

FEDERAL UNIVERSITY OF ABC
GRADUATE COURSE IN COMPUTER SCIENCE

Matheus Campos Fernandes

FUNCTIONAL PROGRAM SYNTHESIS WITH HIGHER-ORDER FUNCTIONS AND
RECURSION SCHEMES

Santo André, SP – Brazil
October 2025

Matheus Campos Fernandes

FUNCTIONAL PROGRAM SYNTHESIS WITH HIGHER-ORDER FUNCTIONS AND
RECURSION SCHEMES

Thesis presented to the Graduate course of Federal University of
ABC, as a partial requirement for obtaining a Doctor's degree
in Computer Science.

Advisor: Prof. Dr. Fabrício Olivetti de França
Coadvisor: Prof. Dr. Emílio de Camargo Franceschini

Santo André, SP – Brazil
October 2025

Matheus Campos Fernandes,

Functional Program Synthesis with Higher-Order Functions and
Recursion Schemes/ Matheus Campos Fernandes. – 2025.

128 p.: il.

Advisor: Prof. Dr. Fabrício Olivetti de França

Coadvisor: Prof. Dr. Emílio de Camargo Franceschini

Thesis (Doutorado) – Federal University of ABC, Graduate Course in
Computer Science, Santo André, 2025.

1. Program Synthesis. 2. Genetic Programming. 3. Functional
Languages. 4. Recursion Schemes. I. Olivetti de França, Fabrício II. de
Camargo Franceschini, Emílio III. Graduate Course in Computer Science,
2025. IV. Title

This exemplar was revised and altered from the original version according to the observations done by the defense board at the defense day, under responsibility of author and advisor's consent.

Santo André, November 24th, 2025

Author's signature: _____

Advisor's signature: _____



MINISTÉRIO DA EDUCAÇÃO

Fundação Universidade Federal do ABC

Avenida dos Estados, 5001 - Bairro Santa Terezinha - Santo André - SP
CEP 09210-580 · Fone: (11) 4996-0017

Ata de Defesa de Tese de Doutorado e Folha de Assinaturas

No dia 31 de Outubro de 2025 às 12:30, no local: <https://meet.google.com/qwr-agud-qcu>, realizou-se a Defesa da Tese de Doutorado, que constou da apresentação do trabalho intitulado "Functional Program Synthesis with Higher-Order Functions and Recursion Schemes" de autoria do candidato, MATHEUS CAMPOS FERNANDES, RA nº 23202110043, discente do Programa de Pós-Graduação em CIÊNCIA DA COMPUTAÇÃO da UFABC, sob orientação do Profº FABRICIO OLIVETTI DE FRANCA. Concluídos os trabalhos de apresentação e arguição, o candidato foi considerado APROVADO pela Banca Examinadora.

E, para constar, foi lavrada a presente ata e folha de assinaturas assinada pelos membros da Banca.

Documento assinado digitalmente
 **FABRICIO OLIVETTI DE FRANCA**
Data: 07/11/2025 06:38:09-0300
Verifique em <https://validar.iti.gov.br>

Dr. FABRICIO OLIVETTI DE FRANCA, UFABC

Presidente - Interno ao Programa

Documento assinado digitalmente
 **MARIO LESTON REY**
Data: 06/11/2025 12:34:41-0300
Verifique em <https://validar.iti.gov.br>

Dr. MARIO LESTON REY, UFABC

Membro Titular - Examinador(a) Externo ao Programa

Documento assinado digitalmente
 **JERONIMO PELLEGRINI**
Data: 06/11/2025 20:51:56-0300
Verifique em <https://validar.iti.gov.br>

Dr. JERONIMO PELLEGRINI, UFABC

Membro Titular - Examinador(a) Externo ao Programa

Documento assinado digitalmente
 **MARCIO PORTO BASGALUPP**
Data: 04/11/2025 09:04:39-0300
Verifique em <https://validar.iti.gov.br>

Dr. MARCIO BASGALUPP, UNIFESP

Membro Titular - Examinador(a) Externo à Instituição



MINISTÉRIO DA EDUCAÇÃO

Fundação Universidade Federal do ABC

Avenida dos Estados, 5001 - Bairro Santa Terezinha - Santo André - SP
CEP 09210-580 - Fone: (11) 4996-0017

Assinado por: **Alcides Miguel Cachulo Aguiar
Fonseca**

Num. de Identificação: 13445055

Data: 2025.11.03 00:26:46 +0000

Dr. ALCIDES FOSENCA, Univ Lisboa

Membro Titular - Examinador(a) Externo à Instituição

Dr. DAVID CORREA MARTINS JUNIOR, UFABC

Membro Suplente - Examinador(a) Interno ao Programa

Dr. ADOLFO GUSTAVO SERRA SECA NETO, UTFPR

Membro Suplente - Examinador(a) Externo à Instituição

Dra. GISELE LOBO PAPP, UFMG

Membro Suplente - Examinador(a) Externo à Instituição

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de
Nível Superior – Brasil (CAPES) – Finance Code 001

Aos Doutores:
Elizabeth Magalhães de Oliveira
Henrique Prado Fernandes

Agradecimentos

Sou grato primeiramente aos meus orientadores, Fabrício e Emílio, pela oportunidade de trabalharmos juntos. Muito obrigado por todas as conversas, ensinamentos, dedicação e apoio e por todo o conhecimento que a orientação de vocês me permitiu adquirir. Nunca, sob nenhuma hipótese, me deixaram faltar nada, e a isso sou muito grato.

Agradeço à minha esposa, Júlia, por todo o suporte acadêmico e não acadêmico, paciência, parceria, incentivo, e por sempre acreditar em mim, sobretudo quando eu mesmo não o fazia.

À minha família, em especial meus pais José e Silvia e minha irmã Beatriz, pelo amor, educação e exemplo, e por terem me provido de condições que possibilitaram trilhar este caminho da forma como fiz.

Agradeço a todos os colegas que tive o prazer de conhecer durante essa jornada na UFABC, e a todos os seus professores e funcionários, por toda a competência e dedicação.

Agradeço também a todos os professores que tive na vida (acadêmicos ou não) por tanto me ensinarem e fomentarem minha curiosidade.

Epigraph

“Our curiosity goes with you on your journey. You walk in the footsteps of those who came before you, and your path guides those who will follow later.”
(The Nomai)

Abstract

Program synthesis is the process of generating a computer program following a set of specifications, which can be a high-level description of the problem and/or a set of input-output examples. The synthesis can be modeled as a search problem in which the search space is the set of all the programs valid under a grammar. As the search space is vast, brute force is usually not feasible, and search heuristics, such as genetic programming, also have difficulty navigating it without guidance. This text presents two novel genetic programming algorithms that synthesize pure, typed, and functional programs: HOTGP and Origami. HOTGP leverages the knowledge provided by the rich data types associated with the specification and the built-in grammar to constrain the search space and improve the performance of the synthesis. Its grammar is based on Haskell’s standard base library (the synthesized code can be directly compiled using any standard Haskell compiler) and includes support for higher-order functions, λ -functions, and parametric polymorphism. Experimental results show that HOTGP is competitive with the state of the art. Additionally, Origami is an algorithm that tackles the challenge of effectively handling loops and recursion by exploring Recursion Schemes. The main advantage of writing a program using Recursion Schemes is that the programs are composed of well-defined templates with only a few parts that need to be synthesized. Preliminary results on a prototype implementation show that, once the choice of which recursion scheme is made, the synthesis process can be simplified. The first implementation of Origami is able to synthesize solutions in several Recursion Schemes and data structures, achieving a considerable improvement over its prototype. Results also show that it is competitive with other GP methods in the literature, as well as LLMs. The latest version of Origami employs AC/DC, a novel procedure designed to improve the search-space exploration. Experimental results in 3 different benchmarks show that it achieves considerable improvement over its previous version by raising success rates on every problem. Compared to similar methods in the literature, it has the highest count of problems solved with success rates of 100%, $\geq 75\%$, and $\geq 25\%$ across all benchmarks. In 18% of all benchmark problems it stands as the only method to reach 100% success rate, being the first known approach to achieve it on any problem in PSB2. It also demonstrates competitive performance to large language models, achieving the highest overall win-rate against Copilot among all GP methods.

Keywords: Program Synthesis; Genetic Programming; Functional Languages; Recursion Schemes

Resumo

Síntese de programas é o processo de gerar um programa de computador a partir de especificações, que pode ser uma descrição de alto nível do problema e/ou um conjunto de exemplos de entrada-saída. A síntese pode ser modelada como um problema de busca, em que o espaço de busca é o conjunto de todos os programas válidos sob uma gramática. Como o espaço de busca é amplo, força bruta geralmente não é viável, e heurísticas de busca, como a programação genética, também têm dificuldade de navegar sem orientação. Este texto apresenta dois novos algoritmos de programação genética que sintetizam programas puros, tipados e funcionais: HOTGP e Origami. HOTGP usa o conhecimento fornecido pelos tipos de dados associados à especificação e à gramática interna para restringir o espaço de busca e melhorar o desempenho da síntese. Sua gramática é baseada na biblioteca padrão da linguagem Haskell (o código sintetizado pode ser compilado diretamente usando qualquer compilador Haskell) e inclui suporte para funções de alta ordem, funções λ e polimorfismo paramétrico. Os resultados experimentais mostram que o HOTGP é competitivo com o estado da arte. Adicionalmente, Origami é um algoritmo que aborda o desafio de lidar com laços e recursão de maneira eficaz, explorando esquemas de recursão. A principal vantagem de se escrever um programa utilizando Esquemas de Recursão é que os programas são compostos por modelos bem definidos com apenas algumas partes que precisam ser sintetizadas. Resultados preliminares de uma implementação protótipo mostram que, uma vez feita a escolha de qual esquema de recursão usar, o processo de síntese pode ser simplificado. A primeira implementação do Origami é capaz de sintetizar soluções em diversos esquemas de recursão e estruturas de dados, alcançando melhora considerável em relação ao seu protótipo. Os resultados também mostram que ele é competitivo tanto com outros métodos de programação genética presentes na literatura quanto com LLMs. A versão mais recente do Origami emprega o AC/DC, um procedimento novo projetado para melhorar a exploração do espaço de busca. Resultados experimentais em 3 benchmarks diferentes demonstram que ele obtém avanços consideráveis em relação à versão anterior, elevando as taxas de sucesso em todos os problemas. Comparado a métodos similares na literatura, apresenta a maior quantidade de problemas resolvidos com taxas de sucesso de 100%, $\geq 75\%$ e $\geq 25\%$ em todos os *benchmarks*. Em 18% dos problemas de todos os *benchmarks*, é o único método a atingir 100% de sucesso, sendo a primeira abordagem conhecida a atingí-lo em qualquer problema do PSB2. Também demonstra desempenho competitivo em relação a LLMs, alcançando a maior taxa geral de vitórias contra o Copilot entre todos os métodos de programação genética.

Palavras-chave: Síntese de Programas; Programação Genética; Linguagens Funcionais; Esquemas de recursão

List of Figures

Figure 2.1	Examples of syntax trees. Leaves (terminals) are represented as ellipses and nodes (functions or non-terminals) as hexagons.	27
Figure 2.2	An example of crossover on two trees.	27
Figure 3.1	Percentage of reduction in the number of nodes caused by the refinement process.	47
Figure 4.1	Distribution of recursion schemes used to solve the full set of PSB1 problems.	68
Figure 6.1	Success rates under different AC and DC configurations. A value of 1 indicates execution every generation, 2 every other generation, 299 only once just before the final (300^{th}) generation, and ∞ indicates the procedure is never executed.	93
Figure 6.2	Percentage of problems won (W), tied (T), and lost (L) by each method compared to Copilot, across General Program Synthesis Benchmark 1 (PSB1) and General Program Synthesis Benchmark 2 (PSB2). Here, W indicates that a given method achieved a higher success rate than Copilot, T indicates equal success rates, and L indicates a lower success rate. CBGP* indicates partial results, as it was evaluated only on PSB1. A reference vertical line is drawn at 50%.	103

List of Tables

Table 3.1	Functions supported by HOTGP.	39
Table 3.2	Laws applied by the simplification system.	43
Table 3.3	Successful solutions found for each problem (% of executions) considering the training (Tr) and test (Te) data sets. HOTGP* lists the results obtained with HOTGP after the simplification procedure. The best values for the test data sets of each problem are highlighted. The <i>collatz-numbers</i> , <i>string-differences</i> , <i>wallis-pi</i> and <i>word-stats</i> problems are omitted as no algorithm was able to find results for those problems.	46
Table 3.4	Node count of the hand-crafted solutions and the smallest correct solutions found by Higher-Order Typed Genetic Programming (HOTGP) and HOTGP*. The node count relative to hand-crafted is shown in parenthesis.	48
Table 4.1	Association between type signatures and its corresponding recursion schemes.	57
Table 4.2	Function set used for solving the PSB1 benchmark problems.	70
Table 4.3	Function and constant set assumed to be provided by the user. All of these are explicitly mentioned in the problem description.	70
Table 4.4	Percentage of runs that returned a perfect solution on the validation set. The bottom part of the table summarizes the result as the number of times each algorithm had the highest percentage, and in how many problems the percentage was greater or equal to a certain threshold. The benchmarks marked with * are only solvable with accumulation.	71
Table 5.1	The complete set of functions available for Origami. Each dataset only had access to the functions that involved its allowed types according to HELMUTH; SPECTOR (2015).	79
Table 5.2	The chosen types for the unbound types in Accu and Hylo. The type is colored in blue when the decision was guided by the canonical solution.	80
Table 5.3	Success rates obtained by Origami for each pattern in each benchmark. The “Best” column shows the highest success rate for that benchmark across all patterns, which is also underlined. We also show in blue the pattern of the canonical solution.	81

Table 5.4	Origami’s success rates compared to HOTGP’s on solved problems. The Δ column shows the relative success rate of Origami with respect to HOTGP.	82
Table 5.5	Success rate with the best values underlined, with the ratio of victories of each algorithm against Origami by the amount of tested problems. The last rows display the amount of problems with a success rate above a certain % for each method.	83
Table 5.6	Comparison between Origami and Copilot. Success rates with the best values underlined, with the ratio of victories of each algorithm against Origami by the amount of tested problems. The last rows display the amount of problems with a success rate above a certain % for each method.	85
Table 6.1	The function set available for Origami _{DSLs}	92
Table 6.2	Success Rates of each pattern on PSB1 for Origami _{AC/DC}	94
Table 6.3	Success Rates of each pattern on PSB2 for Origami _{AC/DC}	95
Table 6.4	Success Rates of each pattern on Polymorphic Program Synthesis Benchmark (PolyPSB) for Origami _{AC/DC}	95
Table 6.5	Success rates of different versions of Origami in PSB1.	97
Table 6.6	Success rates of GP Methods in PSB1.	99
Table 6.7	Success rates of GP Methods in PSB2.	100
Table 6.8	Success rates of GP Methods in PolyPSB.	100
Table 6.9	Success rates of Origami _{AC/DC} and Copilot in PSB1.	102
Table 6.10	Success rates of Origami _{AC/DC} and Copilot in PSB2.	102

List of Acronyms

ADATE Automatic Design of Algorithms Through Evolution

AC Abridge-by-Clipping

AC/DC Abridge-by-Clipping, Diversify-by-Culling

CBGP Code Building Genetic Programming

DC Diversify-by-Culling

DSLS Down-Sampled Lexicase Selection

EA Evolutionary Algorithm

FP Functional Programming

G3P Grammar Guided Genetic Programming

G3P+ Grammar Guided Genetic Programming Extended

GE Grammatical evolution

GP Genetic Programming

HOTGP Higher-Order Typed Genetic Programming

HOF Higher-Order Function

PBE Programming By Examples

PS Program Synthesis

PSB1 General Program Synthesis Benchmark 1

PSB2 General Program Synthesis Benchmark 2

PolyPSB Polymorphic Program Synthesis Benchmark

SFGP Strongly Formed Genetic Programming

STGP Strongly Typed Genetic Programming

UMAD Uniform Mutation by Addition and Deletion

RS Recursion Scheme

LLM Large Language Model

RHH Ramped Half-and-Half

List of Symbols

g_{ac} Interval, in generations, between successive applications of the Abridgment-by-Clipping (AC) procedure.

g_{dc} Interval, in generations, between successive applications of the Diversify-by-Culling (DC) procedure.

Contents

1	Introduction	21
1.1	Objectives	22
1.2	Organization	22
2	Background and Motivation	24
2.1	Genetic Programming	25
2.2	On Functional Programming and Types	28
2.2.1	Benefits to Program Synthesis	29
2.3	Functional PS Algorithms	31
2.4	Strongly Typed Genetic Programming	34
2.5	Final Remarks	36
3	Higher-Order Typed Genetic Programming	37
3.1	Higher-Order Typed Genetic Programming	38
3.1.1	Higher-Order Functions and λ -functions	40
3.1.2	Code Refinements	41
3.2	Results	44
3.2.1	Analysis of the results	45
3.3	Final Remarks	48
4	Origami	50
4.1	Recursion Schemes	51
4.1.1	Folding lists and trees	52
4.1.2	Functors, Fixed points and Functor Algebras	53
4.1.3	An example Recursion Scheme: catamorphism	55
4.1.4	Well-known Recursion Schemes	56
4.2	Origami	57
4.3	Recursion Schemes in PSB1	59
4.3.1	Catamorphism	59
4.3.2	Anamorphism	62
4.3.3	Hylomorphism	64
4.3.4	Accumulation	65

4.3.5	Summarizing PSB1	68
4.4	Origami Proof-of-Concept Results	69
4.5	Final Remarks	72
5	Going Bananas!	73
5.1	Origami Program Synthesis	73
5.1.1	Patterns	74
5.1.2	Genetic Programming	77
5.1.3	Set of functions	78
5.2	Experimental Results	78
5.3	Final Remarks	84
6	AC/DC	86
6.1	Abridge-by-Clipping, Diversify-by-Culling	87
6.2	Extensions to Origami	87
6.2.1	New Patterns	88
6.2.2	Selection mechanism	90
6.2.3	Function set	90
6.3	Experimental Results	91
6.3.1	Abridge-by-Clipping, Diversify-by-Culling (AC/DC) Hyperparameters	93
6.3.2	Results by Scheme	94
6.3.3	Ablation Study	96
6.3.4	Comparison to other GP methods	98
6.3.5	Comparison against LLMs	101
6.4	Final Remarks	104
7	Conclusion	105
7.1	Limitations	106
	Bibliography	109
8	Canonical solutions to PSB1 problems	117
8.1	No recursion	117
8.1.1	Number IO	117
8.1.2	Small or Large	117
8.1.3	Compare String Lengths	118
8.1.4	Median	118
8.1.5	Smallest	118
8.2	Catamorphisms	118
8.2.1	Count Odds	118
8.2.2	Double Letters	119

8.2.3	Super Anagrams	119
8.2.4	String Lengths Backwards	119
8.2.5	Negative To Zero	120
8.2.6	Scrabble Score	120
8.2.7	Mirror Image	120
8.2.8	Vectors Summed	121
8.2.9	Grade	121
8.3	Anamorphisms	122
8.3.1	For Loop Index	122
8.3.2	Digits	122
8.4	Hylomorphisms	123
8.4.1	Collatz Numbers	123
8.4.2	Sum of Squares	123
8.4.3	Even Squares	124
8.5	Accumulations	124
8.5.1	Last Index of Zero	124
8.5.2	Vector Average	125
8.5.3	Checksum	125
8.5.4	String Differences	125
8.5.5	X-Word Lines	126
8.5.6	Word Stats	127

Introduction

Computer programming can be seen as the task of creating a set of instructions that, when executed, can provide as output a solution to a specific problem. This task involves several steps, starting from an abstract description of the solution (*i.e.*, an algorithm) to a concrete implementation written in a programming language.

Given the importance of creating computer programs and the repetitive tasks usually involved, an often sought *holy grail* is the ability to automatically generate source code, either in part or in its entirety. This automatic generation would follow a certain high-level specification, which reduces the burden of manually creating the program. This problem is known as Program Synthesis (PS) (GULWANI, 2010).

A program specification is a high-level description of the objective of the program (KOZA; POLI, 2005). This specification can have different formats, from natural language to a more formal notation. A common approach to specifying a program to solve a problem is providing a set of example inputs and their expected outputs.

In this case, the task of the synthesizer is to find a program that correctly maps each pair of input-output provided by the examples. The main advantage of this approach is that it can be easier to create input-output examples than to write the actual program. On the other hand, providing a representative set containing corner cases might be difficult, possibly making the specification ambiguous and allowing for many alternative programs that do not behave as intended, even if they correctly map those examples.

PS can be modeled as a search where the search space contains the set of all possible programs valid under a pre-specified grammar. The objective is to find a program that meets the given specification. The size of the search space makes it impractical to employ a naïve approach for selecting the best candidate from the enumeration of all possible programs. For this reason, PS is often done using a (meta-)heuristic approach, usually Genetic Programming (GP) (KOZA, 1992).

Traditionally, PS research focuses on imperative programming, for example, by employing stack-based languages such as Push (SPECTOR, 2001a). However, imperative languages expose programmers and synthesizers to errors which arise from implicit state changes (RAY et al., 2017; YU, 1999). On the other hand, pure functional languages remove side effects by construction and prevent this specific category of errors. When combined with strong typing, which some functional languages employ, ill-typed programs are forbidden from even existing, thus providing one more level of defense against bugs.

These are benefits that functional programmers have reaped for a long time, but PS has just started to explore. Recent research in the field has shown that this is a powerful combination (GARROW; LONES; STEWART, 2022), with gaps that warrant exploration, as some functional constructs have still not been considered in the context of PS.

One such example is by using Recursion Schemes (RSs) (MEIJER; FOKKINGA; PATERSON, 1991). These are a reduced set of patterns that can implement a wide variety of recursive algorithms in many different data structures, by just changing the implementation of consumer and producer functions. In a PS setting, this would allow a synthesizer to work on simpler programs, as it would just have to synthesize the consumer/producer functions.

1.1 Objectives

Taking into account the motivation presented thus far, we formulate our research problem statement as follows:

Problem Statement: *PS poses challenges in terms of complexity and cost, primarily attributed to the vast search space involved. This study assesses whether the integration of functional programming principles, strong typing, and recursion schemes can mitigate these challenges, potentially leading to a more efficient synthesis, and identifies potential drawbacks associated with this approach.*

To this end, we propose and evaluate two novel PS algorithms that employ these concepts: Higher-Order Typed Genetic Programming (HOTGP) and Origami. This work also aims to contribute to the literature of PS, GP, and Functional Programming (FP) in general, by providing a theoretical background, literature review, and presenting valuable experiments to those fields.

We highlight that a considerable portion of the contents of the current text has been published in journals, conferences, and as book chapters (FERNANDES; FRANÇA; FRANCESQUINI, 2023, 2024, 2025), in which the candidate was the main author. This text will also assume basic familiarity with the syntax of the ML-language family, specifically the Haskell programming language.

1.2 Organization

The remainder of this work is organized as follows:

- **Chapter 2 - Background and Motivation** provides the theoretical background needed as a foundation for this text, as well as a literature review of recent methods;
- **Chapter 3 - Higher-Order Typed Genetic Programming** introduces and evaluates our first novel algorithm, which expands traditional Genetic Programming by using

functional programming constructs;

- **Chapter 4 - Origami** proposes Origami, a novel algorithm that employs the concept of recursion schemes to provide a framework for evolving recursive programs;
- **Chapter 5 - Going Bananas!** provides and evaluates the first complete implementation of Origami, following the description given in the previous chapter;
- **Chapter 6 - AC/DC** introduces a novel refinement procedure for a more efficient exploration of the search space, while extending Origami and performing a more comprehensive experimental evaluation;
- **Chapter 7 - Conclusion** summarizes the conclusions obtained by the previous chapters, providing some closing thoughts.

Background and Motivation

PS is defined by MANNA; WALDINGER (1980) as “the systematic derivation of a program from a given specification”. Formally, it can be described by two predicates φ and Ψ by the following:

Given an input predicate $\varphi(\mathbf{x})$ and an output predicate $\Psi(\mathbf{x}, z)$, construct a program computing a partial function $z = f(\mathbf{x})$ such that if $\mathbf{x}$ is an input vector satisfying $\varphi(\mathbf{x})$, then $f(\mathbf{x})$ is defined and $\Psi(\mathbf{x}, f(\mathbf{x}))$ is true. (MANNA; WALDINGER, 1971)

In this definition, φ dictates which inputs are valid for the given program, and Ψ compares a given input to a candidate output and checks whether that is a valid program or not.

In more practical terms, GULWANI (2010) defines PS as “the task of discovering an executable program from user intent expressed in the form of some constraints”, and describes it as consisting of three dimensions:

1. the kind of constraints that it accepts as an expression of user intent;
2. the space of programs over which it searches;
3. the search technique it employs.

A user can express their intent by providing Logical Specifications, defining certain properties that must be satisfied by the generated program; by using a natural language description; by giving a detailed trace of a program; by providing a partial or incomplete program; or by providing a set of input and output examples.

In the present work, our interest is focused on the case for input-output examples, in which PS is called Inductive Synthesis or Programming By Examples (PBE). PBE has the advantage of input-output examples being the simplest form of specification (GULWANI, 2010), and usually readily available (FRANKLE et al., 2016).

In the remainder of this chapter, we explore different approaches to the PS problem in terms of the two remaining dimensions. Section 2.1 introduces GP as a search technique for PS and explores some of the challenges faced by its implementations. In Section 2.2, we introduce Functional Programming and Type Theory concepts, discussing how they can benefit PS by constraining the search space. Section 2.3 provides a literature review

of PS methods which synthesize programs that employ Functional Programming concepts. Section 2.4 explores in detail one of those methods: Strongly Typed Genetic Programming (STGP). Finally, Section 2.5 presents concluding thoughts and connects the discussed concepts to the two methods introduced by this text.

2.1 Genetic Programming

The GP algorithm as described by Koza (KOZA, 1992; KOZA; POLI, 2005) is part of the of Evolutionary Algorithm (EA) (HOLLAND, 1975) family. EAs simulate the process of natural selection and evolution to find a locally optimal solution. In general, they work by *evolving* a set of solutions by repeatedly changing and combining them. The search is guided by a selective pressure imposed during the selection of candidates for reproduction.

In the context of PS, GP is employed to evolve candidate programs that aim to satisfy a given specification. However, other uses of GP have been documented, including symbolic regression (KOZA, 1992; KRONBERGER et al., 2024; LENAT, 1977), boolean function synthesis (KALKREUTH et al., 2023; PAWLAK; KRAWIEC, 2018), control policies (NADIZAR; MEDVET; WILSON, 2024; NORDIN; BANZHAF, 1995), and data mining applications (FREITAS, 2002), such as the discovery of classification rules, association rules, and feature selection or construction.

This process is described by Algorithm 1. It starts by randomly generating candidate solutions (called *individuals*) to the problem (Line 2). These individuals compose an initial *population*, which can be initialized using standard GP tree-generation methods.

The *grow* method expands each node by selecting either a terminal or a nonterminal

Algorithm 1: A general representation of the Genetic Programming algorithm

```

1 function evolveProgram(popSize, crossoverRate):
2   pop  $\leftarrow$  initializePop(popSize)
3   while stop criteria not met do
4     nextPop  $\leftarrow$  {}
5     while |nextPop| < popSize do
6       if tossCoinWithProb(crossoverRate) then
7         ind1, ind2  $\leftarrow$  selectTwoIndividuals(pop)
8         ind1, ind2  $\leftarrow$  crossover(ind1, ind2)
9         nextPop  $\leftarrow$  nextPop  $\cup$  {ind1, ind2}
10      else
11        ind  $\leftarrow$  selectOneIndividual(pop)
12        ind  $\leftarrow$  mutate(ind)
13        nextPop  $\leftarrow$  nextPop  $\cup$  {ind}
14    pop  $\leftarrow$  combine(pop, nextPop)

```

until the depth limit is reached, producing trees of irregular shape. In contrast, the *full* method always selects nonterminals until the last level, where only terminals are allowed, producing perfectly balanced trees. Ramped Half-and-Half (RHH) combines both: for each depth between a minimum and maximum, half of the individuals are generated with *full* and the other half with *grow*, increasing structural diversity in the initial population.

This population then undergoes a selection process in which some individuals will be chosen to generate offspring. Generally speaking, “good” individuals should have a higher probability of being selected. To quantify an individual’s quality, we specify a *fitness* measure. Then, to advance the search, the selected individuals can go through two different operations: *mutation* and *crossover*.

Mutation takes an individual and produces a modified copy by applying a perturbation to promote diversity in the population. In contrast, crossover combines two or more individuals, generating a new solution. The new solution could contain the good parts of both individuals, thus generating a better solution.

The last step involves combining the old population with the new one according to some strategy. Popular strategies for this include keeping only the new population, discarding the previous; keeping only the best individuals of the combined populations; or making the probability of an individual surviving in the new population a function of its fitness.

This entire process repeats until some stop criteria are met, such as a maximum number of generations, or when a certain fitness threshold is achieved, or when a certain computational budget is spent, such as time or memory.

In GP, programs are traditionally represented as syntax trees (KOZA; POLI, 2005). In a syntax tree, nodes are functions or operations, leaves are the values that these functions operate on, and the entire program is just an evaluable expression. Figure 2.1a illustrates the representation of the program `(2 + 5) * length("answer")`.

With this representation, the mutation operators simply replace a random subtree with a randomly generated tree, as illustrated in Figure 2.1b. As alternative approaches, we can also replace a single node for another one of the same arity, change the value contained by a leaf node, etc. The crossover operator swaps subtrees of the parents involved in the operation, as illustrated by Figure 2.2. The fitness measure used in this algorithm is any measure of error when evaluating the generated program with the input-output examples.

Even though this is a straightforward representation, others have been proposed. For example, BANZHAF et al. (1998); NORDIN (1994) propose the use of a list of instructions, while SPECTOR (2001b) defends the use of a stack of instructions.

