UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Program Synthesis with Refinement Typed Genetic Programming

José Eduardo Inácio Madeira

Mestrado em Engenharia Informática

Dissertação orientada por:
Prof. Doutor Alcides Miguel Cachulo Aguiar Fonseca

2025

Agradecimentos

O final desta dissertação não é apenas o fechar de um capítulo académico; é o fechar de vários capítulos de vida que, por caminhos mais ou menos caóticos, me trouxeram até aqui e me fizeram crescer como pessoa. E por isso quero falar das pessoas que realmente fizeram a diferença neste percurso.

Começo pelo que sempre esteve na base de tudo, a minha família. Agradeço profundamente à minha mãe, ao meu pai e ao meu avô pelo apoio incondicional, por nunca colocarem pressão para ser o melhor, mas sim para ser alguém de quem eu próprio me orgulhasse. E não posso deixar de agradecer igualmente à minha namorada, que esteve sempre lá para me apoiar nos momentos mais stressantes e para celebrar cada conquista comigo. Sem vocês, este caminho teria sido muito mais difícil, e certamente muito menos feliz.

Seguindo para aqueles que fizeram do percurso académico muito mais do que trabalho, quero agradecer aos membros do LASIGE que, direta ou indiretamente, fizeram parte desta jornada. Pelas conversas de corredor, pelos feedbacks ocasionais e pelos cafés de manhã ou depois de almoço, deixo o meu sincero obrigado. Em especial, e por qualquer ordem, agradeço ao Espada, Batista, Nuno, Ruben, João, Paulo e Catarina, que estiveram sempre dispostos a ajudar, a ouvir um desabafo e a dar força nos momentos menos bons. Levo comigo memórias incríveis convosco, demasiado numerosas para enumerar aqui. Também ao André, que está comigo desde a licenciatura e que sobreviveu a projetos intermináveis, cadeiras com prazos impossíveis e noites inteiras de debug. Obrigado por estares sempre presente.

Por fim, quero agradecer ao meu orientador, Alcides Fonseca, que não me ensinou apenas a fazer investigação, mas a pensar melhor. Obrigado por me desafiar, pela paciência e pelas oportunidades que me abriram portas para mundos que nunca imaginaria, desde conferências até uma summer school do outro lado do planeta. Foi um choque cultural, académico e pessoal que me marcou e me fez crescer.

A todos vocês, obrigado por nunca me deixarem esquecer que o percurso académico também é feito de pessoas, risos e partilhas. Este mestrado termina aqui, mas representa apenas o início de um caminho que só consigo imaginar graças ao apoio que recebi ao longo destes anos.

This thesis was funded by the Fundação para a Ciência e Tecnologia through project Resource Aware Programming `EXPL/CCI-COM/1306/2021` and advanced computing grants `POR011PRE`, 2022.15800.CPCA.A1, and CPCA/A1/402869/2021.

Dedico-te a ti, vovó. Onde quer que estejas, levo-te sempre comigo.

Resumo

Esta dissertação apresenta o desenvolvimento e a avaliação de um novo enquadramento de síntese de programas denominado **Refinement-Typed Genetic Programming (RTGP)**. Este enquadramento combina os princípios formais dos *tipos refinados* com as técnicas de *programação genética*, permitindo a geração automática de programas que são não apenas sintaticamente válidos, mas também semanticamente consistentes com as restrições impostas pelos tipos. O trabalho surge da observação de que, em muitos sistemas de síntese de programas baseados em busca evolutiva, uma parte significativa do esforço computacional é desperdiçada na produção de candidatos inválidos ou incoerentes, conduzindo a uma exploração ineficiente do espaço de soluções. O uso de tipos refinados oferece um meio rigoroso de restringir o espaço de procura, garantindo simultaneamente correção e expressividade.

O sistema proposto foi implementado através da extensão da linguagem Æon, uma linguagem de síntese declarativa concebida para representar programas com propriedades formais explícitas, e da sua integração com uma ferramenta evolutiva GeneticEngine. A integração resultante permite que o processo de síntese seja orientado por informações de tipo e por restrições lógicas expressas nos refinamentos. A arquitetura resultante suporta tanto tipos refinados restritos, que definem condições de validade obrigatórias, como tipos refinados não restritos, que modelam preferências ou objetivos de otimização a serem considerados na função de *fitness*.

A implementação de RTGP introduz várias inovações técnicas relevantes. Em primeiro lugar, a geração dinâmica de gramáticas permite que as produções sintáticas sejam derivadas diretamente do contexto do programa e das anotações de tipo, garantindo coerência entre o espaço gramatical e o sistema de tipos. Em segundo lugar, o mecanismo de validação de refinamentos assegura que apenas indivíduos que satisfazem as propriedades formais especificadas são considerados válidos durante a evolução. Por fim, foram realizadas melhorias ao GeneticEngine, incluindo suporte para otimização multi-objetivo através do algoritmo NSGA-II, suporte da disjunção de multiplos (Meta-Handlers), e uma inicialização mais diversificada de valores numéricos, o que aumentou a robustez e a diversidade populacional.

Do ponto de vista metodológico, o principal objetivo foi investigar até que ponto os tipos refinados podem guiar e restringir o processo evolutivo, melhorando a eficiência da síntese e a qualidade das soluções. Para o efeito, foi conduzida uma avaliação experimental composta por dois componentes: (i) 22 problemas retirados da *Program Synthesis Benchmark Suite 2* e (ii) um caso de estudo adicional centrado na geração de níveis do jogo *Super Mario*. Os exercícios retirados da

benchmark seguem o paradigma clássico de síntese orientada por exemplos de entrada-saída, estes problemas foram utilizados para avaliar a capacidade da linguagem ÆON em gerar soluções sintaticamente válidas e para verificar se o seu sintetizador produzia exclusivamente programas em ÆON bem tipados. Enquanto o caso de estudo final utiliza funções auxiliares personalizadas para medir múltiplos objetivos durante a geração de um nível, e este sim, foi para testar a nossa implementação de síntese de programas com RTGP.

Cada problema da *benchmark* foi executado trinta vezes de forma independente sob duas representações distintas: a *Tree-Based Representation* e a *Grammatical Evolution Representation*. Ambas foram configuradas com os mesmos parâmetros e capacidade computacional, permitindo uma comparação direta entre abordagens. Antes da execução dos ensaios evolutivos, foram implementadas manualmente soluções de referência para garantir que pelo menos uma solução para os problemas seria possível usando o ÆON. Durante a execução, foram recolhidos dados relativos à validade das soluções geradas, o tempo de execução e as métricas de *fitness* associadas a cada indivíduo.

Os resultados da *benchmark* demonstram que o ÆON sem tipos refinados gerou apenas programas bem tipados, embora nem sempre semanticamente válidos face às especificações do problema, soluções que levantavam erros em tempo de execução, sendo por isso consideradas programas inválidos na nossa avaliação. A representação baseada em árvores apresentou taxas superiores de programas válidos e convergência mais estável, beneficiando de uma correspondência direta entre a estrutura da árvore e as regras do sistema de tipos. Esta coerência sintática e semântica reduziu a produção de indivíduos inválidos e facilitou a exploração de regiões promissoras do espaço de procura. Em contraste, a evolução gramatical mostrou maior variabilidade e um potencial exploratório mais amplo, sendo particularmente competitiva em problemas com elevada diversidade sintática. Também observamos que problemas mais simples, que tenham até dois tipos primitivos tiveram mais programas validos, que problemas com mais de dois tipos primitivos e até mesmo com tipos compostos.

Para compreender melhor as diferenças de desempenho entre representações, foi realizada uma análise adicional correlacionando os tipos de dados utilizados em cada problema da *benchmark* com os resultados obtidos. Constatou-se que problemas dominados por operações numéricas beneficiam mais da representação baseada em árvores, devido à sua manipulação mais direta de literais e operadores aritméticos. Por outro lado, problemas com manipulação intensiva de sequências de caracteres mostraram resultados semelhantes ou ligeiramente superiores sob evolução gramatical, provavelmente devido à maior flexibilidade sintática dessa representação.

A análise qualitativa dos resultados do caso de estudo *Super Mario* demonstrou que o RTGP melhora bastante o espaço de procura, gerando programas sintática e semanticamente válidos. Neste cenário, a gramática dotada de tipos refinados, a nossa abordagem com RTGP, produziu exclusivamente indivíduos válidos, enquanto a gramática sem refinamentos gerou apenas candidatos inválidos. O resultado demonstra empiricamente que a utilização de tipos refinados elimina a geração de programas semanticamente incorretos relativamente às propriedades especificadas, conduzindo

a uma exploração mais orientada e eficiente. Além disso, verificou-se que as soluções válidas apresentaram uma convergência sistemática em direção à faixa ótima de *fitness*, evidenciando que estamos a guiar as soluções para programas com maior qualidade na resolução do nosso problema de síntese de programas.

De forma global, a avaliação quantitativa e qualitativa sustenta a hipótese central da dissertação: a programação genética com tipos refinados constitui um mecanismo eficaz para orientar e restringir a síntese de programas, promovendo a geração de soluções corretas e semanticamente coerentes. O RTGP demonstra que é possível unir rigor formal e procura estocástica, obtendo correção e um equilíbrio entre expressividade e eficiência.

Apesar dos resultados promissores, o trabalho reconhece várias limitações e oportunidades de evolução. A atual implementação destina-se a problemas de pequena e média escala, sendo necessário investigar a escalabilidade do sistema para programas de maior dimensão ou com estruturas de dados mais complexas. O desempenho do verificador de tipos e do motor evolutivo poderá ser otimizado, nomeadamente ao nível da gestão de memória e da paralelização do processo evolutivo. Outra vertente relevante é o alargamento da expressividade da linguagem ÆON, de modo a suportar funções de ordem superior e tipos dependentes. Estas melhorias tornariam o sistema aplicável a um conjunto mais vasto de problemas de síntese e permitiriam lidar com cenários reais de desenvolvimento de software.

