the individual task it is solving rather than from a predefined domain. We focus specifically on search guided by automatically generated **properties** that capture useful aspects of a program's output. We introduce PHALCON to demonstrate the framework's viability. PHALCON repeatedly samples programs, selects properties that distinguish incorrect outputs from intended behavior, and uses them to constrain subsequent sampling. PHALCON outperforms state-of-the-art techniques specialized for string and bit-vector transformations, and solves tasks from the challenging Abstraction and Reasoning Corpus. These results establish task-level adaptation as a promising foundation for efficient, general program synthesis.

Preface

Like many research projects, this thesis took a very different direction from where it began. After an unsuccessful attempt to train a neural network using data from a single task, I spent weeks considering what constitutes a "good property." Tilman then suggested assuming that the properties were available and simply trying to search with them. This worked remarkably well, leading me to explore how weakening these properties could cause synthesis to fail, and what could be added to guide the synthesizer back toward a desired solution. This process ultimately gave rise to PHALCON.

I would especially like to thank my supervisors, Tilman Hinnerichs and Sebastijan Dumančić, for helping me shape this thesis into what it is today. Tilman, your feedback was always sharp and to the point; even when I presented a working demonstration, you remained suitably "suspicious". Most importantly, you encouraged me to focus on the "why" rather than only the "what," which was exactly what I needed.

Sebastijan, I have had the pleasure of knowing and working with you for the past five years. I am very grateful that our collaboration extended beyond my bachelor's and master's theses to my work as a research assistant and our literature survey. Your broad perspective and insightful ideas have been invaluable.

Finally, I would like to thank my friends and family for their support and companionship throughout my studies. I am especially grateful to both my former and current housemates. The "in-between" board games were a welcome break from studying, although occasionally, they became the entire study day. I also thank my friends from the association for providing much-needed relief on Thursdays, even if this sometimes resulted in slightly less productive Fridays.

Stef Rasing

Delft, The Netherlands
31st August 2026

Contents

1 Introduction **1**

2 Background **5**
2.1 Inductive Program Synthesis 5
2.2 Properties 7
2.3 Genetic Programming 9

3 Related Work **11**
3.1 Synthesis through Abstract Interpretation 11
3.2 Learning Heuristics with Properties 13

4 Problem Statement **15**
4.1 The Property Selection Problem 15
4.2 The Synthesis-with-Properties Problem 19

5 Methods **21**
5.1 Overview 21
5.2 Iterative Refinement through Conditional Sampling 22
5.3 Synthesis with Properties using Genetic Programming 24
5.4 The Complete PHALCON Algorithm 25

6 Experimental Evaluation **27**
6.1 Experimental Setup 27
6.2 Can PHALCON Compete with Existing Techniques? 30
6.3 Can PHALCON Discover Task-Specific Properties? 34
6.4 Can PHALCON's Properties Be Reused Across Tasks? 38

7 Conclusions and Future Work **43**

A Genetic Programming Implementation **49**

B ARC-AGI-1 Property Grammar **53**

C AI Statement **55**

Chapter 1

Introduction

The program synthesis community faces a crisis because its main use case is increasingly overshadowed by large language models [Austin et al., 2021, Daniotti et al., 2026]. Program synthesis [Gulwani et al., 2017a] automatically generates programs from descriptions of their intended behavior. Yet programmers increasingly turn to large language models (LLMs) to generate code because they can attempt almost any programming task.

Inductive program synthesis (IPS) [Summers, 1977] generates programs from examples, but is typically restricted to a particular class of tasks. Given several input-output pairs, an IPS system searches for a program that reproduces the demonstrated behavior. Such systems can be remarkably effective within their intended domain, but they cannot attempt tasks outside it.

For intuition, consider the following two tasks:

1. Abbreviating a Full Name

Input	Output
`John Doe`	`J Doe`
`Mayur Naik`	`M Naik`
`Nimit Singh`	`N Singh`

2. Counting Character Occurrences

Input 1	Input 2	Output
`Hannah`	`H`	1
`Hannah`	`a`	2
`Hannah`	`x`	0

Many systems solve the first task by exploiting its textual output. Heuristic search [Cropper and Dumančić, 2020] uses handcrafted rules to prioritize programs that produce similar outputs, neurally guided search [Odena et al., 2021, Shi et al., 2022, 2023, Odena and Sutton, 2020] uses a learned model to choose promising construction steps, and abstract interpretation [Cousot and Cousot, 1977, Cousot, 2002, Wang et al., 2018, Guria et al., 2023, Yoon et al., 2023] rule out groups of programs proven incapable of producing the required text. Despite their success on the first task, these systems cannot attempt the second, because its numerical output falls outside their textual focus. Ironically, both tasks come from the same test benchmark, which existing techniques restrict to tasks that fit their specialty.

To compete as an energy-efficient alternative to LLM-based code generation, a single IPS system must handle string transformations, numerical computations, visual reasoning problems, and other cross-domain tasks without being redesigned for each one.

Building such a system is difficult. Domain-specific knowledge helps synthesizers prioritize promising programs and reject unlikely ones. Without knowing the domain in advance, what knowledge can a synthesizer exploit to search efficiently?

Introducing Task-Level Adaptation

This thesis introduces **task-level adaptation**: a framework in which a synthesizer adapts to an individual task rather than an entire domain. By restricting the use of domain knowledge, a synthesizer can handle tasks with different input types, output types, and program instructions. The synthesizer must adapt to the task at hand. To our knowledge, no techniques currently exist within this framework.

Task-level adaptation might also avoid a weakness of domain adaptation: overgeneralization. When designing a system for an entire domain, we must decide in advance which information might be useful across its tasks, regardless of its relevance for individual tasks. For example, neurally guided methods designed for the text domain receive a simplified description of both the input and the output. One detail in this description records whether the output contains a capital letter. This may initially seem useful for text-processing tasks. Yet none of the tasks in the domain depend on capitalization, and the language in which programs are written does not even contain an instruction to change letter case. The system, therefore, considers information that cannot help it solve these tasks. The opposite problem also occurs: information essential to only a few tasks will be excluded because it does not appear useful enough for the domain as a whole. Task-level adaptation avoids both problems by determining what is useful for each task separately.

Guidance Through Properties

Learning from a single task is challenging; search guidance must capture abstract aspects of program behavior while being constructed solely from the task at hand. **Properties** are a sensible way to capture abstractions: each property records one particular detail about a program's output. For a textual output, a property might record its length or first character; for a numerical output, it might record whether the number is positive.

Properties also appear in abstract interpretation and neurally guided search, but those techniques are ill-suited for task-level adaptation. Abstract interpretation requires sufficiently precise properties and propagation rules. Neurally guided search trains a neural model to guide construction but requires more data than a single task provides. We instead use properties directly as inexpensive, automatically generated observations of concrete outputs. They provide a simpler view of program behavior that exposes useful differences between correct and incorrect out-

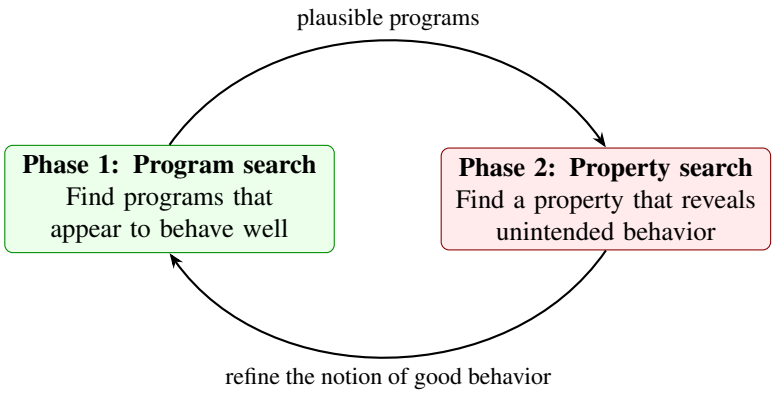


Figure 1.1: The two alternating phases of PHALCON: first, search programs expected to behave correctly; then, identify a property that explains why they do not.

puts without requiring either handcrafted propagation rules or task-specific neural training.

The main contribution of this thesis is a framework for **task-level adaptation through generated properties**. Within this framework, a synthesizer generates properties from the current task, selects those that distinguish the intended behavior from unpromising program behavior, and uses them to guide its search.

PHALCON

We demonstrate this framework's power through PHALCON. PHALCON operates in two alternating phases, shown in Figure 1.1. It iteratively refines its belief about what constitutes good behavior. This belief is represented by a collection of properties. In the first phase, PHALCON searches for programs whose behavior satisfies the properties discovered so far. In the second phase, it searches for a property that distinguishes these programs' behavior from the intended behavior. The newly discovered property is then used to refine PHALCON's characterization of good behavior for the next iteration. The process repeats until PHALCON finds a program that agrees with the task specification.

Our experiments show that PHALCON outperforms state-of-the-art techniques across all tested benchmarks. Notably, PHALCON almost completely solves the SyGuS string transformation domain and solves tasks from the notoriously difficult Abstraction and Reasoning Corpus (ARC).

PHALCON is not the primary contribution of this thesis, but a demonstration of the broader framework. It shows that we can learn useful search guidance within a single task, rather than engineering it for an entire domain. PHALCON's results suggest that task-level adaptation offers a promising foundation for general program synthesis: systems that retain the efficiency of symbolic search while gaining the ability to attempt a far wider range of tasks.

Chapter 2

Background

This chapter introduces the concepts needed to explain the framework of task-level adaptation through generated properties and how PHALCON operates. In Section 2.1, we first define inductive program synthesis (IPS). Then, Section 2.2 formalizes properties and their evaluation across the examples of an IPS task. Finally, Section 2.3 introduces genetic programming (GP), the search technique at the core of PHALCON.

2.1 Inductive Program Synthesis

Program synthesis [Gulwani et al., 2017b] aims to construct a program from a specification of its intended behavior. Such a specification can take many forms, including logical constraints, natural-language instructions, and input-output examples. This thesis focuses specifically on inductive program synthesis (IPS) [Summers, 1977], in which the intended behavior is demonstrated through input-output examples. We made this choice because we can only evaluate properties on concrete inputs and outputs.

An IPS task is a tuple (X, Y^*, G): X is a collection of n input examples, Y^* contains their corresponding target outputs, and the grammar G defines the space of possible programs.

The n input examples are

$$X = [x_1, x_2, \ldots, x_n],$$

where each example x_i consists of m arguments:

$$x_i = \left(x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(m)}\right).$$

The j-th argument has type $\tau^{(j)}$. The corresponding target outputs are

$$Y^* = [y_1^*, y_2^*, \ldots, y_n^*],$$

where every y_i^* is a single output value of type $\tau^{(o)}$.

Input $X^{(1)}$	Target output Y^*
`John Doe`	`J Doe`
`Mayur Naik`	`M Naik`
`Nimit Singh`	`N Singh`

Figure 2.1: The running IPS task. Each row is an input-output example. The desired program abbreviates the first name to its initial and preserves the surname.

The grammar G defines the language of candidate programs through a context-free grammar. Nonterminals correspond to inputs and constants, and its production rules describe the instructions that form the program. A solution is a program from this grammar that produces the required output for every example:

$$f^* \in G \quad \text{such that} \quad \forall i \in \{1, \ldots, n\},\ f^*(x_i) = y_i^*.$$

Figure 2.1 introduces the running task used throughout this thesis. The task takes a single string argument and requires a program to abbreviate the first name while preserving the surname. Later chapters refer back to this task.

The following context-free grammar defines the possible program space for the running task:

```
P (Program)   = S
 S (String)   = x(1)
              | "" | " "
              | concat(S,S)
              | replace(S,S,S)
              | index(S,I)
              | intToString(I)
              | if B then S else S
              | substring(S,I,I)
I (Integer)   = 1 | 0 | -1
              | I + I | I - I
              | length(S)
              | stringToInt(S)
              | if B then I else I
              | indexOf(S,S,I)
B (Boolean)   = True | False
              | I == I
              | prefixOf(S,S)
              | suffixOf(S,S)
              | contains(S,S)
```

A program performing the intended transformation must select the first character of the input and concatenate it with the substring beginning at the first space. In the grammar above, one such program is

```
concat(index(x(1), 1),
       substring(
           x(1),
           indexOf(x(1), " ", 1),
           length(x(1)))).
```

Evaluating this program on each input in Figure 2.1 produces its corresponding target output, solving the running task.