Some notable approaches of GP for general program synthesis include:

- PushGP (HELMUTH; MCPHEE; SPECTOR, 2018a);
- Code Building Genetic Programming (CBGP) (PANTRIDGE; SPECTOR, 2020);
- Grammatical evolution (GE) (O’NEILL; RYAN, 2001);

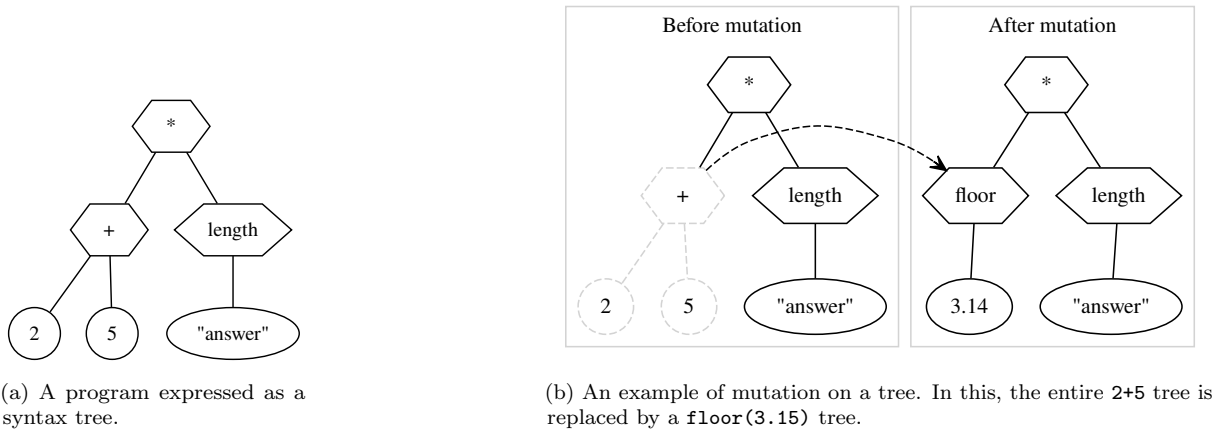


Figure 2.1: Examples of syntax trees. Leaves (terminals) are represented as ellipses and nodes (functions or non-terminals) as hexagons.

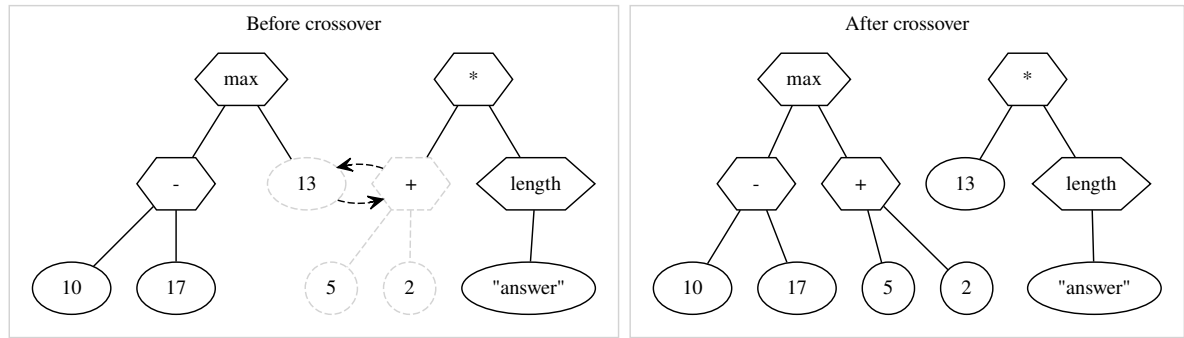


Figure 2.2: An example of crossover on two trees.

- Grammar Guided Genetic Programming (G3P) (MANRIQUE; RÍOS; RODRÍGUEZ-PATÓN, 2009).

GP also has recently been applied in many practical scenarios, including medical (ZHANG et al., 2019) and engineering applications (SHARIATI et al., 2019), project scheduling (LIN; ZHU; GAO, 2020), and COVID-19 pandemic forecasting (SALGOTRA; GANDOMI; GANDOMI, 2020).

However, even though GP has presented some success in standard PS benchmarks (HELMUTH; MCPHEE; SPECTOR, 2018b; HELMUTH; SPECTOR, 2015, 2022; HEMBERG; KELLY; O'REILLY, 2019; KELLY; HEMBERG; O'REILLY, 2019), it still struggles to consistently find solutions to some tasks that are trivial for humans. Some of the problems commonly faced are:

1. Traversing the search space is challenging, as sometimes a small change in the program code significantly impacts its output;
2. Without additional information about the program, the search relies on the completeness of the examples;

3. Some synthesizers can create stateful programs that can have unpredictable behavior depending on how the states are changed.

In the next section, we explore one possible solution to alleviate these problems, which is to employ a statically typed and purely functional paradigm.

2.2 On Functional Programming and Types

In FP, a program is a *pure* function and is defined as the composition of pure functions. A pure function has, by definition, the fundamental property of *referential transparency* (RUSSELL; WHITEHEAD, 1910; SØNDERGAARD; SESTOFT, 1990; STRACHEY, 1967).

This means that any expression (including the whole program which is, by itself, an expression) can safely be substituted by the result of its evaluation. In order to preserve this property, a pure function must have no other effect other than to compute its result, disallowing any side effects such as variable assignments, mutations, and stateful operations (including input and output).

However, iteration is a crucial part of programming, and it is typically expressed in imperative languages by using stateful loops. In order to achieve the same results, functional programs employ *recursion*. Just as the `while` loop is an abstraction of `goto`, functional languages provide abstractions to common recursion patterns, such as `map`, `filter`, and `fold` functions. Those abstractions are implemented as Higher-Order Functions (HOFs) (JONES, 1995), which in this context means a function that receives a function as one of its arguments.

Usually, functional languages also provide the tools to create λ -functions. A λ -function, or anonymous function, or simply lambda, is a function definition not bound to a name. These constructs allow the programmer to define new functions without the formality of naming them. This goes hand-in-hand with HOFs, making it practical to define functions that are just meant to be passed as arguments.

Another more general approach is RSs (MEIJER; FOKKINGA; PATERSON, 1991), which represent abstractions of `fold` and other recursive functions. RSs are a set of common strategies for performing recursion while traversing data structures. For example, consider the `fold` function present in many programming languages, which encapsulates traversing a list and producing an accumulated result, such as counting elements or performing a sum. This function is abstracted by *catamorphism*, one of the many RSs. It generalizes `fold` to any traversable data structures, and can naturally also produce the same behavior as `map` and `filter`.

It is not uncommon (although not technically required) for functional languages to be statically typed, which means that each constant has a type, and functions have argument

types and one output type. As previously stated, the type notation for this text is inspired by the ML family languages, in which we denote concrete types by an upper case name. For example, “42 is an integer” is denoted as `42 :: Int`, while “`"Hello world"` is a string” is `"Hello world" :: String`. Furthermore, functions are always in the curried format $a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_n \rightarrow b$, where a_i is the type of the i^{th} argument of the function and b is its return type: `add :: Int → Int → Int`. We write `[Int]` to represent the type of a list containing elements of type `Int` and `(Int, Bool)` to represent the type of a pair whose first element is of type `Int` and the second of type `Bool`. For other less-common data structures, we can write their names followed by their *type parameters*, such as `Set Int` to represent a set whose elements are of type `Int`, and `Map String Float` for a map whose keys are of type `String` and values are of type `Float`.

Many type systems have the feature of *parametric polymorphism* — also known as generics in some programming languages — which allows the type system to handle not only concrete types but also *type variables*. For example, consider we want to define the type of the `if` function, defined as `if condition branch1 branch2`. The first argument is the condition we want to evaluate, which must always be of type `Bool`. The two remaining arguments are the two possible branches that we pick and can be of any type, as long as they are the same and correspond to the expected return type. So, if the expected return type is `Int`, then both `branch1` and `branch2` must be of type `Int`.

Formally, we denote the type of the `if` function as $\forall a. \text{Bool} \rightarrow a \rightarrow a \rightarrow a$, which can be read as: `if` is a function that, given a value of type `Bool` and two values of the same type `a`, returns a value of type `a`. The letter `a` in this context is a *type variable*, which can be assumed to be any type as per its universal quantifier ($\forall$). In this text, a type variable is always denoted in lowercase-single-letter (*e.g.*, `a`), and $\forall$ will be henceforth implied whenever the type deals with a type variable. The main benefit of having parametric polymorphism is that there is no need for multiple similar functions and data structures whose difference is the types on which they operate.

2.2.1 Benefits to Program Synthesis

The application of FP and strong type systems helps to mitigate the three problems we described. The property of referential transparency constrains the search space to only functions with no side effects. Apart from making the search more effective, the use of pure functions is often associated to a reduction of the number of possible bugs (RAY et al., 2017). Furthermore, it also makes input transformations explicit and predictable, making mutation and crossover much more stable in comparison to imperative code.

Specifically regarding the reliance on example completeness, we argue that the user usually has more information about the program than simply some input-output examples. For example, the user may allow only operations of certain types to be used during the PS,

either by prior knowledge or by preference. The type information can play an essential role in the synthesis process as it can guide the search and constrain the search space making it easier to find programs closer to the correct solution.

In some instances, a well-defined type can constrain the program space to such an extent that only a single solution exists. Take, for example, the type signature $\mathbf{a} \rightarrow \mathbf{a}$. As explained by WADLER (1989), a function that is fully generic in $\mathbf{a}$ cannot manufacture a new value of an unknown type (*parametricity*). Under this constraint, the identity function is the only implementation allowed by that type, as any other would either violate referential transparency or the type information.

Less extreme cases exhibit similar phenomena. A function of type $(\mathbf{a}, \mathbf{b}) \rightarrow \mathbf{a}$ is uniquely determined: it must project the first component of the pair (**fst**), since no other value of type $\mathbf{a}$ can be constructed. Likewise, the type $\mathbf{a} \rightarrow \mathbf{Bool}$ admits only two implementations: a function that always returns **True** and one that always returns **False**. Even for concrete types, similar restrictions occur. For instance, $\mathbf{Bool} \rightarrow \mathbf{Bool}$ allows only four total functions: the identity, negation, the constant-**True** function, and the constant-**False** function. All of these examples illustrate how type information can constrain the space, removing unnecessary candidates (MONTANA, 1995).

Within this same line of reasoning, we can illustrate the differences between imperative and functional paradigms for PS using an example specification: given a list **xs**, return the sum of all elements smaller than 100. A valid imperative style program is presented in Algorithm 2. Changes to any of the initial values of **s** (Line 2), any of the three sections of the **for** loop (Line 3) or the constant 100 (Line 4) can drop the accuracy from 100% to 0%. The choice to access the i^{th} index might come naturally to a programmer, and is a *very* common one, but for the synthesizer it should be no different than accessing the $(i + 1)^{th}$ or 42^{nd} index. Finally, any additional statement inside the loop body that affects the value of either **i** or **s** might also decrease the achieved accuracy.

Let us compare this to a functional solution, such as the one in Algorithm 3. At first glance, we can see that the code is shorter, as the **sum** and **filter** functions abstract away many of the complexities of the imperative solution. Instead of manually iterating a list, the traversal is made implicitly by the **filter** HOF, with the current element being provided as an argument to the predicate function. Thus, the program does not have to

Algorithm 2: An imperative solution to the example specification.

```

1 function solution(xs):
2   s  $\leftarrow$  0
3   for i  $\leftarrow$  0; i < length(xs); i  $\leftarrow$  i + 1 do
4     if xs[i] < 100 then
5       s  $\leftarrow$  s + xs[i]
6   return s

```

Algorithm 3: A functional solution to the example specification.

```

1  solution :: [Int] -> Int
2  solution xs = sum (filter (\x -> x < 100) xs)

```

Algorithm 4: A RS-based solution to the example specification.

```

1  solution xs = fold f xs
2      where f x acc = if x < 100 then x + acc else acc

```

deal with the complexities of keeping track of the counter `i`: initializing it to 0; deciding to stop the iteration when it reaches the length of the list; incrementing it by 1 every time; and using it to index the list. Similarly, the `sum` function abstracts away the complexity of dealing with the `s` accumulator variable: initializing it to 0; updating its value when needed; and returning it at the end. From the synthesizer’s perspective, the only common challenge in both implementations is the discovery of the `x < 100`, which is integral to this problem.

Additionally, in this situation, we could also use the `fold` function to achieve the same results, as shown in Algorithm 4. As previously mentioned, this function is equivalent to the catamorphism RS, but specialized to the `List` data structure. The main advantage from the synthesizer’s perspective is that the path of the recursion is defined by the base structure, and it only has to search for the smaller parts of the program which define the behavior of that specific problem (the function `f`, highlighted in cyan).

We should highlight the importance of the representativeness of the examples, regardless of the paradigm, as it makes the specification less ambiguous. For instance, if the input-example lists do not contain values between 95 and 100, choosing a wrong constant for the filter/if condition, such as 98 or 96, would still achieve 100% accuracy on the training set, even though the code will produce wrong results when one considers all possible inputs.

In the following section, we present a brief review of the literature concerned with PS implementations that generate typed and/or functional programs.

2.3 Functional PS Algorithms

To the best of our knowledge, Automatic Design of Algorithms Through Evolution (ADATE) (OLSSON, 1995) is the earliest example of PS targeting functional code. This work aimed at synthesizing recursive ML language programs using incremental transformations. The algorithm starts with an initial program described by the token “?” that always returns a *don’t know* value, which is equivalent to a function that always returns null. After that, ADATE systematically expands the expression into a pattern matching of

the input type, synthesizing a program for each branch of the pattern match, and replacing the general case with a recursive call.

MONTANA (1995) proposes the STGP algorithm, an adaption of GP that considers the types of each function and terminal during the PS. Its goal is to further constrain the search space by allowing only correctly-typed programs to exist (*i.e.*, programs in which all functions operate on values with the appropriate data types).

STGP and a standard untyped GP were compared by HAYNES et al. (1995) using the “Pursuit Problem”. This problem models a game where four predators pursue a prey. The goal is to create an algorithm for the predators to capture the prey as fast as possible. The prey always runs away from the nearest predator, and the predators only have information about themselves and the prey, but not about the other predators. Results show that a good STGP program can be generated faster than a good GP program. Moreover, the best STGP program has a higher capture rate than the best GP program.

PolyGP (YU, 2001; YU; CLACK, 1998) extends STGP with support to HOFs and λ -functions. It also differs from STGP by using a type unification algorithm instead of a lookup table to determine the concrete types when using polymorphic functions. The λ -functions use the same initialization procedure of the main PS, but the available terminals are limited to the input parameters. As these λ -functions do not have any type restriction, they can be invalid, in which case they must be discarded and regenerated. The algorithm searches for a composition of λ -functions with a user-defined set of terminals and nonterminals as in STGP.

KATAYAMA (2005) proposes MagicHaskell, a breadth-search approach that searches for a correctly-typed functional program using SKIBC (TURNER, 1979) combinators. This simplifies the PS by reducing the search space. MagicHaskell also introduces the use of the de Bruijn lambda to find equivalent expressions and memoization to improve performance (DE BRUIJN, 1972). Additionally, it implements fusion rules to simplify the synthesized program further. This particular approach was reported not to work well with larger problems (PANTRIDGE et al., 2017).

Schema-Guided Inductive Functional Programming (HOFMANN, 2010) follows a data-driven analytical approach to PS. In contrast to most of the methods so far, its process is driven by the shape of the examples rather than by external exploration strategies. Instead of enumerating or evolving candidate programs, it derives the recursive structure of the target function by detecting which RS satisfies the properties presented by the input-output examples.

Strongly Formed Genetic Programming (SFGP) (CASTLE; JOHNSON, 2012) is an extension to STGP. SFGP not only assigns known data-types to terminals but also node-types to functions. A node-type identifies if a given node is a variable, an expression, or an assignment. Each subtree of a function can also be required to be of a certain node-type. The authors argue that this extra information is helpful to build correctly

typed *imperative* programs (*e.g.*, the first child of an assignment must have the “Variable” node-type and match the data-type of the second child). They conducted experiments on 3 datasets, with a reduced grammar that deals mainly with integers, and reported high success rates with a lower computational effort than competing methods.

FORSTENLECHNER et al. (2017) criticize a common technique in GP, which is to provide a different grammar for each problem. They argue that this leads to difficulties in grammar reuse, as they are specifically tailored to each problem. They propose a general grammar to the G3P algorithm, a grammar-guided system (MANRIQUE; RÍOS; RODRÍGUEZ-PATÓN, 2009). Similar to our work, they employ a different grammar for each set of types instead of a different grammar per benchmark problem, encoding its type system into its grammar, effectively enforcing that all syntactically-correct programs are also type-correct. When performing experiments on PSB1, the proposed grammar had difficulty with problems involving characters and strings. The authors then proposed an improved and expanded grammar leading to Grammar Guided Genetic Programming Extended (G3P+) (FORSTENLECHNER et al., 2018).

In this same line, GARROW; LONES; STEWART (2022) compared the generation of Python and Haskell programs using G3P. Their approach supports HOFs, but limits the function arguments to pre-defined commonly used functions. Experimental results showed that the Haskell version consistently outperforms Python in most selected benchmarks. Implementing general λ -functions was left as future work by the authors since that would add complexity to the search space and must be carefully handled as a different construct from the main program.

Notably, SWAN; KRAWIEC; KOCSIS (2019) presented the first algorithm for synthesizing programs that exploit RSs. Their work focuses only on catamorphisms over natural numbers using Peano representation (*i.e.*, the inductive type of natural numbers). The authors evaluate their approach with variations of the Fibonacci sequence, successfully obtaining the correct programs.

SANTOS; SILVA; FONSECA (2020) discuss desiderata for PS approaches by further constraining the search space, similar to what is done by STGP. They propose the use of Refinement Types to this aim. As this is an ongoing project, to the best of our knowledge, there are still no experimental evaluations or comparative results.

PANTRIDGE; HELMUTH; SPECTOR (2022) propose an adaptation of the CBGP algorithm (PANTRIDGE; SPECTOR, 2020) as a means to incorporate elements of functional programming such as HOFs and λ -functions. CBGP uses the same representation of PushGP with three primary constructs: **APP**, to apply a function; **ABS**, to define a function of 0 or more arguments and; **LET**, to introduce local variables in the current scope. It also uses concepts from type theory to ensure the correctness of the polymorphic types. CBGP achieved higher generalization rates for a subset of benchmark problems. However, for other problems, the generalization rate was close to 0. The authors noted

that the evolutionary search avoided using λ -functions and preferred to employ pre-defined functions in HOFs such as `map`. These results show some indirect evidence of the benefits provided by type-safety to PS, in particular, with regard to the stability of the solutions over different executions of the search algorithm.

HE; ARANHA; SAKURAI (2022) investigate the reuse of already synthesized programs as subprograms to be incorporated in the nonterminal set. The main idea is that, if the algorithm has already synthesized solutions to simpler tasks, these solutions can be used to build more complex solutions, in an incremental process. Their results show a significant benefit could be obtained by adding handcrafted modules in 4 selected benchmarks.

A PS algorithm that also employs RSs was recently proposed by HONG; AIKEN (2024). The algorithm uses stochastic search to synthesize programs that use paramorphism, a variation of catamorphism that also allows access to the original structure that was the source for the current accumulated value. It was evaluated on a selection of problems, both from previous benchmarks and derived from the Haskell standard library, achieving better results than its competitors.

Another approach recently explored for PS is the use of Large Language Models (LLMs). Unlike the other approaches described previously, LLMs are not grounded in GP and operate on a textual description of the problem instead of a set of examples. One example is GitHub Copilot, which was evaluated in the context of PS SOBANIA et al. (2024). It was able to synthesize a correct program more often than a selection of GP methods in more than half of the problems in PSB1 and PSB2. However, LLMs are vulnerable to data contamination, as the benchmarks and their solutions may appear in the training corpus, inflating performance through memorization rather than genuine generalization. Studies show that LLMs perform better on problems released before the training data cutoff (LIU et al., 2024). Specifically, ChatGPT achieves a higher success rate by 48 percentage points on problems published before 2021 than on those published after.

As STGP was among the first to propose and employ types for GP, it has naturally influenced following works (CASTLE; JOHNSON, 2012; SANTOS; SILVA; FONSECA, 2020). In the following section, we explore STGP’s technical details, in order to give a concrete example of how types can be integrated into GP, as well as to build the background to understand the algorithms we propose in the chapters to come.

2.4 Strongly Typed Genetic Programming

STGP is described by MONTANA (1995) as:

an enhanced version of genetic programming which enforces data type constraints and whose use of generic functions and generic data types makes it more powerful than other approaches to type constraint enforcement.

In contrast to standard GP, where a given nonterminal must be capable of handling any data type, STGP imposes extra constraints to enforce type-correctness. These constraints state that every possible terminal (*i.e.*, constants and arguments) must have a clearly defined type, and every possible non-terminal (function) must define the type of the arguments it takes and the type of its return value. In other words, it should be well defined that the type of the constant value 15 is `Int` and that the function `AddInt` takes two arguments of type `Int` and returns a value of type `Int`. Another important contribution of the STGP is that the types of the nonterminals can employ parametric polymorphism.

Using type information during evolution constrains the search space by adding two new restrictions:

1. The root node of the tree must return the expected return type of the program;
2. Every non-root node (which is, by definition, an input argument to its parent node) must return the type required by its parent node.

In order to enforce these restrictions, STGP introduces some changes to the standard untyped GP initialization process. When choosing which element to use for a node, only elements that possess the correct type, satisfying both restrictions above, are considered. STGP also keeps track of the depth of the shallowest tree that returns a given type and uses this information to, at each depth level, consider only functions whose arguments would not exceed the maximum defined depth.

Mutation and crossover were also changed, as they must respect the aforementioned restrictions. When a node is selected for mutation, it is simply discarded and replaced by another node of *the same type*. The same algorithm used by the grow initialization is used to generate valid trees of a given type. Crossover starts by randomly selecting the node in the first parent. However, for the second parent, STGP only considers nodes that return *the same type* as the node selected from the first parent. If there is no such node, crossover is not applied, and STGP returns the parents. These two modifications make it so the typing stays consistent, and an ill-typed tree is never generated from any of the two procedures.

The authors also argue in favor of handling runtime errors as part of the evolutionary process, penalizing individuals that present them. This is the opposite of the original GP approach (KOZA, 1992), which enforces that a value must always be returned. For example: a division by zero in STGP would result in an error signal, while in GP it just returns 1. STGP also introduces the `Void` type for functions that do not return anything (*i.e.*, procedures) and local variables which can be statefully changed during computation.

As for the EA part, some changes were also made, namely:

- Steady-state replacement: for each generation, only one individual (or a small number of individuals) is generated and placed in the population rather than generating a whole new population;

- Exponential fitness normalization: this introduces a Parent-Scalar constant hyperparameter $0 < P_{scalar} < 1$. When selecting parents for reproduction, the individuals are ranked by their fitness. The probability of the n^{th} best individual is given by $p(n) = P_{scalar} \times p(n - 1)$.

2.5 Final Remarks

In this chapter, we presented the necessary theoretical background used as the basis for the methods proposed by this text. We defined the concept of PS and described the dimensions that form it. Then, we introduced GP, one of the most common search techniques for PS, and explained in detail how it works. From there, we combined concepts from Functional Programming and Type Theory, and described their possible benefits to GP and PS in general. The detailed description of STGP illustrated those benefits, by extending standard GP in order to enforce strong types and discard ill-typed programs, effectively improving the search process.

In the following chapters, we introduce two new techniques. The first one, HOTGP, is related to STGP in the sense that it also employs strong typing techniques to improve GP, but by additionally allowing more advanced FP constructs such as HOFs and λ -functions. Then we introduce Origami, an idea that bases itself heavily in FP concepts in order to generate recursive programs that employ RSs.

Higher-Order Typed Genetic Programming

This chapter proposes a new GP algorithm, named Higher-Order Typed Genetic Programming (HOTGP), that searches for pure, typed, and functional programs. While still influenced by STGP, HOTGP introduces HOFs and λ -functions, drops the support for impure functions, and uses a general set of functions extracted from Haskell's base library. The main differences, detailed in the next sections, are:

- HOTGP builds programs using a pure functional program paradigm (a subset of the Haskell programming language) while STGP is modeled after a combination of typed-LISP and ADA, allowing impure functions;
- Since HOTGP is designed to only support pure functions, all side effects, including local variables (mutable state) and IO, are disallowed by design;
- As we shifted to a different language, appropriate changes to the set of functions were performed;
- Instead of specifying a strict set of terminals and non-terminals which are specific to each problem, we specify generic sets based on the input and output types¹;
- Moreover, we use a more generic set of non-terminals (all available in the standard Haskell base library) instead of very specific functions that often need to be implemented by the user. This characteristic, combined with the use of a subset of the Haskell language, allows for all the synthesized code to be immediately consumed by a Haskell compiler without modification;
- Finally, HOTGP has support for HOFs (functions that accept λ -functions as input) to handle advanced constructs in the synthesized programs;

The remainder of this chapter is organized as follows. Section 3.1 presents the HOTGP algorithm, its main differences, advantages and challenges. In Section 3.2, we present the results obtained by running HOTGP in a standard PS benchmark, comparing it to the state of the art. Section 3.3 provides a summary of the chapter, highlighting the key points of HOTGP, and discusses future work.

We highlight that the contents of this chapter have been previously presented at the Genetic and Evolutionary Computation Conference (GECCO) (FERNANDES; FRANÇA;

¹This is a design choice that was not explored by STGP nor PolyGP.

FRANCESQUINI, 2023). A considerable portion of the text presented here is an adaptation or extension of the text presented in that paper.

3.1 Higher-Order Typed Genetic Programming

Even though both HOTGP and STGP share the use of strong types, in both experimental evaluations of STGP (HAYNES et al., 1995; MONTANA, 1995), the authors employed a limited function-set specifically crafted for each one of the benchmark problems. For example, to solve the Multidimensional Least Squares Regression problem, they used a minimal set of functions with matrix and vector operators such as `matrix_transpose`, `matrix_inverse`, `mat_vec_mult`, `mat_mat_mult`. Instead, this text uses a more general set of functions, common to all problems, all of which were extracted from the standard Haskell base library.

We argue that, in a practical scenario, providing only the functions needed for each problem is undesirable since it involves giving too much information to the algorithm. This is, in our opinion, not ideal since this piece of information might not be readily available beforehand. A much more reasonable demand on the user is to ask them for the acceptable types that each problem should handle internally. This kind of information usually only requires as much intuition on the problem as providing examples. This position is also defended by FORSTENLECHNER et al. (2017), proposing that providing a specific grammar for each problem results in “bespoke grammars, making them difficult to reuse.”

HOTGP supports primitive types such as integers, floating-point numbers, booleans, and characters, as well as parametric types in the form of pairs, linked lists, and λ -functions. These can be freely combined to form more complex types, *e.g.*, a list of pairs of λ -functions or a string (represented as a list of characters).

As a consequence of using a subset of the Haskell language, HOTGP precludes the use of impure functions. An essential property of pure functions is that, being without side effects, they are easier to compose. Thus, whenever the return type of one function is the same as the input type of another function, they can be composed to form a new, more complex pure function.

The full set of functions allowed by HOTGP is shown in Table 3.1. Most functions are common operations for their specific types. Since we employ a strongly-typed language, we also require conversion functions, such as `intToFloat` and `showInt`. Additional functions of common use include `sum` and `product` for lists of numbers (integers and floating points), `range`, which generates a list of numbers (equivalent to Haskell’s `[x,y..z]`); `zip`, that pairs the elements of two lists given as input; `take`, that returns the first n elements of a list; and `unlines`, that transforms a list of strings into a single string, joining them with a newline character. In particular, `unlines` is needed for the benchmarks requiring the

Table 3.1: Functions supported by HOTGP.

Function Type	Function names
$\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$	addInt, subInt, multInt, divInt, modInt, maxInt, minInt
$\text{Bool} \rightarrow \text{Bool}$	not
$\text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$	and, or
$\text{Bool} \rightarrow a \rightarrow a \rightarrow a$	if
$\text{Float} \rightarrow \text{Float}$	sqrt
$\text{Float} \rightarrow \text{Float} \rightarrow \text{Float}$	addFloat, subFloat, multFloat, divFloat
$a \rightarrow [a]$	singleton
$a \rightarrow [a] \rightarrow [a]$	cons
$[a] \rightarrow a$	head
$[a] \rightarrow [a]$	reverse
$[[a]] \rightarrow [a]$	concat
$a \rightarrow b \rightarrow (a, b)$	toPair
$(a, b) \rightarrow a$	fst
$(a, b) \rightarrow b$	snd
$\text{Char} \rightarrow \text{Char} \rightarrow \text{Bool}$	eqChar
$\text{Char} \rightarrow \text{Bool}$	isLetter, isDigit
$\text{Int} \rightarrow \text{Float}$	intToFloat
$\text{Float} \rightarrow \text{Int}$	floor
$\text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}$	gtInt, ltInt, eqInt
$[a] \rightarrow \text{Int}$	length
$\text{Int} \rightarrow [a] \rightarrow [a]$	take
$\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow [\text{Int}]$	range
$[\text{Int}] \rightarrow \text{Int}$	sumInts, productInts
$[\text{Float}] \rightarrow \text{Float}$	sumFloats, productFloats
$[[\text{Char}]] \rightarrow [\text{Char}]$	unlines
$\text{Int} \rightarrow [\text{Char}]$	showInt
$[a] \rightarrow [b] \rightarrow [(a, b)]$	zip
$(a \rightarrow b) \rightarrow [a] \rightarrow [b]$	map
$(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$	filter

program to print text to the standard output (in our case, since we are working on a pure language, we simply return the output string).

It is worth noting that we included three *constructor* functions: `toPair`, `cons`, and `singleton`. This is a deliberate choice to simplify the function-set. Let us take 2-tuples (pairs) as an example. We must be able to cope with constructions such as `(1, 2)` or `(1 + 2, 3 * 4)` (pairs of literals and pairs of expressions). However, both of these can be represented as applications of `toPair`. The first example can be represented as `toPair 1 2`, which means applying the `toPair` function to the arguments 1 and 2. Similarly, the second example becomes `toPair (addInt 1 2) (multInt 3 4)`.

In other words, the construction of a pair is a simple function application with no special treatment. This has the added benefit of being directly compatible with the mutation

and crossover operators already defined for regular nodes. Under the same reasoning, the evolution process can generate linked lists using a combination of the `cons` and `singleton` functions. For example, the list of the literals 1, 2, 3 can be represented as `cons 1 (cons 2 (singleton 3))`; and the list of the expressions `1 + 2, 3 * 4, 5 - 6` can be represented as `cons (addInt 1 2) (cons (multInt 3 4) (singleton (subInt 5 6)))`. As was also the case with pairs, this has the added benefit of enabling crossover and mutation to happen on just the heads or just the tails of such lists.

HOTGP also allows the user to select which types it can use, to constrain the search space further. Whenever the user selects a subset of the available types, the non-terminal set is inferred from Table 3.1 by selecting only those functions that support the selected types. For example, if we select only the types `Int` and `Bool` we would allow functions such as `addInt`, `and`, `gtInt`, but would not allow functions such as `head`, `floor`, `showInt`.

Future implementations of this algorithm could support ad-hoc polymorphism, employing Haskell’s type classes, so we could simply have `add`, `mult`, `sub` that determine their types by the context instead of having specific symbols for each type.