Adicionalmente, seria útil explorar como a orientação por tipos refinados interage com estratégias de procura alternativas. A investigação de operadores evolutivos híbridos, que combinem análise estática com heurísticas de procura, poderá conduzir a uma síntese mais rápida e diversa. Outro caminho interessante consiste em avaliar o RTGP em domínios aplicados, como geração automática de testes, síntese de configurações em sistemas distribuídos ou otimização de políticas em software embebido. Estes contextos permitiriam avaliar a utilidade prática do método fora do ambiente académico e aferir o seu impacto em tarefas de engenharia de software realistas.

Em conclusão, esta dissertação demonstra que a programação genética com tipos refinados constitui uma abordagem capaz de produzir um enquadramento de síntese de programas simultaneamente expressivo e seguro. O RTGP reduz a geração de candidatos inválidos, acelera a convergência para soluções corretas e comprova ser possível integrar raciocínio formal num processo evolutivo sem sacrificar a flexibilidade da busca. Através da extensão da linguagem ÆON e da sua integração com o GENETICENGINE, este trabalho estabelece as bases para futuras investigações em síntese de programas orientada por tipos e por gramáticas. Aponta-se, assim, para um cenário em que a segurança e correção dos programas gerados, garantidas pelas restrições do sistema de tipos, coexistem com a diversidade proporcionada pelos mecanismos de geração gramatical.

Palavras-chave: Síntese de Programas, Programação Genética, Tipos Refinados, Programação Genética com Tipos Refinados

Abstract

This dissertation presents the design and evaluation of a framework for *Refinement-Typed Genetic Programming (RTGP)*. The framework combines refinement types with genetic programming, enabling the automatic generation of programs that are not only syntactically valid but also semantically consistent with type constraints. The work addresses a common limitation of search-based synthesis systems, where a significant portion of computational effort is spent generating invalid or incoherent candidates, leading to inefficient exploration of the search space. By incorporating refinement types, RTGP restricts the search space in a principled manner while preserving expressiveness.

The approach was implemented by extending the Æon declarative synthesis language, a programming language that integrates a built-in synthesizer for program synthesis problems, and by integrating it with the GeneticEngine framework. The resulting architecture introduces dynamic grammar generation, refinement validation, and mechanisms for extracting fitness information.

The experimental evaluation was conducted in two parts. First, 22 problems from the Program Synthesis Benchmark Suite 2 were used to assess the behavior of the base Æon synthesizer. The results show that simpler problems involving at most two primitive types produced a higher number of valid programs, whereas problems involving multiple primitive or composite types resulted in significantly fewer valid solutions. In this context, invalid programs correspond to candidates that, although syntactically well-formed and well-typed, produce runtime errors or fail to satisfy the intended semantic behavior.

RTGP itself was evaluated in a procedural content generation case study based on Super Mario. In this scenario, grammars with refinement types, corresponding to our RTGP implementation, produced exclusively valid individuals, whereas grammars without refinements generated only invalid candidates. Overall, the results demonstrate that grammars with refinement types effectively constrain the evolutionary search process, significantly increasing the proportion of valid programs while reducing semantically incorrect candidates.

Keywords: Program Synthesis, Genetic Programming, Refinement Types, Refinement-Typed Genetic Programming

Contents

List of Figures xv

List of Tables xvii

List of Listings xx

Acronyms xxi

1 Introduction 1

1.1 Motivation . . . 1

1.2 Objectives and Contributions . . . 2

1.3 Structure of the document . . . 3

2 Background 5

2.1 Program synthesis . . . 5

2.1.1 Overview of Program Synthesis . . . 5

2.1.2 Challenges in Program Synthesis . . . 6

2.2 Genetic programming . . . 7

2.3 Refinement Types . . . 9

2.4 The Æon Programming Language . . . 10

3 Related Work 11

3.1 Syntax-Guided Program Synthesis . . . 11

3.1.1 Program Synthesis via Divide and Conquer . . . 12

3.2 Polymorphic Refined Type Program Synthesis . . . 12

3.2.1 Genetic Programming with MetaHandlers . . . 13

3.2.2 Refinement-Typed Genetic Programming . . . 14

3.3 Refinement Type Checking as a Fitness Function . . . 14

4 Approach to Program Synthesis with Refinement-Typed Genetic Programming 17

4.1 Program Synthesis Problem . . . 17

4.2 Grammar Generation and Refinement Integration . . . 18

4.2.1 Integration with Genetic Programming . . . 20

5 Implementation of Program Synthesis with Refinement-Typed Genetic Programming 21
5.1 GeneticEngine Modifications . . . 21
5.2 Æon Language . . . 23
5.2.1 Language Overview . . . 23
5.2.2 Type System . . . 24
5.2.3 Architecture . . . 25
5.2.4 Context Builder . . . 27
5.2.5 Decorators . . . 29
5.2.6 Grammar Extraction . . . 30
5.2.7 Grammar Representation . . . 31
5.2.8 Synthesis . . . 38

6 Evaluation 41
6.1 Experimental Setup . . . 41
6.2 Part I: Program Synthesis Benchmark Suite 2 (PSB2) . . . 41
6.2.1 Benchmark Setup and Search Configuration . . . 41
6.2.2 Solution Validity . . . 42
6.2.3 Fitness Results . . . 43
6.2.4 Relation Between Exercise Types and Performance . . . 49
6.3 Part II: Super Mario Level Generation Case Study . . . 50
6.3.1 Setup and Grammar . . . 50
6.3.2 Validity and Fitness Results . . . 51
6.4 Summary of Findings . . . 53

7 Conclusion 55

8 Future Work 57

Bibliography 63

List of Figures

2.1 Flow of a Genetic Algorithm. 7

5.1 End-to-end synthesis pipeline, from an Æon specification file to solutions in Æon generated by GeneticEngine. 26
5.2 Compiler pipeline showing parsing, desugaring, type checking, hole extraction, grammar generation, and integration with the synthesis engine. Outlined in green are the contributions of this thesis, while the rest of the pipeline was inherited from the original Æon implementation. 27

6.1 Valid and Invalid Solutions Across PSB2 Exercises. 44
6.1 Valid and Invalid Solutions Across PSB2 Exercises (continued). 45
6.1 Valid and Invalid Solutions Across PSB2 Exercises (continued). 46
6.2 Aggregated fitness values across PSB2 exercises for both representations. Each coloured point corresponds to the aggregated fitness obtained in every valid solution, while the bigger black point represents the mean fitness for each representation. 47
6.2 Aggregated fitness values across PSB2 exercises for both representations (continued). 48
6.2 Aggregated fitness values across PSB2 exercises for both representations (continued). 49
6.3 Valid and Invalid Solutions for Super Mario 52
6.4 Aggregated fitness values for the *Super Mario* case study. Each coloured point corresponds to the aggregated fitness obtained in every valid solution, while the bigger black point represents the mean fitness across all valid solutions. 52

List of Tables

5.1 Supported decorators in ÆON, grouped by purpose. 29
5.2 Search Configuration Parameters in GENETICENGINE and default values. 39

6.1 Search Configuration Parameters in GENETICENGINE for PSB2 runs. 42
6.2 Types and Data Structures Used per Exercise. 43
6.3 Search Configuration Parameters in GENETICENGINE for *Super Mario* runs. 50

List of Listings

2.1 Type System Error. 8
2.2 STGP Possible Solution. 8
2.3 Simple Grammar Example. 9
2.4 Absolute difference function in ÆON with a non-negative return hard constraint. . 10
3.1 Simple SyGus example. 11
3.2 Synthesized solution returned by the SyGus solver. 12
4.1 Example of a simple context-free grammar. 19
4.2 Example of a simple context-free grammar with refinements. 20
5.1 Factorial function written in ÆONCORE syntax. 23
5.2 Factorial function written in ÆON syntax. 23
5.3 Example with hard and soft constraints written in ÆON. 24
5.4 A safe pow function defined in ÆON with refinement types. 24
5.5 Simple Specification File Example. 28
5.6 Context Builder behavior on the knapsack specification. 28
5.7 Transformed Code after decorator resolution and conversion to Core syntax. . . . 28
5.8 Example usage of multiple decorators in ÆON. 30
5.9 ÆON example of a simple synthesis task. 31
5.10 Python Abstract classes representing the non-terminal nodes of a grammar that are generated from the ÆON data types. 32
5.11 Type defined by the user in ÆON. 32
5.12 Python Abstract class representing the non-terminal nodes of a grammar that are generated from a ÆON data type defined by the user in Listing 5.11. 32
5.13 Function in ÆON that calculates the square of an integer. 32
5.14 Python Abstract class representing the non-terminal nodes of a grammar that are generated from a ÆON data type defined by the user in Listing 5.13. 32
5.15 Terminal node representing a literal integer in ÆON. 33
5.16 Predefined Integer and Float operations in ÆON. 33
5.17 Application grammar node for the user-defined function, `id_function`, in Listing 5.9. 34
5.18 Variable grammar node for the user-defined function, `id_function`, in Listing 5.9. 34
5.19 Application and Variable grammar nodes for the ÆON predefined + arithmetic function. 34

5.20 Generated control Flow Grammar nodes for the ÆON predefined types. 35
5.21 Grammar created from all the grammar nodes generated and the starting node. . . 36
5.22 A simple variable with a refined type in ÆON. 37
5.23 Grammar nodes generated for a refined variable `i` with disjunctive refinement ranges. 38
6.1 Excerpt of the grammar extracted from ÆON used for Super Mario level generation. 51

Acronyms

ANF A-normal Form.