2.2 Properties

PHALCON guides synthesis using *properties* of program behavior. A property extracts one particular aspect of an output, such as the length of a string, its first character, or whether an integer is positive. Properties provide a simpler description of behavior than the complete output while preserving distinctions that may be useful during search.

Properties in much of the existing literature are Boolean-valued. Examples include whether a string is empty or whether an integer is positive. This representation is useful, but a single Boolean value can express only one binary distinction. Other values better represent some aspects of an output. For example, the length of a string is more directly represented by an integer than by a collection of predicates such as `length=1`, `length=2`, and so forth.

We therefore allow properties to return values of user-defined types. Let $\mathcal{U}$ denote the collection of property-value types supported by a particular instantiation of the synthesizer. This collection may contain types such as Booleans, integers, text, or even error values. The choice of $\mathcal{U}$ is not fixed, and users may introduce additional types appropriate for the values their grammar manipulates. We require only that we can compare values of the same type for equality.

A property may depend not only on the produced output, but also on the input that produces that output. Therefore, a property is a function of both the input and output. This context is useful, for example, for a property that records whether the output occurs within the input string.

Definition 1 (Property)**.** Let us denote the type of one input example as

$$\mathcal{X} = \tau^{(1)} \times \cdots \times \tau^{(m)},$$

and let $\tau^{(o)}$ be the output type. For a single task input x and output y, a **property** is a function

$$\pi : \mathcal{X} \times \tau^{(o)} \to U_\pi,$$

where $U_\pi \in \mathcal{U}$ is the property-value type associated with π. The result

$$\pi(x_i, y_i)$$

is called the **property value** of output y_i for input x_i.

We evaluate a property separately on each input-output example. We lift π pointwise to an output list

$$Y = [y_1, \ldots, y_n]$$

by defining

$$\pi(X, Y) = [\pi(x_1, y_1), \ldots, \pi(x_n, y_n)] \in U_\pi^n.$$

We call $\pi(X, Y)$ the **property-value vector** of Y. In particular, $\pi(X, Y^*)$ describes the property-value vector of the target outputs.

A candidate program f produces the output list

$$Y_f = [f(x_1), \ldots, f(x_n)].$$

We say that f **satisfies** property π when its property-value vector agrees with that of the target outputs:

$$\pi(X, Y_f) = \pi(X, Y^*).$$

A single property constrains only one aspect of the candidate's behavior. To express several constraints simultaneously, we collect properties in a set Π. We call the set of programs that agree with all properties in Π their agreement set.

Definition 2 (Agreement set)**.** Given a set of properties Π and a grammar G, the possibly infinite **agreement set** of Π is the set of programs in G that satisfy every property in Π:

$$A(\Pi) = \{f \in G \mid \forall \pi \in \Pi,\ \pi(X, Y_f) = \pi(X, Y^*)\}.$$

For example, consider the following three properties:

$$\pi_1(x, y) = \text{length}(y), \quad \pi_2(x, y) = \text{contains}(y, \texttt{" "}), \quad \pi_3(x, y) = \text{index}(y, 1).$$

The property-value vectors for the running task on the target outputs and the outputs of two candidate programs are as follows:

Output source	Output vector	$\pi_1(X, Y)$	$\pi_2(X, Y)$	$\pi_3(X, Y)$
Target outputs Y^*	[`J Doe`, `M Naik`, `N Singh`]	$[5, 6, 7]$	$[\top, \top, \top]$	[`J`, `M`, `N`]
f_1 : `" "`	[`" "`, `" "`, `" "`]	$[1, 1, 1]$	$[\top, \top, \top]$	[`" "`, `" "`, `" "`]
f_2 : $\text{index}(x^{(1)}, 1)$	[`J`, `M`, `N`]	$[1, 1, 1]$	$[\bot, \bot, \bot]$	[`J`, `M`, `N`]

The program f_1 : `" "` agrees with the targets on π_2 but not on π_1 and π_3. The program $f_2 : \text{index}(x^{(1)}, 1)$ agrees only on π_3.

2.3 Genetic Programming

PHALCON's core is genetic programming (GP) [Poli et al., 2007], an evolutionary search technique. GP maintains a population of N candidate programs

$$F \subseteq G \text{ with } |F| = N,$$

each generated from the grammar G. Initially, F contains N random programs up to a specified size. Rather than enumerating G systematically, GP focuses its search on programs that increasingly resemble the desired solution.

GP constructs M new candidate programs using two genetic operations. Crossover combines subexpressions from two parent programs $f_a, f_b \in F$:

$$f' = \text{crossover}(f_a, f_b).$$

Mutation slightly modifies a single program:

$$f' = \text{mutate}(f).$$

From the M candidate programs, we select the N programs with the lowest cost as the next generation. A program's cost measures how far its behavior is from the intended behavior. Because an IPS task contains multiple input-output examples, we measure cost separately for each example and sum their results. Let us denote the cost of the output produced by f on example i by

$$\phi\big(f(x_i), y_i^*\big) \in \mathbb{R}.$$

We define the cost of the complete program by summing these per-example costs:

$$\text{cost}(f) = \sum_{i=1}^{n} \phi\big(f(x_i), y_i^*\big).$$

Summing cost across examples is a design choice. No universally best aggregation rule exists: a candidate may perform well on some examples and poorly on others, and the aggregation determines how these trade-offs are valued. Summation is a simple choice that gives every example equal weight and is invariant to the order in which the examples are presented. It also preserves partial progress: improving the output for one example improves the program's aggregate cost, provided its behavior on the other examples remains unchanged.

For the string task in Figure 2.1, one could use the Levenshtein distance as a cost function:

$$\phi\big(f(x_i), y_i^*\big) = d_{\text{Lev}}\big(f(x_i), y_i^*\big).$$

The Levenshtein distance is the minimum number of character insertions, deletions, and substitutions required to transform one string into another. A program receives the optimal aggregate cost of 0 when it produces every target output:

$$\text{cost}(f) = 0 \quad \Longleftrightarrow \quad \forall i,\ f(x_i) = y_i^*.$$

This high-level account of GP is sufficient for understanding the contributions of this thesis. Appendix A describes the concrete genetic operators, selection mechanism, and synthesis loop.

Chapter 3

Related Work

Properties have previously been used to guide program synthesis in two main ways. First, abstract interpretation techniques use properties to summarize program behavior and prune candidate programs that cannot satisfy the specification. Second, neurally guided techniques use properties to describe synthesis tasks and rank intermediate values through a learned search heuristic.

Both approaches demonstrate that properties can provide effective search guidance. However, this thesis asks whether generated properties can provide such guidance within a single task. This chapter examines why existing property-based techniques are ill-suited for task-level adaptation.

We illustrate both branches using the running task from Figure 2.1. Recall that this task abbreviates a full name, for example, transforming `John Doe` into `J Doe`. Consider the following three Boolean properties:

$$\pi_1(x, y) = \text{length}(y) < \text{length}(x^{(1)}),$$
$$\pi_2(x, y) = \text{contains}(y, \texttt{" "}),$$
$$\pi_3(x, y) = \text{contains}(y, \texttt{"."}).$$

Thus, π_1 records whether the output is shorter than the input, π_2 records whether the output contains a space, and π_3 records whether it contains a period. For every target output in the running task, π_1 and π_2 are true, whereas π_3 is false. The following sections show how the two branches of related work use this information differently.

3.1 Synthesis through Abstract Interpretation

Input-output examples specify the required behavior only at a collection of concrete points. A synthesizer could evaluate each candidate program on these points and reject it individually when it produces an incorrect result. However, this provides no direct way to reject other programs with the same underlying behavioral violation.

Abstract interpretation [Cousot, 2002, Cousot and Cousot, 1977] addresses this limitation by reasoning about an abstract representation of program behavior. Rather than tracking exact values, an abstraction records selected facts about those values. For example, a string abstraction may record whether a string is shorter than its input, contains a space, or contains a period. Multiple concrete strings can consequently share the same abstract description.

Using the three properties introduced above, the target outputs of the running task have the abstract description

$$\alpha(X, Y^*) = \{\pi_1, \pi_2, \neg\pi_3\}.$$

The presence of π_1 and π_2 indicates that these properties hold for every example. In contrast, $\neg\pi_3$ indicates that π_3 is false for every example: none of the target outputs contains a period. If the abstraction cannot determine whether a property is true or false, neither the property nor its negation appears in the set. Absence, therefore, denotes unknown information rather than falsehood.

The main advantage of this representation is that abstract interpretation can eliminate families of programs rather than test and reject each member separately. Consider the family

```
concat(".", S),
```

where any string-producing expression may replace `S`. Every program in this family produces an output containing a period, regardless of the expression substituted for `S`. Its abstract output therefore contains π_3. Because the desired abstract output contains $\neg\pi_3$, the entire family can be pruned without enumerating its individual members.

To reason in this way, the synthesizer requires an *abstract semantics* for the grammar. The ordinary semantics of a grammar rule describes how it acts on concrete values, whereas its abstract semantics describes how it transforms abstract facts. For concatenation, for example, the abstract semantics must establish which length and character-presence properties are guaranteed for the result, given the abstract descriptions of its arguments. These rules must be sound. An abstract rule may be unable to determine whether a property holds, in which case it includes neither the property nor its negation in the abstract description. It may include π only when the property is guaranteed to hold and $\neg\pi$ only when the property is guaranteed not to hold.

Existing synthesis techniques use abstract semantics in several ways. In top-down synthesis [Guria et al., 2023], a partial program can be discarded when every completion it represents has incompatible abstract behavior, as in the provided example. Bidirectional synthesis [Yoon et al., 2023] combines this top-down reasoning with information obtained from complete subexpressions generated through bottom-up enumeration. BLAZE [Wang et al., 2018] instead iteratively refines an automaton representing abstractly correct programs.

Limitations for Task-Level Adaptation

These techniques show that abstraction can provide powerful pruning, but that power depends on two closely related design decisions. First, the properties must expose distinctions relevant to the target programs. Second, the abstract semantics must propagate those distinctions precisely enough to rule out large parts of the search space. A property may describe the targets accurately yet provide little pruning if the abstract semantics cannot determine whether program families satisfy it. Conversely, a precisely propagated property is of little use when it does not distinguish promising programs from the rest of the search space.

In existing systems, domain experts therefore design both the properties and their abstract semantics. Even carefully designed abstractions do not always prune substantially. Constructing a sufficiently precise abstraction is difficult. Automating this process for every individual task would require generating useful properties and deriving sound, effective abstract semantics for all grammar rules during synthesis. This is not feasible within the time available for task-level adaptation. We therefore do not pursue abstract interpretation as the mechanism for exploiting generated properties.

3.2 Learning Heuristics with Properties

A second branch of work uses properties as inputs to neural search heuristics. Neural networks can learn which program components are commonly useful for particular kinds of tasks, but neural networks require a fixed-size representation of the intermediate outputs encountered during search. Property signatures [Odena and Sutton, 2020] provide such a representation by summarizing concrete values through Boolean properties.

A property signature evaluates each property across all examples and aggregates the resulting Boolean values. For a property π, the aggregation is

$$\text{aggregate}(\pi) = \begin{cases} \texttt{AllTrue}, & \text{if } \pi \text{ is true for every example,} \\ \texttt{AllFalse}, & \text{if } \pi \text{ is false for every example,} \\ \texttt{Mixed}, & \text{otherwise.} \end{cases}$$

The signature is the ordered list of these aggregate values for a fixed collection of properties.

If we construct a property signature from the example properties π_1, π_2, and π_3, the target outputs of the running task produce the signature

$$\text{sig}(X, Y^*) = [\texttt{AllTrue}, \texttt{AllTrue}, \texttt{AllFalse}].$$

The first entry indicates that every target output is shorter than its corresponding input. The second indicates that every target contains a space, and the third indicates that none contains a period.