Another important distinction from STGP to HOTGP is the absence of the `Void` type, and constructs for creating local variables. Therefore, impure functions and mutable state are not representable by HOTGP. By construction, HOTGP does not allow side effects and can only represent pure programs. On the other hand, similarly to STGP, runtime errors (such as divisions by zero) can still happen. When they do, the fitness function assigns an infinitely bad fitness value to that solution.

3.1.1 Higher-Order Functions and λ -functions

The main novelty of HOTGP is the use of HOFs. To that end, adding support to λ -functions is essential, as they are first-class values that can be used as arguments to HOFs.

The introduction of lambdas requires additional care when creating or modifying a program. When evaluated, HOTGP’s lambdas only have access to their own inputs, and not to the main program’s. In other words, they do not capture the environment in which they were created or in which they are executed. This means that lambda terminals can be essentially considered “sub-programs” inside our program, and are generated as such. We use the same initialization process from the main programs, using the function type required by the current node and employing the *grow* method. However, two additional constraints must be respected.

Constraint 1 requires all lambdas to use their argument in at least one of its subtrees, which significantly reduces the possibility of the creation of a lambda that just returns a constant value. We argue that, for higher-order-function purposes, a lambda is required to use its argument in order to produce interesting results; otherwise the program could

be simplified eliminating the use of this HOF and returning a constant. Note that this is only true because HOTGP precludes the generation of expressions with side effects.

Constraint 2 takes the form of a configurable maximum depth of the lambda-trees, which is imposed to prevent our programs from growing too large. However, as these lambdas can be nested, this hyperparameter alone is not enough to properly constrain the size of a program. For instance, take a lambda as simple as `\x -> map otherLambda x`. Depending on the allowed types, `otherLambda = \x -> map yetAnotherLambda x` would be a valid function and so on, which could lead to lambdas of arbitrarily large size. Therefore, to prevent excessively large lambda nesting, we constrain nested functions to always be `\x -> x` (the identity function).

To enforce these constraints, similarly to STGP, HOTGP employs type-possibility tables to generate lambdas during both initialization and mutation, at the point where a lambda node must be constructed. For the main program tree, as the argument and output types are known beforehand, both STGP and HOTGP only need to create two tables: one for the *grow* and one for the *full* method. However, HOTGP needs to generate lambdas involving every possible type allowed by the current program. Due to the recursive nature of the table, different argument types can lead up to vastly different type-possibility tables, so we need to keep one separate table for each possible argument type. As a corollary of Constraint 1, those tables are also guaranteed never to grow too large, as they never need to calculate possibilities for depths larger than the maximum lambda depth. These tables also differ from the main tables in the sense that they only consider a node valid if at least one of its subtrees can have an argument leaf as a descendant, enforcing Constraint 2.

In terms of mutation and crossover, lambdas are treated as regular terminals, that is, as atomic entities. Mutation never alters their internal structure: it always replaces a lambda node with a newly generated lambda produced by the method described above. Crossover treats lambdas in the same fashion: they may be selected and moved as a single unit, but no internal component is inspected, modified, or exchanged.

3.1.2 Code Refinements

A well-known difficulty faced by GP algorithms is the occurrence of *bloat* (LUKE; PANAIT, 2006), an unnecessary and uncontrollable growth of a program without any benefit to the fitness function. This happens naturally as some building blocks that apparently do not affect the program’s output survive during successive applications of crossover and mutation. Not only do these bloats make the generated program longer and unreadable, but they can also affect the performance on the test set. For example, consider the task of doubling a number and the candidate solution `x0 * (min x0 900)`. If the training set does not contain input cases such that `x0 > 900`, then this will be a correct solution from the algorithm’s point-of-view.

Algorithm 5: The local search procedure

```

1 function localSearch(tree, path):
2   node  $\leftarrow$  getNode(tree, path)
3   if node = null then return tree
4   bestTree  $\leftarrow$  tree
5   foreach child  $\in$  node.children do
6     if child.outputType  $\neq$  node.outputType then continue
7     newTree  $\leftarrow$  replaceAtPath(tree, path, child)
8     if error(newTree) < error(bestTree) then
9       // found a more correct tree
10      bestTree  $\leftarrow$  newTree
11    else if error(newTree) = error(bestTree) and
12      nNodes(newTree) < nNodes(bestTree) then
13        // found an equally-correct tree, but with fewer nodes
14        bestTree  $\leftarrow$  newTree
15  if bestTree  $\neq$  tree then
16    // do not advance: current subtree changed
17    return localSearch (bestTree, path)
18  nextPath  $\leftarrow$  nextPreOrderPath(tree, path)
19  return localSearch (tree, nextPath)

```

HELMUTH et al. (2017) empirically show that simpler programs often have a higher generalization capability, in addition to being easier to understand and reason about. PANTRIDGE; HELMUTH; SPECTOR (2022), for example, apply a refinement step at the end of the search, repeatedly trying to remove random sections of the program and checking for improvements.

To alleviate the effect of bloats, we also apply a refinement procedure on the best tree found, considering the training data. Refinement starts by applying simplification rules, which remove redundancies from the code:

- Constant evaluations: if there are no argument terminals involved in a certain tree-branch, it can always safely be evaluated to a constant value, *e.g.* `head [4*5, 1+2]  $\equiv$  20;`
- General law-application: the simplifier has access to a table of hand-written simplification procedures, which are known to be true (laws). The complete list of laws is presented in Table 3.2.

After this step, HOTGP applies a Local Search procedure that removes parts of the tree that do not contribute to correctness on the training set or that decrease it. This procedure replaces a node with each of its children, keeping the version that yields the best result according to correctness and tree size, as described by Algorithm 5. It takes as input a *tree* and a *path* identifying the current node, with *path* initially set to the root of the tree

Table 3.2: Laws applied by the simplification system.

Rules	Simplifies to
<code>if True then a else b</code>	<code>a</code>
<code>if False then b else a</code>	<code>a</code>
<code>if b then a else a</code>	<code>a</code>
<code>if (not cond) then a else b</code>	<code>if cond then b else a</code>
<code>a == a</code>	<code>True</code>
<code>a < a</code>	<code>False</code>
<code>a > a</code>	<code>False</code>
<code>a - a</code>	<code>0</code>
<code>a / a</code>	<code>1</code>
<code>mod a a</code>	<code>0</code>
<code>max a a</code>	<code>a</code>
<code>min a a</code>	<code>a</code>
<code>a && a</code>	<code>a</code>
<code>a a</code>	<code>a</code>
<code>a * b/a</code>	<code>b</code>
<code>mod (a * b) a</code>	<code>0</code>
<code>fst (a, b)</code>	<code>a</code>
<code>snd (b, a)</code>	<code>a</code>
<code>b True</code>	<code>True</code>
<code>a False</code>	<code>a</code>
<code>b && False</code>	<code>False</code>
<code>a && True</code>	<code>a</code>
<code>a + 0</code>	<code>a</code>
<code>a - 0</code>	<code>a</code>
<code>a * 1</code>	<code>a</code>
<code>b * 0</code>	<code>0</code>
<code>a / 1</code>	<code>a</code>
<code>mod b 1</code>	<code>0</code>
<code>length (singleton b)</code>	<code>1</code>
<code>length (cons b a)</code>	<code>1 + length a</code>
<code>length (reverse a)</code>	<code>length a</code>
<code>head (singleton a)</code>	<code>a</code>
<code>reverse (singleton a)</code>	<code>singleton a</code>
<code>sum (singleton a)</code>	<code>a</code>
<code>product (singleton a)</code>	<code>a</code>
<code>reverse (reverse a)</code>	<code>a</code>
<code>take (length a) a</code>	<code>a</code>
<code>range a a b</code>	<code>singleton a</code>
<code>a == if cond then a else b</code>	<code>cond</code>
<code>a == if cond then b else a</code>	<code>not cond</code>

being simplified. It then inspects the children of the current node that share its output type (Line 6). For each such child, it builds a copy of the entire tree in which the subtree at the current path is replaced by that child subtree (Line 7). These candidates, along with the original tree, are compared, and the best one in terms of error and number of nodes is kept. If a change occurs, the algorithm continues from the same position because the subtree has just been replaced (Line 13); otherwise, it advances to the next node in pre-order (Line 15). The procedure terminates when no further positions remain to be examined (Line 3).

3.2 Results

In this section, we compare HOTGP to state-of-the-art GP-based program synthesis algorithms found in the literature. For this comparison, we employ the “General Program Synthesis Benchmark Suite” (HELMUTH; SPECTOR, 2015), which contains a total of 29 benchmark problems for inductive program synthesis².

Following the recommended instructions provided by HELMUTH; SPECTOR (2015), we executed the algorithm using 100 different seeds for each benchmark problem. We used the recommended number of training and test instances and included the fixed edge cases in the training data. We also used the same fitness functions described in their paper.

For the evolutionary search, we used a steady-state replacement of 2 individuals per step, with an initial population of 1000, and using a Parent-Scalar of 99.93%. The maximum tree depth was set to 15 for the main program and 3 for λ -functions. The crossover and mutation rates were both empirically set to 50%. We allowed a maximum of 300 000 evaluations with an early stop whenever the algorithm finds a perfectly accurate solution according to the training data.

We report the percentage of correct solutions found within the 100 executions taking into consideration the training and test data sets, before and after the refinement process. To position such results with the current literature, we compare the obtained results against those obtained by:

- **PushGP**, which is the original baseline presented by the benchmark authors (HELMUTH; SPECTOR, 2015);
- Recent improvements of PushGP, namely **Uniform Mutation by Addition and Deletion (UMAD)** (HELMUTH; MCPHEE; SPECTOR, 2018b) and **Down-Sampled Lexicase Selection (DSLS)** (HELMUTH; SPECTOR, 2022);
- **G3P** (FORSTENLECHNER et al., 2017) and **G3P+** (FORSTENLECHNER et al., 2018);

²The full source code for HOTGP can be downloaded from: <https://github.com/mcf1110/hotgp>.

- **CBGP** (PANTRIDGE; HELMUTH; SPECTOR, 2022) and G3P with Haskell and Python grammars (**G3Phs** and **G3Ppy**) (GARROW; LONES; STEWART, 2022).

We have not compared with STGP and PolyGP since their original papers (MONTANA, 1995; YU, 2001) predate this benchmark suite. All the obtained results are reported in Table 3.3. In this table, all the benchmarks that could not be solved with our current function set are marked with “—” in HOTGP columns. For the other approaches, the dash mark means the authors did not test their algorithm for that specific benchmark.

3.2.1 Analysis of the results

Compared to the other algorithms, HOTGP has the highest success rate for the test set in 9 of the benchmark problems, which is a close second to the 11 problems of DSLS. Both are followed by PushGP and CBGP, which got the highest rate for 7 and 5 of the benchmarks, respectively. An important point to highlight is that HOTGP obtained a 100% success rate in 4 problems, a result only matched by CBGP. Specifically, HOTGP was the first method that was able to get a 100% success rate in *negative-to-zero* and *compare-string-lengths*, and presented significant improvements over the competition in *count-odds*, *for-loop-index*, and *small-or-large*. Moreover, HOTGP obtained at least 75% in 7 problems, leaving it tied with CBGP and just 1 problem behind DSLS. Also, it obtained 50% in 10 out of the 29 problems, placing it just a little behind DSLS, but still slightly in front of UMAD.

Overall, these results show that HOTGP is competitive with DSLS and UMAD, while still being as consistent as CBGP, obtaining 100% in more problems than the competition. These 3 methods are very recent and considered the state-of-the-art, positioning HOTGP as a strong method for PS. This brings evidence to the hypothesis that including type information and using a functional paradigm for program synthesis can, indeed, reduce the search space to improve the efficiency of the evolution process.

For example, in the *compare-string-lengths* problem, the input arguments are of the type **String**, and the output is a **Bool** but allowing intermediate **Int** type. Looking at Table 3.1, we can see that there are a few ways to convert a string to a boolean, as we only support functions in the character level. The best we can do is to extract the first character with **head** and then convert the character into a boolean with **isLetter** or **isDigit**. We could, for instance, generate a program that does that for both inputs and compares the results with different boolean operators. We could also apply a **map** function before applying **head**. Also, to convert a string into an integer given the type constraints for this problem, the only solution is to use the **length** function and then apply one of the few operators to compare two integers into a boolean. One example of obtained solution is `((length x1) > (length x0)) && ((length x1) < (length x2))`.

On the other hand, for the *last-index-of-zero* problem, a possible correct solution

Table 3.3: Successful solutions found for each problem (% of executions) considering the training (Tr) and test (Te) data sets. HOTGP* lists the results obtained with HOTGP after the simplification procedure. The best values for the test data sets of each problem are highlighted. The *collatz-numbers*, *string-differences*, *wallis-pi* and *word-stats* problems are omitted as no algorithm was able to find results for those problems.

Benchmark	HOTGP		HOTGP*		PushGP Te	UMAD Te	DSL5 Te	CBGP Te	G3P Te	G3P+		G3Phs		G3Ppy	
	Tr	Te	Tr	Te						Tr	Te	Tr	Te	Tr	Te
checksum	–	–	–	–	0	5	18	–	0	0	0	–	–	–	–
compare-string-lengths	100	100	100	100	7	42	51	22	2	96	0	94	5	12	0
count-odds	46	46	50	50	8	12	11	0	12	4	3	–	–	–	–
digits	–	–	–	–	7	11	28	0	0	0	0	–	–	–	–
double-letters	0	0	0	0	6	20	50	–	0	0	0	–	–	–	–
even-squares	0	0	0	0	2	0	2	–	1	0	0	–	–	–	–
for-loop-index	73	39	73	59	1	1	5	0	8	9	6	–	–	–	–
grade	37	32	39	37	4	0	2	–	31	63	31	–	–	–	–
last-index-of-zero	0	0	0	0	21	56	65	10	22	97	44	0	0	2	2
median	82	73	100	99	45	48	69	98	79	99	59	100	96	39	21
mirror-image	1	1	1	1	78	100	99	100	0	89	25	–	–	–	–
negative-to-zero	100	100	100	100	45	82	82	99	63	24	13	0	0	68	66
number-io	100	100	100	100	98	100	99	100	94	95	83	100	99	100	100
pig-latin	–	–	–	–	0	0	0	–	0	4	3	–	–	–	–
replace-space-with-newline	38	38	38	38	51	87	100	0	0	29	16	–	–	–	–
scrabble-score	–	–	–	–	2	20	31	–	2	1	1	–	–	–	–
small-or-large	28	59	28	59	5	4	22	0	7	39	9	30	4	0	0
smallest	98	95	100	100	81	100	98	100	94	100	73	100	100	99	89
string-lengths-backwards	87	87	89	89	66	86	95	–	68	20	18	0	0	35	34
sum-of-squares	1	1	1	1	6	26	25	–	3	5	5	–	–	–	–
super-anagrams	–	–	–	–	0	0	4	–	21	43	0	30	5	51	38
syllables	0	0	0	0	18	48	64	–	0	53	39	–	–	–	–
vector-average	78	79	80	80	16	92	97	88	5	0	0	67	4	0	0
vectors-summed	34	34	37	37	1	9	21	100	91	28	21	100	68	0	0
x-word-lines	–	–	–	–	8	59	91	–	0	0	0	–	–	–	–
# of Best Results	4		9		1	4	11	4	0	1		1		2	
= 100%	3	3	5	4	0	3	1	4	0	1	0	4	1	1	1
≥ 75%	7	6	7	7	3	7	8	7	4	6	1	5	3	2	2
≥ 50%	8	8	9	10	5	9	13	7	6	8	3	6	4	4	3
> 0%	15	15	15	15	22	21	24	9	17	19	17	8	8	8	7

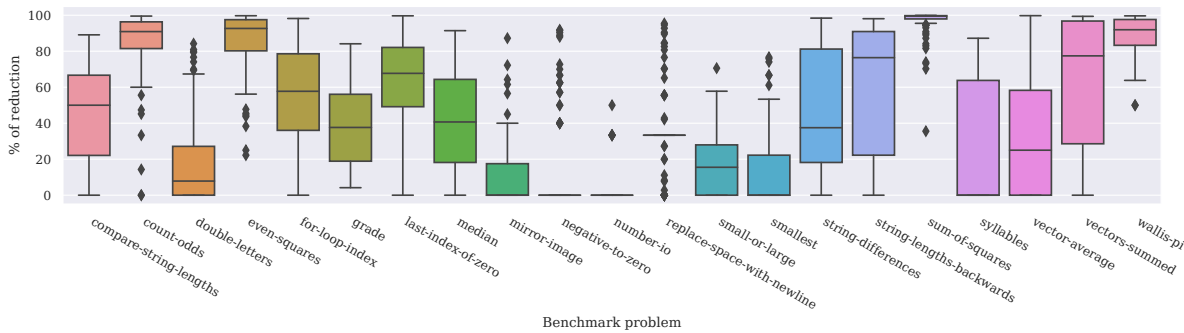


Figure 3.1: Percentage of reduction in the number of nodes caused by the refinement process.

using our function-set is `fst (head (reverse (filter ( $\lambda y \rightarrow 0 == (\text{snd } y)$ ) (zip (range 0 (length x0) 1) x0))))`. So the synthesizer must first enumerate the input, apply a filter to keep only the elements that contain 0, reverse the list, take the first element, and return its index. One of the best obtained solutions was `((length x0) + (if ((head (reverse x0)) == 0) then 1 else 0)) - 2` with 32% of accuracy. It simply checks if the last element is 0, if it is, it returns the length of the list minus one, otherwise it returns the length minus two. This is a possible general case for a recursive solution where it checks the last element and, if it is not zero, recurses with the remainder of the list.

Analysis of the code refinements

As described in Section 3.1.2, the code refinement step always produces an equal or better solution. These improvements are more noticeable on the *median* and *for-loop-index* problems. This is due to the fact that code refinement is sometimes capable of discarding misused numerical constants. For example, one solution to the *median* problem with 99% of accuracy on the training set was `max -96 (min (max x1 x2) (max (min x1 x2) x0))` that only works if the median of the three arguments is greater than -96 , otherwise it will always return a constant value. After the code refinements, HOTGP finds the final and correct solution: `min (max (min x2 x1) x0) (max x1 x2)`.

Another benefit of code refinement is reducing the program size, which can improve the readability of the generated program. Figure 3.1 shows the rate of decrease in the program size after refinements, with a geometric mean of 52%. The refinement process effectiveness varies depending on the nature of the solutions of the problem. For most problems, the end of the upper quartile is within the $> 75\%$ reduction mark, meaning it was not unusual for solutions to get largely simplified. However, more evident results are yielded in problems such as *counts-odds*, *even-squares*, and *sum-of-squares*, which dealt with fewer types (and thus a reduced function-set) and usually reached the maximum evaluation count, therefore were more susceptible to bloat. Notably, *number-io* and *negative-to-zero* had almost no reduction, showing the algorithm could directly find a perfect and near-minimal solution.

To provide further insights into how minimal the correct solutions actually are, and how susceptible to bloat each problem is, Table 3.4 takes the smallest correct solution that HOTGP could find for each problem, and compares them to the handwritten solutions crafted by the authors. Even before the refinement procedure, most of the solutions have the same node count as the handwritten ones, and nearly all of them are reasonably close. The *sum-of-squares* was initially much larger than the manual solution, but after refinement the size reduction is notable. The only correct solution we found for *mirror-image* also has the biggest reduction of the batch, showing a 87% reduction overall.

Table 3.4: Node count of the hand-crafted solutions and the smallest correct solutions found by HOTGP and HOTGP*. The node count relative to hand-crafted is shown in parenthesis.

Benchmark	Hand-crafted	HOTGP	HOTGP*
compare-string-lengths	11	11 (1.0×)	11 (1.0×)
count-odds	4	4 (1.0×)	4 (1.0×)
for-loop-index	7	25 (3.6×)	9 (1.3×)
grade	27	45 (1.7×)	29 (1.1×)
median	9	9 (1.0×)	9 (1.0×)
mirror-image	10	102 (10.2×)	13 (1.3×)
negative-to-zero	3	3 (1.0×)	3 (1.0×)
number-io	4	4 (1.0×)	4 (1.0×)
replace-space-with-newline	8	8 (1.0×)	8 (1.0×)
small-or-large	11	12 (1.1×)	12 (1.1×)
smallest	7	7 (1.0×)	7 (1.0×)
string-lengths-backwards	4	5 (1.2×)	5 (1.2×)
sum-of-squares	7	163 (23.3×)	30 (4.3×)
vector-average	6	6 (1.0×)	6 (1.0×)
vectors-summed	5	5 (1.0×)	5 (1.0×)

3.3 Final Remarks

This chapter presented HOTGP, a GP algorithm that supports HOFs, λ -functions, polymorphic types, and the use of type information to constrain the search space. It also sports a function-set based on the Haskell language using only pure functions in the non-terminals set. Our main arguments in favor of this approach are: i) limiting our programs to pure functions avoids undesirable behaviors; ii) using type-level information and parametric polymorphism reduces the search space directing the GP algorithm towards the correct solution; iii) HOFs eliminate the need of several imperative-style constructs (*e.g.*, for loops).

HOTGP differs from most GP implementations as it actively uses the information of input and output types to constrain the candidate terminals and non-terminals while creating new solutions or modifying existing ones, and to select feasible points of recombination.

We have evaluated our approach with 29 benchmark problems and compared the results with 8 state-of-the-art algorithms from the literature. Overall, HOTGP got favorable results, achieving high success rates in a wide variety of problems, being very competitive with the state of the art.

We also applied code refinements to the best solution found by the algorithm to reduce the occurrence of *bloat* code. This procedure leads to further improvements in the results while at the same time improving the readability of the final program.

Even though we achieved competitive results, we observed that there are still possible improvements. First, our non-terminals set is much smaller than some of the state-of-the-art algorithms (*e.g.*, PushGP). Future work includes carefully examining the impact of adding new functions to the function-set. This inclusion might further simplify the PS or allow us find solutions that are not currently being found. On the other hand, including new functions also expands the search space and can hinder some of our current results.

Our approach could also benefit from a more modular perspective for PS. In a modular approach, the problem is first divided into simpler tasks which are solved independently and then combined to create the complete synthesized program. This approach will require support to different forms of functional composition and the modification of the benchmark to create training data for the different subtasks. Such a synthesizer could also be coupled with Wingman³ (the current implementation of advanced Haskell code generation), which can either synthesize the whole program or guide the process using only the type information, and code holes.

Further research is also warranted concerning more advanced type-level information such as Generalized Algebraic Data Types (GADTs), Type Families, Refinement Types and Dependent Types. More type information could further constrain the search space and, in some situations, provide additional hints to the synthesis of the correct program. Clearly, this must be accompanied by a modification of the current benchmarks and the inclusion of new benchmarks that provides this high-level information about the desired program.

³<https://haskellwingman.dev/>

The previous chapter introduced HOTGP as a GP algorithm for synthesizing functional programs. We demonstrated how FP concepts can constrain the search space and guide the search in PS, leading to competitive results when applied to a GP algorithm.

However, most of the PS algorithms so far have not effectively exploited common programming patterns. Most algorithms, including HOTGP, need to scan the search space to find a correct program without any guidance in terms of structure. While a less restrictive search space can be desirable to allow the algorithm to navigate towards one of the many solutions, a constrained search space, if correctly done, can speed up the search process allowing the search algorithm to focus only on part of the program.

A general and useful pattern is the use of Recursion Schemes (RSs) (MEIJER; FOKKINGA; PATERSON, 1991). This pattern captures the common structure of recursive functions and data structures as combinations of *consumer* and *producer* functions, also known as *fold* and *unfold*, respectively. They are known to be very general and capable of implementing many commonly used algorithms, ranging from data aggregation to sorting. GIBBONS (2003) coined the name *origami programming* and showed many examples of how to write common algorithms using these patterns. The folding and unfolding process can be generalized through Recursion Schemes, which divide the programming task into three simpler steps: (i) choosing the scheme (among a limited number of choices), (ii) choosing a fixed point of a data structure that describes the recursion trace and, (iii) writing the consumer (fold) and producer (unfold) procedure.

In this chapter we studied the problems in PSB1 (HELMUTH; SPECTOR, 2015) and solved them using RSs, reporting a selection of distinct solutions. This examination revealed that most of the solutions for the proposed problems follow the common pattern of folding, unfolding a data structure, or a composition of both. With these observations, we explore the crafting of computer programs following these recursive patterns and craft program templates that follow these patterns along with an explanation of how they can constrain the search space of candidate programs. Our main goal is to find a set of recursion schemes that simplifies the PS process while reducing the search space with type information. We also present the general idea on how to evolve such programs, called *Origami*, a PS algorithm that first determines the (un)folding pattern it will evolve and then evolves the corresponding template.

The remainder of this chapter is organized as follows. In Section 4.1, we introduce

recursion schemes and explain the basic concepts needed to understand our proposal. Section 4.2 presents Origami, a novel algorithm idea to evolve functional programs based on recursion schemes. Section 4.3 outlines some examples of recursive patterns that can be used to solve common programming problems and how Origami could evolve them. In Section 4.4, we show some proof-of-concept results adapting HOTGP to one of the presented patterns and analyze the results. Finally, in Section 4.5, we give some final observations about Origami and describe future work.

We highlight that the contents of this chapter have been previously presented at the Genetic Programming Theory & Practice (GPTP) workshop (FERNANDES; FRANÇA; FRANCESQUINI, 2024). A considerable portion of the text presented here is an adaptation or extension of the text presented in that paper.

4.1 Recursion Schemes

Recursive functions are sometimes referred to as the *goto* of functional programming languages (GIBBONS, 2003; HARVEY, 1992; JONES, 2001; MEIJER; FOKKINGA; PATERSON, 1991). They are an essential part of how to build a program in these languages, allowing programmers to avoid mutable state and other imperative programming constructs. Both *goto* and recursion are sometimes considered harmful when creating a computer program. The reason is that the former can make it hard to understand the program execution flow as the program grows larger, and the latter may lead to problems such as *stack overflows* if proper care is not taken. This motivated the introduction of higher-order functions as the preferred alternative to direct recursion.

By using higher-order functions which apply the function received as input in a recursive pattern to a given value or data structure, we can represent many common recursion schemes. Some of the most well-known examples of these functions are *map*, *filter*, and *fold*. The main advantage of using these general patterns is that the programmer does not need to be as careful (guaranteeing termination, or sane memory usage for example) as they would need to be using direct recursion.

Among these higher-order functions, *fold* is the most general, as it can be used to implement the others. This pattern is capable of representing many recursive algorithms that start with a list of values and return a transformed value. Another common general pattern is captured by the *unfold* function that starts from a seed value and unfolds it, generating a list of values in the process.

While *fold* and *unfold* describe most common patterns of recursive functions, they are limited to recursions that follow a linear path (*e.g.*, a list). In some scenarios, the recursion follows a nonlinear path such as a binary tree (*e.g.*, most comparison-based sort algorithms). To generalize the *fold* and *unfold* operations to different recursive patterns, Recursion Schemes (GARLAND; LUCKHAM, 1973; MEIJER; FOKKINGA; PATERSON,

1991) describe common patterns of generating and consuming inductive data types, not limited to only the consumption or generation of data, but also abstracting the idea of having access to partial results or even backtracking. The main idea is that the recursion path is described by a fixed point of an inductive data type and the program-building task becomes limited to some specific definitions induced by the chosen data structure. This concept is explained in more detail in the following subsection.

4.1.1 Folding lists and trees

A generic inductive list (*i.e.*, linked list) carrying values of type **a** is described in Haskell as:

```
data List a = Nil | Cons a (List a)
```

This is read as “a list can be either empty (**Nil**) or a combination (using **Cons**) of a value of type **a** followed by another list (*i.e.*, its tail)”. In a similar fashion, we could recursively define a binary tree containing values of type **a** as:

```
data Tree a = Empty | Node (Tree a) a (Tree a)
```

From these, we could try to define our first recursion schemes. With the intent of keeping these first examples simple, we will start with a loose and informal definition of the catamorphism, which will be defined more formally in the following sections. To avoid confusion with the actual catamorphism definition and implementation explored in the rest of this chapter, in this section it will be simply called **fold**.

In the context of a list, we can define that **fold** recursively applies an accumulator function **f**, from the right. We start with an initial value *s* (from *seed*), and use **f** to combine it with the last element of the list. Then we use **f** again to combine this result with the penultimate element, and keep going until there are no more elements to combine. Mathematically, it can be expressed as $f(x_0, f(x_1, f(x_2, \dots f(x_{n-1}, s) \dots))$. Or, in Haskell:

```
fold :: (a -> b -> b) -> b -> List a -> b
fold _ s Nil = s
fold f s (Cons x xs) = f x (fold f s xs)
```

This simple pattern can be used to implement a wide variety of functions, such as:

```
sum xs = fold (+) 0 xs
product xs = fold (*) 1 xs
length xs = fold (\_ acc -> acc+1) 0 xs
map f xs = fold (\x acc -> Cons (f x) acc) Nil xs
concat xs = fold (++) Nil xs
```

We can also apply a similar recursion scheme to the `Tree` structure. In this context, `f` must accept the accumulated results of the left and right branches, and combine them with the actual value of that node.

```
foldTree :: (b -> a -> b -> b) -> b -> Tree a -> b
foldTree _ s Empty = s
foldTree f s (Node l x r) = f (foldTree f s l) x (foldTree f s r)
```

Now, we could also define similar functions to the ones we defined for the `List` fold:

```
sumTree tree = foldTree (\l x r -> l+x+r) 0 tree
productTree tree = foldTree (\l x r -> l*x*r) 1 tree
size tree = foldTree (\l _ r -> l+1+r) 0 tree
mapTree f tree = foldTree (\l x r -> Node l (f x) r) Empty tree
```

While this would theoretically work, we needed to define a separate `fold` function for each of the structures we wanted to work with. If we wanted to work with other recursion schemes, not only `fold`, we experience a combinatorial explosion, making it a very labor-intensive process. Thus, a generic approach would be ideal: a `fold` function that automatically works with any new structure that gets implemented — as well as structures that automatically work with any new scheme that we implement.