AST Abstract Syntax Tree.

BNF Backus-Naur Form.

CDGP Counterexample-Driven Genetic Programming.

DNF Disjunctive Normal Form.

EBNF Extended Backus-Naur Form.

GGGP Grammar-Guided Genetic Programming.

GP Genetic Programming.

LLM Large Language Model.

NSGA-II Non-dominated Sorting Genetic Algorithm II.

PSB2 Program Synthesis Benchmark Suite 2.

RTGP Refinement-Typed Genetic Programming.

SMT Satisfiability Modulo Theories.

STGP Strongly-Typed Genetic Programming.

Chapter 1

Introduction

This chapter introduces the problem addressed in this thesis and motivates the use of Refinement-Typed Genetic Programming (RTGP) for program synthesis. It provides an overview of the goals and contributions, and concludes with a roadmap for the remainder of the document.

1.1 Motivation

Program synthesis [19] is a research field that aims to automatically generate computer programs from high-level formal specifications. This approach has the potential to revolutionize software development by significantly reducing the time, effort, and cost required to produce reliable and efficient software. Despite its promise, program synthesis remains a challenging problem due to the vast search space of possible programs.

Traditional synthesis techniques, such as deductive and symbolic approaches, often rely on precise logical constraints or exhaustive enumerations of candidate programs. While effective in constrained domains, these methods tend to suffer from scalability issues when applied to more complex, real-world problems. Moreover, they typically require specifications to be written in highly formal languages, which can limit both accessibility and expressiveness.

Genetic Programming (GP) is an evolutionary computation technique that represents programs as individuals in a population and iteratively improves them through selection and variation operators inspired by natural selection principles [22]. Rather than exhaustively exploring the entire search space, GP performs a stochastic search by applying mutation, crossover, and selection across successive generations to evolve increasingly fit candidate programs. This makes GP particularly suitable for under-specified or partially defined synthesis tasks, where the search space is too large or poorly structured for symbolic methods.

However, this flexibility comes at a cost: GP may generate numerous syntactically valid but semantically incorrect or inefficient programs. For example, when synthesizing a function that computes the absolute value of an integer, a naive GP system might generate `if x > 0 then x else -x`, but also redundant or convoluted variants such as `if x < 0 then x * -1 else x - 0`, which, though correct, are harder to interpret. Without additional guidance, the search process may stagnate or converge on lower quality solutions.

To overcome these limitations, several hybrid techniques have been proposed that combine evolutionary algorithms with formal verification methods. One such approach uses a refinement type checker as a fitness function within Genetic Programming [45], ensuring that candidate programs satisfy semantic constraints throughout the evolutionary process. Similarly, Counterexample-Driven Genetic Programming (CDGP) [8] integrates SMT-based verification into the synthesis loop, using counterexamples to eliminate incorrect candidates and guide the search. These techniques demonstrate that combining formal methods with evolutionary search can significantly improve both correctness and efficiency, particularly in complex synthesis tasks.

Building on prior work on Refinement-Typed Genetic Programming (RTGP) [41], the novelty of this thesis lies in a practical end-to-end realization of the approach. Concretely, we extend and redesign the ÆON [1] programming language and synthesizer to support refinement-based specifications with both hard and soft constraints, and we implement a synthesis pipeline that integrates ÆON with a Strongly Typed, Grammar-Guided Genetic Programming engine. This integration uses refinement types to both restrict the candidate space and to validate synthesized programs, while soft constraints provide optimization objectives for guiding search. Finally, we evaluate the resulting system in two parts: first, we use 22 problems from the Program Synthesis Benchmark Suite to assess the behavior of the base ÆON synthesizer across problem types; second, we evaluate the impact of refinement-typed grammars (our RTGP implementation) in a procedural content generation case study based on Super Mario by comparing refined and non-refined grammars.

1.2 Objectives and Contributions

The objective of this thesis is to redesign and extend ÆON and its synthesizer, and to evaluate the resulting synthesis pipeline, with the goal of generating programs more efficiently and accurately by narrowing the search space through more precise (refinement-based) constraints. In particular, we extend ÆON to express both hard and soft refinement constraints, integrate the language with a strongly typed, grammar-guided GP engine, and empirically assess the resulting system to understand how refinement information affects validity and synthesis outcomes.

This objective is realized through the following contributions:

- **Extension to ÆON:** We extend the ÆON programming language to support the specification of both hard and soft constraints using refinement types. The updated syntax and type system enable developers to express more nuanced requirements, such as functional correctness properties, performance preferences, and partial behavioural constraints—thereby improving both the expressiveness and usability of the language.

- **Synthesis framework integration:** We design and implement a synthesis engine that integrates ÆON with a grammar-guided, strongly typed Genetic Programming framework. This engine supports both single- and multi-objective optimisation.

[1] ÆON is available at `https://github.com/alcides/aeon`

- **Empirical evaluation:** We conduct an experimental evaluation in two parts: (i) a study over 22 problems from the Program Synthesis Benchmark Suite 2 (PSB2) to characterize the behavior of the base ÆON synthesizer across problem types, and (ii) a procedural content generation case study based on *Super Mario*, where we evaluate our RTGP implementation by comparing refined and non-refined grammars to assess the impact of refinement type constraints on search space reduction and solution validity.

1.3 Structure of the document

The remainder of this document is organized as follows:

- **Chapter 2 - Background:** Introduces the foundational concepts for this thesis: program synthesis, genetic programming, refinement types, and a brief overview of ÆON as the programming language used in this work. This chapter establishes the terminology and technical context required for the subsequent chapters.

- **Chapter 3 - Related Work:** Reviews prior research relevant to this dissertation, including syntax-guided synthesis, divide-and-conquer strategies, refinement-typed synthesis, the use of Meta-Handlers in genetic programming, and approaches that integrate type checking or formal verification into the synthesis loop. The chapter positions RTGP within this context and highlights open gaps.

- **Chapter 4 - Approach to Program Synthesis with RTGP:** Presents the methodological core of the work: how a synthesis problem is specified, how grammars are generated and refined from program context, and how these elements integrate with a genetic search procedure. This chapter defines the abstractions and interfaces used throughout the system.

- **Chapter 5 - Implementation of Program Synthesis with RTGP:** Details the concrete system: the redesign and extension of the ÆON language (syntax, type system with refinements, architecture, context builder, decorators, and grammar extraction) and the integration with a grammar-guided, strongly-typed genetic programming engine. It also documents framework level improvements and the overall synthesis pipeline.

- **Chapter 6 - Evaluation:** Describes the experimental setup and reports results from a two-part evaluation. First, it uses 22 problems from the Program Synthesis Benchmark Suite 2 to characterize the behavior of the base ÆON synthesizer across problem types, analyzing solution validity and aggregated fitness, and relating performance differences to the underlying data types and structures. Second, it evaluates our RTGP implementation in a *Super Mario* level generation case study by comparing our RTGP configuration (refined grammars) against a baseline using non-refined grammars, highlighting how refinements constrain the search space and increase the proportion of valid individuals.

- **Chapter 7 - Conclusion:** Summarizes the main contributions and findings of this thesis, reflects on the limitations of the proposed approach, and discusses the broader implications of the work.
- **Chapter 8 - Future Work:** Outlines promising directions for extending and improving the system, including support for dependent types, integration with LLM tools, and other open research challenges identified during the course of this work.

Chapter 2

Background

This chapter summarizes the foundational concepts required for the rest of the thesis. We introduce program synthesis and its main paradigms, review genetic programming as a search-based synthesis technique, and present refinement types as a mechanism for expressing and enforcing semantic constraints.

2.1 Program synthesis

This section provides a brief introduction to program synthesis and establishes the terminology used throughout the dissertation. We first present the problem and common specification styles, and then discuss the main challenges that synthesis systems face.

2.1.1 Overview of Program Synthesis

Program synthesis [19] is the task of automatically generating programs from high-level user intent. The intent may be expressed in various forms, such as logical specifications, input-output examples, partial programs, or even natural language. This paradigm has been successfully applied to several domains in software engineering, including compiler optimisations [42], automated bug fixing [30], and code refactoring [40].

Despite its potential, program synthesis remains a challenging problem. It requires not only interpreting the user's intent correctly but also exploring a vast space of potential programs to find one that is correct, efficient, and interpretable. Ambiguity in user intent and the combinatorial explosion of candidate programs are major obstacles. Nonetheless, the field has seen rapid advances and remains an active area of research in both academia and industry.

Program synthesis typically follows two main approaches: deductive and inductive synthesis. Deductive synthesis [27] involves specifying the desired behaviour formally, often through logical constraints or detailed specifications. The synthesis process employs logical inference and reasoning to systematically generate programs guaranteed to satisfy these explicit constraints. This method ensures correctness by construction, making it particularly suited to domains requiring strong formal guarantees. On the other hand, inductive synthesis [44] operates by learning generalizable solutions from specific input-output examples [12, 32]. Rather than requiring a fully for-

malized specification upfront, inductive approaches infer program logic from observed instances, making them particularly useful for scenarios where explicit specifications are difficult or impractical to define, or where the problem domain is highly complex and ambiguous.

More recently, additional paradigms have emerged. Program synthesis [5] with large language models (LLMs) trained on code and text to translate user-written descriptions into executable programs. This makes synthesis accessible to non-experts but introduces challenges in disambiguation and correctness. Another example is search-based synthesis [17], which includes techniques such as genetic programming (GP) and treats synthesis as a search or optimization problem over a space of candidate programs. These approaches use fitness functions and heuristic strategies to evolve or iteratively refine programs based on correctness and performance criteria.

In summary, modern program synthesis spans a wide spectrum of approaches, each balancing correctness, usability, and scalability depending on the target application and the form of user intent.