Signatures can also describe the values that subexpressions produce during bottom-up search. Let the values produced by a subexpression f be

$$Y_f = [f(x_1), \ldots, f(x_n)].$$

We can compute a signature from these intermediate values to the target outputs:

$$\mathrm{sig}(Y_f, Y^*).$$

A neurally guided heuristic h_θ computes

$$h_\theta\left(\mathrm{sig}(X, Y^*), \mathrm{sig}(Y_f, Y^*)\right) \in [0, 1].$$

This value estimates the likelihood that the subexpression f appears in a solution program. The synthesizer uses this estimate as a heuristic for prioritizing subexpressions during bottom-up search.

The neural network is trained on randomly generated programs and inputs. Each generated program provides a known solution, and evaluating it on the generated inputs produces an input-output task. Subexpressions that occur in the generated program can then serve as positive training examples. From this data, the network learns how relationships between task signatures and subexpression signatures correlate with membership in a solution. The signatures expose semantic information about the generated task and subexpressions, while the network learns how to use that information to guide search.

Several synthesis systems use property signatures in this way. BUSTLE [Odena et al., 2021] uses a comparatively small model whose inputs are based on property signatures. CROSSBEAM [Shi et al., 2022] uses a larger model that additionally represents previously generated subexpressions. LAMBDABEAM [Shi et al., 2023] adapts this approach to synthesize programs containing lambda expressions.

Limitations for Task-Level Adaptation

Like abstract-interpretation techniques, these methods normally rely on a collection of properties chosen by the designer. It may be possible to automate this part by generating the properties that constitute a signature. Doing so, however, does not remove the need to learn how signatures should guide search. The heuristic is trained across large collections of synthesis tasks: BUSTLE uses hundreds of thousands of data points, whereas CROSSBEAM uses millions. Such training is viable when its cost can be amortized across an entire domain, but not when a new model must be trained for every individual task. It is therefore unsuitable for the form of task-level adaptation considered in this thesis.

Chapter 4

Problem Statement

The preceding chapter showed that existing property-based synthesis techniques are not suitable for task-level adaptation. Abstract interpretation requires properties together with sound abstract semantics designed for the grammar, while neurally guided synthesis requires a search heuristic trained across many data points. A task-level adaptive synthesizer can perform neither form of adaptation within a single task. We must therefore devise a new way to construct and exploit search guidance using only the individual IPS task being solved.

This chapter introduces *task-level adaptation through generated properties*, the thesis's main contribution. This framework consists of two parts. First, the synthesizer must automatically generate candidate properties and select those that provide useful guidance. Second, it must search for programs whose outputs exhibit the target behavior with respect to the selected properties. This chapter formalizes both problems.

4.1 The Property Selection Problem

Recall that an IPS task is a tuple (X, Y^*, G), where X contains the inputs, Y^* contains their target outputs, and G is a grammar defining the candidate programs. Task-level adaptation lets the synthesizer use these three components for the current task, but not guidance designed or learned for a domain of related tasks. In particular, the properties used for guidance must be constructed from the current task rather than supplied as a predefined domain-specific collection.

Generating Candidate Properties

To select task-specific properties, we first need a sufficiently varied collection of candidates. We avoid using separate domain knowledge and instead reuse the task's grammar. The grammar already contains rules for its inputs, constants, and operations. It does not, however, contain an expression representing the output whose behavior a property must observe. We therefore extend G with a dedicated output

symbol y of type $\tau^{(o)}$:

$$G_\pi = G \cup \{y : \tau^{(o)}\}.$$

We retain only expressions from G_π that contain y and whose result type belongs to $\mathcal{U}$. These constraints ensure that properties depend on y and are well-typed.

We enumerate expressions from G_π bottom-up to a fixed maximum size. The size of an expression is the number of operators it contains, which approximates its computational complexity. Constants do not increase this size because they do not add computation. The resulting finite collection is denoted by $\Pi_{\text{candidate}}$.

This construction uses only information already present in the current IPS task. It does depend on the user-defined collection of property-value types $\mathcal{U}$, which we define to contain every type in the property grammar. Consequently, it can produce properties for different input and output types without requiring a separately engineered property family for each domain.

Measuring Agreement and Information

A property agrees with a candidate program when the program's outputs exhibit the same behavior as the target outputs with respect to that property. Because the grammar may contain infinitely many programs, we cannot measure agreement over the complete program space. We instead measure agreement relative to a finite collection of sampled programs.

Definition 3 (Agreement rate)**.** Let $F \subseteq G$ be a nonempty finite collection of sampled programs and let $\Pi \subseteq \Pi_{\text{candidate}}$. The **agreement rate** of Π with respect to F is

$$\alpha_F(\Pi) = \frac{|\{f \in F \mid \forall \pi \in \Pi : \pi(X, Y_f) = \pi(X, Y^*)\}|}{|F|}.$$

Equivalently,

$$\alpha_F(\Pi) = \frac{|\mathcal{A}(\Pi) \cap F|}{|F|}.$$

The agreement rate is the portion of sampled programs whose outputs exhibit the target behavior with respect to every property in Π. A low agreement rate indicates that the selected properties distinguish the target outputs from a large portion of the sampled program behavior.

Suppose that a program is drawn uniformly from F. Its probability of agreeing with the target behavior for every property in Π is precisely $\alpha_F(\Pi)$. We measure the information conveyed by observing this agreement using its negative log-likelihood.

Definition 4 (Property information)**.** The **information** of a property set Π with respect to F is

$$I_F(\Pi) = -\log_2 \alpha_F(\Pi).$$

If $\alpha_F(\Pi) = 0$, we define

$$I_F(\Pi) = +\infty.$$

Properties

$$\pi_1(x, y) = \text{contains}(y, \texttt{"."}), \qquad \pi_2(x, y) = \text{index}(y, 1).$$

The target vectors are

$$\pi_1(X, Y^*) = [\bot, \bot, \bot], \qquad \pi_2(X, Y^*) = [\texttt{J}, \texttt{M}, \texttt{N}].$$

Program f	Output vector Y_f	$\pi_1(X, Y_f)$	$\pi_2(X, Y_f)$
`"."`	[., ., .]	$[\top, \top, \top]$	[., ., .]
`" "`	[" ", " ", " "]	$[\bot, \bot, \bot]$	[" ", " ", " "]
toString(1)	[1, 1, 1]	$[\bot, \bot, \bot]$	[1, 1, 1]
index($x^{(1)}$, 1)	[J, M, N]	$[\bot, \bot, \bot]$	[J, M, N]
substr($x^{(1)}$, 1, 5)	["John ", "Mayur", "Nimit"]	$[\bot, \bot, \bot]$	[J, M, N]

Program f	$f \in \mathcal{A}(\{\pi_1\})$	$f \in \mathcal{A}(\{\pi_2\})$	$f \in \mathcal{A}(\{\pi_1, \pi_2\})$
`"."`	No	No	No
`" "`	Yes	No	No
toString(1)	Yes	No	No
index($x^{(1)}$, 1)	Yes	Yes	Yes
substr($x^{(1)}$, 1, 5)	Yes	Yes	Yes
Agreement rate $\alpha_F(\cdot)$	$\frac{4}{5}$	$\frac{2}{5}$	$\frac{2}{5}$
Information $I_F(\cdot)$	$-\log_2 \frac{4}{5} \approx 0.32$	$-\log_2 \frac{2}{5} \approx 1.32$	$-\log_2 \frac{2}{5} \approx 1.32$

Figure 4.1: Agreement rates and information of two properties over five sampled programs. Property π_1 agrees with four programs, whereas π_2 agrees with two. The same two programs agree with both properties, so their combined agreement rate is $\frac{2}{5}$, and their combined information is approximately 1.32 bits.

As the agreement rate decreases, agreement with the selected properties becomes less likely and hence more informative. If no sampled program agrees with all selected properties, the agreement rate is zero, and the information is infinite.

Figure 4.1 illustrates the agreement rate and information using five programs sampled for the running task and two exemplar properties.

Marginal Contribution

A useful property set should not contain properties that merely reproduce distinctions already made by the other selected properties. We measure a property's contribution by the additional information it provides when added to the remaining selected properties.

Definition 5 (Marginal information contribution)**.** Let Π be a property set such that

$\alpha_F(\Pi) > 0$, and let $\pi \in \Pi$. The **marginal information contribution** of π to Π is

$$\Delta_F(\pi \mid \Pi) = I_F(\Pi) - I_F\big(\Pi \setminus \{\pi\}\big).$$

Equivalently,

$$\Delta_F(\pi \mid \Pi) = \log\left(\frac{\alpha_F\big(\Pi \setminus \{\pi\}\big)}{\alpha_F(\Pi)}\right).$$

The marginal contribution is nonnegative because adding a property cannot increase the agreement rate. It is zero precisely when removing π does not change the agreement rate:

$$\alpha_F\big(\Pi \setminus \{\pi\}\big) = \alpha_F(\Pi).$$

In that case, π makes no additional distinction among the sampled programs and is redundant with respect to the remaining properties. We prefer property sets in which every selected property makes a substantial contribution.

Consider the properties in Figure 4.1 again, with $\Pi = \{\pi_1, \pi_2\}$. Removing π_1 leaves the joint information unchanged at 1.32. Hence,

$$\Delta_F(\pi_1 \mid \Pi) = I_F(\Pi) - I_F(\{\pi_2\}) = 1.32 - 1.32 = 0.$$

In contrast, removing π_2 decreases the information from 1.32 to 0.32, giving

$$\Delta_F(\pi_2 \mid \Pi) = I_F(\Pi) - I_F(\{\pi_1\}) = 1.32 - 0.32 = 1.$$

Thus, π_1 is redundant with respect to π_2, while π_2 contributes one additional bit of information to π_1.

Avoiding Overly Restrictive Property Sets

A property set should be informative, but it should not constrain the target behavior so precisely that no sampled program satisfies all of its properties. Such a set distinguishes the target from every sampled program, but provides no attainable intermediate guidance.

For example, consider the property

$$\pi_{\text{exact}}(x, y) = y.$$

Agreement with this property requires a candidate program to produce the complete target output vector:

$$\pi_{\text{exact}}(X, Y_f) = \pi_{\text{exact}}(X, Y^*) \quad \Longleftrightarrow \quad Y_f = Y^*.$$

If the sample F contains no exact solution, then

$$\alpha_F(\pi_{\text{exact}}) = 0 \qquad \text{and} \qquad I_F(\pi_{\text{exact}}) = +\infty.$$

Although the singleton set $\{\pi_{\text{exact}}\}$ is maximally informative under our measure, it merely restates the original synthesis objective. It does not identify a coarser aspect of the target behavior that candidate programs can satisfy before they produce the exact target outputs. We therefore require the selected set to have a nonzero agreement rate:

$$\alpha_F(\Pi) > 0.$$

Consequently, every property in the selected set must also have a nonzero agreement rate:

$$\forall \pi \in \Pi,\ \alpha_F(\{\pi\}) > 0.$$

The Property Selection Problem

We thus seek a property set with two characteristics. First, the set should have a low but nonzero agreement rate, so that it distinguishes the target behavior from many sampled programs. Second, each selected property should contribute non-redundantly to the set's agreement rate.

Definition 6 (Property selection problem)**.** Given a collection of candidate properties $\Pi_{\text{candidate}}$ and a nonempty finite collection of sampled programs $F \subseteq G$, the **property selection problem** is to select a set $\Pi \subseteq \Pi_{\text{candidate}}$ whose agreement rate $\alpha_F(\Pi)$ is as small as possible, subject to

$$\alpha_F(\Pi) > 0$$

and

$$\Delta_F(\pi \mid \Pi) > 0 \qquad \text{for every } \pi \in \Pi.$$

Minimizing the joint agreement rate maximizes the information provided by the selected properties. Requiring a nonzero joint agreement rate avoids overly restrictive property sets, while requiring a positive marginal contribution ensures that every selected property makes a distinction not already made by the remaining properties.