4.1.2 Functors, Fixed points and Functor Algebras

In order to make the recursion schemes generic, we start by noticing that both `List` and `Tree` are “functors”. Functors are type constructors `f :: Type -> Type` which support a `fmap` function, with the purpose of distributively applying a function to each of the nested elements of the Functor¹. This is expressed in Haskell by the following type class, and its instances for `List` and `Tree`:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
instance Functor List where
    fmap _ Nil = Nil
    fmap f (Cons x xs) = Cons (f x) (fmap f xs)
instance Functor Tree where
    fmap _ Empty = Empty
    fmap f (Node l x r) = Node (fmap f l) (f x) (fmap f r)
```

¹The functor must also satisfy the two functor laws: `fmap id = id` and `fmap (f ∘ g) = fmap f ∘ fmap g`

Given a functor `f`, we say that a type `p` is a fixed point if `p` is *isomorphic* to `f p`. Two types are isomorphic when there exist functions converting each to the other with neither loss of information nor change of structure: applying one after the other returns the original value. We can represent this by the following data type and helper function:

```
data Fix f = MkFix (f (Fix f))
```

```
unfix :: Fix f -> f (Fix f)
```

```
unfix (MkFix x) = x
```

`MkFix` is a *value constructor* allowing us to create values of type `Fix`. `unfix` is a helper function that extracts one layer of our nested structure. Together they witness the isomorphism mentioned above: applying `unfix` followed by `MkFix` (and vice-versa) returns the original value. In other words, `MkFix ∘ unfix = id` and `unfix ∘ MkFix = id`, which means `Fix f` and `f (Fix f)` contain exactly the same information.

The `Fix` type allows us to write recursive programs for data structures with non-recursive definitions that are very similar to those targeting data structures with recursive definitions. This becomes very useful to our purposes if we eliminate the recursive definition from the data structures by adding a second parameter:

```
data ListF a b = NilF | ConsF a b
```

```
data TreeF a b = EmptyF | NodeF b a b
```

```
-- In these functors, fmap applies the function
-- to the recursive structure, not to the elements.
```

```
instance Functor (ListF a) where
```

```
  fmap :: (b -> c) -> ListF a b -> ListF a c
```

```
  fmap f NilF = NilF
```

```
  fmap f (ConsF a b) = ConsF a (f b)
```

```
instance Functor (TreeF a) where
```

```
  fmap :: (b -> c) -> TreeF a b -> TreeF a c
```

```
  fmap f EmptyF = EmptyF
```

```
  fmap f (NodeF l a r) = NodeF (f l) a (f r)
```

In combination with `Fix`, these definitions are isomorphic to `List` and `Tree`, in the sense that they still allow us to carry the same information. Formally, we say that `Fix (ListF a)` is isomorphic to `List a`, by the means of the following morphisms:

```
fromList :: List a -> Fix (ListF a)
```

```
fromList Nil = MkFix NilF
```

```
fromList (Cons x xs) = MkFix (ConsF x (fromList xs))
```

```

toList :: Fix (ListF a) -> List a
toList (MkFix NilF) = Nil
toList (MkFix (ConsF x xs)) = Cons x (toList xs)

```

A similar reasoning could be applied to evidence the isomorphism between `Tree` and `Fix TreeF` with the functions `fromTree` and `toTree`. Notice that `ListF` and `TreeF` are functors over `b`, which means that `fmap` will not apply the function on the elements, but on the tail of the list.

Finally, RS arise from two categorical constructions: algebras and coalgebras. Given an endofunctor f , an f -algebra is a pair (a, h) where a is the carrier object and $h : fa \rightarrow a$ is the structure map. The carrier a is the domain in which the functor's shape is interpreted, and the structure map specifies how one f -structured layer is collapsed into a single value. Operationally, this corresponds to the mechanism underlying folds.

Dually, an f -coalgebra is a pair (a, h) with $h : a \rightarrow fa$. Here the carrier a provides the space of seeds from which data will be generated, and the structure map specifies how a seed expands into one f -structured layer. This is the mechanism underlying unfolds.

These constructions supply the abstract basis for recursion schemes: algebras determine how structures are consumed; coalgebras determine how they are produced. With these definitions, we have all we need to properly define our first recursion scheme.

4.1.3 An example Recursion Scheme: catamorphism

The application of an algebra into a fixed point structure is called *catamorphism*:

```

cata :: Functor f => (f a -> a) -> Fix f -> a
cata alg xs = alg (fmap (cata alg) (unfix xs))

```

$$\begin{array}{ccc}
 \text{Fix } f & \xrightarrow{\text{cata alg}} & a \\
 \text{unfix} \downarrow & & \uparrow \text{alg} \\
 f (\text{Fix } f) & \xrightarrow{\text{fmap (cata alg)}} & f a
 \end{array}$$

Given an algebra `alg`, `cata` *peels* the outer layer of the fixed-point `data`, maps itself to the nested structure, and applies the algebra to the result. In short, the procedure traverses the structure to its deepest layer and applies `alg` recursively accumulating the result. In this sense, `cata` is similar to the `foldr` function for lists.

Notice that `cata` works with any functor, as long as we can create a fixed-point from it, which is true for the `ListF` and `TreeF` data structures.

As an example, we can implement the `sum` function for lists and trees in terms of `cata`:

```

sum :: List Int -> Int
sum ls = cata sumAlg (fromList ls) where

```



```

-- starting value
sumAlg NilF = 0
-- combine the value x to the accumulator
sumAlf (ConsF x acc) = x + acc

sumTree :: Tree Int -> Int
sumTree t = cata sumAlg (fromTree t) where
  -- starting value
  sumAlg EmptyF = 0
  -- combine the value x to the accumulators
  sumAlg (NodeF lAcc x rAcc) = x + lAcc + rAcc

```

Although the general idea of defining the fixed-point of a data structure and implementing the catamorphism may look like over-complicating standard functions, the end result allows us to focus on much simpler implementations. In the special case of a list, we just need to specify the neutral element (`NilF`) and how to combine two elements (`ConsF x y`). All the inner mechanics of how the whole list is reduced is abstracted away in the `cata` function.

4.1.4 Well-known Recursion Schemes

While catamorphism is one of the simplest and most common RSs, there are less frequent patterns that hold some useful properties when building recursive programs. The most well-known recursion schemes (including the ones already mentioned) are:

- **catamorphism / anamorphism:** also known as folding and unfolding, respectively. The catamorphism aggregates the information stored in the inductive type. The anamorphism generates an inductive type starting from a seed value;
- **accumulation:** this is similar to catamorphism by reducing a structure, but allowing an extra accumulating parameter;
- **paramorphism / apomorphism:** these Recursion Schemes work as catamorphism and anamorphism. However, at every step they allow access to the original downwards structure;
- **histomorphism / futumorphism:** histomorphism allows access to every previously consumed element from the most to the least recent and futumorphism allows access to the elements that are yet to be generated.

And, of course, we can also combine these morphisms creating the *hylomorphism* (anamorphism followed by catamorphism), *metamorphism* (catamorphism followed by anamorphism), and *chronomorphism* (combination of futumorphism and histomorphism).

In the following, section we will explain some possible ideas on how to exploit these patterns in the context of program synthesis, and show different examples of programs developed using this pattern.

4.2 Origami

The main idea of Origami is to reduce the search space by breaking down the synthesis process into smaller steps. An overview of the proposed approach is outlined in Algorithm 6.

Algorithm 6: Origami Program Synthesis

```

1 function origami( $x, y, types$ ):
2   recScheme  $\leftarrow$  pickRecursionScheme(types)
3   indTypes  $\leftarrow$  pickInductiveType(types)
4   template  $\leftarrow$  pickTemplate(recScheme, indTypes, types)
5   fitness  $\leftarrow$  createFitnessFunction(template, indTypes)
6   return evolveProgramFromTemplate(template, fitness)

```

In Line 2 of the algorithm we determine the recursion scheme of the program. Since there are just a few known morphisms and the distribution of use cases for each morphism is highly skewed, some options (from the most naive to more advanced ones) are:

- Do the search heuristically (*e.g.*, following a flowchart²)
- Run multiple searches in parallel with each one of the templates
- Integrate this decision as part of the search (*e.g.*, encode into the chromosome)
- Use the type information to pre-select a subset of the templates (see. Table 4.1)
- Use the description of the problem together with a language model

Table 4.1: Association between type signatures and its corresponding recursion schemes.

Type signature	Recursion Scheme
$f\ a \rightarrow b$	catamorphism, accumulation
$a \rightarrow f\ b$	anamorphism
$a \rightarrow b$	hylomorphism

Specifically to the use of type information, as we can see in Table 4.1, the type signature can constrain the possible recursion schemes, thus reducing the search space of this choice. There are also some specific patterns in the description of the program that can help us

²<https://hackage.haskell.org/package/recursion-schemes-5.2.2.4/docs/docs/flowchart.svg>

choose one of the templates. For example, whenever the problem requires returning the position of an element, we should use accumulation.

After the choice of recursion scheme, in Line 3 we choose the appropriate base (inductive) data type. The most common choices are natural numbers, lists, and rose trees. Besides these choices, one could provide custom data structures if needed. This choice could be done employing the same same methods used in Line 2.

Line 4 deals with the choice of which specific template of evolvable functions (further explained in the next sections) will specify the parts of the program that must be evolved returning a template function to be filled by the evolutionary process. Once this is done, we can build the fitness function (Line 5) that will receive the evolved functions, wrap them into the recursion scheme, and evaluate them using the training data. Finally, we run the evolution (Line 6) to find the correct program.

To illustrate the process, let us go through the process to generate a solution to the problem *count-odds* from PSB1 (HELMUTH; SPECTOR, 2015):

Count Odds *Given a vector of integers, return the number of integers that are odd, without use of a specific even or odd instruction (but allowing instructions such as modulo and quotient).*

We can start by determining the type signature of this function:

```
countOdds :: [Int] -> Int
```

Line 2 As the type signature suggests, we are collapsing a list of values into a value of the same type. So, we should pick one of the *catamorphism* variants. Let us pick the plain catamorphism.

Line 3 In this step we need to choose a base inductive type. Since the type information tells us we are working with lists, we can use the list functor.

Line 4 As the specific template we choose the reduction to a value (from a list of integers to a single value).

We are now at this point of the code generation where we depart from the following template:

```
countOdds :: [Int] -> Int
countOdds ys = cata alg (fromList ys)

alg :: ListF Int Int -> Int
alg xs = case xs of
    NilF -> e1
    ConsF x y -> e2
```

Line 5 We still need to fill up the *gaps* `e1` and `e2` in the code. At this point, the piece of code `NilF -> e1` can only evolve to a constant value as it must return an integer and it does not have any integer available in scope. The piece of code `ConsF x y -> e2` can only evolve to operations that involve `x`, `y`, and integer constants.

Line 6 Finally, the evolution can be run. As the evolution of the functions inside the recursion scheme is well determined by the input-output types of the main function, the final solution should be:

```
countOdds :: [Int] -> Int
countOdds xs = cata alg xs

alg :: ListF Int Int -> Int
alg xs = case xs of
    NilF -> 0
    ConsF x y -> mod x 2 + y
```

4.3 Recursion Schemes in PSB1

With the purpose of motivating the use of RSs for PS in a practical scenario, we conducted an investigation of PSB1 (HELMUTH; SPECTOR, 2015) in order to find recurrent patterns and recursion schemes. To identify the different templates that can appear, we manually solved the entire PSB1 in such a way that the evolvable *gaps* of the recursion schemes have a well determined and concrete function type. In this section, we present and analyse each of the patterns we found. Particularly, we will highlight one example of each template, but the entire set of solutions is available in the appendix (Chapter 8). In the following examples, the evolvable parts of the solution are shown in underlined green, making it more evident the number and size of programs that must be evolved by the main algorithm.

It should be noted that we made some concessions in the way some programs were solved. In particular, our solutions are only concerned with returning the required values and disregard any IO operations (for instance, *print the result with a string "The results is"*) as we do not see the relevancy in evolving this part of the program at this point. This will be part of the full algorithm for a fair comparison with the current state-of-the-art.

4.3.1 Catamorphism

In the previous sections, we gave the definition of a catamorphism. Notice that all of the following solutions follow the same main form `cata alg (fromList data)`, which changes the input argument into a fixed form of a list and apply the algebra of the catamorphism.

Specifically for the catamorphism, we observed four different templates that we will exemplify in the following, from the simplest to the more complicated approaches. In what follows, we will only present the definition of the `alg` function.

Reducing a structure

The most common use case of catamorphism is to reduce a structure to a single value, or $f\ a \rightarrow a$. In this case, the algebra follows a simple function that is applied to each element and combined with the accumulated value. This template was already illustrated in the beginning of Section 4.2 with the example of `countOdds`:

Count Odds *Given a vector of integers, return the number of integers that are odd, without use of a specific even or odd instruction (but allowing instructions such as modulo and quotient).*

```
-- required primitives: constant int, mod, +
alg :: ListF Int Int -> Int
alg NilF = 0
alg (ConsF x acc) = mod x 2 + acc
```

Evolvable expressions: *given a function of type $[a] \rightarrow b$, we need to evolve:*

- i) `alg NilF`: *an expression of type b ;*
- ii) `alg (ConsF x acc)`: *an expression of type b , with the values $x :: a$ and $acc :: b$ in scope.*

Regenerating the structure: mapping

The higher-order function `map` is a `fold` that processes and reassembles the structure. Any function that can be implemented using a `map` can also be implemented through catamorphism.

Double Letters *Given a string, print (in our case, return) the string, doubling every letter character, and tripling every exclamation point. All other non-alphabetic and non-exclamation characters should be printed a single time each.*

```
-- required primitives: if-then-else, (<>), ([])
-- user provided: constant '!', constant "!!!", isLetter
alg NilF = []
alg (ConsF x xs) = if x == '!'
```

```

then "!!!" <> xs
else if isLetter x then [x,x] <> xs
else x:xs

```

The main difference from the previous example is that, in this program, at every step the intermediate result (`xs`) is a list. Notice that the `ConsF` case is still constrained in such a way that we can either insert the character `x` somewhere in `xs`, or change `x` into a string and concatenate to the result.

Evolvable expressions: given a function of type $[a] \rightarrow [b]$, we need to evolve:

i) *alg NilF*: an expression of type $[b]$;

ii) *alg (ConsF x xs)*: an expression of type $[b]$, with the values $x :: a$ and $xs :: [b]$ in scope.

Generating a function

Another possible application of catamorphism is in functions that return a function, or in Haskell notation $f\ a \rightarrow f\ a \rightarrow b$, which is read as: a function that takes two arguments of type $f\ a$ (e.g., a list of values) and returns a value of type b . This signature is equivalent to its curried form which is $f\ a \rightarrow (f\ a \rightarrow b)$: a function that takes a value of type $f\ a$ and returns a function that takes an $f\ a$ and returns a value of b . While generating a function that returns a function seems to add complexity, the type constraints can help guiding the synthesis more efficiently than if we were to interpret it as a function of two arguments.

Super Anagrams Given strings x and y of lowercase letters, return true if y is a super anagram of x , which is the case if every character in x is in y . To be true, y may contain extra characters, but must have at least as many copies of each character as x does.

```

-- required primitives: delete, constant bool
-- elem, (∈)
alg NilF = \ys -> True
alg (ConsF x f) = \ys -> elem x ys && f (delete x ys)

```

For the base case, the empty string (which will also be the end of the first string), we assume that this is a super anagram returning `True`. For the second pattern, we must remember that `xs` is supposed to be a function that receives a list and returns a boolean value. So, we first check that the second argument is not null, that `x` is contained in `ys`, and then evaluate `xs` passing `ys` after removing the first occurrence of `x`. Notice that for

the `NilF` case we are not limited to returning a constant value, we can apply any function to the second argument that returns a boolean. Thus, any function `String -> Bool` will work. Even though we have more possibilities for the base case, we can grow the tree carefully to achieve a proper solution. The same goes for the second case in which we add more possible programs as we have in our possession a char value, a string and a function that process a string.

Evolvable expressions: given a function of type $[a] \rightarrow [a] \rightarrow b$, we need to evolve:

- i) *alg NilF*: an expression of type b with the value $ys :: [a]$ in scope;
- ii) *alg (ConsF x xs)*: an expression of type b , with the values $x :: a$, $ys :: [a]$ and $f :: [a] \rightarrow b$ in scope.

Combination of patterns

More complex programs often combine two or more different tasks represented as functions that return tuples $(f\ a \rightarrow (b, c))$. If both tasks are independent and are implemented with a catamorphism, they are equivalent to applying different functions in every element of the n -tuple. The evolution process would be the same as per the previous template but we would evolve one function for each output type.

4.3.2 Anamorphism

Anamorphism can be seen as the dual of catamorphism, in the sense that it is derivated from a coalgebra:

```
ana :: Functor f => (a -> f a) -> a -> Fix f
ana coalg seed = MkFix (fmap (ana coalg) (coalg seed))
```

$$\begin{array}{ccc}
 a & \xrightarrow{\text{ana coalg}} & \text{Fix } f \\
 \text{coalg} \downarrow & & \uparrow \text{MkFix} \\
 f\ a & \xrightarrow{\text{fmap (ana coalg)}} & f\ (\text{Fix } f)
 \end{array}$$

In this function, we first apply `coalg` to `seed`, generating a structure of type `f` (usually a singleton), then we map the `ana coalg` function to the just generated data to further expand it, finally we enclose it inside a `Fix` structure to obtain the fixed point. In the context of lists, this procedure is known as `unfold` as it departs from one value and unrolls it into a list of values.

Its evolutionary template is composed of the `coalg` function, which can either generate a new value and continue the recursion; or choose to terminate the structure, effectively

providing a base-case, which ends the recursion. The most common case is when there is a predicate that handles deciding when to follow each path: one returning a terminator, and the other the data structure containing one element and one seed value. For this recursion scheme we only identified a single template in which the first argument is used as the initial seed and any remaining argument (if of the same type) is used as a constant when building the program. Of course, during the program synthesis, we may test any permutation of the use of the input arguments.

For Loop Index *Given 3 integer inputs start, end, and step, print the integers in the sequence $n_0 = \text{start}$, $n_i = n_{i-1} + \text{step}$ for each $n_i < \text{end}$, each on their own line.*

```
-- required primitives: (==), (+)
forLoopIndex :: Int -> Int -> Int -> [Int]
forLoopIndex start end step = toList (ana coalg start)
  where
    coalg seed = case seed == end of
      True -> NilF
      False -> ConsF seed (seed + step)
```

In this program, the **start** argument is the starting seed of the anamorphism and the **step** and **end** are used when defining each case. The **case** predicate must evolve a function that takes the seed as an argument and returns a boolean. To evolve such a function we are limited to the logical and comparison operators. As the type of the seed is well determined, we must compare it with values of the same type, which can be constants or one of the remaining arguments. After that, we must evolve two programs, one that creates the element out of the seed (a function of type `Int -> Int`) and the generation of the next seed.

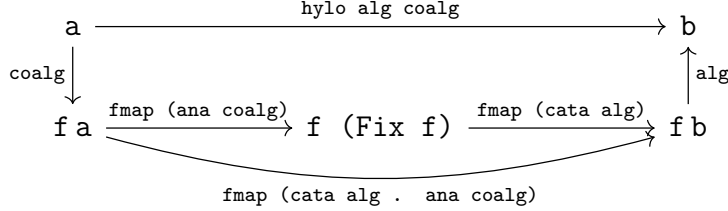
Evolvable expressions: *given a function of type $a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_n \rightarrow [b]$, we need to evolve:*

- i) initial seed: an expression of type b , with all the arguments in scope;*
- ii) predicate: and expression of type `Bool`, with `seed :: b` and all the arguments in scope;*
- iii) next seed: an expression of type b , with `seed :: b` and all the arguments in scope;*

4.3.3 Hylomorphism

Hylomorphism is the fusion of both catamorphism and anamorphism:

```
hylo :: Functor f => (f b -> b) -> (a -> f a) -> a -> b
hylo alg coalg = alg . fmap (cata alg . ana coalg) . coalg
```



Given both an algebra and a coalgebra, `hylo` uses an initial seed to generate a structure with the coalgebra, and immediately consumes it with the algebra. This template works the same as evolving the functions for both schemes.

Collatz Numbers Given an integer, find the number of terms in the Collatz (hailstone) sequence starting from that integer.

```
-- required primitives: constant int, (==)
-- (+), (*), mod, div
alg NilF = 1
alg (ConsF x acc) = 1 + acc

coalg x = case x == 1 of
    True -> NilF
    False -> ConsF x (if mod x 2 == 0
                        then div x 2
                        else div (3*x + 1) 2)
```

The single input argument is used as the initial seed. In the coalgebra, if the current seed is equal to 1 the process terminates, otherwise we generate the next hailstone number. The algebra in this case is simply the `length+1`, counting the number of generated values and offsetting it by 1 in the `NilF` case, to account for the value 1 that was dropped during the anamorphism.

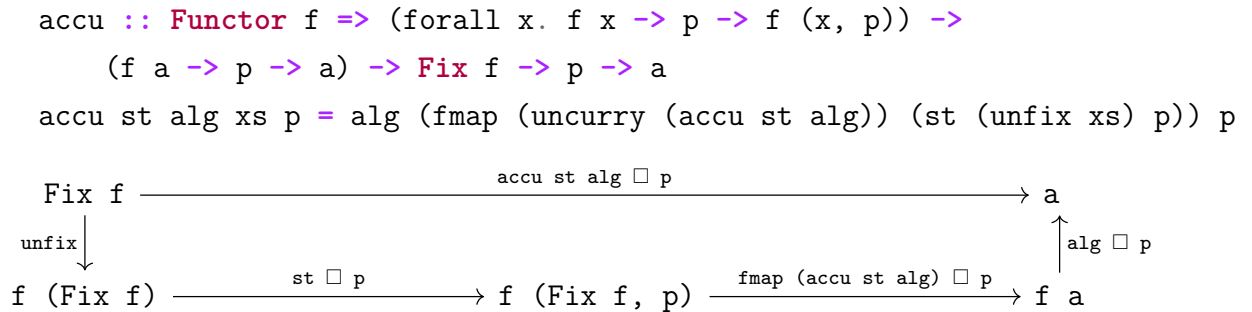
Evolvable expressions: given a function of type `a -> b`, we need to evolve:

- i) `alg NilF`: an expression of type `b`;
- ii) `alg (ConsF x xs)`: an expression of type `b`, with the values `x :: a` and `acc :: b` in scope.

- iii) *coalg predicate*: and expression of type *Bool*, with $x :: b$ in scope;
- iv) *coalg next seed*: an expression of type *b*, with $x :: b$ in scope;

4.3.4 Accumulation

In some situations our solution needs to accumulate the results and gain access to the partial results. For this purpose we can implement *accumulation*, which requires an algebra and an accumulator strategy *st*. In this diagram, we use the notation $st \sqsubseteq p$ as a shorthand for $\backslash x \rightarrow st\ x\ p$.



This pattern first traverses the structure from the top, using the accumulator strategy *st* to annotate each element of type *a* with some additional information: the accumulating parameter of type *p*. Then, it uses the algebra $f\ a \rightarrow p \rightarrow a$ in from bottom to top, just like catamorphism, but allowing access to the accumulating parameter *p* at that level. This template must be carefully used because it can add an additional degree of freedom through *p*, since this can be of any type, not limited by any of the main program types, thus expanding the search space. To avoid such a problem, we will use accumulations in very specific use-cases as described in the following sub-sections.

Indexing data

Whenever the problem requires the indexing of the data structure, we can use the accumulating parameter to store the index of each value of the structure and, afterwards, use this information to process the final solution. With this template, the parameter should be of type *Int*.

Last Index of Zero Given a vector of integers, at least one of which is 0, return the index of the last occurrence of 0 in the vector.

```

-- required primitives: if-then-else, (+), (==)
-- (<>), constant int, Maybe, Last
lastIndexZero :: [Int] -> Int

```

```

lastIndexZero xs = accu st alg (fromList xs) 0
  where
    st NilF s = NilF
    st (ConsF x xs) s = ConsF x (xs, s+1)

    alg NilF i = -1
    alg (ConsF x acc) i = if x == 0 && acc == -1
                          then i
                          else acc

```

The accumulator strategy of this program has the purpose of indexing our list, starting from 0 and incrementing it by 1 at every step. When the list is indexed, we build the result from the bottom up by signaling that we have not found a zero by initially returning -1. Whenever $x == 0$ and the currently stored index is -1, the program returns the index stored in that level (s). Otherwise, it just returns the current acc .

Evolvable expressions: given a function of type $[a] \rightarrow b$, we need to evolve:

- i) initial parameter: an expression of type Int ;
- ii) updated parameter: an expression of type Int , with $s :: Int$ in scope;
- iii) $alg\ NilF\ s$: an expression of type b , with $i :: Int$ in scope;
- iv) $alg\ (ConsF\ x\ xs)$: an expression of type b , with $i :: Int$, $x :: a$ and $acc :: b$ in scope;

Naturally, this pattern only requires accumulation as a means to build an indexed list. If we allow our solution to have an indexed list structure in the first place, this pattern is trivially transformed into a catamorphism:

```

data IListF a b = INilF | IConsF Int a b deriving Functor
toIndexedList :: [a] -> Fix (IListF a)
toIndexedList xs = go 0 xs where
  go _ [] = MkFix INilF
  go i (x:xs) = MkFix (IConsF i x (go (i+1) xs))

-- required primitives: if-then-else, (+), (==)
-- (<>), constant int, Maybe, Last
lastIndexZero :: [Int] -> Int
lastIndexZero xs = cata alg (toIndexedList xs)

```

```

where
  alg INilF = -1
  alg (IConsF i x acc) = if x == 0 && acc == -1
                        then i
                        else acc

```

A combination of catamorphisms

In some cases the recursive function is equivalent to the processing of two or more catamorphisms, with a post-processing step that combines the results. If all the operations involved are associative, this can be expressed in terms of an accumulation, as it doesn't matter whether we perform the computation top to bottom or bottom to top.

A simple example is the average of the values of a vector, in which we need to sum the values and count the length of the vector, combining both final results with the division operator. This template of accumulation constrains the type of the accumulator to a tuple of the returning type of the program.

***Vector Average** Given a vector of floats, return the average of those floats. Results are rounded to 4 decimal places.*

```

-- required primitives: (+), (/)
vecAvg :: [Double] -> Double
vecAvg xs = accu st alg (fromList xs) (0.0, 0.0)
where
  st NilF _ = NilF
  st (ConsF x xs) (s1, s2) = ConsF x (xs, (s1 + x, s2 + 1))

  alg NilF (s1, s2) = s1 / s2
  alg (ConsF x acc) s = acc

```

We illustrate this solution by splitting the accumulator expression into two distinct expression, one for each element of the tuple. While $s1 + x$ accumulates the sum of the values of the list, $s2 + 1$ increments the accumulator by one at every step. In this template, the final solution is the combination of the values at the final state of the accumulator, thus in the `alg` function we just need to evolve a function of the elements of the state.

***Evolvable expression:** given a function of type $[a] \rightarrow b$, we need to evolve:*

- i) initial value: an expression of type (b, b) ;*
- ii) updated seeds: an expression of type (b, b) , with $x :: a, xs :: [a], s1 :: b$ and $s2 :: b$ in scope;*

iii) *post-processing: an expression of type b , with $s1 :: b$ and $s2 :: b$ in scope.*

Naturally, this could also be converted into two catamorphisms if we allow for a post-processing step of type $b \rightarrow b \rightarrow b$ (assuming that the reducing operation is associative):