2.1.2 Challenges in Program Synthesis

When working with program synthesis, there are three core challenges that need to be addressed. These challenges, which we refer to as the *Three Pillars of Machine Programming* [18], are crucial in developing effective synthesis systems.

- **Intention**: The first challenge is understanding the user's goals, what they truly want from the synthesis process. Program synthesis requires both semantic and syntactic constraints, but how they are set has a substantial impact on the design and performance of the synthesis system. While input-output examples have become popular as a specification tool, they are not necessarily adequate for all problems. Dealing with under-specification is an important part of the intention challenge. When there are numerous solutions that match the user's needs, how does the system know which one the user wants? One possible solution is to accept the uncertainty and return any legitimate program, allowing the user to change the result. However, in practice, choosing the best answer can mean the difference between a tool that is truly useful and one that fails to meet expectations.

- **Invention**: Once the user's intent is understood, the next challenge is to generate a program that fulfills the requirements. This is perhaps the most difficult aspect of synthesis, as it often requires inventing new algorithms or approaches to solve the problem at hand. Various strategies have been developed to address this complexity, including enumerative search, which explores programs in a size-bounded order; constraint-based synthesis, which encodes the problem as a SAT or SMT query; stochastic methods, such as genetic programming or MCMC sampling, that use heuristics to explore large, complex spaces; neural-guided synthesis, where learned models prioritize promising regions of the search space; and hybrid approaches, which combine symbolic reasoning with statistical guidance. Much of the research effort in synthesis focuses on improving these search strategies to balance completeness, scalability, and performance.

- **Adaptation**: Traditionally, program synthesis has been seen as the process of creating entirely new algorithms from scratch. However, much of software development involves working with existing code, either improving, debugging, or optimizing it. This pillar focuses on the application of synthesis techniques to tasks other than the creation of new software, such as code maintenance and enhancement. In practice, program synthesis can play a significant role in helping the ongoing software development processes.

2.2 Genetic programming

Genetic Programming (GP) [34] is a machine learning technique used to automatically create computer programs or models through an evolutionary process inspired by the theory of natural selection and genetics. This approach has been successfully applied in a wide range of domains [1], including symbolic regression [31], program synthesis [33], automated program repair [51], software optimisation [25], financial modeling [4], and bioinformatics [50]. These applications demonstrate GP's versatility in solving complex real-world problems where traditional methods may be limited. The process involves generating a population of random computer programs, called individuals, typically represented as trees, and then evaluating their fitness against a given problem domain. The fittest programs are then selected for reproduction via a selection algorithm, and through methods like crossover and mutation, new generations of programs are created. This process continues until a stopping criterion is met, and the best program found in the process is considered the output of the genetic programming algorithm. Figure 2.1 is a simple and illustrative example of how the flow of a genetic algorithm works.

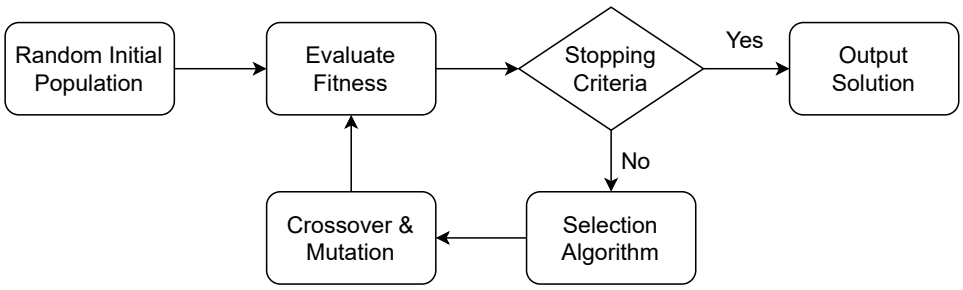


Figure 2.1: Flow of a Genetic Algorithm.

Strongly-Typed Genetic Programming. Strongly-Typed Genetic Programming (STGP) [29] is an improved approach to genetic programming. This technique combines programming techniques with the capability of genetic algorithms to produce high-quality programs that meet specific criteria. Within STGP, program development takes place by ensuring compliance with predetermined rules relating to variables and operations using an established type system. In contrast, standard GP does not enforce such strict type compliance.

Implementing a type system structure offers numerous advantages, in particular, enhanced program reliability and correctness.

An advantage of STGP is its capacity to generate semantically precise programs; this means that not only is their syntactic correctness ensured, but also they are valid for the type system. The restriction of data types when generating a program through STGP is an effective tool for dealing with certain bugs and errors often encountered by standard GP.

Listing 2.1 presents a example of an error that could be faced in a program generated by standard GP. A function called *add_numbers* takes two inputs, *a* and *b*, and returns their sum. However, in line 4, we can witness that the program is trying to sum an integer to a string which would raise a TypeError while trying to compile that program. With STGP, this type of bug can easily be avoided by restricting the data types of *a* and *b* input parameters to be only integers, as we can notice in Listing 2.2, preventing any type errors from occurring during compilation time. Listing 2.2 presents a semantically precise program that is both syntactically correct and valid for the type system, which can be an example of an STGP generated solution.

```
1 def add_numbers(a, b):
2     return a + b
3
4 result = add_numbers(2, '3') # This will result in a TypeError
```

Listing 2.1: Type System Error.

```
1 def add_integers(a : int, b : int) -> int:
2     return a + b
3
4 result : int = add_integers(2, 3)
```

Listing 2.2: STGP Possible Solution.

Grammar-Guided Genetic Programming. Grammar-Guided Genetic Programming (GGGP) [28] is another method for constraining the search space in genetic programming using grammar. A Grammar consists of a collection of rules defined by a set of terminal symbols, a set of non-terminal symbols and a starting symbol that describes all valid expressions that can be formed within a given context. We can look at the grammar example in Listing 2.3, which is written in Backus-Normal-Form (BNF) [24], it describes a simple arithmetic expression. That grammar consists of five production rules defined by five non-terminal symbols (<*expr*>, <*term*>, <*digit*>, <*op*>, and *S*), and 17 terminal symbols (*0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /, (,)*) and the starting symbol is *S*. In the context of GGGP, the grammar defines the set of syntactically valid programs that the genetic programming algorithm can evolve by randomly expanding each non-terminal grammar symbol until no more expansions are possible. Consequently, all individuals generated by the genetic programming algorithm will be valid concerning the grammar, ensuring that the programs generated conform to the desired syntax. This process allows domain-specific constraints to be encoded in the synthesis process.

By encoding hard constraints in the grammar, the search space can be reduced and the performance of genetic programming can be improved.

```
S       ::= <expr>
<expr>  ::= <term> | <expr> <op> <term>
<term>  ::= <num> | (<expr>)
<num>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<op>    ::= + | - | * | /
```

Listing 2.3: Simple Grammar Example.

The ideas introduced in this section establish the foundation for the synthesis framework proposed later in this thesis. A more detailed discussion of related approaches, such as Meta-Handlers, and Refinement-Typed Genetic Programming (RTGP), is presented in Chapter 3.

2.3 Refinement Types

Refinement types are built upon standard type systems by allowing logical predicates to be attached to types, enabling more precise reasoning about program values. Instead of treating a variable as simply an `int`, for instance, a refinement type can specify that it must be a non-negative integer. This logical predicate refines the type, restricting its admissible values to those that satisfy the condition.

The main goal of refinement types is to improve static verification. By embedding logic into the type system, many runtime errors, such as out-of-bounds accesses, division by zero, or illegal states, can be caught at compile time. A refinement type system verifies not only that values match their base types, but also that the associated predicates hold. As a result, programs become more robust and self-documenting, with types explicitly describing both structure and expected behaviour.

Refinement Types for ML [15] introduced a refinement type system for the ML programming language, showing how logical predicates can be integrated into a statically typed language while preserving decidable type checking. The ML's type system is extended with logical annotations and subtyping support, making it possible to encode and verify complex correctness properties within types.

Refinement type systems bring several key benefits. They offer improved expressiveness, allowing programmers to encode rich invariants directly in type signatures, which constrains valid program behaviours and reduces ambiguity. They also enable static verification through automatic checking of predicates using theorem provers or SMT solvers, helping to ensure program correctness without execution. Moreover, these systems typically support subtyping, where a type with stronger (more restrictive) predicates is considered a subtype of a more general one, and incorporate inference mechanisms that extend traditional type inference with refinement logic, maintaining a balance between expressiveness and decidability.

Refinement types greatly affect software reliability and security. They effectively prevent many programming errors in practice, by ensuring that invariants like buffer sizes, index ranges and data structure properties hold. Furthermore, as types incorporate both the structural and behavioural aspects of program entities, they act as a form of formal documentation to clarify use intentions and constraints for various components in the system.

Several practical refinement type systems have emerged across languages and domains. Examples include *Liquid Haskell* [47] for Haskell, *Liquid Java* [16] for Java, *Flux* [26] for Rust.

2.4 The ÆON Programming Language

The work in this thesis builds on ÆON [41], a refinement-typed language designed to express synthesis problems along with the constraints that guide program generation. ÆON uses refinement types as its primary mechanism for specifying semantic properties of values, which enables writing specifications that are both concise and precise.

The initial version of ÆON provided a solid foundation but also exposed practical limitations for more realistic synthesis workflows. In particular, it lacked support for soft constraints and multi-objective optimisation, restricting the ability to express optimisation preferences beyond strict correctness. Its specification syntax was also limited in expressiveness, making it cumbersome to encode richer behavioural properties and to support composite data types such as lists. In addition, the original synthesis engine relied on basic genetic programming techniques, which struggled to scale to more complex problems due to limited mechanisms for effective search pruning or feedback-guided exploration.