4.2 The Synthesis-with-Properties Problem

After selecting properties, the synthesizer must use them to find programs with the required abstract behavior. Unlike abstract interpretation, the setting considered here does not assume abstract semantics that can propagate properties through partial programs. Unlike neural guidance, it does not assume a model trained to predict which constructions will satisfy them. The selected properties are instead requirements that can be checked by executing a complete candidate program on X and comparing its property-value vectors with those of Y^*.

Definition 7 (Synthesis-with-Properties Problem)**.** Given an IPS task (X, Y^*, G) and a set of properties Π, find one or more programs

$$F_\Pi \subseteq G$$

such that

$$\forall f \in F_\Pi,\ \forall \pi \in \Pi, \qquad \pi(X, Y_f) = \pi(X, Y^*).$$

Equivalently, the returned programs must satisfy

$$F_\Pi \subseteq \mathcal{A}(\Pi).$$

Satisfying the selected properties does not imply satisfying the complete IPS specification. Properties retain only selected aspects of behavior, meaning that $\mathcal{A}(\Pi)$ may contain programs that produce outputs different from the targets. Every returned program must therefore still be checked against the original target outputs. If none is an exact solution, its behavior can provide further evidence for selecting additional properties.

Together, the two problems describe the central challenge of task-level adaptation through generated properties. Property selection determines which aspects of the target behavior are informative for the current task, and synthesis with properties turns those aspects into a population of better-directed candidate programs.

Chapter 5

Methods

The previous chapter formulated task-level adaptation through generated properties as two problems. The *property selection problem* asks which properties distinguish the target outputs from unpromising program behavior. The *synthesis-with-properties problem* asks for programs whose outputs agree with the target outputs on the selected properties. This chapter presents PHALCON, a method that alternates between the solutions to these two problems.

PHALCON does not select a complete property set before synthesis. Instead, it begins without guidance and searches for a program satisfying the task specification. If that search does not produce a solution, the programs encountered during the search reveal how the current search differs from the intended behavior. PHALCON selects one property that captures such a difference and searches again, now rewarding programs that satisfy every property selected so far. This way, PHALCON iteratively refines its notion of good behavior until it finds a solution.

5.1 Overview

Figure 5.1 summarizes the method. Its input is an IPS task (X, Y^*, G) together with parameters for property generation and genetic programming (GP). First, PHALCON extends the task grammar with the output symbol y and enumerates the finite set of candidate properties $\Pi_{\text{candidate}}$ as described in Section 4.1. It then initializes the set of selected properties to $\Pi_0 = \varnothing$.

At iteration t, GP searches for programs that agree with Y^* on every property in Π_t. If it finds a solution, PHALCON returns it immediately. Otherwise, the population produced by GP is used to select an additional property π_t, which is then added to the set of selected properties. The enlarged property set is used for the objective for the next GP run. The loop continues until an exact solution is found or a termination condition is met.

The following sections describe the two alternating components. We first explain how repeated conditional sampling discriminates observed behavior from intended behavior. We then define how GP is used for property-based synthesis.

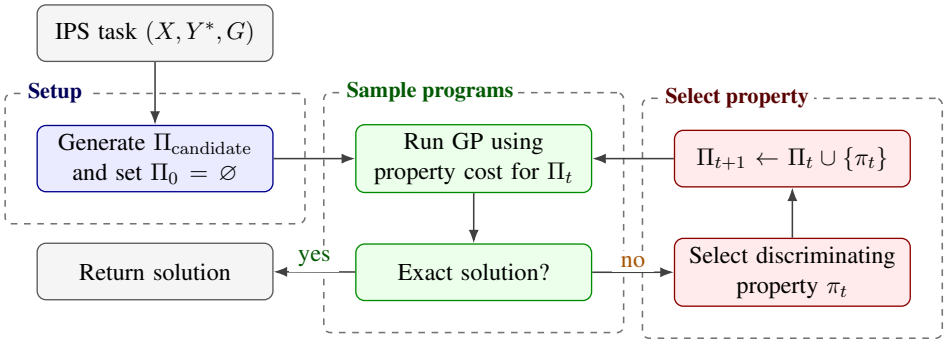


Figure 5.1: Overview of the PHALCON synthesis algorithm. During setup, PHALCON generates the candidate properties and initializes the selected property set. It then uses GP to sample programs satisfying the current property set. If it finds no exact solution, it selects a new property from the sampled behavior, extends the selected property set, and repeats the sampling phase.

5.2 Iterative Refinement through Conditional Sampling

Sampling without guidance produces many programs whose behavior is unrelated to the task. Properties for such a sample may distinguish arbitrary behavior from the target, but reveal little about the boundary between ideal and almost-ideal behavior. Instead, PHALCON alternates between two stages:

1. sample programs satisfying the current notion of good behavior, as defined by the selected property set; and
2. use the sampled programs to refine the notion of good behavior by discriminating actual behavior from intended behavior.

At iteration t, the selected property set Π_t defines the current agreement set

$$\mathcal{A}(\Pi_t) = \{f \in G \mid \forall \pi \in \Pi_t : \pi(X, Y_f) = \pi(X, Y^*)\}.$$

The current agreement set contains programs that we currently believe exhibit good behavior. Using GP, we obtain a sample $F_t \subseteq \mathcal{A}(\Pi_t)$. For each candidate property, PHALCON measures its agreement rate relative to this sample:

$$\widehat{\alpha}_{F_t}(\pi) = \frac{|\{f \in F_t \mid \pi(X, Y_f) = \pi(X, Y^*)\}|}{|F_t|}$$

and selects a property with the smallest nonzero agreement rate:

$$\textsc{SelectProperty}(F_t, \Pi_{\text{candidate}}, X, Y^*) = \operatorname*{arg\,min}_{\substack{\pi \in \Pi_{\text{candidate}} \\ \widehat{\alpha}_{F_t}(\pi) > 0}} \widehat{\alpha}_{F_t}(\pi).$$

Consider again the five programs in Figure 4.1. Suppose the first round selects

$$\pi_1(x, y) = \text{index}(y, 1), \qquad \pi_1(X, Y^*) = [\texttt{J}, \texttt{M}, \texttt{N}].$$

The next synthesis run is guided toward programs in $\mathcal{A}(\{\pi_1\})$ and might produce the following batch:

Program f	Output vector Y_f	$\pi_2(X, Y_f)$
$\text{substr}(x^{(1)}, 1, 6)$	`["John D","Mayur ","Nimit "]`	$[5, 6, 6]$
$\text{substr}(x^{(1)}, 1, 7)$	`["John Do","Mayur N","Nimit S"]`	$[5, 6, 6]$
$\text{concat}(\text{substr}(x^{(1)}, 1, 3), \texttt{" "})$	`["Joh ","May ","Nim "]`	$[4, 4, 4]$
$\text{concat}(\text{substr}(x^{(1)}, 1, 2), \texttt{" "})$	`["Jo ","Ma ","Ni "]`	$[3, 3, 3]$
$\text{concat}(\text{index}(x^{(1)}, 1), \texttt{" "})$	`["J ","M ","N "]`	$[2, 2, 2]$

Every program in this batch satisfies π_1, since its outputs begin with the correct character. The candidate property

$$\pi_2(x, y) = \text{indexOf}(y, \texttt{" "})$$

has a target vector

$$\pi_2(X, Y^*) = [2, 2, 2]$$

and agrees only with the last program. Its conditional agreement rate is therefore

$$\widehat{\alpha}_{F_1}(\pi_2) = \frac{1}{5}.$$

Selecting π_2 refines the current notion of good behavior: outputs must not only begin with the correct character but also include a space immediately after.

Idealized Information Analysis

We can quantify the benefit of conditional resampling under idealized assumptions. The lowest agreement rate for a single unconditional sample of N programs is

$$\widehat{\alpha}_F(\Pi) = \frac{1}{N},$$

which contains

$$I_F(\Pi) = -\log_2 \widehat{\alpha}_F(\Pi) = -\log_2 \frac{1}{N} = \log_2 N$$

bits of information. If we divide the same number of samples into $K = \frac{N}{B}$ batches of size B, the maximum accumulated information is

$$K \log_2 B = \frac{N}{B} \log_2 B.$$

Notice that the amount of information grows linearly with N for conditional batch sampling, rather than logarithmically for single sampling.

The enormous advantage of conditional sampling is best shown through an example. One batch of 100 samples provides at most

$$\log_2 100 \approx 6.64$$

bits. Five conditional batches of size 20 can instead yield up to

$$5 \log_2 20 \approx 21.61$$

bits. The same amount of information from a single batch would require approximately

$$2^{21.61} \approx 3.2 \times 10^6$$

programs. Thus, under these idealized assumptions, conditional resampling can extract as much information from 100 sampled programs as from an unconditional batch of approximately 3.2 million programs.

This comparison is an upper bound, not a guarantee. It assumes that each batch is drawn from the current agreement set and that exactly one program satisfies the selected property. In practice, agreement rates may be higher, and GP may fail to satisfy all properties. Nevertheless, the analysis captures the central advantage: each iteration of PHALCON refines a progressively more task-relevant region of the search space.

5.3 Synthesis with Properties using Genetic Programming

After selecting properties, PHALCON must produce a new sample whose outputs exhibit the corresponding target behavior. It uses genetic programming (GP) for this purpose. GP searches directly over complete executable programs and requires only a numerical cost function.

The GP population size equals the desired sample size. Moreover, PHALCON maintains output diversity: no two programs in the population may produce the same output vector. This prevents semantically equivalent programs from occupying multiple sample positions without providing additional evidence for property selection.

An Example-Wise Property Cost

Consider an IPS task that contains n input-output examples, $X = (x_1, \ldots, x_n)$ and $Y^* = (y_1^*, \ldots, y_n^*)$. For a program f and a selected property set Π, PHALCON defines the property cost

$$\mathrm{cost}_\Pi(f) = \sum_{\pi \in \Pi} \sum_{i=1}^{n} \mathbf{1}\left[\pi(x_i, f(x_i)) \neq \pi(x_i, y_i^*)\right].$$

GP minimizes this cost. A cost of zero means that the program belongs to $\mathcal{A}(\Pi)$. Once the entire population has zero cost, it forms the desired sample for the selected property set. When $\Pi = \varnothing$, every program has zero cost, so the initial population makes the first sample.

The cost function counts disagreements separately for each example. Under the formal definition of agreement, a property is unsatisfied as soon as it disagrees on at least one example. Using this binary criterion directly as cost would make a program that agrees on all but one example indistinguishable from one that agrees on none. Example-wise cost instead gives GP partial credit and lets it approach the required behavior one example at a time.

Feedback from Unsuccessful Search

GP is stochastic and may fail to produce a population that satisfies every selected property. In that case, the population is not a conditional sample from $\mathcal{A}(\Pi_t)$. Nevertheless, its distinct output behaviors provide additional feedback about the properties that remain unsatisfied. PHALCON uses this population to select properties and provides additional guidance for the next run.

GP Termination

A GP run terminates under three conditions. First, if GP finds an exact solution to the input-output examples, PHALCON immediately returns that program. Second, the run terminates when every member of the population has zero cost; it then forms the next conditional sample. Third, the run terminates after a fixed number of generations without cost improvement; in that case, the current population is a sample that does not satisfy all properties.

5.4 The Complete PHALCON Algorithm

Algorithm 1 combines iterative property selection with GP-based synthesis. The selected property set is initially empty. Each run either finds an exact solution or returns the final population. This population is then used to select a property with the smallest positive agreement rate. The selected property is added to the fitness objective for the next run. After a user-defined number of property-selection iterations, PHALCON performs one final GP run using the complete set of properties selected during the preceding iterations. No further property is selected after this run. If the final run does not find an exact solution, PHALCON terminates and reports failure.