```
-- required primitives: (+), (/)
vecAvg :: [Double] -> Double
vecAvg xs = f (cata alg1 (fromList xs)) (cata alg2 (fromList xs))
  where
    f s1 s2 = s1 / s2

    alg1 NilF = 0.0
    alg1 (ConsF x acc) = acc + x

    alg2 NilF = 0.0
    alg2 (ConsF x acc) = acc + 1
```

4.3.5 Summarizing PSB1

From the 29 problems considered here and implemented by a human programmer, 17% were trivial enough and did not require any recursion scheme; 41% were solved using catamorphism; 20% of used accumulation (although we were required to constrain the accumulator function). Anamorphism accounted for only 7% of the problems and hylomorphism for 14%. The distribution of the usage of each recursion scheme is shown in Figure 4.1.

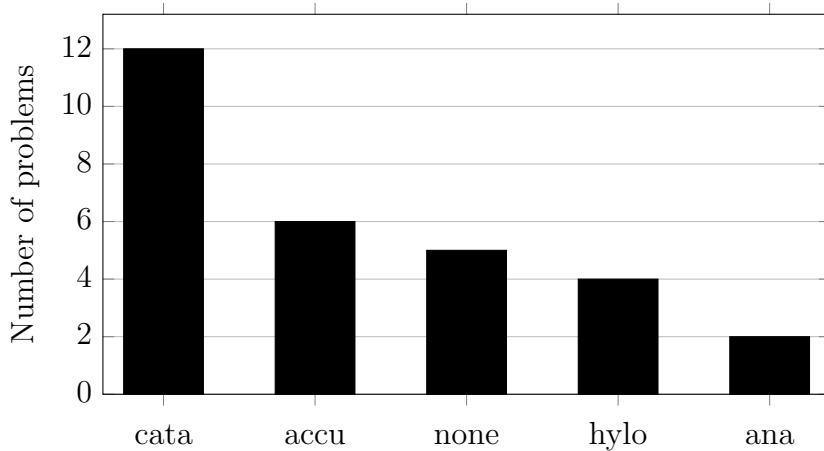


Figure 4.1: Distribution of recursion schemes used to solve the full set of PSB1 problems.

In the next section we report a simple experiment with a subset of these problems as a proof-of-concept of our approach.

4.4 Origami Proof-of-Concept Results

The main objective of this chapter is to introduce the ideas of using recursion schemes to solve programming challenges and to verify whether the current benchmark problems can be solved using this approach. In this section, we will show how using the catamorphism template can help improve the overall performance of a GP approach.

For this purpose, we adapted HOTGP to generate only the necessary *evolvable* part of a catamorphism (in here, implemented as a `foldr`) and tested most of the benchmarks that can be solved by this specific template. Specifically, we asked HOTGP to generate an expression for the `alg (ConsF x xs)` pattern. For the `alg NilF`, we used default *empty* values depending on the data-type: 0 for `Int` and `Float`, `False` for `Bool`, the space character for `Char`, and empty lists for lists and strings. Naturally, this is a simplification of the template, as all of the benchmarks we are interested in happen to use these values for the null pattern. In order to properly generate the recursion patterns, this part of the function should also be considered in the evolution.

We set the maximum depth of the tree to 5, as the expressions we want to generate are always smaller than that. This also has an effect in keeping the generated programs more readable. All the other parameters use the same values described in the previous chapter. To position Origami within the current literature, we compare the obtained results against the same algorithms HOTGP was compared to in Section 3.2.

Furthermore, we removed the benchmarks that would need the algorithm to output a function (Mirror Image, Vectors Summed, and Grade), as this is not currently supported by HOTGP. For completeness, we also tested the benchmarks that can be solved by accumulations (even though this adaptation does not support it) to show that once committed to a template (*e.g.*, catamorphism), the algorithm cannot find the correct solution if it requires a different template (*e.g.*, accumulation).

Table 4.2 shows the function set we used for the presented experimental evaluation, while Table 4.3 lists the set of functions and constants we assume that should be provided by the user, as they are contained in the problem description. Some of them can be replaced by `case-of` instructions (*e.g.*, `isVowel`, `isLetter`, `scrabbleScore`), which can increase the difficulty of obtaining a solution.

Analyzing the results depicted in Table 4.4, one can notice that when comparing the standard HOTGP with Origami, the latter always obtains an equal or better number of perfect runs, except on the accumulation benchmarks. Not only that, but the number of problems that are always solved increased from 1 to 4, and those higher than 75% increased from 3 to 6, a significant improvement in success rate. When compared to related work, out of the 7 solvable benchmarks, Origami had the best results in 6 of them. The only exception being the *replace-space-with-newline*. Overall, once we choose the correct template, the synthesis step becomes simpler.

Table 4.2: Function set used for solving the PSB1 benchmark problems.

Type class	Functions
Numbers	<code>fromIntegral</code> , <code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>^</code> <code>div</code> , <code>quot</code> , <code>mod</code> , <code>rem</code> <code>abs</code> , <code>min</code> , <code>max</code>
Logical	<code><</code> , <code><=</code> , <code>></code> , <code>>=</code> , <code>==</code> , <code>/=</code> , <code>&&</code> , <code>not</code> , <code> </code>
Lists	<code>cons</code> , <code>snoc</code> , <code><></code> , <code>head</code> , <code>tail</code> , <code>init</code> , <code>last</code> , <code>null</code> , <code>length</code> , <code>delete</code> , <code>elem</code>
Tuple	<code>fst</code> , <code>snd</code>
Map	<code>findMap</code> , <code>insertWith</code>
General purpose	<code>if-then-else</code> , <code>case</code> , <code>uncurry</code> , <code>fromEnum</code> , <code>toEnum</code> , <code>id</code>

Table 4.3: Function and constant set assumed to be provided by the user. All of these are explicitly mentioned in the problem description.

Type	Functions/Constants
<code>Int -> Bool</code>	<code>(< 1000)</code> , <code>(>= 2000)</code>
<code>String</code>	<code>"small"</code> , <code>"large"</code> , <code>"!!!!"</code> , <code>"ABCDF"</code> , <code>"ay"</code>
<code>Char</code>	<code>'!'</code> , <code>' '</code> , <code>'\n'</code>
<code>Int</code>	<code>0</code> , <code>1</code> , <code>64</code>
<code>Char -> Bool</code>	<code>isVowel</code> , <code>isLetter</code>

Table 4.4: Percentage of runs that returned a perfect solution on the validation set. The bottom part of the table summarizes the result as the number of times each algorithm had the highest percentage, and in how many problems the percentage was greater or equal to a certain threshold. The benchmarks marked with * are only solvable with accumulation.

Benchmark	Origami	HOTGP	DSLS	UMAD	PushGP	G3P	CBGP	G3P+	G3Phs	G3Ppy
checksum*	0	–	1	<u>5</u>	0	0	–	0	–	–
count-odds	<u>100</u>	50	11	12	8	12	0	3	–	–
double-letters	<u>94</u>	0	50	20	6	0	–	0	–	–
last-index-of-zero*	0	0	<u>62</u>	56	21	22	10	44	0	2
negative-to-zero	<u>100</u>	<u>100</u>	82	82	45	63	99	13	0	66
replace-space-with-newline	60	38	<u>100</u>	87	51	0	0	16	–	–
scrabble-score	<u>100</u>	–	31	20	2	2	–	1	–	–
string-lengths-backwards	<u>100</u>	89	95	86	66	68	–	18	0	34
syllables	<u>84</u>	0	64	48	18	0	–	39	–	–
vector-average*	0	80	88	<u>92</u>	16	5	88	0	4	0
# of Best Results	6	1	2	2	0	0	0	0	0	
= 100	<u>4</u>	1	1	0	0	0	0	0	0	0
≥ 75	<u>6</u>	3	4	4	0	0	2	0	0	0
≥ 50	<u>7</u>	4	<u>7</u>	5	2	2	2	0	0	1
> 0	7	5	<u>10</u>	<u>10</u>	9	6	3	7	1	3

4.5 Final Remarks

Our main hypothesis with this chapter is that, by starting the program synthesis fixing one of the recursion schemes, we simplify the process of program synthesis. For this purpose we used a general set of benchmarks widely used in the literature. Within this benchmark suite, we observed that, in most cases, the evolvable part of the programs becomes much simpler, to the point of being trivial. However, some of them require a pre-processing of the input arguments with some general use functions (such as `zip`) to keep this simplicity, or an adaptation to the output type as to return a function instead of a value.

In some cases, more complicated functions can be evolved with the help of a human interaction by asking additional information such as *when should the recursion stop?*. Also, in many cases, the type signature of each one of the evolvable programs already constrains the search space. For example, the pattern `alg NilF` must return a value of the return type of the program without using any additional information, thus the space is constrained to constant values of the return type.

Analysing the minimal function set required to solve all these problems, one can formulate a basic idea about the adequate choice based on the signature of the main function and on any user-provided type/function.

Also, every problem in the benchmark could be solved by just a handful of patterns. As the problems become more difficult and other patterns emerge, we can resort to more advanced recursions such as *dynomorphism* when dealing with dynamic programming problems, for example. Also, none of these problems required a recursive pattern with a base structure different from a list. In the future, we plan to test other benchmarks and introduce new ones that require different structures to test our approach.

One challenge to this approach is how to treat the templates containing multiple evolvable parts. For example, anamorphism requires the evolution of three expressions: one that generates the next element, one to generate the next seed, and one predicate to check for the stop condition. We will consider a multi-gene approach (SEARSON; LEAHY; WILLIS, 2010) or a collaborative co-evolution strategy (GREFENSTETTE; DALEY, 1996; SOULE; HECKENDORN, 2008).

As a final consideration, we highlight the fact that most of the programs can be further simplified if we annotate the output type with monoids. In functional programming, and Haskell in particular, monoids are a class of types that have an identity value (`mempty`) and a binary operator (`<>`) such that `mempty <> a = a <> mempty = a`. With these definitions we can replace many of the functions and constants described in Table 4.2 and Table 4.3 with `mempty` and `<>`, reducing the search space.

Going Bananas!

Chapter 4 introduced Origami as a technique for synthesizing programs by leveraging common recursive patterns as described by RSs. We showed that, for PSB1 (HELMUTH; SPECTOR, 2015), each problem can be fully solved choosing one of four RSs (cata, accu, ana, or hylo). However, these results were simply a proof-of-concept, and preliminary experiments were conducted as a way to show the feasibility of the algorithm.

In this chapter, we present the first full implementation of the Origami algorithm. This implementation aims to evaluate Origami in its complete form against PSB1, while also enabling potential improvements as well experiments in other benchmarks, in the future.

The remainder of this chapter is organized as follows. Section 5.1 presents and discusses implementation details of Origami. In Section 5.2, we report and analyze the experimental results of this implementation in PSB1, comparing it to the state of the art. Finally, Section 5.3 presents closing thoughts, highlighting the main developments of this chapter and discussing current limitations.

We highlight that the contents of this chapter have been previously presented at the 34th Brazilian Conference on Intelligent Systems (BRACIS) (FERNANDES; FRANÇA; FRANCESQUINI, 2025). A considerable portion of the text presented here is an adaptation or extension of the text presented in that paper.

5.1 Origami Program Synthesis

Origami’s implementation follows a Koza-style Genetic Programming (KOZA, 1992) (tree representation). The main distinctions to traditional approaches are the introduction of immutable nodes (ensuring a certain Recursion Scheme); and the type-safety of the genetic operators using the same approach taken by HOTGP.

The implementation is based on *patterns*, which are used to represent different RSs. A pattern is composed of immutable nodes and a set of evolvable slots that, when replaced with expressions, can be evaluated. The immutable nodes describe the main definition of the RS (see Section 5.1.1) and are fixed once we choose the pattern, while the evolvable slots represent the inner mechanisms that need to be synthesized to correspond to the expected behavior described by the dataset. These slots have a well-defined output type (inferred from the problem description), and a well-defined set of bindings to which the expression has access.

In this chapter we focus on the six different patterns that comprehend the minimal set required to solve PSB1. All the manual solutions are publicly available on GitHub¹, and are referred to as *canonical* solutions in the remainder of this text. When a pattern is used by the canonical solution to solve a problem, we also refer to it as the *canonical pattern* for that problem. Naturally, Origami is not limited to these patterns, and more could be included as needed. The next section details these six patterns.

5.1.1 Patterns

NoScheme

This is the simplest pattern in Origami, as it does not employ any recursion at all. It is represented by the following code:

```
f arg0 ... argn = slot1
```

This pattern has just a single slot, which has all the arguments in scope and returns a value of the same type as the output of the program. Its main use is to accommodate for problems that do not require any recursion.

Catamorphism over Indexed List

This pattern captures the most common RS observed in PSB1, and arguably in practical scenarios as well, *i.e.*, folding a list from the right. This pattern is commonly used in Haskell as `foldr`. In Meijer-notation (MEIJER; FOKKINGA; PATERSON, 1991), this would be represented by the banana brackets $\langle b, \oplus \rangle$, where b is the initial value and $\oplus$ is the combining function. In the context of Origami, it can be represented as:

```
f arg0 ... argn = cata alg arg0 where
  alg INil = slot1
  alg (ICons i x acc) = slot2
```

In a problem with arguments of type $i_0 \dots i_n$ and of output type o , where $i_0 \equiv [e]$ ², this pattern's slots are typed as follows:

- slot₁ :: o , with nothing in scope;
- slot₂ :: o , with scope $\{ i :: i_0 ; x :: e ; acc :: o ; arg_0 :: i_0 \dots arg_n :: i_n \}$.

This pattern will be referred simply as *Cata* in the remainder of this thesis.

¹<https://github.com/folivetti/origami-programming/>

²The notation $i_0 \equiv [e]$ is a restriction such that i_0 can be decomposed into the type $[e]$, which is the type of a list with elements of some type e .

Curried Catamorphism over Indexed List

This pattern captures a common variation of the Catamorphism, and can be represented by the following code:

```
f arg0 arg1 = cata alg arg0 arg1 where
  alg INil = \ys -> slot1
  alg (ICons i x f) = \ys -> slot2
```

As a problem of type $i_0 \rightarrow i_1 \rightarrow o$ can also be seen in its curried form as $i_0 \rightarrow (i_1 \rightarrow o)$, we can employ Catamorphism to accumulate a *function* over the first argument, and then apply this function to the second argument. This is useful, as mentioned in Chapter 4, when we need to apply a Catamorphism over the zip of two lists.

In a problem with arguments of type i_0 , i_1 ³ and of output type o , where $i_0 \equiv [e]$, this pattern's slots are typed as follows:

- slot₁ :: o , with scope $\{ ys :: i_1 \}$;
- slot₂ :: o , with scope $\{ i :: \text{Int} ; x :: e ; f :: i_1 \rightarrow o ; ys :: i_1 \}$.

For brevity, this will be referred to as simply *CurriedCata*. Note that both this and the previous pattern use Indexed Linked List as the data structure, allowing the program to access each element's index and value. For the remaining patterns we employ a regular list since it is enough to solve their problems (as shown by the canonical solutions presented in Chapter 8).

Anamorphism to a List

This pattern is commonly used in Haskell as `unfold`, which is used to generate a list. In Meijer-notation (MEIJER; FOKKINGA; PATERSON, 1991), this would be represented by the concave lenses $\llbracket g, p \rrbracket$ where g is the generator function, and p is the predicate. In the context of Origami, it can be represented by the following code:

```
f arg0 ... argn = ana coalg slot1 where
  coalg seed = if slot2 then []
               else slot3 : slot4
```

In a problem with arguments of type $i_0 \dots i_n$ and of output type o , where $o \equiv [e]$, this pattern's slots are typed as follows:

- slot₁ :: i_0 , with scope $\{ arg_0 :: i_0 \dots arg_n :: i_n \}$;
- slot₂ :: Bool , with scope $\{ seed :: i_0 ; arg_0 :: i_0 \dots arg_n :: i_n \}$;

³Notice that this pattern can only be applied to problems with exactly two arguments.

- $\text{slot}_3 :: e$, with scope $\{ \text{seed} :: i_0 ; \text{arg}_0 :: i_0 \dots \text{arg}_n :: i_n \}$;
- $\text{slot}_4 :: i_0$, with scope $\{ \text{seed} :: i_0 \}$.

Note that while we do not enforce arg_0 to be used in slot_1 , it must be of the same type as arg_0 , as all of the solutions for PSB1 respected this constraint. For brevity, this will be referred to as simply *Ana* in the rest of this thesis.

Accumulation over a List

This pattern captures using an accumulation strategy before using a `foldr`, and can be represented by the following code:

```
f arg0 ... argn = accu st alg arg0 slot1
  where
    st [] s = []
    st (x : xs) s = x : (xs, slot2)
    alg [] s = slot3
    alg (x : acc) s = slot4
```

In a problem with arguments of type $i_0 \dots i_n$ and of output type o , where $i_0 \equiv [e]$, and given a type a , this pattern's slots are typed as follows:

- $\text{slot}_1 :: a$, with scope $\{ \text{arg}_0 :: i_0 \dots \text{arg}_n :: i_n \}$;
- $\text{slot}_2 :: a$, with scope $\{ x :: e ; xs :: [e] ; s :: a ; \text{arg}_0 :: i_0 \dots \text{arg}_n :: i_n \}$;
- $\text{slot}_3 :: o$, with scope $\{ s :: a ; \text{arg}_0 :: i_0 \dots \text{arg}_n :: i_n \}$;
- $\text{slot}_4 :: o$, with scope $\{ x :: e ; \text{acc} :: o ; s :: a ; \text{arg}_0 :: i_0 \dots \text{arg}_n :: i_n \}$.

This is the first pattern whose types are not fully determined by the type of the arguments and the expected output type: the accumulator type a . Types such as this will be referred to as *unbound types*. To keep the implementation simple, we assume unbound types are known and provided by the user. The exploration of different types is an interesting challenge that warrants dedicated research. This pattern will be referred to as simply *Accu* in the rest of this thesis.

Hylomorphism through a List

This pattern captures an Anamorphism followed by a Catamorphism, such as applying `foldr` to the result of `unfold`, in Haskell. In Meijer-notation (MEIJER; FOKKINGA; PATERSON, 1991), this would be represented by the envelopes $\llbracket (c, \oplus), (g, p) \rrbracket$. In Origami, it is represented by the following code:

```

f arg0 ... argn = hyl0 alg coalg arg0 where
  coalg seed = if slot1 then [] else slot2 : slot3
  alg [] = slot4
  alg (x : acc) = slot5

```

In a problem with arguments of type $i_0 \dots i_n$ and of output type o , **and given a type a** , this pattern's slots are typed as follows:

- slot₁ :: Bool, with scope { seed :: i_0 ; arg₀ :: $i_0 \dots i_n$ };
- slot₂ :: a , with scope { seed :: i_0 ; arg₀ :: $i_0 \dots i_n$ };
- slot₃ :: i_0 , with scope { seed :: i_0 ; arg₀ :: $i_0 \dots i_n$ };
- slot₄ :: o , with nothing in scope;
- slot₅ :: o , with scope { x :: a ; acc :: o ; arg₀ :: $i_0 \dots i_n$ }.

This pattern also contains an unbound type: the intermediary list has elements of type a . This pattern will be referred to as *Hyl₀*.

5.1.2 Genetic Programming

Origami synthesizes the evolvable slots using a GP (KOZA, 1992) algorithm. Since the patterns require more than a single slot, we represent each solution as a collection of programs represented as expression trees, in a multi-gene representation (SEARSON; LEAHY; WILLIS, 2010). Each element of this collection corresponds to one of the slots.

The GP starts with an initial random population of 1 000 individuals, and iterates by applying either crossover to a pair of parents, or mutation to a single parent, generating 1 000 new individuals in total. The entire population is replaced by the offspring population.

The initial population is generated using a *ramped half-and-half* method, where half of the individuals are generated using the *full* method and half using the *grow* method. The maximum depth for each method varies between 1 and 5. The parental selection is performed using a tournament selection of size 10.

Following a simple GP algorithm, in Origami the mutation randomly selects one of the evolvable slots, then picks one point in the tree at random to be replaced by a new subtree generated at random using the grow method, with a maximum depth of $5 - d_{current}$. Crossover also starts by picking one of the slots at random, then performing one of these two actions with equal probability: i) swap the entire slot of one parent with the same slot of the other parent; ii) swap two subtrees of the same output type from each parent.

5.1.3 Set of functions

In HOTGP (Chapter 3), the selected functions focused on providing a minimal set of operations, including HOFs, that would enable the synthesis of programs under the functional programming paradigm.

With Origami, however, the main focus is assessing RSs as the only means of synthesizing recursive programs. Therefore, we designed our function-set to avoid implicitly recursive functions, like `map`, `filter`, `sum`, and `product`. We acknowledge that this might remove shortcuts and potentially make the synthesis of certain problems harder. Notice that the recursion happens in the immutable nodes that describe the RSs, so the recursion is provided rather than evolved. Additionally, the set of operations includes functions equivalent to those used by other methods, in particular those implemented by PushGP (HELMUTH; MCPHEE; SPECTOR, 2018a). As a result, Origami has a larger set of operations than HOTGP. The full set of functions is presented in Table 5.1. Once an execution is finished, the champion’s slots are refined using the same procedure that was used in HOTGP (Section 3.1.2, sans the hand-written simplification rules). To refine a tree, we pick the root node and check if replacing it with any of its children leads to a correctly-typed solution with an equal or better fitness. If so, we replace it with the best child; otherwise, we keep the original node. This process continues recursively, traversing the tree and greedily replacing nodes with their children when needed. This procedure applies Occam’s Razor to choose a simpler solution (HELMUTH et al., 2017), making sure the fitness in the training set is never worse.

5.2 Experimental Results

To evaluate our approach we conducted experiments to perform an automatic search for different patterns in the PSB1 (HELMUTH; SPECTOR, 2015) context. For each of the 29 datasets, we sequentially tried each pattern in increasing order of complexity: NoScheme; Cata, if `arg0` is a list; CurriedCata, if the problem has two arguments and `arg0` is a list; Ana, if the return type is a list; Accu, if `arg0` is a list; Hylo.

For each dataset, we executed 30 seeds of each pattern starting from the simplest and testing other patterns if none of the seeds succeeded in finding a solution (*i.e.*, the success rate was 0%). Each seed followed the instructions provided by PSB1, using the recommended number of training and test instances, and included the fixed edge cases in the training data, as well as using the fitness functions described in (HELMUTH; SPECTOR, 2015). We also made the same adaptations to the benchmarks as in Chapter 3 and Chapter 4, which are similar to the ones made by SOBANIA et al. (2024). Specifically, we changed the input of the grade benchmark from `(arg0, arg1, arg2, arg3, arg4)` to `[(arg0, 'A'), (arg1, 'B'), (arg2, 'C'), (arg3, 'D')], arg4`; and, since we

Table 5.1: The complete set of functions available for Origami. Each dataset only had access to the functions that involved its allowed types according to HELMUTH; SPECTOR (2015).

Operations	Types
add, sub, mult, div, quot, mod, rem, min, max	Int -> Int -> Int
abs, succ, pred	Int -> Int
add, sub, mult, div, min, max	Float -> Float -> Float
abs, sqrt, sin, cos, succ, pred	Float -> Float
fromIntegral	Int -> Float
floor, ceiling, round	Float -> Int
lt, gt, gte, lte	Int -> Int -> Bool
lt, gt, gte, lte	Float -> Float -> Bool
and, or	Bool -> Bool -> Bool
not	Bool -> Bool
if	Bool -> a -> a -> a
eq, neq	a -> a -> Bool
show	a -> [Char]
charToInt	Char -> Int
intToChar	Int -> Char
isLetter, isSpace, isDigit	Char -> Bool
length	[a] -> Int
cons, snoc	a -> [a] -> [a]
mappend	[a] -> [a] -> [a]
elem	a -> [a] -> Bool
delete	a -> [a] -> [a]
null	[a] -> Bool
head, last	[a] -> a
tail, init	[a] -> [a]
zip	[a] -> [b] -> [(a, b)]
replicate	Int -> a -> [a]
enumFromThenTo	Int -> Int -> Int -> [Int]
reverse	[a] -> [a]
splitAt	Int -> [a] -> ([a], [a])
intercalate	[a] -> [a] -> [a]
fst	(a, b) -> a
snd	(a, b) -> b
mkPair	a -> b -> (a, b)
apply	(a -> b) -> a -> b
singleton	a -> b -> Map a b
insert	a -> b -> Map a b -> Map a b
insertWith	((b, b) -> b) -> a -> b -> Map a b -> Map a b
fromList	[(a, b)] -> Map a b

only generate pure programs, we adapted the output to return the results instead of printing them on: checksum, digits, even-squares, for-loop-index, grade, pig-latin, replace-space-w-nl., string-differences, syllables, and word-stats.

Note that we deliberately placed the patterns with unbound types at the end of the sequence. Therefore, the unbound type in both Accu and Hylo is only decided after all other patterns have failed. For the benchmarks that Origami failed to find a solution with the other patterns, we applied one of these two patterns choosing the type that was known to be correct according to the canonical solutions. For the cases in which the canonical solutions did not use Accu or Hylo, we chose a reasonable type as needed (see Table 5.2).

The maximum tree depth was set to 5 for each slot. As Origami is based in HOTGP, which was empirically shown to be robust to changes to the crossover rate, we set it to the same value as HOTGP (50%). We allowed a maximum of 300 000 evaluations with an early stop whenever the algorithm finds a perfectly accurate solution according to the

training data. For patterns in which termination is not guaranteed, namely Ana and Hylo, a maximum number of iterations was imposed (empirically set to 10 000). Non-termination is also an issue that can occur with CurriedCata. Specifically, Origami was synthesizing solutions with the slot `alg (Cons i x f) = \ys -> f (f ys)`, essentially creating a “fork bomb”. To tackle this, a maximum execution budget is enforced when the evaluation of a single iteration of a slot executes more than 10 000 operations. In this case, the program is assigned an infinitely bad fitness. This limit was reached by less than 0.5% of the individuals.

Table 5.3 shows the percentage of executions in which Origami was able to synthesize a solution that completely solved the test set (*i.e.*, success rate). Origami found a solution for all of the problems that were canonically solved by NoScheme as well as Cata. Surprisingly, it was also able to synthesize a solution for for-loop-index by using NoScheme, even though the canonical solution used Ana, and for grade by using Cata when the canonical solution used CurriedCata. Nonetheless, we also ran these problems with their canonical patterns and discovered Origami was also able to synthesize solutions, albeit less often. Moreover, Origami was able to find the solutions for 3 out of the 4 canonical CurriedCata problems, and 2 out of the 3 Ana problems. Accu and Hylo, however, appear to be the most difficult patterns to synthesize, as no solution for problems that canonically involve these patterns was found.

Considering the 4 canonical Accu problems, checksum and word-stats are historically hard, with few methods ever finding a solution. The same can be said for Hylo in the wallis-pi and collatz-numbers problems.

In vector-average, the canonical solution involved using Accu to compute both the sum and the count as a pair in the `st` slots, and using the `alg` slots to perform the division as a post-processing step, finally obtaining the average. The solution that got closer to the intended result was the following:

Table 5.2: The chosen types for the unbound types in Accu and Hylo. The type is colored in blue when the decision was guided by the canonical solution.

Dataset	Accu	Hylo
checksum	<code>Int</code>	<code>Int</code>
collatz-numbers	–	<code>Int</code>
digits	–	<code>Int</code>
pig-latin	–	<code>[Char]</code>
string-differences	<code>Int</code>	<code>(Char, Char)</code>
sum-of-squares	–	<code>Int</code>
vector-average	<code>(Float, Int)</code>	<code>Int</code>
wallis-pi	–	<code>Float</code>
word-stats	<code>((Int, Int), (Int, Int))</code>	<code>[Char]</code>
x-word-lines	<code>Int</code>	<code>[Char]</code>

Table 5.3: Success rates obtained by Origami for each pattern in each benchmark. The “Best” column shows the highest success rate for that benchmark across all patterns, which is also underlined. We also show in blue the pattern of the canonical solution.

Dataset	NoScheme	Cata	CurriedCata	Ana	Accu	Hylo	Best
checksum	0	0	—	—	0	0	0
collatz-numbers	0	—	—	—	—	0	0
compare-string-lengths	<u>90</u>	—	—	—	—	—	90
count-odds	0	<u>40</u>	—	—	—	—	40
digits	0	—	—	0	—	0	0
double-letters	0	<u>3</u>	—	—	—	—	3
even-squares	0	—	—	<u>3</u>	—	—	3
for-loop-index	<u>90</u>	—	—	67	—	—	90
grade	0	<u>100</u>	10	—	—	—	100
last-index-of-zero	0	<u>70</u>	—	—	—	—	70
median	<u>97</u>	—	—	—	—	—	97
mirror-image	<u>93</u>	—	—	—	—	—	93
negative-to-zero	0	<u>87</u>	—	—	—	—	87
number-io	<u>100</u>	—	—	—	—	—	100
pig-latin	0	0	—	—	—	0	0
replace-space-w-nl.	0	<u>3</u>	—	—	—	—	3
scrabble-score	0	<u>100</u>	—	—	—	—	100
small-or-large	<u>53</u>	—	—	—	—	—	53
smallest	<u>100</u>	—	—	—	—	—	100
string-differences	0	0	0	—	0	0	0
string-lengths-back.	0	<u>97</u>	—	—	—	—	97
sum-of-squares	0	—	—	—	—	0	0
super-anagrams	0	0	<u>73</u>	—	—	—	73
syllables	0	<u>7</u>	—	—	—	—	7
vector-average	0	0	—	—	0	0	0
vectors-summed	0	0	<u>20</u>	—	—	—	20
wallis-pi	0	—	—	—	—	0	0
word-stats	0	0	—	—	0	0	0
x-word-lines	0	0	0	—	0	0	0