This thesis builds upon the original ÆON language, redesigning and extending it to better support modern synthesis workflows. The improved ÆON introduces support for both hard and soft constraints, a more expressive and flexible syntax for specifications, and a refined type system to capture complex correctness properties. Furthermore, it integrates with a grammar-guided, strongly-typed genetic programming engine, enabling both single- and multi-objective optimisation tasks.

A core feature of ÆON is its ability to specify hard constraints using refinement types. For example, consider the following function that computes the absolute difference between two integers. Its return type is refined to ensure the result is always non-negative:

```
1 def abs_diff: (x: Int) -> (y: Int) -> {z: Int | z >= 0} {
2     if x > y then x - y else y - x;
3 }
```

Listing 2.4: Absolute difference function in ÆON with a non-negative return hard constraint.

Chapters 5 and 6 build on these foundations to present the language extensions and synthesis pipeline, and to report the experimental evaluation.

Chapter 3

Related Work

This chapter presents alternative approaches to program synthesis and prior work combining genetic programming with refinement types.

3.1 Syntax-Guided Program Synthesis

SyGuS [3] is a program synthesis approach that uses logical specification and syntactic templates to limit the space of valid implementations. In practice, the user provides a program with some holes, and a logical specification that the full program needs to implement. A typical SyGuS encoding of this problem will search for expressions that satisfy both a semantic constraint specified by a logical formula and a syntactic constraint specified by a context-free grammar. To check the satisfiability of the semantic constraint written in the specification, SyGuS uses SMT [6] (Satisfiability Modulo Theories) solvers. SMT Solvers are capable of handling complex logical constraints, allowing them to efficiently solve many practical instances of SyGuS problems. The SyGuS approach has numerous practical applications in several domains such as program optimization, finding bugs, software verification and program completion. A simple example is synthesizing a function $f(x) = x + x$ from a small grammar and input-output constraints, as shown in Listing 3.1.

```
(set-logic LIA)

(synth-fun double ((x Int)) Int
  ((Start Int (x 0 1
               (+ Start Start)
               (- Start Start)
               (* Start Start)))))

(declare-var x Int)

(constraint (= (double 0) 0))
(constraint (= (double 1) 2))
(constraint (= (double 2) 4))
(constraint (= (double 3) 6))

(check-synth)
```

Listing 3.1: Simple SyGus example.

In this example, `synth-fun` declares the function to be synthesized, named `double`, which takes an integer and returns an integer. The grammar restricts the search space to expressions formed using constants, the variable `x`, and basic arithmetic operations. The `constraint` statements define the required input-output behaviour that the synthesized function must satisfy. Finally, the `check-synth` command invokes the synthesis procedure.

A SyGuS solver would return a correct implementation such as Listing 3.2:

```
(define-fun double ((x Int)) Int
  (+ x x))
```

Listing 3.2: Synthesized solution returned by the SyGus solver.

This demonstrates how SyGuS can automatically generate a correct implementation that respects both syntactic and semantic constraints.

3.1.1 Program Synthesis via Divide and Conquer

An enumerative approach that yields an effective performance for solving SyGuS instances involves the generation of expressions derived from syntax rules while simultaneously trimming them down to only the expressions relevant to the current context. This approach has shown promising results on small to medium-sized benchmarks. However, it has limitations when it comes to larger solutions. This is because it requires searching a large space of possible expressions, which becomes infeasible for more extensive solutions. To counter this scalability issue a tool called EUSOLVER [2] was developed. It is a prototype implementation of a divide-and-conquer enumerative algorithm for solving SyGuS problems. This technique is based on the idea that, although the overall solution expression to a synthesis problem may be huge and complex, the key individual components of the solution are frequently small and simple. An individual component in this context could be either a *term* or a *predicate*. After the divide-and-conquer algorithm enumerates terms and predicates separately, it attempts to combine them into a conditional expression. This combination is achieved using decision tree learning techniques [7, 36]. By doing that, the search space is reduced, and the algorithm's performance is improved. In sum, the divide and conquer strategy recursively divides the problem into smaller sub-problems until a solution is reached. To improve solution quality, the developers of the EUSOLVER tool extended the divide and conquer algorithm to an *anytime* algorithm, a continuous search algorithm to find further compact solutions beyond the first produced one.

3.2 Polymorphic Refined Type Program Synthesis

Synquid [35] is another program synthesis framework for verifying and generating correct software programs based on a certain specification. Synquid differs from SyGuS in that it uses refined polymorphic types to restrict the search space of possible solutions. The use of refined polymorphic types allows Synquid to perform more precise type inference, which results in guiding the search towards more efficient and correct programs. This framework combines ideas from refinement type

checking, liquid type inference [37] and program synthesis altogether. One of the key advantages of Synquid is its ability to verify and synthesize more complex programs thanks to automatic refinement inference. Nonetheless, to synthesize complete programs, this framework must complete accurate corrections and limit the components, functions, and variables that functions can use.

3.2.1 Genetic Programming with MetaHandlers

GeneticEngine [11] is an open-source genetic programming framework[1] that combines STGP and GGGP to solve single- and multi-objective problems. It allows users to define a fitness function that evaluates the performance of candidate solutions for a given problem while providing domain information about the solution structure through datatypes used to represent grammars. This framework introduces the concept of Meta-Handlers, which are type annotations that allow users to encode expressive constraints on the grammar in the GGGP framework. These annotations provide a more formal and practical approach than BNF and its extensions and allow users to override the tree-generation process. As such, Meta-Handlers serve both as refinement types and as a mechanism for enforcing structural constraints during program generation. Meta-Handlers can represent liquid types as refinement types that act as additional constraints during the initialization phase of GP. They also allow more precise type information to be specified in genetic programming, similarly to how liquid types are used in traditional programming languages.

Within GeneticEngine, candidate solutions (individuals) can be represented using different encodings that determine how programs are generated, mutated, and recombined during evolution. In this work, we consider two representations: *Tree-Based Genetic Programming* and *Grammatical Evolution*. In the Tree-Based representation [49], individuals are directly encoded as abstract syntax trees (ASTs) that follow the structure defined by the grammar. Each node corresponds to a production rule or constructor, while its children represent the required arguments. Because the tree structure is derived from the grammar's typed datatypes, type correctness and structural constraints can be enforced during generation and variation. This allows genetic operators such as crossover and mutation to operate on subtrees while preserving grammatical validity. In contrast, Grammatical Evolution [38] represents individuals as linear genomes composed of integer sequences. These genomes are mapped to programs by selecting grammar productions according to the integer values in the sequence. The mapping process translates the linear genotype into a tree-like phenotype that conforms to the grammar. While this representation provides greater syntactic flexibility and can explore a wider range of structures, it may also generate invalid or semantically inconsistent candidates when implicit constraints are violated during mapping.

Both representations explore the same grammar-defined search space but differ in how candidate solutions are encoded and manipulated during evolution. The Tree-Based approach maintains a direct correspondence between genotype and phenotype, whereas Grammatical Evolution relies on an indirect mapping from a linear genome to a program structure.

[1] GeneticEngine is available at `https://github.com/alcides/GeneticEngine`

3.2.2 Refinement-Typed Genetic Programming

Refinement-Typed Genetic Programming (RTGP) [14] is a genetic programming paradigm that tightly integrates a refinement type system into the evolutionary search process. It can be seen as a hybrid of STGP and GGGP, featuring an advanced type system with polymorphism, dependent types, and refinement types to embed domain-specific knowledge directly into the representation of programs.

In summary, refinement types refine the traditional type system by merging it with logical assertions about program behaviour, as seen in more detail in section 2.3. This dual approach of specification and verification enables the detection of bugs at compile time and encourages a deeper understanding of program semantics, thereby contributing to the development of robust software systems.

The key idea is to use refinement types, types annotated with logical predicates, to both constrain the search space of programs and to encode the desired behavioural specifications (fitness criteria) of the target program. In this setting, logical predicates act as hard constraints, while behavioural specifications act as soft constraints used to guide the search. This approach is argued to be more expressive than previous grammar or type-constrained GP techniques, since complex semantic constraints that were hard to enforce with context-free grammars or basic type systems can now be encoded within the type system itself.

In practical terms, RTGP provides a unified way to specify a GP problem: the representation, the search space restrictions, the fitness evaluation, and the execution context are all expressed in one specification language, Æon, via refined types and functions.

This yields a more user friendly interface for problem description, while also tackling two major challenges in program evolution, the exponentially large search space and the lack of a continuous fitness gradient, within a single framework.

3.3 Refinement Type Checking as a Fitness Function

Tseng et al. [45] propose to use GP to combine existing functions to build more complex, high-level functions. This model integrates refinement type checking, a formal verification method, with a genetic programming search technique. Breaking down programs into smaller components or chunks reduces the search space required for synthesizing solutions and enables solving larger synthesis problems effectively. Moreover, using this model, users can take advantage of built-in functions or external libraries that provide complex logic at no cost. The input-output types of these individual small scale software units, called “program pieces”, are specified by refining them beforehand to deliver intended behaviour when composed together like puzzles. To evaluate fitness results obtained from Refinement Type Checking error frequency count produces an innovative fitness function, used during GP evaluation, resulting in selecting correct Programs without any typing errors being corrected due to their behaviours being predicted through assurance based on refinement types.

A drawback of this method is that it relies on a standard GP algorithm, which may lead to larger search spaces. Although composing "program pieces" helps reduce the search space, relying on standard GP techniques may reintroduce scalability issues, as these do not inherently guide the search with semantic constraints. Despite these concerns, it is clear that introducing both genetic programming and formal verification tactics through this proposed model indicates a promising pathway towards more efficient and scalable program synthesis.

Chapter 4

Approach to Program Synthesis with Refinement-Typed Genetic Programming

This chapter presents the approach taken to perform program synthesis using Refinement-Typed Genetic Programming (RTGP). The methodology leverages both hard and soft constraints to guide the synthesis process, ensuring that the generated programs are not only semantically correct but also optimised according to user-defined objectives.