Algorithm 1 PHALCON

Require: IPS task (X, Y^*, G),
Property generation parameters θ_π,
GP parameters θ_{GP},
Maximum number of iterations N
Ensure: An exact solution, or FAILURE

$\Pi_{\mathrm{candidate}} \leftarrow$ GENERATEPROPERTIES(G, X, Y^*, θ_π)
$\Pi \leftarrow \varnothing$
while $|\Pi| < N$ **do**
    $(f^*, F) \leftarrow \mathrm{GP}(X, Y^*, G, \Pi, \theta_{\mathrm{GP}})$ ▷ f^*: solution or $\perp$; F: population
    if $f^* \neq \perp$ **then**
        return f^*
    end if
    $\pi \leftarrow$ SELECTPROPERTY$(F, \Pi_{\mathrm{candidate}}, X, Y^*)$
    $\Pi \leftarrow \Pi \cup \{\pi\}$
end while
$(f^*, F) \leftarrow \mathrm{GP}(X, Y^*, G, \Pi, \theta_{\mathrm{GP}})$ ▷ Final GP run
if $f^* \neq \perp$ **then**
    return f^*
end if
return FAILURE

Chapter 6

Experimental Evaluation

We introduce task-level adaptation as an alternative to domain-level adaptation; therefore, it should achieve competitive synthesis performance without relying on domain-specific guidance. We further argue that task-level adaptation can overcome the over-generalization inherent in domain-level guidance by identifying properties specific to the task being solved, rather than using a fixed collection of properties intended to apply across an entire domain. Finally, we investigate whether properties discovered for previous tasks can be transferred to new tasks to reduce the number of refinement iterations required.

This chapter evaluates whether PHALCON meets these expectations. We organize the evaluation around three research questions.

$\mathcal{Q}_1$: Can PHALCON compete with domain-level adaptation?

$\mathcal{Q}_2$: Does PHALCON discover task-specific properties?

$\mathcal{Q}_3$: Can properties discovered for one task be reused for other tasks?

We first describe the common experimental setup, including the benchmarks, parameter settings, and execution environment. The remainder of the chapter addresses each research question separately. For every question, we present the relevant experiment, analyze its results, and state the conclusion supported by those results.

6.1 Experimental Setup

We evaluate PHALCON on three different benchmarks: the string and bit-vector tracks of the Syntax-Guided Synthesis (SyGuS) competitions [Padhi et al., 2019], and the training set of the Abstraction and Reasoning Corpus (ARC-AGI-1) [Chollet, 2019]. The two SyGuS benchmarks allow comparison with established domain-specific synthesis methods. ARC-AGI-1 evaluates whether the same task-level adaptation mechanism can operate in a more difficult setting, with a substantially larger program space.

Benchmarks

SyGuS strings. The SyGuS string benchmark consists of string transformation tasks. Each task includes a grammar with a shared set of basic string operations and task-specific constants.

The original benchmark contains groups of closely related tasks that differ only in the number of examples. Including all such variants would disproportionately weight essentially the same transformation. We therefore remove these redundant variants, leaving 124 distinct string-transformation tasks.

SyGuS bit-vectors. The SyGuS bit-vector benchmark consists of tasks over fixed-width bit-vectors. These tasks require programs to express transformations using logical, arithmetic, shift, and conditional operations. As in the string benchmark, each task supplies a grammar containing both generally available operations and task-specific rules or constants.

Again, some benchmark instances are closely related and differ primarily in the number or repetition of their input-output examples. To limit this redundancy and control the cost of evaluating candidate programs, we retain tasks containing at most ten input-output examples. After this filtering step, the evaluation contains 151 bit-vector tasks.

ARC-AGI-1. ARC-AGI-1 consists of transformation tasks on two-dimensional colored grids. Tasks may require identifying objects, detecting spatial relationships, extending patterns, copying or moving shapes.

Unlike the SyGuS benchmarks, ARC-AGI-1 does not provide a program grammar. We therefore use a publicly available general-purpose ARC-AGI-1 grammar [Hodel, 2023]. This grammar contains a large collection of instructions and intermediate types for manipulating grids, objects, coordinates, integers, and collections. We use the same grammar across the ARC-AGI-1 tasks. This makes the search space considerably larger than the two SyGuS benchmarks, but it provides a stronger test of whether task-specific properties can guide search in a vast program space.

Our evaluation uses all 400 tasks in the ARC-AGI-1 training partition. We use the training partition because its tasks span a wide range of synthesis difficulty and are less difficult than those in the evaluation partition. Some transformations can be expressed using only a small number of grammar instructions, whereas others require programs containing tens of instructions.

Parameter Settings

Table 6.1 summarizes the parameters used. Apart from two exceptions, we use a common configuration across all benchmarks.

Parameter	Value	Exception
GP population size	20	
Candidate population size per generation	2000	10000 for ARC-AGI-1
Maximum stagnant generations	10	
Sample size	20	
Maximum selected properties	20	
Maximum property size	3	4 for bit-vectors
Repetitions per task	5	
Nominal timeout per run	30 minutes	

Table 6.1: Parameters used in the experiments.

Genetic-programming parameters. The GP population contains 20 programs and also serves as the sample for property selection. This keeps property evaluation inexpensive while retaining a useful range of program behavior. Each generation considers 2000 candidate programs for strings and bit-vectors and 10000 for ARC, whose larger search space requires broader exploration. A GP run terminates after 10 generations without fitness improvement.

PHALCON parameters. The sample size equals the GP population size of 20. PHALCON selects at most 20 properties, although successful runs generally require fewer. The maximum property size is 3 for strings and ARC, and 4 for bit-vectors. These are the largest property spaces that we can generate under a minute.

Property grammars. For the SyGuS benchmarks, we generate properties from each task's grammar. For ARC-AGI-1, we exclude selected rules from property generation to enumerate deeper properties efficiently. We primarily retain rules that extract or summarize parts of their inputs, guided by the intuition that properties should capture smaller aspects of the output. This restriction applies only to property generation; program synthesis still uses the complete grammar. We provide more details in Appendix B.

Execution environment We implemented PHALCON in Julia and ran all experiments on the Delft AI Cluster [TU Delft Research Engineering and Infrastructure Team, 2026] using one CPU core per run. String and ARC-AGI-1 runs received 64GB of memory, while bit-vector runs received 32GB.

Because PHALCON is stochastic, we ran each task five times with independent random seeds. The reported results correspond to the mean performance across runs, along with a variability band. This band represents the best-case and worst-case performance across the five runs for each problem instance.

Memory control. To avoid memory overflow, we prune subexpressions that produce values more than twice the size of the task's largest input or target output.

Timeout. Each run has a 30-minute timeout, checked at the start of every PHALCON iteration. Because an active Julia function cannot be interrupted, the final iteration may finish after this limit. We report the actual elapsed runtime.

6.2 Can PHALCON Compete with Existing Techniques?

Task-level adaptation is a viable alternative to domain-level adaptation only if its greater generality does not substantially reduce synthesis performance. We therefore compare PHALCON with domain-specific baselines on all three benchmarks.

We measure performance using the percentage of tasks solved, both overall and as a function of program evaluations and execution time. We treat performance relative to program evaluations as the primary efficiency measure because execution times depend on implementation, hardware, and system-specific optimizations. Runtime still indicates the practical cost of each method.

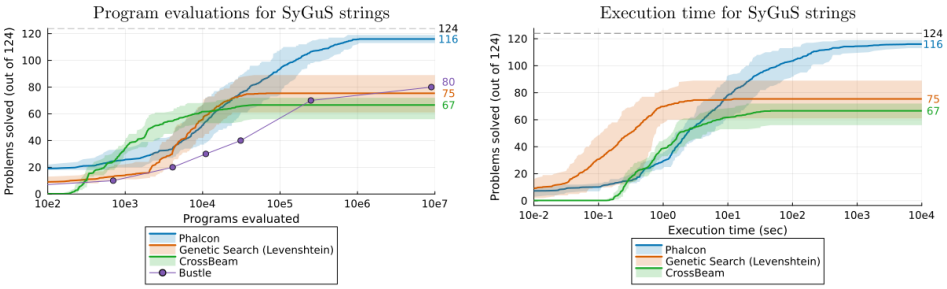
SyGuS String Transformations

We compare PHALCON with BUSTLE [Odena et al., 2021], CROSSBEAM [Shi et al., 2022], and GP using Levenshtein distance as cost. BUSTLE and CROSSBEAM use heuristics learned from large collections of random string-transformation tasks, whereas the GP baseline directly rewards textual similarity. All three baselines, therefore, exploit guidance specialized to string transformations.

Figure 6.1 shows the results. PHALCON achieves the highest final accuracy, solving 116 of the 124 tasks. Its advantage is statistically significant relative to the baselines: BUSTLE ($p = 3.21 \cdot 10^{-7}$), CROSSBEAM ($p = 7.07 \cdot 10^{-12}$), and GP ($p = 3.02 \cdot 10^{-8}$).

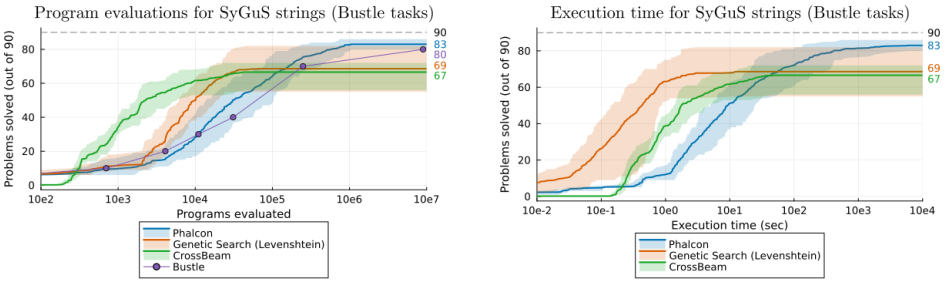
The efficiency results are more mixed. CROSSBEAM reaches its final performance using fewer program evaluations than PHALCON for the same performance, while GP requires a similar number, and BUSTLE requires more. In terms of execution time, PHALCON and CROSSBEAM perform comparably, while GP is faster. We omit the published BUSTLE runtimes because they seem unreasonably fast compared to their excessive number of program evaluations.

These results show that PHALCON not only competes with the string-specific baselines but also exceeds their final accuracy. It also requires fewer evaluations than BUSTLE and is similar to GP, although it does not match CROSSBEAM's evaluation efficiency. This difference is unsurprising because CROSSBEAM uses a learned heuristic trained across many tasks. By contrast, PHALCON constructs its guidance from the task currently being solved. Its ability to outperform BUSTLE and Levenshtein-guided GP in both final accuracy and evaluation efficiency demonstrates that PHALCON can learn useful guidance within a single task.



(a) Tasks solved over program evaluations. (b) Tasks solved over execution time.

Figure 6.1: Performance of PHALCON, BUSTLE, CROSSBEAM, and GP with Levenshtein-distance cost on the SyGuS string benchmark.



(a) Tasks solved over program evaluations. (b) Tasks solved over execution time.

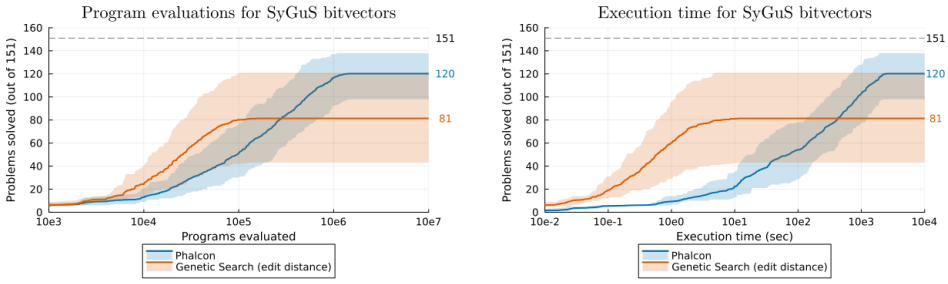
Figure 6.2: Performance on the subset of SyGuS string tasks supported by all compared methods.

Comparison on the Shared Task Subset

The complete string benchmark also includes tasks with non-string outputs, such as Boolean and integer values. Because BUSTLE, CROSSBEAM, and Levenshtein-based GP are restricted to string-valued outputs, their published evaluations include only 90[1] tasks. This inability to attempt every task motivates task-level adaptation. Nevertheless, comparing only the supported tasks separates this generality advantage from performance within the baselines' intended domain.