```
accu st alg arg0 (last arg0, length arg0)
```

```
where
```

```
st [] s = []
```

```
st (x : xs) s = x : (xs, s)
```

```
alg [] s = min 0 (last arg0)
```

```
alg (x : acc) s = acc + ((max (x - acc) x) / (snd s))
```

Origami took a different approach from the canonical solution, by storing the length of the input in the second element of the tuple while having no use for the first element. The `st` section had no other purpose than to transmit this pre-processing step to the `alg` section. This solution got a perfect score during training but failed in testing for certain

Table 5.4: Origami’s success rates compared to HOTGP’s on solved problems. The Δ column shows the relative success rate of Origami with respect to HOTGP.

Dataset	Origami	HOTGP	Δ	Dataset	Origami	HOTGP	Δ
scrabble-score	100	0	100	number-io	100	100	0
mirror-image	93	1	92	sum-of-squares	0	1	-1
super-anagrams	73	0	73	median	97	99	-2
last-index-of-zero	70	0	70	small-or-large	53	59	-6
grade	100	37	63	compare-string-lengths	90	100	-10
for-loop-index	90	59	31	count-odds	40	50	-10
string-lengths-back.	97	89	8	negative-to-zero	87	100	-13
syllables	7	0	7	vectors-summed	20	37	-17
double-letters	3	0	3	replace-space-w-nl.	3	38	-35
even-squares	3	0	3	vector-average	0	80	-80
smallest	100	100	0				

cases. If we were to replace $\min 0$ (last $\arg_0$) by 0 and $\max (x - \text{acc}) x$ by x , then this solution would be correct.

The Hylo solution for sum-of-squares employed `coalg` to generate a list of all the numbers from 0 to $\arg_0$, and then used `alg` to square each number and sum them. Even though this was the simplest Hylo solution, its 5 different slots incur an increased search space in relation to other patterns, which seems challenging for the algorithm.

Table 5.4 compares Origami’s results to HOTGP’s. There was a substantial increase (> 30) in the success rate in 6 problems. In the 17 problems where the absolute difference is < 30 , we highlight syllables, double-letters and even-squares problems, as those were problems for which HOTGP was not able to synthesize a solution, whereas Origami was successful at least once. The two problems with a more noticeable decrease are replace-space-w-nl. and vector-average. These can be explained by the change in function-set between the two algorithms, as HOTGP’s solutions were arguably simpler due to having `map` and `filter` for replace-space-w-nl. and `sum` for vector-average. In a practical scenario, the inclusion of these functions would likely lead to a correct solution but, as previously noted, removing them was a conscious decision to enable the proper assessment of the impact of Recursion Schemes in PS. It would also allow for composite solutions, such as using Ana with a `map` inside instead of relying on Hylo to find the entire pattern, which might be easier to synthesize.

To assess how Origami fares with relation to the best methods found in the literature, we compare its results to those obtained by different versions of PushGP (HELMUTH; MCPHEE; SPECTOR, 2018b; HELMUTH; SPECTOR, 2015, 2022), Grammar Guided Genetic Programming (G3P) (FORSTENLECHNER et al., 2017), the extended grammar version of G3P (here called G3P+) (FORSTENLECHNER et al., 2018), Code Building Genetic Programming (CBGP) (PANTRIDGE; HELMUTH; SPECTOR, 2022), G3P with Haskell and Python grammars (G3Phs and G3Ppy) (GARROW; LONES; STEWART,

Table 5.5: Success rate with the best values underlined, with the ratio of victories of each algorithm against Origami by the amount of tested problems. The last rows display the amount of problems with a success rate above a certain % for each method.

Dataset	Origami	HOTGP	DSLS	UMAD	PushGP	CBGP	G3P	G3P+	G3Phs	G3Ppy
checksum	0	0	18	5	0	–	0	0	–	–
compare-string-lengths	90	100	51	42	7	22	2	0	5	0
count-odds	40	50	11	12	8	0	12	3	–	–
digits	0	0	28	11	7	0	0	0	–	–
double-letters	3	0	50	20	6	–	0	0	–	–
even-squares	3	0	2	0	2	–	1	0	–	–
for-loop-index	90	59	5	1	1	0	8	6	–	–
grade	100	37	2	0	4	–	31	31	–	–
last-index-of-zero	70	0	65	56	21	10	22	44	0	2
median	97	99	69	48	45	98	79	59	96	21
mirror-image	93	1	99	100	78	100	0	25	–	–
negative-to-zero	87	100	82	82	45	99	63	13	0	66
number-io	100	100	99	100	98	100	94	83	99	100
pig-latin	0	0	0	0	0	–	0	3	–	–
replace-space-w-nl.	3	38	100	87	51	0	0	16	–	–
scrabble-score	100	0	31	20	2	–	2	1	–	–
small-or-large	53	59	22	4	5	0	7	9	4	0
smallest	100	100	98	100	81	100	94	73	100	89
string-lengths-back.	97	89	95	86	66	–	68	18	0	34
sum-of-squares	0	1	25	26	6	–	3	5	–	–
super-anagrams	73	0	4	0	0	–	21	0	5	38
syllables	7	0	64	48	18	–	0	39	–	–
vector-average	0	80	97	92	16	88	5	0	4	0
vectors-summed	20	37	21	9	1	100	91	21	68	0
x-word-lines	0	0	91	59	8	–	0	0	–	–
Win ratio vs. Origami		$\frac{9}{25}$	$\frac{10}{25}$	$\frac{9}{25}$	$\frac{7}{29}$	$\frac{5}{14}$	$\frac{3}{29}$	$\frac{5}{29}$	$\frac{2}{11}$	$\frac{0}{11}$
= 100%	4	4	1	3	0	4	0	0	1	1
≥ 75%	10	7	8	7	3	7	4	1	3	2
≥ 50%	13	10	13	9	5	7	6	3	4	3
≥ 25%	14	13	16	13	7	7	7	7	4	5
> 0%	19	15	24	21	22	9	17	17	8	7

2022), as well as with Github Copilot (SOBANIA et al., 2024). In some of those works, only a subset of the problems was chosen, often avoiding the most difficult ones (*i.e.*, not previously solved by any other method), or problems not solvable by the proposed method itself. The results are reported in Table 5.5 (“–” indicates the authors did not test their method on that specific problem).

Among the other GP algorithms, the best performer (DSLS) achieves a higher success rate than Origami in only 10 problems. Considering that they both get a 0 in pig-latin, Origami outperforms DSLS in 14 problems. Notably, Origami frequently outperforms CBGP and the G3P variants. It also has the highest number of problems solved with 100%, $\geq 75\%$, and $\geq 50\%$, and is second-place in $\geq 25\%$. When we consider problems to

which Origami found at least one solution, we note that it outperforms HOTGP, CBGP, and all the G3P variations, placing Origami at the fourth place. It is also worth noting that Origami outperforms HOTGP in both the number of best results and amount of problems above all thresholds, which demonstrates it is a substantial improvement over HOTGP.

We also compare with the results obtained using Copilot on PSB1, as reported by SOBANIA et al. (2024). In that paper, the authors tested Copilot with a different formulation of the program synthesis problem: instead of receiving example of input-outputs, they made the problem description and the function signature available to Copilot. This input format can be more difficult to process, as it requires the extraction of useful information from a textual description, but contains additional information that may be implicit in the input-output format. As shown in Table 5.6, out of the 29 problems, Copilot had better results in 14 of them, and Origami is equivalent or outperforms Copilot in 15 problems. While Copilot solves more problems (at least once) than Origami, it struggles with consistency, and does not achieve 100% success rate on any of the tested problems.

5.3 Final Remarks

This chapter presents the first full implementation of Origami, a GP algorithm proposed in Chapter 4, and builds on its previous work, HOTGP. Origami’s main differential is the use of RSs, well-known constructs in functional programming that enable recursive algorithms to be defined in a unified manner. The main motivation for using these in the PS context is enabling recursive programs to be synthesized in a controlled manner, without sacrificing expressiveness.

We evaluate our approach in the 29 problems in the PSB1 dataset, which is known to be solvable by just a handful of RSs. In general, Origami performs better than other similar methods, synthesizing the correct solution more often than other methods in most problems. It was also able to obtain the highest count of problems with success rate $= 100\%$, $\geq 75\%$ and $\geq 50\%$ among the GP methods. Furthermore, Origami achieved comparable results to Github Copilot, solving some problems that the LLM achieved 0% score. We should stress that the problem formulation is different for both approaches, indicating that combining LLM with GP and RSs could be beneficial to improve the results. These experimental results suggest that using Recursion Schemes to guide the search is a promising research avenue.

Currently, the main challenge of Origami appears to be dealing with harder RSs, such as Accumulation and Hylomorphism. Different evolutionary mechanisms, such as other selection methods and mutation/crossover operators, should be evaluated in this context to understand if they can positively impact the search process.

Table 5.6: Comparison between Origami and Copilot. Success rates with the best values underlined, with the ratio of victories of each algorithm against Origami by the amount of tested problems. The last rows display the amount of problems with a success rate above a certain % for each method.

Dataset	Origami	Copilot
checksum	0	<u>89</u>
collatz-numbers	0	<u>73</u>
compare-string-lengths	<u>90</u>	70
count-odds	40	<u>98</u>
double-letters	3	<u>88</u>
even-squares	3	<u>11</u>
for-loop-index	<u>90</u>	72
grade	<u>100</u>	84
last-index-of-zero	<u>70</u>	61
median	97	79
mirror-image	93	70
negative-to-zero	87	99
number-io	<u>100</u>	93
pig-latin	0	<u>54</u>
replace-space-w-nl.	3	87
scrabble-score	<u>100</u>	35
small-or-large	53	51
smallest	<u>100</u>	66
string-lengths-back.	<u>97</u>	60
sum-of-squares	0	<u>90</u>
super-anagrams	<u>73</u>	55
syllables	7	<u>96</u>
vector-average	0	<u>92</u>
vectors-summed	20	<u>87</u>
x-word-lines	0	<u>1</u>
Win ratio vs. Origami		$\frac{14}{29}$
= 100%	<u>4</u>	0
≥ 75%	10	<u>12</u>
≥ 50%	13	<u>22</u>
≥ 25%	14	<u>23</u>
> 0%	19	<u>26</u>

After introducing the concept of Origami in Chapter 4, Chapter 5 presented its first full implementation (hereafter, `OrigamiBNNS`). This version was capable of synthesizing recursive programs across multiple recursion schemes and consistently outperformed other GP methods.

Nevertheless, one limitation observed in `OrigamiBNNS` is its lack of population diversity, as it frequently spends considerable time exploring solutions that specialize in a similar subset of the training examples. Even with syntactic differences, they had a similar prediction error throughout all the examples.

In this chapter, we introduce a new diversity mechanism called AC/DC that aims to remove part of the individuals specialized in the same test cases while replacing them with new randomly generated programs, allowing a more efficient exploration of the search space (see Section 6.1). Beyond that, we also implement and evaluate:

- A new RS (histomorphism), as well as variations of catamorphism for other data-structures (namely `Map` and `Set`);
- A new selection strategy: Down-Sampled Lexicase Selection (DSLS) (HELMUTH; SPECTOR, 2022);
- A less restrictive function set, so that Origami can synthesize more expressive programs;
- A more comprehensive experimental evaluation by evaluating Origami in two new benchmark suites, namely PSB2 (HELMUTH; KELLY, 2021) and PolyPSB (PANTRIDGE; HELMUTH, 2023);

The remainder of this chapter is organized as follows. In Section 6.1, we introduce and describe AC/DC, which is the main contribution of this chapter. In Section 6.2, we present other proposed extensions to the Origami algorithm. In Section 6.3, we empirically evaluate this version of Origami and analyze the results.

We highlight that the contents of this chapter constitute a paper which is yet to be published. A considerable portion of the text presented here is an adaptation or extension of the text presented in the paper.

6.1 Abridge-by-Clipping, Diversify-by-Culling

The main contribution of this chapter is a proposal of a two-part diversification procedure called Abridge-by-Clipping, Diversify-by-Culling (AC/DC).

Diversify-by-Culling (DC) groups individuals sharing identical error vectors, and removes all of them except the one with the fewest nodes (*i.e.*, *culling*, since smaller programs potentially have a higher generalization capability (HELMUTH et al., 2017)). Using the error vector removes individuals that specialize in the same examples, even if their outputs differ in the incorrect predictions. This procedure is executed every g_{dc} generations.

As DC consistently reduces the population count, new individuals are randomly generated, using the same procedure employed during the initialization, to replace the removed ones. This aims to periodically introduce new genetic material, mitigating premature convergence and leading to a more diverse population and more comprehensive exploration of the search space.

A complementary step involves applying an Abridge-by-Clipping (AC) procedure to every individual in the population every g_{ac} generations. Previously, Origami_{BNNs} and HOTGP applied simplification (*i.e.*, *abridging*) to the best individual it found at the end of the search, only as a post-processing step (see Section 5.1). This procedure first performs a behavior-keeping change by evaluating branches which do not rely on any input, replacing them with constant leaves (*i.e.*, *clipping*). Then, it applies the local search procedure previously presented in Section 3.1.2 (Algorithm 5), which tries to replace each node by each of its children, reevaluating the program and keeping changes that do not reduce the fitness value.

Extending this clipping procedure to the entire population improves the fitness of certain individuals, as some parts of the program may actually increase the error (see Section 3.1.2). Furthermore, as it eliminates segments of the program that do not contribute to the overall fitness, making it part of the evolution creates space for mutation and crossover operators to explore more variations without exceeding the maximum size constraints.

As a result of the application of AC, the chance that some individuals share the same error vector increases. This, in turn, creates an opportunity for DC which will then eliminate redundant solutions and introduce novel genetic material into the population. Therefore, AC should be executed at least before every DC. In other words, it is desirable that $g_{dc} = k \cdot g_{ac}$ for some $k \in \mathbb{N}^+$. An empirical analysis for the choice of the g_{ac} and g_{dc} values is shown in Section 6.3.1.

6.2 Extensions to Origami

This section proposes enhancements to the Origami algorithm: three new patterns (including a new RS), a new selection strategy, and an extended function set.

6.2.1 New Patterns

For the current work, we manually solved two additional benchmarks:

- **PSB2** (HELMUTH; KELLY, 2021): The General Program Synthesis Benchmark 2 (PSB2) is the successor to PSB1, introducing more challenging problems;
- **PolyPSB** (PANTRIDGE; HELMUTH, 2023): A benchmark that was introduced to highlight the capabilities of CBGP in handling polymorphic data and functions, which is also supported by Origami. Since this benchmark lacks an official name, we unofficially refer to it as the Polymorphic Program Synthesis Benchmark (PolyPSB).

The canonical solutions for these two new benchmarks are also available on GitHub¹. They required one previously unused RS (histomorphism), as well as catamorphism variations for 2 different data structures: **Set** and **Map**, which can be encoded into 3 additional patterns. These additions do not incur any changes to the algorithm, as Origami is capable of supporting any RS in general.

Catamorphism over a Set

This pattern is very similar to the regular *Cata*, but traversing a **Set** instead of an indexed list:

```
f arg0 ... argn =
  cata alg (fromSet arg0) where
    alg SNilF = slot1
    alg (SConsF x acc) = slot2
```

In a problem with arguments of type $i_0 \dots i_n$ and of output type o , where $i_0 \equiv \mathbf{Set} \ e^2$, the slots are typed as follows:

- slot₁ :: o , with nothing in scope;
- slot₂ :: o , with scope $\{ x :: e ; acc :: o ; arg_0 :: i_0 \dots arg_n :: i_n \}$.

This pattern will be referred simply as *Cata Set* in the remainder of this work.

Catamorphism over a Map

Similarly, this pattern represents folding over a **Map**, while allowing simultaneous access for the current key and value:

¹<https://github.com/folivetti/origami-programming>

²The notation $i_0 \equiv \mathbf{Set} \ e$ is a restriction such that i_0 can be decomposed into the type $\mathbf{Set} \ e$, which is the type of a set with elements of some type e .

```

f arg0 ... argn =
  cata alg (fromMap arg0) where
    alg MNilF = slot1
    alg (MConsF k v acc) = slot2

```

In a problem with arguments of type $i_0 \dots i_n$ and of output type o , where $i_0 \equiv \text{Map } a \ b$, the slots are typed as follows:

- slot₁ :: o , with nothing in scope;
- slot₂ :: o , with scope $\{ k :: a ; v :: b ; acc :: o ; arg_0 :: i_0 \dots arg_n :: i_n \}$.

This pattern will be referred simply as *Cata Map* in the remainder of this work.

Histomorphism over Indexed List

This pattern is a generalization of catamorphism, giving it access to the entire history of the recursion so far. It is specifically useful in problems that have the need to compare the current element with the previous elements in some way. It is represented by the following:

```

f arg0 ... argn = histo alg (fromList arg0)
  where
    alg INilF = slot1
    alg (IConsF i x table) = slot2
  where
    acc = extract table
    tableAsList = tbl2List table

```

When compared to the regular *Cata*, the only difference is that instead of being able to access just the last value returned by `alg`, it has access to `table`, containing the entire history of the values returned by it.

This table is traditionally encoded in a representation that allows for any data structure to record its history. Since this pattern only deals with indexed lists, for convenience, we introduced two bindings: `tableAsList` has the previously computed elements as a regular Haskell linked-list; and `acc` contains the latest value returned by `alg`.

In a problem with arguments of type $i_0 \dots i_n$ and of output type o , where $i_0 \equiv [e]$, the slots are typed as follows:

- slot₁ :: o , with nothing in scope;
- slot₂ :: o , with scope $\{ i :: \text{Int} ; x :: e ; acc :: o ; tableAsList :: [e] ; arg_0 :: i_0 \dots arg_n :: i_n \}$.

This pattern will be referred simply as *Histo* in the remainder of this work.

6.2.2 Selection mechanism

We also consider the Down-Sampled Lexicase Selection (DSLS) (HELMUTH; SPECTOR, 2022) as a selection strategy, instead of the tournament selection that was previously used in `OrigamiBNNs`. This change was motivated by its positive impact on the PushGP algorithm, which could potentially be translated to `Origami`.

Lexicase Selection aims to maintain a diverse population, in terms of the subset of examples that each individual solves. The goal is to combine genetic material from programs that successfully solve different subsets of the examples, hoping that the resulting offspring solves both subsets simultaneously.

The standard Lexicase Selection algorithm (SPECTOR, 2012) iterates through the examples in shuffled order, at each step selecting only the individuals that have the lowest error for the current example. This process stops when only one individual remains, or when the examples are exhausted (in which case a random individual is selected from the remaining ones).

In contrast, DSLS (HELMUTH; SPECTOR, 2022) is a variation that shuffles a smaller random subset of examples, instead of the entire dataset. This approach makes selection not only faster, as fewer programs need to be evaluated, but empirically leads to better results. In experimental evaluations on PushGP, DSLS was reported as a substantial improvement over standard Lexicase Selection, being robust to a wide range of subsample sizes. Evidence suggests that its success is due to allowing more individuals to be evaluated, as well as submitting them through a “changing environment” (HELMUTH; SPECTOR, 2022), in which the set of examples is constantly changing.

In the context of this work, DSLS complements the selection pressure from AC/DC, which promotes a diverse population that fails in different cases and increases the chance of producing solutions that cover complementary errors.

6.2.3 Function set

The previous chapters involving `Origami` aimed to assess whether representing recursion using RSs was beneficial for PS. For that intent, `OrigamiBNNs` needed to synthesize solutions using RSs as the *only* means of recursion. Thus, its function set was purposefully restricted, not allowing most implicitly recursive functions (such as `sum` and `product`). If `OrigamiBNNs` needed to perform those operations, it would need to synthesize them purely through the RSs, which it was able to do with varying degrees of success.

In the current chapter, we aim to test `Origami` as a feature-complete synthesizer, which means that it should have access to the set of functions that are commonly provided by the Haskell standard library. We highlight, however, that a function set can be undesirably large, making the search space larger than it needs to be. Furthermore, certain functions can trivialize some problems, leading to an unfair comparison to other methods. To

achieve a balance, we provide a function set that is intended to be equivalent to the ones provided by other synthesizers, such as PushGP (HELMUTH; SPECTOR, 2015) and CBGP (PANTRIDGE; HELMUTH; SPECTOR, 2022), using the most similar functions available in the Haskell standard library. The function set is provided in detail in Table 6.1.

6.3 Experimental Results

This chapter also extends previous chapters by performing a more comprehensive experimental evaluation of Origami. While the previous evaluation (Section 5.2) already tackled PSB1 (HELMUTH; SPECTOR, 2015) (29 problems), we now incorporate both PSB2 (HELMUTH; KELLY, 2021) (25 problems) and PolyPSB (PANTRIDGE; HELMUTH, 2023) (17 problems), with a total of 71 problems. Each problem prescribes which type of operations should be available, as well as providing ways to obtain or generate input/output examples. They also define a small number to be used as the count of examples that are visible to the synthesizer (train cases), as well as a larger number indicating how many examples are hidden, reserved to evaluate the generalization capabilities of the synthesized program (test cases).

Our evaluation is centered on the assessment of the new version of Origami, denoted Origami_{AC/DC}. This variant integrates the entire set of improvements presented in this chapter and will be used by most of the experiments. However, to allow verifying the contribution of the AC/DC procedures alone (see Section 6.3.3), we also introduce a separate version of the algorithm, Origami_{DSLS}, which contains just the extensions presented on Section 6.2, and does not contain AC/DC.

The main experimental evaluation follows the same strategy used to evaluate Origami_{BNNs} in Section 5.2. For each problem, we perform 30 independent runs (initialized with different seeds) for each pattern in the following order: 1. NoScheme; 2. Cata List/Set/Map; 3. CurriedCata; 4. Histo; 5. Ana; 6. Accu; 7. Hylo.

The execution only advances to the next pattern when none of the 30 seeds was able to find a solution that solves the test dataset. This simulates a realistic scenario, where there would be no previous information regarding which scheme is capable or better suited to solve the problem, so all of them need to be tested. Also, a pattern is only attempted if the input/output type of the problem matches the one expected by the pattern. For example, CurriedCata will only be executed for problems with two arguments where the first one is a list. Note that it is trivial to decide which Cata variation to use, as it is uniquely determined by the type of the first argument. This specific order was chosen so simpler patterns are given preference, while also avoiding the decision of the unbound types in Accu and Hylo when another pattern solves the problem.

Table 6.1: The function set available for Origami_{DSL}.

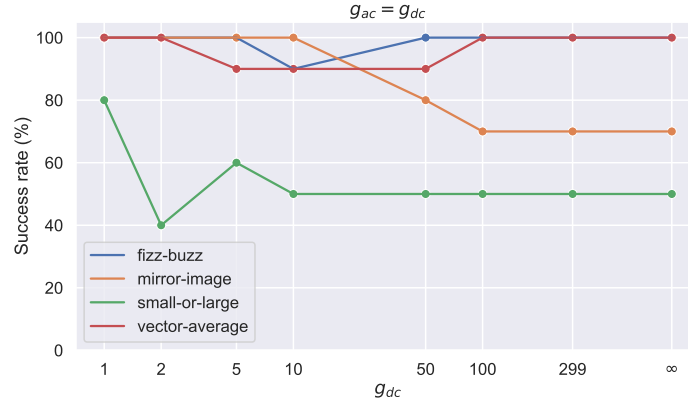
Operations	Types
addInt, subInt, multInt, divInt, quotInt, modInt, remInt, minInt, maxInt	Int -> Int -> Int
absInt, incrementInt, decrementInt	Int -> Int
sumInt, productInt	[Int] -> Int
addFloat, subFloat, multFloat, divFloat, minFloat, maxFloat, powFloat	Float -> Float -> Float
absFloat, sqrt, sin, cos, incrementFloat, decrementFloat	Float -> Float
sumFloat, productFloat	[Float] -> Float
ltInt, gtInt, gteInt, lteInt	Int -> Int -> Bool
ltFloat, gtFloat, gteFloat, lteFloat	Float -> Float -> Bool
fromIntegral	Int -> Float
floor, ceiling, round	Float -> Int
and, or	Bool -> Bool -> Bool
not	Bool -> Bool
if	Bool -> a -> a -> a
eq, neq	a -> a -> Bool
showInt	Int -> [Char]
showFloat	Float -> [Char]
showBool	Bool -> [Char]
showChar	Char -> [Char]
charToInt, digitToInt	Char -> Int
intToChar	Int -> Char
isLetter, isSpace, isDigit	Char -> Bool
toLower, toUpper	Char -> Char
length	[a] -> Int
cons, snoc, mappend, delete	a -> [a] -> [a]
elem	a -> [a] -> Bool
null	[a] -> Bool
head, last	[a] -> a
tail, init	[a] -> [a]
zip	[a] -> [b] -> [(a,b)]
replicate	Int -> a -> [a]
enumFromThenTo	Int -> Int -> Int -> [Int]
take, drop	Int -> [a] -> [a]
takeWhile	(a -> Bool) -> [a] -> [a]
any, all	(a -> Bool) -> [a] -> Bool
reverse	[a] -> [a]
splitAt	Int -> [a] -> ([a], [a])
intercalate	[a] -> [[a]] -> [a]
fst	(a, b) -> a
snd	(a, b) -> b
pair	a -> b -> (a, b)
apply	(a -> b) -> a -> b
singleton	k -> v -> Map k v
insert, insertWith	k -> v -> Map k v -> Map k v
lookup	k -> Map k v -> Maybe v
fromList	[(k, v)] -> Map k v
keys	Map k v -> [k]
fromList	[a] -> Set a
toList	Set a -> [a]
insert, delete	a -> Set a -> Set a
member	a -> Set a -> Bool
union, intersection, difference	Set a -> Set a -> Set a
gtOrd, ltOrd, gteOrd, lteOrd	Ord a => a -> a -> Bool

6.3.1 AC/DC Hyperparameters

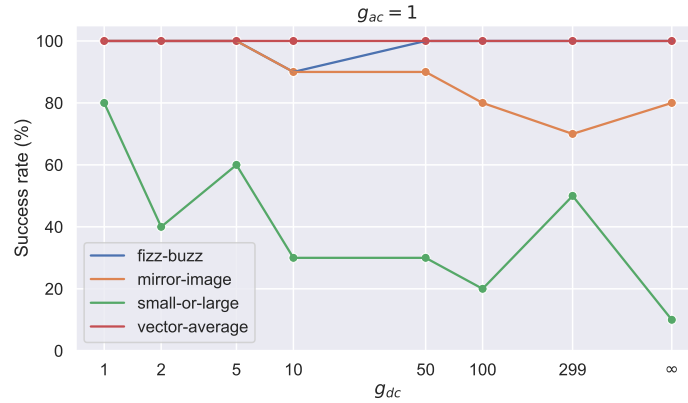
Before carrying out the main evaluation, we empirically determined the values for g_{ac} and g_{dc} , which are the execution frequencies of the AC and DC procedures, respectively (see Section 6.1). We executed 10 independent runs of Origami_{AC/DC} (with different seeds) on a subset of 4 problems: *fizz-buzz* (PSB2), *mirror-image*, *small-or-large*, and *vector-average* (PSB1).

In the first experiment, we tested the values $\{1, 2, 5, 10, 50, 100, 299, \infty\}$ with $g_{ac} = g_{dc}$. Figure 6.1a shows the *success rates* of each evaluation, which measures the percentage of executions (in this case, 10) for which Origami_{AC/DC} synthesized a solution that successfully handles all the test cases.

The overall peak at $g_{dc} = 1$ indicates that the best performance is achieved when both were set to 1, meaning that AC and DC were applied every generation. Next, we fixed $g_{ac} = 1$ and tested different values for g_{dc} (Figure 6.1b). The results confirmed that performing AC and DC at every generation ($g_{ac} = g_{dc} = 1$) is the configuration that empirically seems to obtain the best performance.



(a) AC and DC executed at equal intervals.



(b) AC executed every generation, DC executed at varied intervals.

Figure 6.1: Success rates under different AC and DC configurations. A value of 1 indicates execution every generation, 2 every other generation, 299 only once just before the final (300th) generation, and ∞ indicates the procedure is never executed.

6.3.2 Results by Scheme

We executed Origami_{AC/DC} following the experimental framework described earlier. Table 6.2, Table 6.3, Table 6.4 show the success rates for each pattern in the tested problems. Underlined values show the highest rate for a given problem, and blue values indicate its canonical pattern — that is, the pattern that was used in the manual implementation presented as the canonical solution.

The bottom rows summarize, for each pattern, the number of problems whose canonical solution use this pattern (as “Total canonical”) and how many of these problems were solved at least once using that pattern (as “Solved canonical”). They also report the number of problems solved using a given pattern, but whose canonical solution corresponded to a different pattern (as “Solved non-canonical”).

Overall, Origami_{AC/DC} demonstrates that it is able to synthesize solutions using all the provided patterns.

Table 6.2: Success Rates of each pattern on PSB1 for Origami_{AC/DC}.

Dataset	NoScheme	Cata List	CurriedCata	Histo	Ana	Accu	Hylo	Best
checksum	0	0	0	0	–	<u>70</u>	–	70
collatz-numbers	0	–	–	–	–	–	<u>0</u>	0
compare-string-lengths	<u>97</u>	–	–	–	–	–	–	97
count-odds	0	<u>100</u>	–	–	–	–	–	100
digits	0	–	–	–	<u>0</u>	–	0	0
double-letters	0	<u>63</u>	–	–	–	–	–	63
even-squares	0	–	–	–	<u>38</u>	–	–	38
for-loop-index	77	–	–	–	<u>97</u>	–	–	97
grade	0	<u>100</u>	<u>97</u>	–	–	–	–	100
last-index-of-zero	0	<u>93</u>	–	–	–	–	–	93
median	<u>100</u>	–	–	–	–	–	–	100
mirror-image	<u>100</u>	–	–	–	–	–	–	100
negative-to-zero	0	<u>100</u>	–	–	–	–	–	100
number-io	<u>100</u>	–	–	–	–	–	–	100
pig-latin	0	<u>0</u>	–	0	0	0	0	0
replace-space-with-newline	0	<u>20</u>	–	–	–	–	–	20
scrabble-score	0	<u>100</u>	–	–	–	–	–	100
small-or-large	<u>67</u>	–	–	–	–	–	–	67
smallest	<u>100</u>	–	–	–	–	–	–	100
string-differences	0	0	<u>0</u>	0	0	0	0	0
string-lengths-backwards	0	<u>100</u>	–	–	–	–	–	100
sum-of-squares	0	–	–	–	–	–	<u>30</u>	30
super-anagrams	0	0	<u>63</u>	–	–	–	–	63
syllables	0	<u>80</u>	–	–	–	–	–	80
vector-average	<u>100</u>	–	–	–	–	<u>87</u>	–	100
vectors-summed	0	0	<u>87</u>	–	–	–	–	87
wallis-pi	0	–	–	–	–	–	<u>0</u>	0
word-stats	0	0	–	0	–	<u>0</u>	0	0
x-word-lines	0	0	–	0	0	<u>0</u>	0	0
Total canonical	6	9	4	0	3	4	3	
Solved canonical	6	8	3	0	2	2	1	
Solved non-canonical	2	1	0	0	0	0	0	

Table 6.3: Success Rates of each pattern on PSB2 for Origami_{AC/DC}.

Dataset	NoScheme	Cata List	CurriedCata	Histo	Ana	Accu	Hylo	Best
basement	0	0	–	0	–	40	–	40
bouncing-balls	0	–	–	–	–	–	0	0
bowling	0	0	–	0	–	0	0	0
camel-case	0	0	–	0	–	0	0	0
coin-sums	0	0	87	–	–	–	–	87
cut-vector	0	0	–	0	–	0	0	0
dice-game	20	–	–	–	–	–	–	20
find-pair	0	0	–	20	–	–	–	20
fizz-buzz	87	–	–	–	–	–	–	87
fuel-cost	0	100	–	–	–	–	–	100
gcd	0	–	–	–	–	–	23	23
indices-of-substring	0	0	0	0	0	0	0	0
leaders	0	40	–	37	–	–	–	40
luhn	0	0	–	0	–	0	0	0
mastermind	0	0	0	0	–	0	0	0
middle-character	33	–	–	–	–	–	–	33
paired-digits	0	0	–	60	–	–	–	60
shopping-list	0	0	0	0	–	0	0	0
snow-day	0	–	–	–	–	–	10	0
solve-boolean	0	0	–	0	–	0	0	0
spin-words	0	0	–	0	0	0	0	0
square-digits	0	–	–	–	0	–	0	0
substitution-cipher	0	0	–	0	0	0	0	0
twitter	0	0	–	0	0	0	0	0
vector-distance	0	0	0	0	–	0	0	0
Total canonical	3	4	3	4	0	3	4	
Solved canonical	3	1	1	3	0	1	2	
Solved non-canonical	0	1	0	0	0	0	0	

Table 6.4: Success Rates of each pattern on PolyPSB for Origami_{AC/DC}.

Dataset	NoScheme	Cata List	Cata Set	Cata Map	Hylo	Best
area-of-rectangle	100	–	–	–	–	100
centimeters-to-meters	97	–	–	–	–	97
count-true	0	100	–	–	–	100
filter-bounds	0	–	77	–	–	77
first-index-of-true	97	100	–	–	–	100
get-vals-of-key	0	100	–	–	–	100
max-applied-fn	0	–	–	–	7	7
min-key	–	–	–	97	–	97
set-cartesian-product	0	–	0	–	0	0
set-symmetric-difference	100	–	–	–	–	100
sets-with-element	0	–	100	–	–	100
simple-encryption	0	100	–	–	–	100
sum-2-vals	100	–	–	–	–	100
sum-2-vals-polymorphic	100	–	–	–	–	100
sum-2d	97	–	–	–	–	97
sum-vector-vals	0	100	–	–	–	100
time-sheet	0	73	–	–	–	73
Total canonical	6	6	2	1	1	
Solved canonical	6	6	2	1	1	
Solved non-canonical	1	0	0	0	0	

Specifically, it found a solution at least once in all of the 15 problems that are canonically solved by NoScheme. Moreover, in 3 problems (*vector-average*, *for-loop-index*, *first-index-of-true*), a NoScheme solution was found even though it was not the canonical pattern. This is not completely unexpected, as our goal while writing the canonical solutions was to employ RSs as much as possible. However, with the extended function set available, sometimes the implicit recursion of other functions (such as `sum` and `takeWhile`) is enough for the problem to be solved in NoScheme. Nevertheless, for a more complete assessment of Origami_{AC/DC}, it was still evaluated on those problems using their canonical patterns, being able to solve all of them at least once.

For the next pattern in our sequence, *Cata*, $\frac{18}{22}$ (82%) of the canonical problems were solved, even for the new data structures (`Map` and `Set`) in PolyPSB. Like previously described, *grade* and *leaders* were solved using *Cata*, as well as with their canonical *CurriedCata* and *Histo* patterns, respectively. For the remaining patterns, Origami_{AC/DC} solves $\frac{3}{4}$ (75%) of *Histo*, $\frac{2}{3}$ (67%) of *Ana*, $\frac{4}{7}$ (57%) of *CurriedCata*, $\frac{3}{7}$ (43%) of *Accu*, and $\frac{3}{8}$ (38%) of the *Hylo* problems.

There is an apparent correlation between the chosen pattern and its success rate. In general, some patterns solve fewer problems than others, and also present lower success rates, *i.e.*, for the subset of problems they are able to solve, they do so with low consistency. This trend is supported by the previous assessment of Origami_{BNNS}, and can likely be attributed to a combination of two main factors:

- **Inherent problem complexity:** Patterns such as *Accu*, *Ana*, and *Hylo* tend to solve more difficult problems, while NoScheme addresses simpler problems. Evidence for this is that other methods in the literature present a similar behavior for those problems (see Section 6.3.4);
- **Pattern complexity:** Some patterns inherently make solutions harder to evolve. For example, *CurriedCata* solutions require an accumulated function, making programs more sensitive to small changes. *Histo* involves more bindings, while *Accu* and *Ana* have four slots and *Hylo* has six, which significantly expands the search space. In addition, unlike *Cata*, *Accu*, and *Histo*, which have guaranteed termination, *Ana* and *Hylo* can produce infinite recursions, increasing the difficulty of finding correct solutions.

6.3.3 Ablation Study

In this section, we perform an Ablation Study (SHEIKHOESLAMI, 2019): a controlled experiment in which components of a system are systematically removed to evaluate their individual impact on overall performance. By comparing the 3 different versions of Origami—Origami_{BNNS}, Origami_{DSLs}, and Origami_{AC/DC}—this study allows us to assess the effect of AC/DC independently of the other enhancements introduced in this chapter.

Table 6.5: Success rates of different versions of Origami in PSB1.

Dataset	Origami _{AC/DC}	Origami _{DSL_S}	Origami _{BNN_S}
checksum	70	23	0
compare-string-lengths	97	100	90
count-odds	100	100	40
double-letters	63	60	3
even-squares	38	13	3
for-loop-index	97	90	90
grade	100	100	100
last-index-of-zero	93	93	70
median	100	100	97
mirror-image	100	87	93
negative-to-zero	100	100	87
number-io	100	100	100
replace-space-with-newline	20	33	3
scrabble-score	100	100	100
small-or-large	67	40	53
smallest	100	100	100
string-lengths-backwards	100	100	97
sum-of-squares	30	13	0
super-anagrams	63	90	73
syllables	80	73	7
vector-average	100	100	0
vectors-summed	87	97	20

The best results obtained for each method in PSB1 are presented in Table 6.5. For most of the problems, both Origami_{AC/DC} and Origami_{DSL_S} represent a considerable improvement over Origami_{BNN_S}, showing the positive overall impact of the new function set and selection method. Among these, we highlight the improvements on *checksum*, *count-odds*, *double-letters*, *even-squares*, *replace-space-with-newline*, *small-or-large*, *syllables*, *sum-of-squares*, *vector-average* and *vectors-summed*. Specifically, the major difference in performance in *vector-average* stems from the introduction of the **sum** operator, making it solvable using NoScheme instead of the canonical Accu. The only problem with a lower success rate is *super-anagrams*, where Origami_{AC/DC} got a marginally lower success rate than Origami_{BNN_S}, even though Origami_{DSL_S} presented a considerable improvement.

Comparing the results from Origami_{AC/DC} to Origami_{DSL_S} illustrates the effects of applying AC/DC to the evolution. In this benchmark, Origami_{AC/DC} improves $\frac{8}{29}$ (28%) problems, with a mean³ improvement of $1.62\times$. The most pronounced improvement is *checksum*, whose success rate increased from 23% to 70% (a delta of 47%, corresponding to a $3\times$ improvement). Conversely, Origami_{DSL_S} achieves higher success rates in $\frac{4}{29}$ (14%) problems, specifically *compare-string-lengths*, *replace-space-with-newline*, *super-anagrams*, and *vectors-summed*, with Origami_{AC/DC} representing a mean decrease of $0.77\times$. As a whole, Origami_{AC/DC} yields more improvements than regressions on this dataset, indicating that the addition of AC/DC generally benefits the evolutionary process.

³All ratio means we report are computed as geometric means.

6.3.4 Comparison to other GP methods

To evaluate the performance of $\text{Origami}_{\text{AC/DC}}$, we compare its results to those of established GP methods from the literature, including:

- **PushGP** variants:
 - **PushGP₀**, the baseline algorithm for program synthesis, which evolves programs in the stack-based Push language (HELMUTH; KELLY, 2021; HELMUTH; SPECTOR, 2015);
 - **PushGP_{UMAD}**, which replaces its standard mutation with UMAD (HELMUTH; MCPHEE; SPECTOR, 2018b);
 - **PushGP_{DSLs}**, which employs DSLs (HELMUTH; SPECTOR, 2022) (see Section 6.2.2);
- **CBGP**, Code-Building Genetic Programming, a typed functional GP system that supports polymorphism and HOFs (PANTRIDGE; HELMUTH, 2023; PANTRIDGE; HELMUTH; SPECTOR, 2022; PANTRIDGE; SPECTOR, 2020).

Table 6.6, Table 6.7, Table 6.8 show the results when comparing to other state-of-the-art methods.

Across all 3 benchmarks, $\text{Origami}_{\text{AC/DC}}$ has the highest $= 100\%$, $\geq 75\%$, and $\geq 25\%$ rates when compared to the other methods. It also has the highest $\geq 50\%$ rates on PSB1 and PolyPSB, and is second to $\text{PushGP}_{\text{DSLs}}$ on PSB2. For the $> 0\%$ category, it ties with CBGP on PolyPSB, and is second to $\text{PushGP}_{\text{DSLs}}$ on both PSB1 and PSB2.

Overall, in $\frac{13}{71}$ (18%) of the problems across all benchmarks, $\text{Origami}_{\text{AC/DC}}$ is the only method to obtain 100% success rate: *count-odds*, *grade*, *scrabble-score*, *string-lengths-backwards*, *vector-average*, *fuel-cost*, *area-of-rectangle*, *get-vals-of-key*, *set-symmetric-difference*, *sets-with-element*, *simple-encryption*, *sum-2-vals*, and *sum-vector-vals*. Moreover, $\text{Origami}_{\text{AC/DC}}$ is the only method to achieve 100% success rate on a PSB2 problem, in *fuel-cost*.

In PSB1, $\text{Origami}_{\text{AC/DC}}$ achieves the highest success rate (including ties) in $\frac{25}{29}$ (86%) problems, with lower rates only in *digits*, *replace-space-with-newline*, *vectors-summed*, and *x-word-lines*. In PSB2, it achieves the highest success rate (including ties) in $\frac{10}{25}$ (40%) problems, with $\text{PushGP}_{\text{DSLs}}$ obtaining higher rates in 8 problems. The remaining 7 problems have not been solved by any method to date. In PolyPSB, it achieves the highest success rate (including ties) in $\frac{15}{17}$ (88%) problems, as CBGP has the highest rate in *sum-2d* and *max-applied-fn*.

Across the three benchmarks, $\text{Origami}_{\text{AC/DC}}$ strictly outperformed all other methods in $\frac{37}{71}$ (52%) problems, with an average improvement of $3.35\times$ over the second-best method (excluding the *leaders* problem, in which $\text{Origami}_{\text{AC/DC}}$ is the only method to achieve

Table 6.6: Success rates of GP Methods in PSB1.

	Origami _{AC/DC}	PushGP _{DSLs}	PushGP _{UMAD}	PushGP ₀	CBGP
checksum	70	18	5	0	–
compare-str.-len.	97	51	42	7	22
count-odds	100	11	12	8	0
digits	0	28	11	7	0
double-letters	63	50	20	6	–
even-squares	38	2	0	2	–
for-loop-index	97	5	1	1	0
grade	100	2	0	4	–
last-index-of-zero	93	65	56	21	10
median	100	69	48	45	98
mirror-image	100	99	100	78	100
negative-to-zero	100	82	82	45	99
number-io	100	99	100	98	100
replace-sp.-with-nl.	20	100	87	51	0
scrabble-score	100	31	20	2	–
small-or-large	67	22	4	5	0
smallest	100	98	100	81	100
string-len.-back.	100	95	86	66	–
sum-of-squares	30	25	26	6	–
super-anagrams	63	4	0	0	–
syllables	80	64	48	18	–
vector-average	100	97	92	16	88
vectors-summed	87	21	9	1	100
x-word-lines	0	91	59	8	–
= 100%	10	1	3	0	4
≥ 75%	15	8	7	3	7
≥ 50%	20	13	9	5	7
≥ 25%	21	16	13	7	7
> 0%	22	24	21	22	9

Table 6.7: Success rates of GP Methods in PSB2.

	Origami _{AC/DC}	PushGP _{DSL}	PushGP ₀
basement	<u>40</u>	2	1
bouncing-balls	0	<u>3</u>	0
camel-case	0	<u>4</u>	1
coin-sums	<u>87</u>	39	2
dice-game	<u>20</u>	1	0
find-pair	<u>20</u>	<u>20</u>	4
fizz-buzz	<u>87</u>	74	25
fuel-cost	<u>100</u>	67	50
gcd	<u>23</u>	20	8
indices-of-substring	0	<u>4</u>	0
leaders	<u>40</u>	0	0
middle-character	33	<u>79</u>	57
paired-digits	<u>60</u>	17	8
snow-day	<u>10</u>	7	4
solve-boolean	0	<u>5</u>	<u>5</u>
square-digits	0	2	0
substitution-cipher	0	<u>86</u>	60
twitter	0	<u>52</u>	31
= 100%	<u>1</u>	0	0
≥ 75%	<u>3</u>	2	0
≥ 50%	4	<u>5</u>	3
≥ 25%	<u>7</u>	6	5
> 0%	11	<u>17</u>	13

Table 6.8: Success rates of GP Methods in PolyPSB.

	Origami _{AC/DC}	CBGP
area-of-rectangle	<u>100</u>	59
centimeters-to-meters	<u>97</u>	92
count-true	<u>100</u>	<u>100</u>
filter-bounds	<u>77</u>	13
first-index-of-true	<u>100</u>	<u>100</u>
get-vals-of-key	<u>100</u>	12
max-applied-fn	7	<u>24</u>
min-key	<u>97</u>	31
set-symmetric-difference	<u>100</u>	50
sets-with-element	<u>100</u>	4
simple-encryption	<u>100</u>	96
sum-2-vals	<u>100</u>	94
sum-2-vals-polymorphic	<u>100</u>	<u>100</u>
sum-2d	97	<u>100</u>
sum-vector-vals	<u>100</u>	16
time-sheet	<u>73</u>	2
= 100%	<u>10</u>	4
≥ 75%	<u>14</u>	7
≥ 50%	<u>15</u>	9
≥ 25%	<u>15</u>	10
> 0%	<u>16</u>	<u>16</u>

a nonzero success rate). In $\frac{20}{71}$ (28%) problems, it tied with other approaches as the best performer, showing no improvement. This group includes the *find-pair* problem (where Origami_{AC/DC} tied with PushGP_{DSL} at a 20% success rate), the 13 problems that were not solved by any method, and 6 problems in which Origami_{AC/DC} and at least one other method both achieved 100% success rates. Origami_{AC/DC} was outperformed in $\frac{14}{71}$ (20%) problems. Among these, 9 problems were solved by other methods but not by Origami_{AC/DC}. In the 5 problems Origami_{AC/DC} solved but did not achieve the highest success rate, the top method obtained a $2.19\times$ higher success rate on average.

6.3.5 Comparison against LLMs

LLMs have been recently tested in many variations of the PS task. In one of such works (SOBANIA et al., 2024), GitHub Copilot is evaluated in solving both PSB1 and PSB2. However, comparing these results to Origami_{AC/DC} is not straightforward, as we need to consider the risk of *data contamination*, and the fact that Copilot has access to the problem textual description rather than input-output examples.

First, data contamination means that the benchmarks used for the evaluation, as well as their intended solutions, may have been included in the corpora used to train these LLMs. Thus, their performance might simply reflect the memorization of such solutions, rather than a genuine capability of solving novel tasks in a generalizable fashion. This concern is supported by the literature, with evidence that LLM performances tend to be higher on problems that were public before the training data cutoff date (LIU et al., 2024). As the information regarding the dataset used for training these models is not publicly available, there is no way to verify whether or not that is the case for GitHub Copilot and these benchmarks.

Secondly, the fundamental difference in the *type of information* each method receives makes them arguably operate on different tasks. Origami, as a traditional PBE algorithm, receives a set of structured input-output examples. In contrast, LLMs operate on natural language and therefore need a textual description of the problem. These two types of specification are not only distinct in format, but also in the actual task they describe. A textual description often contains implicit context and information that is not available in the input-output pairs, which LLMs can effectively leverage when synthesizing the program. On the other hand, input-output examples offer a low-level, more detailed description of the expected behavior, which is especially valuable in edge cases.

With those challenges in mind, a comparison between Origami_{AC/DC} and Copilot is shown in Table 6.9 and Table 6.10.

Table 6.9: Success rates of Origami_{AC/DC} and Copilot in PSB1.

	Origami _{AC/DC}	Copilot
checksum	70	89
collatz-numbers	0	73
compare-string-lengths	97	70
count-odds	100	98
double-letters	63	88
even-squares	38	11
for-loop-index	97	72
grade	100	84
last-index-of-zero	93	61
median	100	79
mirror-image	100	70
negative-to-zero	100	99
number-io	100	93
pig-latin	0	54
replace-space-with-newline	20	87
scrabble-score	100	35
small-or-large	67	51
smallest	100	66
string-differences	0	9
string-lengths-backwards	100	60
sum-of-squares	30	90
super-anagrams	63	55
syllables	80	96
vector-average	100	92
vectors-summed	87	87
x-word-lines	0	1
= 100%	11	0
≥ 75%	16	12
≥ 50%	20	22
≥ 25%	21	23
> 0%	22	26

Table 6.10: Success rates of Origami_{AC/DC} and Copilot in PSB2.

	Origami	Copilot
basement	40	95
camel-case	0	31
coin-sums	87	12
cut-vector	0	1
dice-game	20	0
find-pair	20	41
fizz-buzz	87	89
fuel-cost	100	97
gcd	23	80
indices-of-substring	0	82
leaders	40	67
luhn	0	6
middle-character	33	98
paired-digits	60	88
shopping-list	0	75
snow-day	10	10
spin-words	0	96
square-digits	0	55
substitution-cipher	0	78
twitter	0	89
vector-distance	0	79
= 100%	1	0
≥ 75%	3	12
≥ 50%	5	14
≥ 25%	7	16
> 0%	9	20

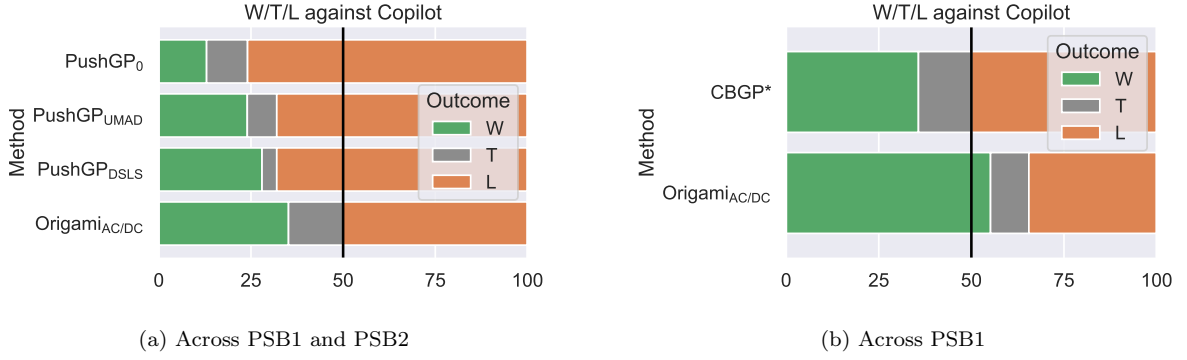


Figure 6.2: Percentage of problems won (W), tied (T), and lost (L) by each method compared to Copilot, across PSB1 and PSB2. Here, W indicates that a given method achieved a higher success rate than Copilot, T indicates equal success rates, and L indicates a lower success rate. CBGP* indicates partial results, as it was evaluated only on PSB1. A reference vertical line is drawn at 50%.

The results in PSB1 show that the performance of both algorithms is broadly comparable, with a similar number of problems solved in the $> 0\%$, $\geq 25\%$, $\geq 50\%$, and $\geq 75\%$ categories. The main distinction appears in the $= 100\%$ category, where Origami_{AC/DC} achieves perfect success rates in 11 problems, while Copilot does not reach 100% in any case. Similarly, in PSB2 benchmark, Copilot achieves higher counts in most categories ($> 0\%$, $\geq 25\%$, $\geq 50\%$, and $\geq 75\%$) while Origami_{AC/DC} leads only in the $= 100\%$.

Overall, a larger number of problems are solved at least once ($> 0\%$) by Copilot ($\frac{46}{54}$ (85%)) than Origami_{AC/DC} ($\frac{31}{54}$ (57%)), across both benchmarks. However, Copilot is not able to solve any problem with 100% success rate. This indicates that its higher problem coverage comes with less reliability, as it occasionally fails to produce a solution even for simple problems.

We also consider which method has a higher success rate on a problem-by-problem basis. Figure 6.2 presents a comparison of Origami_{AC/DC} and other GP methods against Copilot.

Considering both benchmarks, Origami_{AC/DC} wins against Copilot in $\frac{19}{54}$ (35%), ties in $\frac{8}{54}$ (15%), and loses in $\frac{27}{54}$ (50%). This represents the highest win rate among the methods evaluated on both PSB1 and PSB2.

As CBGP was only evaluated on PSB1, which was shown to be the less challenging of the two, Figure 6.2b shows an isolated comparison on just this benchmark. In this scenario, Origami_{AC/DC} achieves a W/T/L record of 16/3/10 (ties are omitted in the table when both methods scored 0). This corresponds to a 55% win rate over Copilot, which is higher than CBGP under the same conditions. In contrast, Origami_{AC/DC} achieves 3/17/5 against Copilot on PSB2, or a 12% win rate, indicating that Copilot performs strongly on this harder benchmark.

6.4 Final Remarks

This main contribution of this chapter is a novel procedure called Abridge-by-Clipping, Diversify-by-Culling (AC/DC). This procedure aims at a more efficient exploration of the search space, as it leads to a more diverse population and creates more opportunities for mutation and crossover. This chapter also continues the development of Origami, by applying the DSLS selection method, expanding the set of functions available, and introducing 3 new patterns that allow it to synthesize a new RS (histomorphism), as well as variations of catamorphism for Maps and Sets.

We evaluated this improved version of Origami (Origami_{AC/DC}) in 3 different benchmarks, two of which were not considered in previous work. Overall, Origami_{AC/DC} demonstrates that it is able to synthesize solutions using all the provided patterns. We observed a strong correlation between the canonical scheme used by each problem and whether Origami_{AC/DC} can solve them at least once, as well as its success rate. Most notably, Origami_{AC/DC} is able to solve all problems that canonically use NoScheme, and solves $\geq 50\%$ of the problems in all other RSs.

The changes brought by this chapter also achieved a considerable improvement over the previous version of Origami (Origami_{BNNs}), having a higher success rate in all of the PSB1 problems. AC/DC also led to meaningful improvements in several problems in PSB1, with a very reduced number of regressions.

When compared to the state of the art of GP, Origami_{AC/DC} achieved the highest number of problems solved with success rates of $= 100\%$, $\geq 75\%$, and $\geq 25\%$ on all benchmarks, and is either tied or in second place in $> 0\%$. In 18% of the problems across the entire benchmark, Origami_{AC/DC} is the only method to obtain 100% success rate, being the first known method to achieve this in a PSB2 problem.

Origami_{AC/DC} also presents a performance that is competitive to state-of-the-art LLMs, such as GitHub Copilot, especially in PSB1, with the highest overall win-rate over Copilot when compared to other GP methods.

For future work, we believe that a deeper investigation into some of the PSB2 problems might lead to insights into what might be preventing Origami from finding solutions more frequently, or from finding solutions at all. These could result in strategies to improve the overall success rate of Origami across all benchmarks and patterns.

Another current limitation is that Origami_{AC/DC} goes through the RSs in a linear fashion, exhausting all the allowed runs in a pattern before advancing to the next one. This could be done more effectively by, for example, applying heuristics that would bias the search towards the most likely pattern. Or, more interestingly, applying a co-evolutionary approach, with one population per pattern being evolved in parallel. Exchanging genetic material among the populations has the potential to make the search more effective overall, constituting a promising avenue of work.

Conclusion

This text aimed to contribute mainly to the fields of PS, GP and FP. Its main objective was applying FP concepts with strong types, including Recursion Schemes, to PS. This was motivated by the benefits this combination brings to the table, which could potentially lead to improvements in the PS process. Therefore, in Chapter 1, we defined the following as our problem statement.

Problem Statement: *PS poses challenges in terms of complexity and cost, primarily attributed to the vast search space involved. This study assesses whether the integration of functional programming principles, strong typing, and recursion schemes can mitigate these challenges, potentially leading to a more efficient synthesis, and identifies potential drawbacks associated with this approach.*

In order to properly contextualize this work, we presented a general theoretical background of the aforementioned fields. We conducted a literature review, describing related work and detailing well-known methods which have been largely influential, identifying potential improvement opportunities. Then, two novel PS methods were proposed.

HOTGP extends on the standard GP algorithm by employing λ -functions and higher-order functions. These abstractions are native to FP and enable the algorithm to synthesize short solutions to problems that would require increased complexity in an imperative setting. Furthermore, HOTGP has a simplification procedure which leads to shorter programs, which are both more readable and generalize better. We positioned this method with the current literature by evaluating it and comparing it to state-of-the-art methods. Overall, HOTGP obtained good results, being competitive with the literature and showing considerable improvements in certain problems. The synthesized solutions were comparable to the hand-crafted ones, in terms of size and complexity.

Our second proposed method is called Origami, and explores the knowledge of the already established and deeply researched Recursion Schemes field. We argue that, by having access to a handful of pre-established patterns, an algorithm would be able to navigate the search space of recursive programs more effectively, providing a better way of synthesizing functional programs.

We first conducted an analysis of the most widely used PS benchmark, PSB1 (HELMUTH; SPECTOR, 2015), by manually implementing solutions using Recursion Schemes and were able to identify recurrent patterns on the solutions. We conducted initial experiments by adapting HOTGP to evolve the non-constant part of the most common pattern,

catamorphism, in order to assess the validity of this algorithm. The preliminary results show that, when using the correct pattern, an algorithm can quickly synthesize a correct solution.

Following, we presented the first complete implementation of Origami, using a multi-gene approach that uses patterns and slots. We evaluated our approach in PSB1, finding that, in general, Origami performs better than other similar methods, synthesizing the correct solution more often than other methods in most problems. It was also able to obtain the highest count of problems with success rate = 100%, $\geq 75\%$ and $\geq 50\%$ among the GP methods. It also achieved comparable results to Github Copilot, solving some problems that the LLM could not solve.

Finally, we also proposed a novel procedure called AC/DC. This procedure aims at a more efficient exploration of the search space, as it leads to a more diverse population and creates more opportunities for mutation and crossover. We applied this procedure to Origami, and also continued its development by applying the DSLS selection method, expanding the set of functions available, and introducing 3 new patterns that allow it to synthesize a new RS (histomorphism), as well as variations of catamorphism for Maps and Sets. This improved version (Origami_{AC/DC}) was evaluated in 3 different benchmarks, two of which were not considered previously. Experimental results show that, together, these advancements synthesize solutions using all provided RSs, achieving a considerable improvement over its previous version by raising success rates on every problem. Compared to other similar methods in the literature, it has the highest count of problems solved with success rates of 100%, $\geq 75\%$, and $\geq 25\%$ across all benchmarks. In 18% of all benchmark problems it stands as the only method to reach 100% success rate, being the first known approach to achieve it on any problem in the General Program Synthesis Benchmark 2. It also demonstrates competitive performance to LLMs, achieving the highest overall win-rate against Copilot among all GP methods.

7.1 Limitations

This section addresses some current limitations of Origami, which we believe warrant more research and could potentially lead to meaningful scientific contributions.

Currently, both implementations of Origami go through the RSs in a linear fashion, exhausting all the allowed runs in a pattern before advancing to the next one. This could be done more effectively by, for example, applying heuristics that would bias the search towards the most likely pattern. Or, more interestingly, by applying a co-evolutionary approach.

In terms of co-evolution, one option is to have one population for each *slot* that we want to evolve. To exemplify, consider we want to evolve a solution using catamorphism and hylomorphism. These two schemes share the same evolvable expressions for the algebra,

so it could be evolved once and evaluated in both schemes at the same time.

Another approach would be to keep the multi-gene representation, and provide each *pattern* with its own dedicated population, in which individuals will compete amongst themselves and be evaluated with the same fitness function.

Regardless, here we envision a significant opportunity to also share genetic information between different populations. For instance, crossover could be applied in two individuals of different populations. Additionally, some patterns only require minor adaptations to be converted into other patterns. For example, we can easily convert an accumulation into a catamorphism by discarding the **st** expressions, and replacing any reference to the second parameter of **alg** with freshly generated expressions, and a catamorphism into an accumulation by generating expressions for **st**, and injecting the second parameter of **alg** into some part of the expression. We can also trivially combine a catamorphism and an anamorphism into a hylomorphism, given that their types match. All of these are ideas that could lead to interesting results, and can only be explored by an algorithm that employs these RSs.

Additionally, in the text, we have already shown that, from the types of the problem alone, we can discard some patterns that will never be able to provide results of the desired type. However, more information is also available in the example set itself, which can allow us to further narrow down some patterns. For example, a problem of type $[a] \rightarrow [b]$ can be solved by catamorphism, but could be solved by other schemes. However, if we verify that every output has the same length of the input, this is a strong indicator that we want the catamorphism, specifically one that represents a **map** operation. Similarly, in a problem of type $[a] \rightarrow [a]$, if the elements are always in the same order, and the output is always equal or shorter than the input, this could also indicate that we want a catamorphism that represents a **filter** operation. These are invariants that can be derived from the dataset itself, and could be used to bias the search in such a way that it searches for certain schemes more than others.

Also from the training dataset, we could potentially derive information that trivializes the search for specific slots. For example, suppose we are searching for a **Cata** solution, and the examples provide the expected output for the empty list. We can always guarantee that, if a solution that uses the **Cata** pattern exists, its expression for the **NilF** slot must be the output for the empty list. In this situation, as this output is already provided by the dataset, the search can focus just on the **ConsF**, effectively reducing the search space in half. The exact same reasoning can also be applied for **Accu** and **Histo**.

An additional limitation is that **Origami** does not address the issue of making sure that a co-algebra, such as those used in anamorphism or in hylomorphism, terminates. We currently impose a fixed arbitrary memory consumption limit and discard solutions as soon as they exceed it. In general, as termination is undecidable, there will always be instances where it is impossible to determine whether a program halts. Nevertheless, it may be

possible to identify specific cases where termination can be formally proved or disproved, or at the very least apply heuristics that reduce the frequency of non-terminating programs.

Another idea that could improve on how quickly we find solutions is that some constants are more often used than others, such as 0, 1, -1, "", [], etc. This knowledge could be incorporated into the GP process, by prioritizing those constants and trying all of them before moving to generating a random constant.

Another potential avenue of work is Interactive Synthesis HELMUTH et al. (2023), in which we start from a largely reduced number of examples. As the synthesis progresses and a solution is found, the synthesizer asks the user to check if the program generalized correctly by generating unseen examples. These examples are generated heuristically, aiming to allow the user to quickly identify if that is the correct program or not. If the examples show that the solution is actually incorrect, they are included in the train set and the search continues. While this is not a problem that literature benchmarks have, as the scarcity of examples is artificial, such a system would take Origami to a production-ready level.

Bibliography

- BANZHAF, W. et al. **Genetic programming**: an introduction: on the automatic evolution of computer programs and its applications. 1998. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.
- CASTLE, T.; JOHNSON, C. G. Evolving High-Level Imperative Program Trees with Strongly Formed Genetic Programming. In: GENETIC PROGRAMMING, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 2012. p.1–12.
- DE BRUIJN, N. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. **Indagationes Mathematicae (Proceedings)**, v.75, n.5, p.381–392, 1972. Disponível em: <<https://www.sciencedirect.com/science/article/pii/1385725872900340>>.
- FERNANDES, M. C.; FRANÇA, F. O. de; FRANCESQUINI, E. HOTGP - Higher-Order Typed Genetic Programming. In: GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2023. p.1091–1099. (GECCO '23).
- FERNANDES, M. C.; FRANÇA, F. O. de; FRANCESQUINI, E. Origami: (un)folding the abstraction of recursion schemes for program synthesis. In: WINKLER, S. et al. (Ed.). **Genetic Programming Theory and Practice XX**. Singapore: Springer Nature Singapore, 2024. p.263–281. Disponível em: <https://doi.org/10.1007/978-981-99-8413-8_14>.
- FERNANDES, M. C.; FRANÇA, F. O. de; FRANCESQUINI, E. Going Bananas! - Unfolding Program Synthesis with Origami. In: INTELLIGENT SYSTEMS, Cham. **Anais...** Springer Nature Switzerland, 2025. p.3–18.
- FORSTENLECHNER, S. et al. A Grammar Design Pattern for Arbitrary Program Synthesis Problems in Genetic Programming. In: GENETIC PROGRAMMING, Cham. **Anais...** Springer International Publishing, 2017. p.262–277.
- FORSTENLECHNER, S. et al. Extending Program Synthesis Grammars for Grammar-Guided Genetic Programming. In: PARALLEL PROBLEM SOLVING FROM

- NATURE – PPSN XV, Cham. **Anais...** Springer International Publishing, 2018. p.197–208.
- FRANKLE, J. et al. Example-Directed Synthesis: a type-theoretic interpretation. In: ANNUAL ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 43., New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2016. p.802–815. (POPL '16). Disponível em: <<https://doi.org/10.1145/2837614.2837629>>.
- FREITAS, A. A. **Data mining and knowledge discovery with evolutionary algorithms**. 2002. Springer Science & Business Media.
- GARLAND, S. J.; LUCKHAM, D. C. Program schemes, recursion schemes, and formal languages. **Journal of Computer and System Sciences**, v.7, n.2, p.119–160, 1973.
- GARROW, F.; LONES, M. A.; STEWART, R. Why Functional Program Synthesis Matters (in the Realm of Genetic Programming). In: GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE COMPANION, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2022. p.1844–1853. (GECCO '22).
- GIBBONS, J. **The fun of programming**. 2003.
- GREFENSTETTE, J.; DALEY, R. Methods for competitive and cooperative co-evolution. In: ADAPTATION, COEVOLUTION AND LEARNING IN MULTIAGENT SYSTEMS: PAPERS FROM THE 1996 AAAI SPRING SYMPOSIUM. **Anais...** 1996. p.45–50.
- GULWANI, S. Dimensions in Program Synthesis. In: INTERNATIONAL ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF DECLARATIVE PROGRAMMING, 12., New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2010. p.13–24. (PPDP '10). Disponível em: <<https://doi.org/10.1145/1836089.1836091>>.
- HARVEY, B. Avoiding recursion. **Hoyles, C., Noss, R. Learning Mathematics and LOGO. The MIT Press Cambridge, Massachusetts, USA**, v.393426, 1992.
- HAYNES, T. et al. Strongly Typed Genetic Programming in Evolving Cooperation Strategies. In: INTERNATIONAL CONFERENCE ON GENETIC ALGORITHMS, 6., San Francisco, CA, USA. **Proceedings...** Morgan Kaufmann Publishers Inc., 1995. p.271–278.

- HE, Y.; ARANHA, C.; SAKURAI, T. Incorporating Sub-Programs as Knowledge in Program Synthesis by PushGP and Adaptive Replacement Mutation. In: GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE COMPANION, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2022. p.554–557. (GECCO '22). Disponível em: <<https://doi.org/10.1145/3520304.3528891>>.
- HELMUTH, T. et al. Improving Generalization of Evolved Programs through Automatic Simplification. In: GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2017. p.937–944. (GECCO '17). Disponível em: <<https://doi.org/10.1145/3071178.3071330>>.
- HELMUTH, T. et al. Human-Driven Genetic Programming for Program Synthesis: a prototype. In: COMPANION CONFERENCE ON GENETIC AND EVOLUTIONARY COMPUTATION, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2023. p.1981–1989. (GECCO '23 Companion). Disponível em: <<https://doi.org/10.1145/3583133.3596373>>.
- HELMUTH, T.; KELLY, P. PSB2: the second program synthesis benchmark suite. In: GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2021. p.785–794. (GECCO '21). Disponível em: <<https://doi.org/10.1145/3449639.3459285>>.
- HELMUTH, T.; MCPHEE, N. F.; SPECTOR, L. Program synthesis using uniform mutation by addition and deletion. In: GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE. **Proceedings...** 2018. p.1127–1134.
- HELMUTH, T.; MCPHEE, N. F.; SPECTOR, L. Program Synthesis Using Uniform Mutation by Addition and Deletion. In: GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2018. p.1127–1134. (GECCO '18). Disponível em: <<https://doi.org/10.1145/3205455.3205603>>.
- HELMUTH, T.; SPECTOR, L. General Program Synthesis Benchmark Suite. In: ANNUAL CONFERENCE ON GENETIC AND EVOLUTIONARY COMPUTATION, 2015., New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2015. p.1039–1046. (GECCO '15). Disponível em: <<https://doi.org/10.1145/2739480.2754769>>.
- HELMUTH, T.; SPECTOR, L. Problem-solving benefits of down-sampled lexicase selection. **Artificial Life**, v.27, n.3–4, p.183–203, 2022.

- HEMBERG, E.; KELLY, J.; O'REILLY, U.-M. On Domain Knowledge and Novelty to Improve Program Synthesis Performance with Grammatical Evolution. In: GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2019. p.1039–1046. (GECCO '19). Disponível em: <<https://doi.org/10.1145/3321707.3321865>>.
- HOFMANN, M. **Schema-guided inductive functional programming through automatic detection of type morphisms**. 2010. Tese Otto-Friedrich-Universität Bamberg, Fakultät Wirtschaftsinformatik und
- HOLLAND, J. H. Adaptation in natural and artificial systems. an introductory analysis with applications to biology, control and artificial intelligence. **Ann Arbor: University of Michigan Press**, 1975.
- HONG, Q.; AIKEN, A. Recursive Program Synthesis using Paramorphisms. **Proc. ACM Program. Lang.**, New York, NY, USA, v.8, n.PLDI, Jun. 2024. Disponível em: <<https://doi.org/10.1145/3656381>>.
- JONES, M. P. A system of constructor classes: overloading and implicit higher-order polymorphism. **Journal of functional programming**, v.5, n.1, p.1–35, 1995.
- JONES, N. D. The expressive power of higher-order types or, life without CONS. **Journal of Functional Programming**, v.11, n.1, p.55–94, 2001.
- KALKREUTH, R. et al. General Boolean Function Benchmark Suite. In: ACM/SIGEVO CONFERENCE ON FOUNDATIONS OF GENETIC ALGORITHMS, 17. **Proceedings...** 2023. p.84–95.
- KATAYAMA, S. Systematic search for lambda expressions. **Trends in functional programming**, v.6, p.111–126, 2005.
- KELLY, J.; HEMBERG, E.; O'REILLY, U.-M. Improving Genetic Programming with Novel Exploration - Exploitation Control. In: GENETIC PROGRAMMING, Cham. **Anais...** Springer International Publishing, 2019. p.64–80.
- KOZA, J. R. **Genetic Programming**: on the programming of computers by means of natural selection. 1992. Bradford Books. Cambridge, MA: Bradford Books, 1992. (Complex Adaptive Systems).
- KOZA, J. R.; POLI, R. Genetic Programming. In: BURKE, E. K.; KENDALL, G. (Ed.). **Search Methodologies**: introductory tutorials in optimization and decision support techniques. Boston, MA: Springer US, 2005. p.127–164. Disponível em: <https://doi.org/10.1007/0-387-28356-0_5>.

- KRONBERGER, G. et al. **Symbolic Regression**. 2024. Chapman & Hall / CRC Press.
- LENAT, D. B. Automated theory formation in mathematics. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE - VOLUME 2, 5., San Francisco, CA, USA. **Proceedings...** Morgan Kaufmann Publishers Inc., 1977. p.833–842. (IJCAI'77).
- LIN, J.; ZHU, L.; GAO, K. A genetic programming hyper-heuristic approach for the multi-skill resource constrained project scheduling problem. **Expert Systems with Applications**, v.140, p.112915, 2020.
- LIU, Z. et al. No need to lift a finger anymore? assessing the quality of code generation by chatgpt. **IEEE Transactions on Software Engineering**, 2024.
- LUKE, S.; PANAIT, L. A comparison of bloat control methods for genetic programming. **Evolutionary computation**, v.14, n.3, p.309–344, 2006.
- MANNA, Z.; WALDINGER, R. A Deductive Approach to Program Synthesis. **ACM Trans. Program. Lang. Syst.**, New York, NY, USA, v.2, n.1, p.90–121, jan 1980. Disponível em: <<https://doi.org/10.1145/357084.357090>>.
- MANNA, Z.; WALDINGER, R. J. Toward Automatic Program Synthesis. **Commun. ACM**, New York, NY, USA, v.14, n.3, p.151–165, mar 1971. Disponível em: <<https://doi.org/10.1145/362566.362568>>.
- MANRIQUE, D.; RÍOS, J.; RODRÍGUEZ-PATÓN, A. Grammar-Guided Genetic Programming. In: **Encyclopedia of Artificial Intelligence**. p.767–773. Disponível em: <<https://doi.org/10.4018/978-1-59904-849-9.ch114>>.
- MEIJER, E.; FOKKINGA, M.; PATERSON, R. Functional programming with bananas, lenses, envelopes and barbed wire. In: FUNCTIONAL PROGRAMMING LANGUAGES AND COMPUTER ARCHITECTURE: 5TH ACM CONFERENCE CAMBRIDGE, MA, USA, AUGUST 26–30, 1991 PROCEEDINGS 5. **Anais...** Springer, 1991. p.124–144.
- MONTANA, D. J. Strongly typed genetic programming. **Evolutionary computation**, v.3, n.2, p.199–230, 1995.
- NADIZAR, G.; MEDVET, E.; WILSON, D. G. Naturally interpretable control policies via graph-based genetic programming. In: EUROPEAN CONFERENCE ON GENETIC PROGRAMMING (PART OF EVOSTAR). **Anais...** Springer, 2024. p.73–89.

- NORDIN, P. A Compiling Genetic Programming System that Directly Manipulates the Machine Code. In: **Advances in Genetic Programming, Volume 1**. Disponível em: <<https://doi.org/10.7551/mitpress/1108.003.0019>>.
- NORDIN, P.; BANZHAF, W. Genetic programming controlling a miniature robot. In: WORKING NOTES FOR THE AAAI SYMPOSIUM ON GENETIC PROGRAMMING. **Anais...** MIT, Cambridge, MA, USA, AAAI, 1995. v.61, p.67.
- OLSSON, R. Inductive functional programming using incremental program transformation. **Artificial intelligence**, v.74, n.1, p.55–81, 1995.
- O'NEILL, M.; RYAN, C. Grammatical evolution. **IEEE Transactions on Evolutionary Computation**, v.5, n.4, p.349–358, 2001.
- PANTRIDGE, E. et al. On the Difficulty of Benchmarking Inductive Program Synthesis Methods. In: GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE COMPANION, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2017. p.1589–1596. (GECCO '17). Disponível em: <<https://doi.org/10.1145/3067695.3082533>>.
- PANTRIDGE, E.; HELMUTH, T. Solving Novel Program Synthesis Problems with Genetic Programming Using Parametric Polymorphism. In: GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2023. p.1175–1183. (GECCO '23). Disponível em: <<https://doi.org/10.1145/3583131.3590502>>.
- PANTRIDGE, E.; HELMUTH, T.; SPECTOR, L. Functional Code Building Genetic Programming. In: GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2022. p.1000–1008. (GECCO '22).
- PANTRIDGE, E.; SPECTOR, L. Code Building Genetic Programming. In: GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE, 2020., New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2020. p.994–1002. (GECCO '20). Disponível em: <<https://doi.org/10.1145/3377930.3390239>>.
- PAWLAK, T. P.; KRAWIEC, K. Competent geometric semantic genetic programming for symbolic regression and boolean function synthesis. **Evolutionary computation**, v.26, n.2, p.177–212, 2018.
- RAY, B. et al. A Large-Scale Study of Programming Languages and Code Quality in GitHub. **Commun. ACM**, New York, NY, USA, v.60, n.10, p.91–100, sep 2017. Disponível em: <<https://doi.org/10.1145/3126905>>.

- RUSSELL, B.; WHITEHEAD, A. N. **Principia Mathematica, Vol. I**. 1910. Cambridge University Press. Cambridge: Cambridge University Press, 1910.
- SALGOTRA, R.; GANDOMI, M.; GANDOMI, A. H. Time series analysis and forecast of the COVID-19 pandemic in India using genetic programming. **Chaos, Solitons & Fractals**, v.138, p.109945, 2020.
- SANTOS, P.; SILVA, S.; FONSECA, A. Refined Typed Genetic Programming as a User Interface for Genetic Programming. In: GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE COMPANION, 2020., New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2020. p.251–252. (GECCO '20). Disponível em: <<https://doi.org/10.1145/3377929.3390042>>.
- SEARSON, D. P.; LEAHY, D. E.; WILLIS, M. J. GPTIPS: an open source genetic programming toolbox for multigene symbolic regression. In: INTERNATIONAL MULTICONFERENCE OF ENGINEERS AND COMPUTER SCIENTISTS. **Proceedings...** Citeseer, 2010. v.1, p.77–80.
- SHARIATI, M. et al. Application of Extreme Learning Machine (ELM) and Genetic Programming (GP) to design steel-concrete composite floor systems at elevated temperatures. **Steel Compos. Struct**, v.33, n.3, p.319–332, 2019.
- SHEIKHOESLAMI, S. **Ablation programming for machine learning**. 2019.
- SOBANIA, D. et al. A Comparison of Large Language Models and Genetic Programming for Program Synthesis. **IEEE Transactions on Evolutionary Computation**, p.1–1, 2024.
- SØNDERGAARD, H.; SESTOFT, P. Referential transparency, definiteness and unfoldability. **Acta Informatica**, v.27, p.505–517, 1990.
- SOULE, T.; HECKENDORN, R. B. Improving performance and cooperation in multi-agent systems. **Genetic Programming Theory and Practice V**, p.221–237, 2008.
- SPECTOR, L. Autoconstructive evolution: push, pushgp, and pushpop. In: GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE. **Proceedings...** 2001. (GECCO '01).
- SPECTOR, L. Autoconstructive evolution: push, pushgp, and pushpop. In: GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE (GECCO-2001). **Proceedings...** 2001. v.137.
- SPECTOR, L. Assessment of problem modality by differential performance of lexicon case selection in genetic programming: a preliminary report. In: ANNUAL

- CONFERENCE COMPANION ON GENETIC AND EVOLUTIONARY COMPUTATION, 14., New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2012. p.401–408. (GECCO '12). Disponível em: <<https://doi.org/10.1145/2330784.2330846>>.
- STRACHEY, C. Fundamental Concepts in Programming Languages. **Higher Order Symbol. Comput.**, USA, v.13, n.1–2, p.11–49, Apr. 1967. Disponível em: <<https://doi.org/10.1023/A:1010000313106>>.
- SWAN, J.; KRAWIEC, K.; KOCSIS, Z. A. Stochastic synthesis of recursive functions made easy with bananas, lenses, envelopes and barbed wire. **Genetic Programming and Evolvable Machines**, USA, v.20, n.3, p.327–350, sep 2019. Disponível em: <<https://doi.org/10.1007/s10710-019-09347-3>>.
- TURNER, D. A. A new implementation technique for applicative languages. **Software: Practice and Experience**, v.9, n.1, p.31–49, 1979.
- WADLER, P. Theorems for free! In: FUNCTIONAL PROGRAMMING LANGUAGES AND COMPUTER ARCHITECTURE. **Proceedings...** 1989. p.347–359.
- YU, G. T. **An analysis of the impact of functional programming techniques on genetic programming**. 1999. TeseUniversity of London, University College London (United Kingdom).
- YU, T. Polymorphism and Genetic Programming. In: GENETIC PROGRAMMING, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 2001. p.218–233.
- YU, T.; CLACK, C. PolyGP: a polymorphic genetic programming system in haskell. In: GENETIC PROGRAMMING 1998: PROCEEDINGS OF THE THIRD ANNUAL CONFERENCE, University of Wisconsin, Madison, Wisconsin, USA. **Anais...** Morgan Kaufmann, 1998. p.416–421. Disponível em: <http://www.cs.mun.ca/~tinayu/index_files/addr/public_html/pgp.new.pdf>.
- ZHANG, F. et al. Genetic programming of macrophages to perform anti-tumor functions using targeted mRNA nanocarriers. **Nature communications**, v.10, n.1, p.1–16, 2019.

Canonical solutions to PSB1 problems

Here we present the complete set of solutions to the PSB1 benchmark. All of these solutions were written by the authors in order to get a better understanding of the recurrent patterns that could be used to solve the benchmark.

8.1 No recursion

In this section we outline the solutions to the easiest problems that do not require any recursive scheme. We should note that by wrapping the inputs of these problems in a list with a single element, we could fit them into the discussed recursive patterns. However, it would just add unnecessary complications to the code.

Even though the solutions to these problems are trivial, for the sake of completeness, we give the description of each problem followed by their solutions. By their simplicity, these solutions dispense the need for additional explanations.

8.1.1 Number IO

Given an integer and a float, print their sum.