4.1 Program Synthesis Problem

Program synthesis involves automatically generating a program that satisfies a given specification. The specification may be expressed as input-output examples, formal assertions, or even natural language descriptions. In this section, we describe the formalism used to define program synthesis problems with Refinement-Typed Genetic Programming described in Section 3.2.2.

Components of a Program Synthesis Problem

To define a program synthesis problem with RTGP, we consider the following components:

- **Hard Refinements**: These are constraints that must be satisfied by any valid synthesized program. These are encoded as refinement types in the function signature and serve as strict correctness conditions.
- **Soft Refinements**: These are optional constraints that guide the search process toward more desirable solutions without enforcing strict correctness requirements. These may include preferences for certain programming constructs, performance optimisations, or other user-defined criteria.
- **Hole Information**: Metadata about incomplete parts of the program (referred to as"holes") that need to be synthesized. This includes the type, context and any relevant constraints. Hole information is used to construct the grammar from which candidate solutions are derived.

Problem Specification Process

Defining and solving a program synthesis problem with RTGP begins with the user providing a specification that outlines the problem to be solved. From this, the system performs context analysis by extracting details such as the types and lexical context of incomplete code segments. These details are then used to construct the grammar components that define the space of valid candidate programs. Once the grammar is in place, a fitness function is built, primarily based on soft constraints, which evaluates the quality of generated programs and may include additional checks such as syntax tree depth or other heuristics.

4.2 Grammar Generation and Refinement Integration

In this section, we present an approach for extracting and dynamically generating grammars to support program synthesis tasks. By leveraging context information and refinement types, this method optimizes the synthesis process by reducing the synthesis search space, ensuring that the generated programs are both efficient and semantically relevant.

Grammar Extraction

Grammar extraction begins by analyzing the program's incomplete components, referred to as holes, to identify the holes type and context. This information is then used to generate the grammar rules from which valid program candidates are derived. Grammar extraction is a crucial step in narrowing the synthesis search space to constructs that are relevant to the user's current codebase. The following steps outline the process:

- **Context Analysis:** Traverse the user's codebase to identify relevant constructs such as variables, data types, and function definitions. This ensures that the extracted grammar includes only elements that are in scope for the synthesis process.
- **Hole Identification:** Identify holes in the program that require synthesis. This involves three steps: **Location Identification**, which consists of traversing the program's Abstract Syntax Tree (AST) to locate the holes; **Context Collection**, where the system gathers information about the lexical scope of each hole, including visible variable bindings, type declarations, and function definitions; and **Type Inference**, which infers the expected types of each hole to provide a blueprint for the constructs that will comprise synthesized solutions.

Dynamic Grammar Generation

After extracting the necessary context and typing information, a grammar tailored to the local scope and the expected hole type is dynamically generated for each hole. The process begins with **Contextual Grammar Construction**, in which grammar rules are built from the previously extracted context and types. These rules are adapted on a per-hole basis to reflect the constructs available in scope.

The generated grammar consists of two types of nodes. **Non-Terminal Nodes** represent abstract syntactic categories that can be decomposed into sequences of non-terminal and terminal nodes. These typically include data types, function signatures, and expressions, serving as the structural skeleton of permissible expressions. **Terminal Nodes**, by contrast, represent concrete constructs such as variable names, literals, and function calls. They correspond to the actual values used in the generated code and form the leaves of the syntax tree.

Simple Context-Free Grammar Example

Let us consider the following simple context-free grammar that generates arithmetic expressions. The non-terminals define high-level structures like expressions and terms, while the terminals specify variables and numeric values. This grammar enables the synthesis of valid arithmetic expressions, such as $X + 3$ or $Y * 7$, while ensuring syntactic correctness.

```
Expr    ::= Term Op Term | Term
Op      ::= + | - | * | /
Term    ::= Var | Number
Var     ::= X | Y
Number  ::= [0-9]+
```

Listing 4.1: Example of a simple context-free grammar.

Integration of Refinement Types

Refinement types allow developers to specify logical predicates over types, constraining the permissible values of function parameters and return values. By integrating refinement types into the grammar generation process, we can ensure that the synthesized programs meet the specified requirements. The following steps outline the integration of refinement types into the grammar generation process:

- **Refinement Type Extraction:** Parse the program to locate refinement annotations (e.g., `{x: Int |x > 0}`), which specify logical constraints on values of parameters and return types.
- **Translation to Constraints:** Convert the extracted refinement types into a symbolic representation (e.g., Sympy expressions). Simplify these expressions and convert them into a canonical form, such as disjunctive normal form (DNF).
- **Constraint Handling:** Extract bounded intervals from the DNF representation. These bounds are then encoded as metadata (e.g., Meta-Handlers) and used to filter or annotate grammar productions. This allows the synthesizer to generate only values that satisfy the refinement constraints.

Context-Free Grammar with Refinements Example

As shown in Listing 4.2, we extend the grammar in Listing 4.1 to incorporate refinement constraints. For instance, we might restrict the values that the Number can have to specific ranges. The following

grammar guides the synthesis process to produce only expressions that respect these constraints.

```
Expr    ::= Term Op Term | Term
Op      ::= + | - | * | /
Term    ::= Var | Number
Var     ::= X | Y
Number  ::= [0 .. 10] | [20 .. 30]
```

Listing 4.2: Example of a simple context-free grammar with refinements.

4.2.1 Integration with Genetic Programming

Upon analysis of a program specification, the synthesis pipeline generates three core components required for integration with a genetic programming (GP) engine: a grammar, a target type, and a fitness function. The grammar is generated dynamically according to the context of synthesis, integrating available variables, functions, operators, and logical constraints defined by refinement types. The target type, which aligns with the type of the program fragment to be synthesized, functions as the root symbol for grammar expansion. The fitness function represents the optimization objectives, incorporating soft constraints or evaluation metrics, and assigns scores to candidate programs.

The components are provided to the GP engine to direct the search process. The initial population is generated by expanding the non-terminal symbols of the grammar into complete program trees, thereby ensuring that all individuals maintain syntactic and semantic validity. Evolution occurs via genetic operators that are both grammar-aware and type-safe, including mutation and crossover, which maintain the structural and type constraints of the grammar. Fitness evaluation includes both strict constraints, such as refinement checks, and flexible objectives, enabling the GP engine to assess and develop programs towards correctness and optimum performance.

This integration helps constrain the search space by ensuring that candidate programs are syntactically valid, context-aware, and guided by logical specifications.

Benefits of Refinement-Typed Genetic Programming

This approach offers several key advantages. It ensures *contextual relevance* by generating only code that aligns with the local lexical scope and typing environment. Through refinement types, it promotes *semantic correctness*, enforcing logical properties beyond simple type checking. Finally, the grammar generation process contributes to a constrained and focused *search space*, which, supports a more accurate synthesis, though the actual impact depends on the problem domain and fitness landscape.

Chapter 5

Implementation of Program Synthesis with Refinement-Typed Genetic Programming

This thesis extends ÆON, an existing prototype functional programming language, to support program synthesis guided by both hard and soft constraints. Hard constraints encode correctness conditions that synthesized programs must satisfy, while soft constraints encode optimization goals that guide the search toward preferred solutions.To achieve this goal, we implement a synthesis system that constrains and directs the search space using Refinement-Typed Genetic Programming.

The proposed implementation allows users to express their intent by writing ÆON specifications that combine hard constraints (which must be satisfied) and soft constraints (which guide optimization). These specifications serve a dual purpose: ensuring program correctness and steering the synthesis of complete or partial solutions. The Context Builder extracts key information from each specification, which is then translated by the ÆON synthesizer and used as input to the GENETICENGINE framework.

ÆON is responsible for parsing the specification, generating the grammar, and constructing the fitness function, while GENETICENGINE performs the evolutionary search over the resulting solution space. This modular integration enables the synthesis of semantically valid programs that align with user-defined constraints.

5.1 GENETICENGINE Modifications

GENETICENGINE was the framework chosen for this work due to its modular design, extensibility, and native support for type-safe, grammar-based synthesis. However, the original framework presented several limitations that restricted its applicability to more complex or realistic synthesis tasks. Specifically, it lacked support for multi-objective problems with a dynamic or unknown number of objectives; limited the expressiveness of type-level constraints by not allowing unions of Meta-Handlers; and imposed restrictive bounds on numeric literals that limited diversity in generated solutions. To address these limitations, this thesis introduces the following improvements, each discussed in more detail below.

Multi-objective Support with NSGA-II

Originally, GeneticEngine offered limited support for multi-objective optimisation. While it included an implementation of Lexicase Selection, this strategy implicitly assumed that the number of objectives was known and fixed at compile time. Such an assumption is not compatible with synthesis scenarios where objectives may be derived dynamically from the program context, as is the case in RTGP, where soft refinement constraints generate objectives in a data-dependent manner.

To address this limitation, we extended GeneticEngine with support for an arbitrary and dynamically inferred number of objectives. These objectives are lazily extracted from the synthesis context, allowing the framework to adapt to problems where the optimisation criteria are not known in advance, but instead emerge from refinement annotations and metadata generated during program analysis.

Building on this dynamic objective inference, we integrated support for the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [10], a widely adopted evolutionary algorithm for multi-objective optimisation. Unlike Lexicase Selection, which evaluates objectives in a sequential and stochastic manner, NSGA-II explicitly maintains a Pareto front of non-dominated solutions and applies a ranking and crowding distance mechanism to balance convergence and diversity.

This makes it particularly well suited for RTGP, where multiple, potentially conflicting, objectives must be optimised simultaneously without prior knowledge of their number or ordering.