Figure 6.2 shows the comparison on this shared subset. PHALCON again achieves the highest final accuracy. Although its margin is smaller than on the complete benchmark, the differences remain statistically significant relative to the baselines: BUSTLE ($p = 5.41 \cdot 10^{-3}$), CROSSBEAM ($p = 4.19 \cdot 10^{-8}$), and GP ($p = 5.84 \cdot 10^{-4}$).

[1]The BUSTLE and CROSSBEAM papers report 89 tasks. They do not include `problem_11440431`, although it also has a string output. Including this task gives a shared subset of 90 tasks.



(a) Tasks solved over program evaluations.

(b) Tasks solved over execution time.

Figure 6.3: Performance of PHALCON and GP with Hamming-distance fitness on the SyGuS bit-vector benchmark.

Thus, PHALCON's advantage does not arise only from its ability to handle additional output types. It also performs competitively within the narrower string domain for which the baselines were designed.

SyGuS Bit-Vector Transformations

For the bit-vector benchmark, we compare PHALCON with GP guided by Hamming distance. This baseline rewards candidate outputs according to their bitwise similarity.

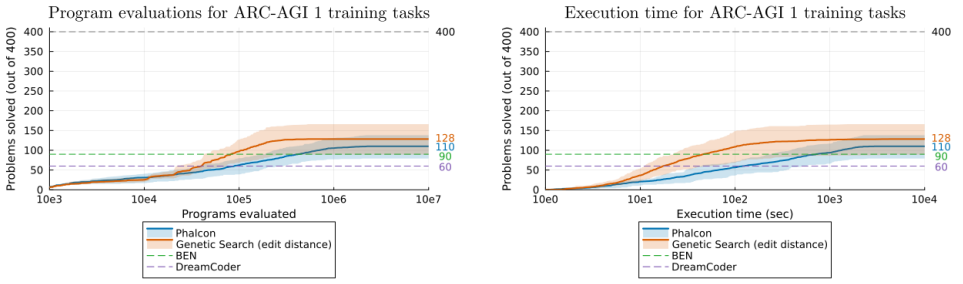
Figure 6.3 shows the results. PHALCON solves 120 tasks, compared with 81 for GP. This difference is statistically significant ($p = 3.26 \cdot 10^{-6}$). However, GP reaches its final accuracy using fewer program evaluations and substantially less execution time.

The progression of the curves helps explain this result. Hamming distance supplies useful guidance immediately, allowing GP to solve many tasks early in the search. Its performance then stabilizes after approximately 10 seconds. PHALCON initially progresses more slowly because it must discover its guidance during synthesis, but it eventually exceeds GP's final accuracy. Moreover, its curve continues to increase near the 30-minute timeout (around $10^{3.3}$ seconds). The experiment therefore establishes a higher observed accuracy for PHALCON, while leaving open the question of how many additional tasks it could solve under a larger budget.

ARC-AGI-1 Training Tasks

For ARC, we compare PHALCON with GP guided by grid-cell Hamming distance and with the published results of BEN [Witt et al., 2023] and an ARC-AGI-1 adaptation of DreamCoder [Ellis et al., 2020, Bober-Irizar and Banerjee, 2024].

BEN decomposes the input and output grids into components using predefined features such as color, size, and shape. It then searches for mappings between the



(a) Tasks solved over program evaluations.

(b) Tasks solved over execution time.

Figure 6.4: Performance of PHALCON and GP with grid-cell Hamming-distance fitness on the ARC-AGI-1 training tasks. The horizontal lines denote the overall performance of BEN and DreamCoder.

input and output components and combines them into a solution. BEN solves 90 of the 400 training tasks, with a reported average runtime of 23.20 seconds per task.

DreamCoder is a neuro-symbolic synthesis system that learns reusable abstractions across tasks. These abstractions reduce the search space for later tasks by making previously useful program structures directly available. The ARC-AGI-1 adaptation considered here solves 60 of the 400 tasks. It does not report directly comparable execution times.

These comparisons must be interpreted cautiously. Neither BEN nor DreamCoder uses the same grammar as PHALCON. GP is the only baseline evaluated using the same grammar as PHALCON.

Figure 6.4 shows the results. PHALCON solves 110 tasks, exceeding BEN's 90 ($p = 8.31 \cdot 10^{-4}$) and DreamCoder's 60 ($p = 2.28 \cdot 10^{-5}$). GP solves 128 tasks and therefore significantly outperforms PHALCON ($p = 2.93 \cdot 10^{-4}$). GP also requires fewer program evaluations and less execution time.

As in the bit-vector experiment, domain-specific distance gives GP useful guidance from the start of the search. PHALCON must instead spend part of its budget discovering suitable properties. Unlike on bit-vectors, it does not overtake GP within the available time. Its performance continues to increase near the timeout, so the result does not indicate where its accuracy would stabilize with a larger budget.

Nevertheless, PHALCON solves more tasks than the published BEN and DreamCoder systems while searching over a broader, multi-typed grammar.

Answer to $\mathcal{Q}_1$

The experiments show that PHALCON can compete with domain-level adaptation. It achieves the highest final accuracy on both SyGuS benchmarks and outperforms the published BEN and DreamCoder results on ARC. The only baseline with higher final accuracy is Hamming-guided GP on ARC.

PHALCON generally takes longer to reach its final performance. This reflects the principal trade-off of task-level adaptation: domain-specific methods begin with useful guidance, whereas PHALCON must derive its guidance from the current task. That initial advantage does not always persist. On bit-vectors, GP progresses faster initially but stabilizes at a lower accuracy, while PHALCON continues solving tasks. The string results show the strongest outcome, with PHALCON solving almost the entire benchmark.

We therefore answer $\mathcal{Q}_1$ affirmatively: task-level adaptation is competitive with domain-level guidance and can achieve better performance. Its principal limitation is not final accuracy but the time and program evaluations required to discover and exploit task-specific properties.

Runtime Analysis

The preceding results suggest that faster task-level adaptation would allow PHALCON to realize more of its potential within a fixed budget. We therefore examine where it spends its execution time.

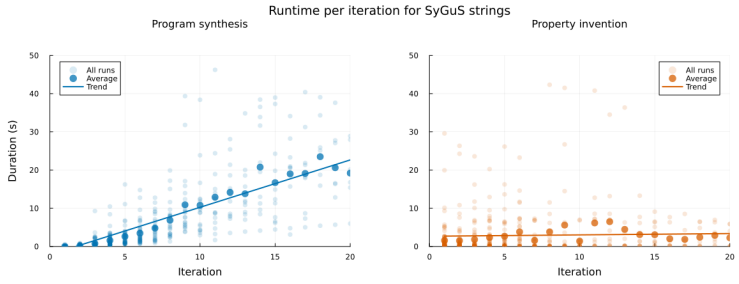
Each iteration consists of two phases: sampling programs with GP and selecting a new property. Sampling may become increasingly expensive because every additional property enlarges the objective. Property selection, by contrast, evaluates the same set of candidates against a fixed-size sample and should remain approximately constant across iterations. We therefore measure the average time spent in each phase at every iteration. For strings and bit-vectors, we include an entire benchmark run; for ARC, we use a sample of 100 tasks.

Figure 6.5 shows the results. In all three domains, GP sampling cost scales approximately linearly with the number of selected properties. Property selection remains broadly constant for strings and ARC. The bit-vector results form two approximate bands, producing a slight upward trend on average.

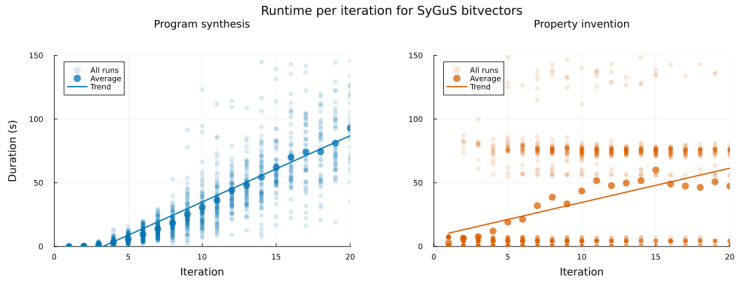
Overall, GP sampling dominates the runtime, particularly in later iterations. Future improvements should therefore focus primarily on the synthesis-with-properties problem: reducing the cost of finding programs that satisfy a collection of properties.

6.3 Can PHALCON Discover Task-Specific Properties?

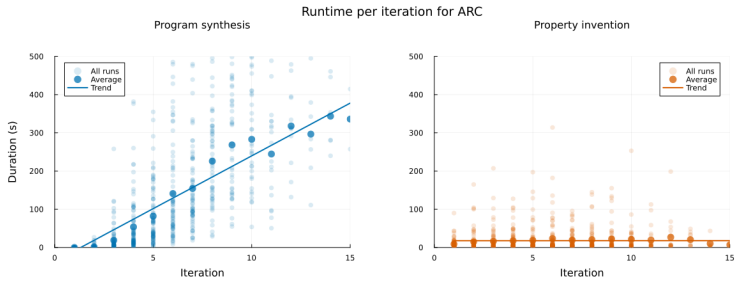
Task-level adaptation should avoid the overgeneralization seen in domain-level adaptation by selecting information relevant to the current task. We investigate whether PHALCON can do so through two case studies: the running string-transformation task and one ARC-AGI-1 task. These case studies illustrate how the selected properties relate to the behavior required by each specification.



(a) SyGuS strings.



(b) SyGuS bit-vectors.



(c) ARC-AGI-1.

Figure 6.5: Average time spent on GP sampling and property selection at each PHALCON iteration.

A SyGuS String-Transformation Task

We first analyze PHALCON's synthesis process for the running task. The task abbreviates the first name to its initial while preserving the surname.

PHALCON solves this task within a few seconds after selecting the two properties shown in Table 6.2. The first property records the position of the first space in the output. Its target vector $[2, 2, 2]$ requires every output to place the first space at the second position.

The second property searches for a space in the output, beginning at the position of the first space in the corresponding input. Its target vector is $[-1, -1, -1]$, indicating that no such additional space may occur. Together, the properties constrain the location and number of spaces without prescribing the complete output.

Property π_i	Target property-value vector $\pi_i(X, Y^*)$
`indexOf(y, " ", 1)`	$[2, 2, 2]$
`indexOf(y, " ", indexOf(x(1), " ", 1))`	$[-1, -1, -1]$

Table 6.2: Properties selected by PHALCON for the running task.

```
1  concat(
2    index(x(1), 1),
3    substring(
4      replace(x(1), concat(" ", x(1)), concat(" ", " ")),
5      indexOf(x(1), " ", 1),
6      length(x(1))))
```

Figure 6.6: Program synthesized by PHALCON for the running task.

Both properties are closely tailored to the required transformation. They focus on the space between the abbreviated first name and the surname, rather than generic features such as capitalization, the presence of digits, or length. Their usefulness also depends on their specificity: a domain-tailored property would never indicate whether the first space occurs at position 2, or prohibit spaces after the position where one occurred in the input.

Figure 6.6 shows the synthesized program. It selects the first input character and concatenates it with the substring beginning at the first space. This behavior matches the exemplar solution from Chapter 2. The expression on line 4 is syntactically more complicated than necessary, but is semantically equivalent to `x(1)`: the string `concat(" ", x(1))` is longer than `x(1)` and therefore cannot occur within it. The redundant expression is an artifact of GP's genetic operations.

An ARC-AGI-1 Task

We next analyze the ARC-AGI-1 task shown in Figure 6.7. The required transformation places a red background behind the cyan object in each input grid.

Table 6.3 shows the three properties selected by PHALCON. Their values include collections of cells and complete objects and are therefore omitted from the figure; we describe their target behavior below.

The first property, `frontiers(y)`, extracts monochromatic rows and columns spanning the complete grid. On the target outputs, these frontiers describe the parts of the black background that remain unchanged. The property, therefore, constrains cell placement in that region.