```
-- required primitives: fromIntegral, (+)
numberIO :: Int -> Double -> Double
numberIO x y = fromIntegral x + y
```

8.1.2 Small or Large

Given an integer n , print “small” if $n < 1000$ and “large” if $n \geq 2000$ (and nothing if $1000 \leq n < 2000$).

```
-- required primitives: multi-way if
-- provided primitives: (< 1000), (>= 2000),
-- "small", "large"
smallOrLarge :: Int -> String
smallOrLarge x = if | (< 1000) x -> "small"
                    | (>= 2000) x -> "large"
                    | otherwise -> ""
```

8.1.3 Compare String Lengths

Given three strings $n1$, $n2$, and $n3$, return true if $\text{length}(n1) < \text{length}(n2) < \text{length}(n3)$, and false otherwise.

```
-- required primitives: length, (<), (&&)
strLengths :: String -> String -> String -> Bool
strLengths a b c = length a < length b
                  && length b < length c
```

8.1.4 Median

Given 3 integers, print their median.

```
-- required primitives: min, max, (+), (-)
medianOfThree :: Int -> Int -> Int -> Int
medianOfThree a b c = (a + b + c) - (min a (min b c))
                    - (max a (max b c))
```

8.1.5 Smallest

Given 4 integers, print the smallest of them.

```
-- required primitives: min
smallest :: Int -> Int -> Int -> Int -> Int
medianOfThree a b c d = min a (min b (min c d))
```

8.2 Catamorphisms

Here we report the problems that were solved by using catamorphism. The next three problems have already been presented and dissected in the text. For the sake of completeness, we replicate them here:

8.2.1 Count Odds

Given a vector of integers, return the number of integers that are odd, without use of a specific even or odd instruction (but allowing instructions such as modulo and quotient).

```
-- required primitives: constant int, mod, +
countOdds :: [Int] -> Int
countOdds = cata alg . fromList where
    alg NilF = 0
    alg (ConsF x acc) = mod x 2 + acc
```

8.2.2 Double Letters

Given a string, print (in our case, return) the string, doubling every letter character, and tripling every exclamation point. All other non-alphabetic and non-exclamation characters should be printed a single time each.

```
-- required primitives: if-then-else, (<>), ([])
-- user provided: constant '!', constant "!!!", isLetter
doubleLetters :: String -> String
doubleLetters = cata alg . fromList where
  alg NilF      = []
  alg (ConsF x xs) = x == '!'
                      then "!!!" <> xs
                      else if isLetter x then [x,x] <> xs
                      else x:xs
```

8.2.3 Super Anagrams

Given strings x and y of lowercase letters, return true if y is a super anagram of x , which is the case if every character in x is in y . To be true, y may contain extra characters, but must have at least as many copies of each character as x does.

```
-- required primitives: delete, constant bool
-- elem, (∩)
superAnagram :: String -> (String -> Bool)
superAnagram = cata alg . fromList where
  alg NilF = \ys -> True
  alg (ConsF x f) = \ys -> elem x ys && f (delete x ys)
```

8.2.4 String Lengths Backwards

Given a vector of strings, print the length of each string in the vector starting with the last and ending with the first.

```
-- required primitives: length, snoc
strLenBack :: [String] -> [Int]
strLenBack = cata alg . fromList where
  alg NilF = []
  alg (ConsF x xs) = snoc (length x) xs
```

The initial condition is simply the empty list and we simply append the length of each element to the end of the resulting list (snoc).

8.2.5 Negative To Zero

Given a vector of integers, return the vector where all negative integers have been replaced by 0.

```
-- required primitives: max, (:), int constants
negativeToZero :: [Int] -> [Int]
negativeToZero = cata alg . fromList where
    alg NilF = []
    alg (ConsF x xs) = max 0 x : xs
```

The initial condition is the neutral element of a list and we just cons (:) maximum between 0 and each element to the resulting list.

8.2.6 Scrabble Score

Given a string of visible ASCII characters, return the Scrabble score for that string. Each letter has a corresponding value according to normal Scrabble rules, and non-letter characters are worth zero.

```
-- required primitives: findMap, constant int, (+)
-- user provided primitives: scores, a hash table
-- mapping the characters to their scores
scrabbleScore :: String -> Int
scrabbleScore = cata alg . fromList where
    alg NilF = 0
    alg (ConsF x xs) = getScore x + xs
```

This solution assumes that the user can provide a function that returns the correspondence between any character to a score.

8.2.7 Mirror Image

Given two vectors of integers, return true if one vector is the reverse of the other, and false otherwise.

```
-- required primitives: init, last, null, not, (Eq), (==)
mirrorImage :: [Int] -> ([Int] -> Bool)
mirrorImage = cata alg . fromList where
    alg NilF ys = null ys
    alg (ConsF x xs) ys = (not.null) ys && last ys == x && xs (init ys)
```

For this problem we must return a function that takes the second list and returns a boolean. As such, for the `NilF` case we return whether the argument is null signaling that all of the elements were consumed. In the `ConsF` case we must check if we still have elements to consume, compare the last element to `x` and apply `xs` to the list `ys` without the last value.

8.2.8 Vectors Summed

Given two equal-sized vectors of integers, return a vector of integers that contains the sum of the input vectors at each index.

```
-- required primitives: tail, head, (+), (:)
sumOfVecs :: [Int] -> ([Int] -> [Int])
sumOfVecs = cata alg . fromList where
  alg NilF ys = []
  alg (ConsF x xs) ys = if null ys then []
                        else (x + head ys : xs (tail ys))
```

When there is no element to consume from the first list, we simply return the empty list, as there is nothing to add. When we do have at least one element to consume, we check if we still have an element of the second list, if we do, we add those two together and concatenate to the remainder of the list.

8.2.9 Grade

Given 5 integers, the first four represent the lower numeric thresholds for achieving an A, B, C, and D, and will be distinct and in descending order. The fifth represents the student's numeric grade. The program must print Student has a X grade., where X is A, B, C, D, or F depending on the thresholds and the numeric grade.

```
-- required primitives: constant "ABCDF", if-then-else
-- (<), head, tail.
grade :: [(Double, Char)] -> (Double -> Char)
grade = cata alg . fromList where
  alg NilF n = 'F'
  alg (ConsF x xs) n = if n >= fst x then snd x else xs n
```

Assuming we provide a list of tuples where the first element is the threshold and the second is the grade, we can traverse this structure and compare the second argument value with the threshold, if it is within this bound, we return the corresponding grade, otherwise we keep processing. The empty case means that the only option is to fail the student with an 'F'.

8.3 Anamorphisms

Here we report the problems that were solved by using anamorphism.

8.3.1 For Loop Index

Given 3 integer inputs start, end, and step, print the integers in the sequence $n_0 = \text{start}$, $n_i = n_{i-1} + \text{step}$ for each $n_i < \text{end}$, each on their own line.

```
-- required primitives: (==), (+)
forLoopIndex :: Int -> Int -> Int -> [Int]
forLoopIndex start end step = toList (ana coalg start) where
    coalg seed = case seed == end of
        True -> NilF
        False -> ConsF seed (seed + step)
```

This is already explained in depth in the text.

8.3.2 Digits

Given an integer, print that integer's digits each on their own line starting with the least significant digit. A negative integer should have the negative sign printed before the most significant digit.

```
-- required primitives: constant int, (==)
-- abs, (<), rem, quot, if-then-else
digits :: Int -> [Int]
digits x = toList $ ana coalg x where
    coalg x =
        case x == 0 of
            True -> NilF
            False -> ConsF (if abs x < 10
                            then (x `rem` 10)
                            else abs (x `rem` 10))
                            (x `quot` 10)
```

Because this problem requires that we handle negative values, the program that produces the next element should verify if we are at the last digit. In case we are not, we should calculate the absolute value to ensure that we do not display the minus sign. Even though this solution is larger than the previous problem, the search process is simplified for the fact that there is a single argument, thus all the operations must involve the argument and constant values.

8.4 Hylomorphisms

Here we report the problems that were solved by using hylomorphism.

8.4.1 Collatz Numbers

Given an integer, find the number of terms in the Collatz (hailstone) sequence starting from that integer.

```
-- required primitives: constant int, (==)
-- (+), (*), mod, div
collatz :: Int -> Int
collatz = hylo alg coalg where
  alg NilF = 1
  alg (ConsF x acc) = 1 + acc

  coalg x =
    case x == 1 of
      True -> NilF
      False -> ConsF x (if mod x 2 == 0
                        then div x 2
                        else div (3*x + 1) 2)
```

This is already explained in depth in the text.

8.4.2 Sum of Squares

Given integer n , return the sum of squaring each integer in the range $[1, n]$.

```
-- required primitives: (^), (+), int constants
sumOfSquares :: Int -> Int
sumOfSquares n = hylo alg coalg n where
  alg NilF = 0
  alg (ConsF x xs) = x^2 + xs

  coalg x = case x == 0 of
    True -> NilF
    False -> ConsF x (x - 1)
```

The initial condition is the neutral element of addition and we just add the square of each value.

8.4.3 Even Squares

Given an integer n , print all of the positive even perfect squares less than n on separate lines.

```
-- required primitives: constant int, (+),
-- (^), (>=), mod
evenSquares :: Int -> [Int]
evenSquares n = hylo alg coalg n where
  alg NilF = []
  alg (ConsF x xs) = if x >= n then xs else x : xs
  coalg seed = case seed <= 1 of
    True -> NilF
    False -> ConsF (seed ^ 2) (seed - 1 - (1 - mod seed 2))
```

In this program, the starting seed must be a constant value, while the single argument is used to form the condition for stopping growing the structure.

8.5 Accumulations

Here we report the problems that were solved by using accumulation. The following two problems are already explored in the text, and replicated here for completeness.

8.5.1 Last Index of Zero

Given a vector of integers, at least one of which is 0, return the index of the last occurrence of 0 in the vector.

```
-- required primitives: if-then-else, (+), (==)
-- (<>), constant int, Maybe, Last
lastIndexZero :: [Int] -> Int
lastIndexZero xs = accu st alg (fromList xs) 0 where
  st NilF s = NilF
  st (ConsF x xs) s = ConsF x (xs, s+1)

  alg NilF i = -1
  alg (ConsF x acc) i = if x == 0 && acc == -1
                        then i
                        else acc
```

8.5.2 Vector Average

Given a vector of floats, return the average of those floats. Results are rounded to 4 decimal places.

```
-- required primitives: (+), (/)
vecAvg :: [Double] -> Double
vecAvg xs = accu st alg (fromList xs) (0.0, 0.0) where
  st NilF _ = NilF
  st (ConsF x xs) (s1, s2) = ConsF x (xs, (s1 + x, s2 + 1))

  alg NilF (s1, s2) = s1 / s2
  alg (ConsF x acc) s = acc
```

8.5.3 Checksum

Given a string, convert each character in the string into its integer ASCII value, sum them, take the sum modulo 64, add the integer value of the space character, and then convert that integer back into its corresponding character (the checksum character). The program must print Check sum is X, where X is replaced by the correct checksum character.

```
-- required primitives: mod, fromEnum, toEnum
-- constant 64, constant ' ', (+)
checksum :: String -> Char
checksum = accu st alg (fromList xs) 0 where
  st NilF s = NilF
  st (ConsF x xs) s = ConsF x (xs,
    toEnum (mod (fromEnum x + fromEnum s) 64))
  alg NilF s = toEnum (fromEnum s + 32)
  alg (ConsF x xs) s = xs
```

The base case (NilF) is the white space character that must be included into the calculation, after that, at every step, we add the current character value to the value of x, calculates the modulus 6 and convert it back to a character (as constrained by the type of xs).

8.5.4 String Differences

Given 2 strings (without whitespace) as input, find the indices at which the strings have different characters, stopping at the end of the shorter one. For each such index, print a line containing the index as well as the character in each string.

```

-- required primitives: if-then-else, (:), (+), (/=)
-- constant int, null, head, tail
stringDiffs :: String -> (String -> [(Int, (Char, Char))])
stringDiffs xs ys = accu st alg (fromList xs) 0 ys where
    st NilF s = NilF
    st (ConsF x xs) s = ConsF x (xs, s+1)
    alg NilF s zs = []
    alg (ConsF x xs) s zs =
        if null zs
        then []
        else if x /= head zs
            then (s, (x, head zs)) : xs (tail zs)
            else xs (tail zs)

```

The accumulator will then store the index of each pair as it traverse the lists. Afterwards, we compare each character and, if they are different, we store it into the final result, other wise we skip it.

8.5.5 X-Word Lines

Given an integer X and a string that can contain spaces and newlines, print the string with exactly X words per line. The last line may have fewer than X words.

```

-- required primitives: tuples, constant int
-- constants ' ' and '\n', if-then-else, (/=), (==)
-- (+), bimap, fst, second, const, snoc, (⋈)
xWordLines :: Int -> String -> String
xWordLines n xs = accu st alg (fromList xs) 1 where
    st NilF s = NilF
    st (ConsF x xs) s =
        ConsF x (xs, if x /= ' ' && x /= '\n'
            then s
            else s + 1)

    alg NilF s = []
    alg (ConsF x xs) s = if
        | x == ' ' && mod s n == 0 -> '\n' : xs
        | x == '\n' && mod s n /= 0 -> ' ' : xs
        | otherwise -> x : xs

```

For this program, we need the accumulator to hold a value of how many words have been seen so far, we do that by checking for a whitespace or carriage return. If the current char is none of those, we simply concatenate the char into the end of the accumulator. If the char is one of those, we check if we already have n words, if we do we reset the counter and insert the carriage return into the string, otherwise we add one to the counter of words and insert the whitespace. In our base library, the functions `fst`, `snd` return the first and second element of a tuple, respectively. The functions `(first, second, bimap)`, applies a function to the first, second, or two functions to both elements of a tuple.

8.5.6 Word Stats

Given a file, print the number of words containing n characters for n from 1 to the length of the longest word. At the end of the output, print a line that gives the number of sentences and line that gives the average sentence length.

```
-- required primitives: Map, if-then-else
-- (/=), constants ' ', '\n', first, fst, snd
-- insertWith, constant int
wordDist :: String -> Map Int Int
wordDist xs = accu st alg (fromList xs) 0 where
  st NilF s = NilF
  st (ConsF x xs) s =
    ConsF x (xs,
      if x /= ' ' && x /= '\n'
      then s + 1
      else 0)
  alg NilF s = singleton s 1
  alg (ConsF x xs) s = if x == ' ' || x == '\n'
    then insertWith (+) s 1 xs
    else xs

-- required primitives: constant int, constant '\n'
-- if-then-else
lineCount :: String -> Int
lineCount = cata alg . fromList
  where
    alg NilF = 1
    alg (ConsF x xs) = xs + if x == '\n' then 1 else 0

-- required primitives: tuples, constant '\n'
```



```

-- if-then-else, first, second, uncurry, (/)
avgLineLen :: String -> Double
avgLineLen xs = accu st alg (fromList xs) (0.0, 0.0)
  where
    f x = if x /= '\n'
          then s + 1
          else s
    g x = if x /= '\n'
          then s
          else s + 1
    st NilF s = NilF
    st (ConsF x xs) s = ConsF x (xs, (f (fst s), g (snd s)))
    alg NilF (a, b) = a / (b + 1)
    alg (ConsF x xs) s = xs

```

This problem is possibly the most challenging of the benchmark to be synthesized all at once. One should notice that this is actually three problems into one. The first program deals with calculating the word distribution. It initializes the accumulator as a tuple of a count of char and an empty map. While traversing the structure, it adds 1 to the count until it reaches the end of a word. When it does, it adds the word length with a count of one. The algebra should start adding the last count of the accumulator as it will reach the end of a string without breaking a word. After that it just returns the map containing the word distribution.

The second program simply count the occurrences of '\n' indicating a new line. The last program, increment one counter for every character of a line and the second for every carriage returns, dividing the result in the end.

We also highlight the fact that folds that do not have any post-processing could always be combined by storing the accumulators into an n -tuple and applying the functions in their corresponding elements. We do not show that as the resulting program would become too large to report here.