Union of Meta-Handlers

GeneticEngine introduced the concept of Meta-Handlers as a mechanism for encoding additional semantic constraints on grammar nodes, thereby restricting the space of admissible values during the GP initialization process. In its original form, each grammar element could only be associated with a single Meta-Handler, limiting the expressiveness of type-level constraints in cases where multiple conditions needed to be enforced.

This work extends that model by supporting the *union of Meta-Handlers*, allowing multiple constraints to be associated with a single grammar node. Instead of replacing or overriding previous constraints, all associated Meta-Handlers are preserved and evaluated as alternatives, so a candidate is valid if it satisfies at least one of them. This effectively defines the valid space as the union of their individual domains and enables a more expressive and modular way of encoding complex refinement types, including disjunctive or composite predicates.

For example, consider a variable x annotated with the refinement type $x : \{\, \mathbb{Z} \mid 0 \leq x \leq 10 \,\} \cup \{\, \mathbb{Z} \mid 20 \leq x \leq 30 \,\}$. In the original GeneticEngine design, only one of these interval constraints could be attached to the corresponding grammar node. With support for unions, both constraints are retained as distinct Meta-Handlers, and a candidate value is considered valid if it satisfies at least one of them. As a result, the system correctly permits values within either of the two intervals, while excluding all values that fall outside them, including those in the gap between 10 and 20.

Expanded Search Space for Number Values

In its original form, GeneticEngine applied implicit bounds when initializing numeric literals, restricting integers and floating-point values to a limited range (e.g., between 0 and 100). While such bounds simplify the search space, they can significantly limit the engine's ability to discover valid solutions in problems that require values outside this predefined interval.

In this work, these restrictions were removed. Numeric initialisation now spans the full range permitted by the system, from the smallest to the largest representable value. For instance, while a variable previously initialised as an integer literal could only assume values in the range $[0, 100]$, it can now take any value within the full integer domain supported by the system. This modification improves the engine's ability to explore arithmetic-heavy domains, increasing the chance of discovering numerically valid and diverse solutions.

5.2 Æon Language

This section describes the Æon language as the specification front-end for our synthesis pipeline. We outline the language's main design goals and features, and then detail the type system and architecture that enable refinement-typed, grammar-guided synthesis.

5.2.1 Language Overview

Æon [14] is a functional programming language implemented in Python using the Lark parsing toolkit. It combines polymorphism with both hard and soft refinement types to achieve high expressiveness and flexibility in program synthesis tasks.

The language is designed to be user-friendly and intuitive, borrowing elements of Python-like syntax to lower the barrier for new users. Internally, Æon is built on top of a lower-level core language called ÆonCore, which has a more verbose and strict syntax. While ÆonCore adheres rigidly to typing rules, Æon introduces syntactic sugar to improve usability and readability.

To illustrate the difference, consider the factorial function written in both languages. The ÆonCore version (Listing 5.1) is harder to read and write compared to the more intuitive Æon version (Listing 5.2).

```
let factorial : (x:Int) -> Int = (\n ->
        if  == [Int] n 0  then 1
        else  * [Int] n  (factorial (- [Int] n 1 )))
```

Listing 5.1: Factorial function written in ÆonCore syntax.

```
def factorial (n:Int) : Int {
    if n == 0 then 1 else (n * factorial(n - 1))
}
```

Listing 5.2: Factorial function written in Æon syntax.

Æon supports two forms of refinement: hard and soft refinements. Hard refinements are statically verifiable through an SMT solver, whereas soft refinements are too expressive for static verification and must be checked at runtime.

Listing 5.3 shows an example with both hard refinements (`len ps ⩾ 1`, `balance > 0`, `balance ⩽ sum toBuy` and `toBuy subset ps`) and soft refinements (`@minimize_int (- len toBuy )`). This example also introduces the hole operator, `?hole`, which denotes an incomplete expression to be filled during synthesis. The synthesizer replaces this hole with a valid expression that conforms to the expected type and satisfies all applicable constraints. This example represents a variation of the classic knapsack problem. The function receives a list of integers `ps : List[Int]` with at least one element, and a positive balance. It must return a subset of `ps` whose sum does not exceed the balance, while also preferring solutions that return a longer list (i.e., maximizing its length).

```
@minimize_int(- (len toBuy))
def knapsack : (ps : List[Int] | len ps >= 1) ->
                (balance : Int | balance > 0) ->
                (toBuy : List[Int] |
                    isSubset toBuy ps &&
                    balance <= sum toBuy) {
    ?hole : List[Int]
}
```

Listing 5.3: Example with hard and soft constraints written in Æon.

5.2.2 Type System

The type system used in Æon was not developed as part of this thesis. It was previously introduced and follows the theoretical foundations presented in the paper *Refinement Types: A Tutorial* [23]. It supports both type checking and inference for programs annotated with refinement types, integrating three core judgments. The first is **check**, which determines whether an expression satisfies a given refinement type. The second is **synthesize**, which infers the most precise type for a given expression. Lastly, **entailment** verifies, using logical reasoning via an SMT solver, whether one refinement implies another. These rules are applied during the type-checking phase, ensuring that expressions are type-safe and annotated accordingly.

Refinement Types in Æon

Æon uses refinement types to express precise constraints over values, allowing users to encode richer semantic invariants directly in type annotations. For instance, in Listing 5.4, the function `pow` takes two arguments: a base `b` and an exponent `e`, both constrained to be between 1 and 100. The function computes the power using the `Math_pow` function, which is defined in one of Æon's standard libraries. The type annotations ensure that the inputs are always within the specified range.

By expressing such constraints in the type system, Æon enables early detection of incorrect programs and guarantees that synthesized expressions adhere to user-defined preconditions.

```
import "Math.ae";

def pow : (b:Int | b >= 1 && b <= 100) ->
          (e:Int | e >= 1 && e <= 100) -> Int =
                \x -> \y -> Math_pow x y;
```

Listing 5.4: A safe pow function defined in ÆON with refinement types.

5.2.3 Architecture

ÆON [14] is a functional programming language composed of several modules, each responsible for a distinct phase in the compilation pipeline, including parsing, type checking, and refinement handling. In this thesis, we extend its architecture with synthesis components to support refinement-type-guided synthesis and multi-objective optimization.

At a high level, the main architectural modules are:

- **sugar**: Parses the *.ae* file using ÆON syntax. This module desugars surface-level code into a lower-level representation compatible with ÆONCORE.
- **frontend**: Transforms the output of the *sugar* module into the core internal representation (ÆONCORE AST), preserving all relevant typing and syntactic information.
- **core**: Defines the abstract syntax tree (AST), all base types, terms, and language primitives supported in ÆON.
- **typing**: Performs type checking and inference based on refinement types. It annotates each expression node and stores hole information for synthesis when required.
- **backend**: Implements low-level features like pattern-matching-based evaluation and variable context management, and supports structural transformations needed for type checking and synthesis.
- **synthesis**: Originally, this module provided random expression generation based on type annotations. In this work, it was replaced by a more powerful synthesis pipeline built on top of GENETICENGINE.

To support refinement typed synthesis, this thesis introduced a new *synthesis* module that replaces the original one and extensive modifications were made to the *sugar* and *typing* modules. These changes were necessary to integrate decorators, import resolution, and grammar-based synthesis with user-defined constraints.

Key Modifications

This work introduces several architectural changes to support the new synthesis model:

- **Decorator and Fitness Handling**: Decorators such as `@minimize_int` are now parsed and interpreted during the *desugar* phase, where they are transformed into structured metadata associated with the corresponding function or synthesis target. This metadata explicitly encodes optimisation intent (e.g., whether a value should be minimised or maximised, and its

expected type) and is later consumed by the Context Builder to generate the appropriate fitness function. As a result, the optimisation criteria used by GeneticEngine are no longer defined externally or through manual configuration, but are instead derived directly from the program annotations, allowing the synthesis process to remain both self-contained and semantically aligned with the user's intent.

- **Import Resolution**: Support for importing libraries and other Æon files was added. All imported functions and types are merged and made available as candidate building blocks during synthesis.
- **A-normal Form Transformation**: Code normalization into A-normal form (ANF) [39], an intermediate representation of programs in functional programming language compilers, was moved from the parsing stage to the type-checking stage. This ensures normalization is applied after type refinements are processed, enabling accurate type inference and simplifying grammar construction.

System Overview and Compiler Design

Figure 5.1 presents a high-level view of the complete synthesis pipeline, from an Æon specification to the generation of candidate solutions. This process comprises several stages, including parsing, desugaring, type checking, constraint extraction via the Context Builder, and the construction of both a synthesis grammar and a fitness function, which are subsequently passed to GeneticEngine. A more detailed perspective of the internal workflow is provided in Figure 5.2, which illustrates the full compiler pipeline, including the parser, desugaring phase, type checking, hole identification, dynamic grammar generation, and the final integration with the synthesis engine. Together, these figures highlight how Æon systematically transforms a high-level, refinement-typed specification into a structured search space that can be effectively explored by genetic programming.

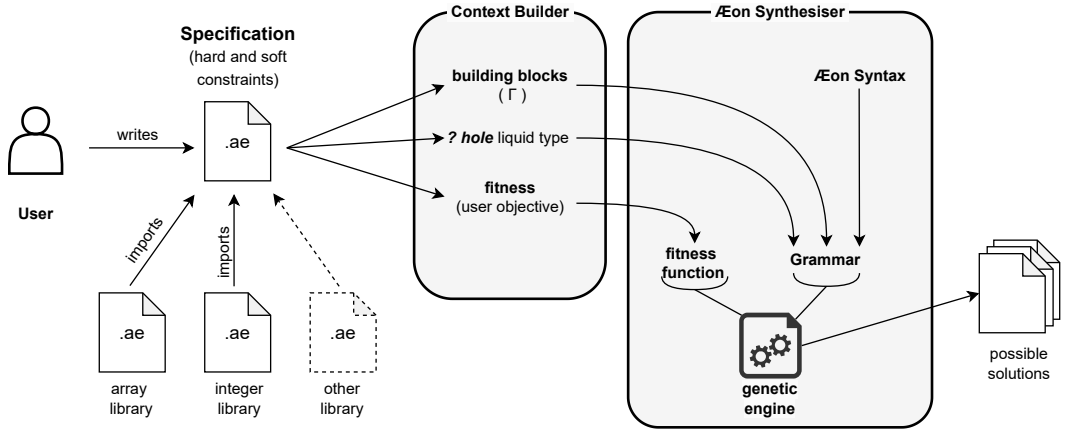


Figure 5.1: End-to-end synthesis pipeline, from an Æon specification file to solutions in Æon generated by GeneticEngine.