The second property extracts the non-background objects and selects the object whose lowest cell has the greatest row index. Because row indices increase downwards, this selects the lowermost object. In the target outputs, the selected object is cyan; when the cyan and red objects tie, the implementation returns the cyan object due to its tie-breaking order. Agreement with this property, therefore, requires that

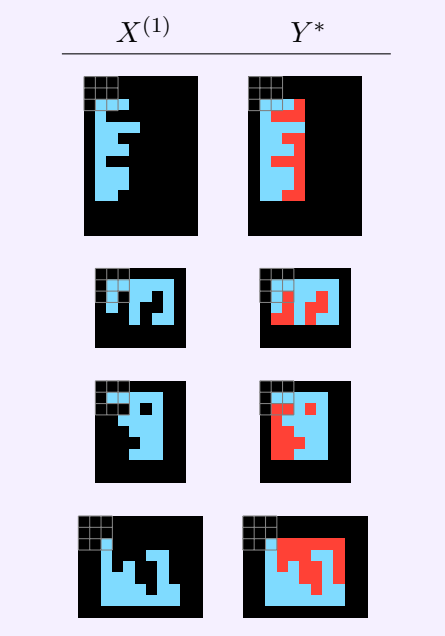


Figure 6.7: An exemplar ARC-AGI-1 task for analysis.

Property π_i
`frontiers(y)`
`argmax_by(objects(y, ⊤, ⊤, 0), lowermost)`
`argmax_by(objects(y, ⊤, ⊤, 2), uppermost)`

Table 6.3: Properties selected by PHALCON for the ARC-AGI-1 task.

no object is placed below the cyan object.

The third property again extracts objects, this time ignoring red as the background color, and selects the object whose highest cell has the greatest row index. It consequently selects the object with the lowest upper boundary. This object is also the cyan object in the target outputs. Together, the second and third properties constrain any object's vertical placement relative to the cyan object.

These properties depend on the task's spatial structure. Rather than merely recording general facts such as the colors present in the output, they constrain full-grid background lines and the objects occurring at particular vertical extremes.

Figure 6.8 shows the synthesized program. It extracts the cyan object from the input, computes its backdrop, and underfills this region with red. Underfilling changes only cells of the most common color, which is black in these inputs, and therefore preserves the original objects.

```
1  underfill(
2    x(1),
3    red,
4    mapply(
5      backdrop,
6      objects(x(1), ⊥, ⊥, 0)))
```

Figure 6.8: Program synthesized by PHALCON for the ARC-AGI-1 task.

Answer to $\mathcal{Q}_2$

Both case studies show that PHALCON can discover properties tailored to an individual task. For the string transformation, it identifies the position of the space separating the first and last name. For the ARC-AGI-1 task, it identifies spatial relationships between the background, the cyan object, and the newly added red region.

The properties can also return values other than Booleans, allowing a compact property to express a precise constraint. For example,

$$\pi(x, y) = \texttt{indexOf(y, " ", 1)}$$

directly returns the position of the first space. Matching its target value requires the space to occur at position 2.

We therefore answer $\mathcal{Q}_2$ affirmatively: PHALCON can derive informative, task-specific guidance from a single specification.

6.4 Can PHALCON's Properties Be Reused Across Tasks?

PHALCON selects properties one at a time because iterative resampling makes them gradually more task-specific. After selecting a property, it resamples programs that satisfy that property and uses their behavior to select the next one. This conditional sampling reveals distinctions that may not have seemed useful in the preceding population.

Repeating this process for every task is expensive. If properties discovered for earlier tasks remain useful, selecting several of them at once could reduce the number of refinement iterations. We investigate this possibility using PHALCON^{Transferable}, a variant that maintains a library of properties selected for previous tasks.

Selecting Previously Discovered Properties

At each iteration, PHALCON^{Transferable} searches its library before generating a new property. The method considers only library properties with

$$0 < \widehat{\alpha}_{F_t}(\pi) \leq 0.3.$$

Such a property agrees with at most 30% of the sampled programs while still satisfying at least one. PHALCON^{Transferable} selects multiple eligible properties during the same iteration. It generates a new property only when the library contains no eligible candidate and terminates after at most 20 iterations.

Properties that distinguished different behaviors in their original tasks may become redundant in a new task. To limit this problem, we compare their agreement patterns on the current sample. For each property, define

$$A_{F_t}(\pi) = \{f \in F_t \mid \pi(X, Y_f) = \pi(X, Y^*)\}.$$

Two properties with the same set $A_F(\pi)$ make the same distinction between the sampled programs with respect to the target. We process properties in increasing order of agreement rate and retain only properties with distinct agreement patterns.

This test removes properties that are observably redundant in the current sample. Still, it cannot predict how useful they will remain after the search is conditioned on the other selected properties. This is precisely the information normally obtained by selecting one property, resampling, and only then selecting the next.

Because the library grows as tasks are processed, the order affects the experiment. We randomize the string and bit-vector task order independently in each of the five runs. For ARC, we process tasks in increasing order of exemplar-solution size, measured by the number of operations in the reference program [Hodel, 2023], and randomize tasks with the same solution size.

SyGuS String Transformations

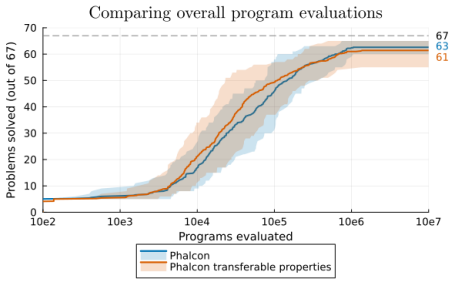
Property transfer requires compatible input and output types, which ironically subjects us to the limitations of domain-level adaptation. We must therefore restrict this experiment to the 67 SyGuS tasks that map a single-string input to a single-string output.

Figure 6.9 compares the two methods. Their results are similar. PHALCON^{Transferable} uses slightly fewer program evaluations and less execution time, but also achieves slightly lower final accuracy. Reusing properties, therefore, provides no clear overall improvement on the string tasks.

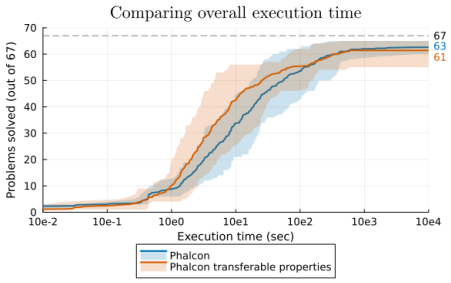
SyGuS Bit-Vector Transformations

Figure 6.10 shows the results for bit-vectors. PHALCON^{Transferable} requires slightly more program evaluations and execution time and solves fewer tasks overall.

Standard PHALCON continues solving bit-vector tasks near the timeout. A small loss of search efficiency can therefore prevent several runs from solving within the available budget. Selecting several properties early does not provide enough additional guidance to compensate for this loss.

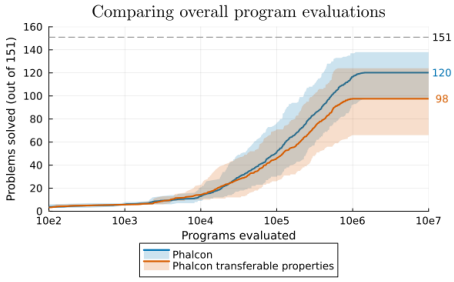


(a) Tasks solved over program evaluations.

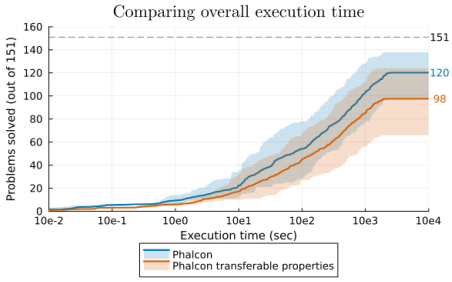


(b) Tasks solved over execution time.

Figure 6.9: Performance of PHALCON and PHALCON^{Transferable} on the compatible SyGuS string tasks.



(a) Tasks solved over program evaluations.



(b) Tasks solved over execution time.

Figure 6.10: Performance of PHALCON and PHALCON^{Transferable} on the SyGuS bit-vector benchmark.

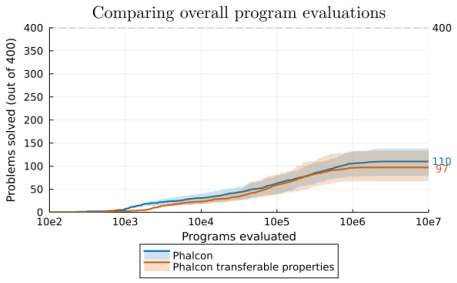
ARC-AGI-1 Training Tasks

Figure 6.11 shows a similar result on ARC. PHALCON^{Transferable} requires more program evaluations and execution time and achieves lower final accuracy. As in the bit-vector experiment, both methods continue solving tasks near the timeout. The transferred properties do not guide search efficiently enough to offset the cost of using a larger property set.

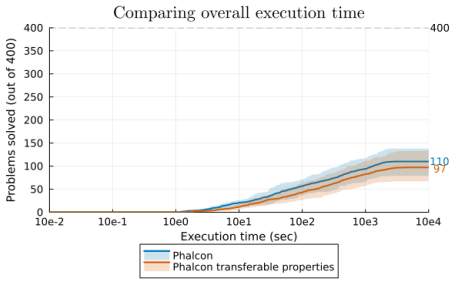
Why Did Property Transfer Not Improve Performance?

Property transfer should reduce the number of refinement iterations. Table 6.4 shows that it succeeds in this respect. The average number of iterations decreases by 19% for strings, 32% for bit-vectors, and 30% for ARC. Apparently, fewer iterations do not improve performance.

We suspect that PHALCON^{Transferable} picks more properties in total, making synthesis take longer. Table 6.5 shows that PHALCON^{Transferable} uses 38% more properties for strings, 105% more for bit-vectors, and 93% more for ARC. It therefore



(a) Tasks solved over program evaluations.



(b) Tasks solved over execution time.

Figure 6.11: Performance of PHALCON and PHALCON^{Transferable} on the ARC-AGI-1 training tasks.

Domain	PHALCON	PHALCON^{Transferable}	Change
Strings	4.7	3.8	-19%
Bit-vectors	8.1	5.5	-32%
ARC-AGI-1	4.3	3.0	-30%

Table 6.4: Average number of refinement iterations for successful runs.

Domain	PHALCON	PHALCON^{Transferable}	Change
Strings	4.7	6.5	$+38\%$
Bit-vectors	8.1	16.6	$+105\%$
ARC-AGI-1	4.3	8.3	$+93\%$

Table 6.5: Average number of selected properties for successful runs.

needs substantially more properties to solve tasks in fewer iterations.

The larger property sets also make synthesis more expensive. Every property adds comparisons to the GP fitness function and further restricts the population that GP must produce. The transferred properties thus reduce the number of resampling steps, but they provide less information per property and increase the cost of each subsequent synthesis phase. This trade-off is most pronounced for bit-vectors and ARC, where the property sets approximately double in size, and performance decreases the most.

Answer to $\mathcal{Q}_3$

The experiment shows why PHALCON selects properties one at a time. Although selecting several transferred properties reduces the number of iterations, it requires larger property sets and does not improve overall performance. A property that distinguishes the current population may add little information after another property

restricts that population. Conditional resampling avoids this problem by evaluating each new property against programs satisfying all previously selected properties.

We therefore answer $\mathbf{Q_3}$ negatively for the tested strategy. Property reuse may still help if a future method can predict each property's contribution after conditioning.

Chapter 7

Conclusions and Future Work

Large language models have changed what users expect from code generation. Given a task in almost any common programming language, an LLM can at least attempt a solution. Inductive program synthesis offers more explicit specifications and systematic search, but most existing techniques achieve their efficiency by specializing in a particular domain. A method designed for string transformations, for example, may be highly effective within that domain but unable to handle tasks with a different output type.

This thesis argued that this specialization is a fundamental obstacle to general program synthesis. Existing methods commonly adapt their representations, heuristics, or semantics to a domain of related tasks. We call this approach *domain-level adaptation*. Besides restricting which tasks a synthesizer can attempt, domain-level adaptation inevitably over-generalizes. It may therefore retain information irrelevant to an individual task while omitting distinctions that matter only to a small number of tasks.

We introduced *task-level adaptation* as an alternative framework. Rather than receiving guidance designed or learned for a domain, a task-level adaptive synthesizer derives its guidance from the individual task it is currently solving. More specifically, this thesis proposed task-level adaptation through generated properties. The synthesizer constructs properties from the supplied task grammar and uses them to describe selected aspects of the target behavior. It therefore requires no separately engineered property collection, abstract semantics, or heuristics learned from other tasks.

To demonstrate this framework's viability, we developed PHALCON. PHALCON captures task-specific behavior in an incrementally selected property set. It alternates between two phases. First, it samples programs that satisfy the properties selected so far, producing examples of behavior that currently appear promising. Second, it selects a new property on which many sampled programs differ from the target outputs. Repeating these phases progressively refines the boundary between behavior that satisfies the current guidance and behavior that satisfies the complete specification.

Our experiments show that task-level adaptation is possible and competitive. PHALCON achieves higher final accuracy than almost all evaluated baselines across the SyGuS string, SyGuS bit-vector, and ARC-AGI-1 benchmarks. PHALCON generally requires more time and program evaluations because it must discover guidance during synthesis rather than begin with domain-specific knowledge. Nevertheless, it nearly solves the complete string benchmark and continues to improve near the time limit for bit-vectors and ARC. The case studies further show that its generated properties capture highly specific aspects of individual tasks.

The experiments also reveal areas for improvement in the framework. Most of PHALCON's runtime is spent searching for programs that satisfy an expanding set of properties, while property generation and selection remain comparatively inexpensive. Future work should therefore focus on the synthesis-with-properties problem: constructing conditional samples in fewer generations, or developing a search procedure better suited to satisfying generated behavioral constraints.

Our transfer experiment provides a second direction for future work. Selecting several properties reused from earlier tasks reduces the number of refinement iterations but produces larger property sets and does not improve end-to-end performance. The difficulty is predicting which properties will remain informative after conditioning on the others. A stronger transfer method might reuse not only a property but also the structures that satisfy it. Such information could preserve the benefits of task-level selection while reducing the cost of rediscovering similar behavior.

For program synthesis to compete with the broad applicability of LLM-based code generation, it must move beyond systems restricted to tasks anticipated by their designers. Task-level adaptation provides a path toward this goal: it allows a synthesizer to approach tasks from different domains while avoiding guidance generalized across tasks for which it may be irrelevant. Through PHALCON, this thesis demonstrates that a synthesizer can learn guidance from the task at hand, bringing us one step closer to generalized program synthesis.

Bibliography

J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

M. Bober-Irizar and S. Banerjee. Neural networks for abstraction and reasoning: Towards broad generalization in machines. *arXiv preprint arXiv:2402.03507*, 2024.

F. Chollet. On the measure of intelligence, 2019. URL https://arxiv.org/abs/1911.01547.

P. Cousot. Abstract interpretation: Theory and practice. In D. Bosnacki and S. Leue, editors, *Model Checking of Software, 9th International SPIN Workshop, Grenoble, France, April 11-13, 2002, Proceedings*, volume 2318 of *Lecture Notes in Computer Science*, pages 2–5. Springer, 2002. doi: 10.1007/3-540-46017-9_2. URL https://doi.org/10.1007/3-540-46017-9_2.

P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In R. M. Graham, M. A. Harrison, and R. Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi: 10.1145/512950.512973. URL https://doi.org/10.1145/512950.512973.

A. Cropper and S. Dumančić. Learning large logic programs by going beyond entailment, 2020. URL https://arxiv.org/abs/2004.09855.

S. Daniotti, J. Wachs, X. Feng, and F. Neffke. Who is using ai to code? global diffusion and impact of generative ai. *Science*, 391(6787):831–835, 2026. doi: 10.1126/science.adz9311. URL https://www.science.org/doi/abs/10.1126/science.adz9311.

K. Ellis, C. Wong, M. Nye, M. Sablé-Meyer, L. Cary, L. Morales, L. Hewitt, A. Solar-Lezama, and J. B. Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv preprint arXiv:2006.08381*, 2020.

S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 07 2017a. ISSN 2325-1107. doi: 10.1561/2500000010. URL https://doi.org/10.1561/2500000010.

S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Found. Trends Program. Lang.*, 4(1-2):1–119, 2017b. doi: 10.1561/2500000010. URL https://doi.org/10.1561/2500000010.

S. N. Guria, J. S. Foster, and D. V. Horn. Absynthe: Abstract interpretation-guided synthesis. *Proc. ACM Program. Lang.*, 7(PLDI):1584–1607, 2023. doi: 10.1145/3591285. URL https://doi.org/10.1145/3591285.

M. Hodel. Arc-dsl: A domain specific language for the abstraction and reasoning corpus. Technical report, GitHub, 2023. URL https://github.com/michaelhodel/arc-dsl/blob/main/arc_dsl_writeup.pdf.

A. Odena and C. Sutton. Learning to represent programs with property signatures. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=rylHspEKPr.

A. Odena, K. Shi, D. Bieber, R. Singh, C. Sutton, and H. Dai. BUSTLE: bottom-up program synthesis through learning-guided exploration. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=yHeg4PbFHh.

S. Padhi, U. Abhishek, A. Fu, E. Polgreen, and A. Reynolds. Benchmarks for sygus competition. https://github.com/SyGuS-Org/benchmarks, 2019.

R. Poli, W. Langdon, N. Mcphee, and J. Koza. Genetic programming: An introductory tutorial and a survey of techniques and applications, 11 2007.

K. Shi, H. Dai, K. Ellis, and C. Sutton. Crossbeam: Learning to search in bottom-up program synthesis. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=qhC8mr2LEKq.

K. Shi, H. Dai, W. Li, K. Ellis, and C. Sutton. Lambdabeam: Neural program search with higher-order functions and lambdas. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/a10da26f47120217c1b7c2aeb2979048-Abstract-Conference.html.

P. D. Summers. A methodology for lisp program construction from examples. *J. ACM*, 24(1):161–175, Jan. 1977. ISSN 0004-5411. doi: 10.1145/321992.322002. URL https://doi.org/10.1145/321992.322002.

TU Delft Research Engineering and Infrastructure Team. Delft ai cluster (daic). https://daic.tudelft.nl, 2026. Accessed: 2026-06-09.

X. Wang, I. Dillig, and R. Singh. Program synthesis using abstraction refinement. *Proc. ACM Program. Lang.*, 2(POPL):63:1–63:30, 2018. doi: 10.1145/3158151. URL https://doi.org/10.1145/3158151.

J. Witt, S. Rasing, S. Dumančić, T. Guns, and C.-C. Carbon. A divide-align-conquer strategy for program synthesis. *arXiv preprint arXiv:2301.03094*, 2023.

Y. Yoon, W. Lee, and K. Yi. Inductive program synthesis via iterative forward-backward abstract interpretation. *Proc. ACM Program. Lang.*, 7(PLDI):1657–1681, 2023. doi: 10.1145/3591288. URL https://doi.org/10.1145/3591288.

Appendix A

Genetic Programming Implementation

Algorithm 2 shows our genetic programming implementation. It starts by generating a set of *extensions*, which are subprograms of size up to $d = 1$ used as building blocks during mutations. We have defined size as the number of operations in an expression, reflecting its computational complexity. The initial population contains the N lowest-cost extensions of the task's output type. A heuristic cost function H computes a program's cost. Note that this is slightly different from the fitness used throughout the thesis; instead of maximizing fitness, we now want to minimize cost.

After initialization, the algorithm proceeds iteratively. In each generation, it produces M candidate offspring. To generate these candidates, the algorithm randomly selects two parent programs, f_a and f_b, from the current population. These parents are then used to produce four offspring: one obtained by mutating f_a, one by mutating f_b, one by applying crossover between f_a and f_b, and one by mutating the resulting crossover program. This design ensures that we explore both isolated mutations and recombined structures.

Candidate offspring are added to the population C, ensuring that its size does not exceed N. We retain only the best programs and ignore programs that produce identical outputs.

If any program in the population satisfies the specification, the algorithm immediately returns it as a solution.

If the entire population has a cost of zero, we have sampled N programs satisfying all properties, and the search terminates by returning C.

The process repeats until we observe no improvement in heuristic cost for T consecutive generations, at which point the search terminates, and we return the population C.

Choice of genetic operators

The effectiveness of GP strongly depends on the mutation and crossover operators, as these determine how we explore new programs. We designed these operators to mimic common program editing actions performed by humans, such as replacing, inserting, or deleting instructions.

Crossover

The operation

$$f' = \text{crossover}(f_a, f_b)$$

replaces a randomly selected subprogram of f_a with a randomly selected subprogram of the same type from f_b.

Mutation

The operation

$$f' = \text{mutate}(f)$$

randomly applies one of the following transformations:

1. **Replace** a random subprogram with a randomly generated extension of the same type. For example:

$$X + 1 \quad \Rightarrow \quad X + (1 + X)$$

2. **Insert** a new operation into the program. We fill in any missing operands introduced with a random extension. For example:

$$X + 1 \quad \Rightarrow \quad X + (1 \times _) \quad \Rightarrow \quad X + (1 \times X)$$

3. **Delete** a random operation by replacing a randomly selected node with one of its children. For example:

$$X + 1 \quad \Rightarrow \quad X$$

Algorithm 2 Genetic Programming

Require:
Population size: N
Candidate population size: M
Cost function: H
Maximum extension size: d
Termination threshold: T

$E \leftarrow$ all extensions up to size d
$C \leftarrow N$ best programs according to H up to size d

loop
    $C_{\text{current}} \leftarrow C$

    loop $\frac{M}{4}$ times
        Randomly select parents $f_a, f_b \in C_{\text{current}}$
        $c_1 \leftarrow$ mutate(f_a)
        $c_2 \leftarrow$ mutate(f_b)
        $c_3 \leftarrow$ crossover(f_a, f_b)
        $c_4 \leftarrow$ mutate(c_3)

        $\text{add_to_population}(C, \{c_1, c_2, c_3, c_4\})$ ▷ ignores duplicate outputs

        if C contains a solution f^* **then**
            return f^*
        end if

        if C has a total cost of zero **then**
            return C
        end if
    end loop

    if C has not improved for T iterations **then**
        return C
    end if
end loop

Appendix B

ARC-AGI-1 Property Grammar

The ARC-AGI-1 grammar is too large for efficient property search, so we restrict it. Our restrictions are motivated by the intuition that properties record details of program behavior.

To that end, we define a type hierarchy that indicates a type's size. Concretely for ARC-AGI-1, this yields the following hierarchy from large to small:

Type	**Meaning**
$\mathrm{GridContainer}$	Vector of grids
Grid	Matrix of integers
$\mathrm{Objects}$	Vector of objects
Object	Arbitrary shape with colors
$\mathrm{Indices}$	Vector of integer tuples
$\mathrm{IntContainer}$	Vector of integers
$\mathrm{IntTuple}$	Tuple of integers
$\mathrm{Integer}$	Integer value
$\mathrm{Boolean}$	Boolean value

Properties may perform only operations that shrink their inputs. For example, the operation

$$\mathrm{Integer} = \mathrm{mostcolor}(\mathrm{Grid})$$

is allowed, as it maps a Grid to an $\mathrm{Integer}$. However, the operation

$$\mathrm{IntTuple} = \mathrm{astuple}(\mathrm{Integer}, \mathrm{Integer})$$

is not allowed, as it maps an $\mathrm{Integer}$ into an $\mathrm{IntTuple}$, which is higher in the hierarchy.

Operations that preserve the same level in the hierarchy are also forbidden. For example,

$$\mathrm{Grid} = \mathrm{rotate90}(\mathrm{Grid})$$

is not allowed, since it does not reduce the input type.

Appendix C

AI Statement

For this thesis, I used artificial intelligence to help articulate my thoughts in writing. Specifically, I used large language models to check grammar and spelling, provide feedback on my writing, and assist with programming tasks, primarily creating graphs and processing data. I did not use AI to generate ideas or formulate original thoughts. All ideas, concepts, arguments, techniques, and conclusions presented in this thesis are my own.