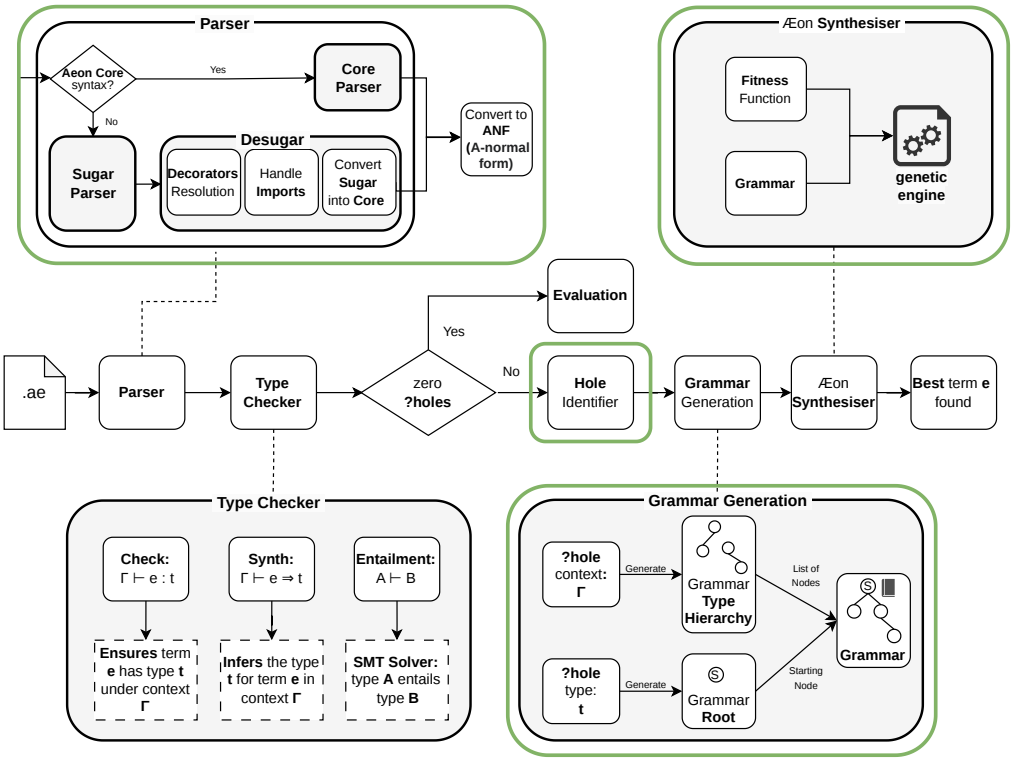


Figure 5.2: Compiler pipeline showing parsing, desugaring, type checking, hole extraction, grammar generation, and integration with the synthesis engine. Outlined in green are the contributions of this thesis, while the rest of the pipeline was inherited from the original Æon implementation.

5.2.4 Context Builder

After the specification is type-checked and normalized into A-normal form, the Context Builder extracts the necessary information for synthesis. The *Context Builder* is a core component of the Æon synthesis pipeline, responsible for translating user-written specifications into the structured inputs required by the GeneticEngine framework. Given a user specification in a *.ae* file, the Context Builder extracts three key elements:

- **Building Blocks** (Γ): A mapping that includes all visible definitions in scope, such as variable names, function declarations, and their associated types. This forms the set of building blocks available for synthesis.
- **Hole Type**: The type of the hole to be synthesized, including any attached refinement constraints. This is inferred by the type checker and serves as the root type for grammar generation.
- **Fitness Specification**: A representation of the user's optimization goals, derived from soft refinement annotations (e.g., `@minimize_int`). These are translated into a fitness function passed to GeneticEngine.

To illustrate, consider the simplified example in Listing 5.5, which defines a function `test` that takes a positive integer and returns a negative integer. The hole expression `?r` represents an incomplete expression that must be synthesized. From this example, the Context Builder infers that the input context contains a variable `k:Int |k > 0`, and the output type is `Int |z < 0`. Since no soft constraint is provided in this case, there is no guiding towards a specific solution. The first solution found that satisfies the hard constraints will be returned.

```
import aeon/libraries/.

def test : (k:Int | k > 0) -> {z:Int | z < 0} {
    ?r
}
```

Listing 5.5: Simple Specification File Example.

In more complex examples, such as the knapsack problem shown in Listing 5.3, the Context Builder identifies additional constructs in scope, including variables `ps: List[Int]` and `balance : Int`, and infers the hole type `List[Int]`. The fitness function in this case is derived from the soft refinement `@minimize_int(-len toBuy)`, which expresses a domain-specific predicate.

```
@minimize_int(- len toBuy)
def knapsack: (ps: List[Int] | len ps >= 1) ->
                (balance: Int | balance > 0) ->
                (toBuy: List[Int] |
                    isSubset toBuy ps &&
                    balance <= sum toBuy) {
    ?hole
}
```

Listing 5.6: Context Builder behavior on the knapsack specification.

Using this information, the Context Builder generates the necessary input for the synthesis pipeline. The grammar root is determined by the type of the hole (`List[Int]`), the grammar rules are constructed based on all in-scope definitions extracted into Γ, and the fitness function is synthesized from the soft constraint `@minimize_int(- len toBuy)`. The code in Listing 5.7 shows the internal form after the Context Builder processes the input.

```
let knapsack: (ps: List[Int] | len ps >= 1) ->
                (balance: Int | balance > 0) ->
                (toBuy: List[Int] |
                    isSubset toBuy ps && balance <= sum toBuy)
    = (\x -> ?hole : List[Int]) in
let __internal__fitness_function_knapsack : Int = - (len toBuy)
```

Listing 5.7: Transformed Code after decorator resolution and conversion to Core syntax.

If no soft refinements are present, the search returns the first programs that satisfy the hard constraints. Otherwise, expressions such as `(- len toBuy)` are compiled into fitness functions that score candidates for single- or multi-objective optimization, depending on the decorators.

5.2.5 Decorators

Æon provides a set of decorators, syntactic annotations prefixed with `@`, to specify additional synthesis behavior, soft constraints, or configuration flags. Inspired by Python's syntax, they are interpreted during the *sugar* phase and influence synthesis without affecting runtime semantics.

Table 5.1 provides an overview of all supported decorators and their primary roles.

Decorator	Description
Fitness-Oriented Decorators	
`@minimize_int(expr)`	Minimize an integer expression. Generates a hidden fitness function.
`@minimize_float(expr)`	Same as above but for float expressions.
`@multi_minimize_float(expr)`	Enables many-objective synthesis with a list of float objectives.
Grammar and Control Decorators	
`@hide(x, y, ...)`	Hides specific variables or constants from the synthesis grammar to reduce overfitting.
`@disable_control_flow`	Removes control flow nodes (e.g., `if` expressions) from the grammar.
`@allow_recursion`	Enables recursive calls to the function itself during synthesis.
Synthesis Configuration Decorators	
`@objective_number(n)`	Declares the number of objectives. Useful when the backend does not support dynamic objective discovery.
`@error_fitness(value)`	Sets a fallback fitness score in case of runtime exceptions during evaluation.

Table 5.1: Supported decorators in Æon, grouped by purpose.

These decorators are attached to function definitions and processed during the *sugar* phase. They are stripped from the program's AST and translated into internal metadata that influences how the *ContextBuilder* and GeneticEngine handle grammar generation, fitness evaluation, and control flow.

Some decorators, such as `@minimize_int` and `@multi_minimize_float`, directly define optimization goals. Others, like `@hide` or `@disable_control_flow`, control what constructs are included in the synthesis grammar. This separation between functional code and synthesis metadata allows users to write clean, expressive specifications while maintaining fine-grained control over the synthesis process.

Usage Example of Decorators

In the following example, the function *fun* has a soft constraint (`@minimize_int`), allows recursion during synthesis (`@allow_recursion`), and assigns a high error value if evaluation fails (`@error_fitness`).

```
@minimize_int(2023 - fun(289))
@error_fitness(9999)
@allow_recursion
def fun(i:Int) : Int {
  (?hole: Int) * i
}
```

Listing 5.8: Example usage of multiple decorators in Æon.

Multi-Objective Support

Thanks to our extensions to GeneticEngine, the synthesizer can now handle multiple objectives of the same or different types. For example, users can specify several `@minimize_int` decorators within the same function to independently minimize different expressions. Each decorator contributes a component to the overall fitness vector, enabling expressive multi-objective optimization. This modular design ensures that decorators serve as a flexible and intuitive mechanism to guide the synthesis process in Æon.

5.2.6 Grammar Extraction

The Æon programming language introduces a dynamic and context-sensitive approach to grammar construction for use within the GeneticEngine framework. Unlike traditional synthesis systems that rely on statically defined, overly general grammars, often resulting in inefficient exploration, Æon constructs grammars at runtime based on the user's active specification.

Rather than generating a broad and static grammar that attempts to cover the entire language space, Æon extracts a grammar tailored to the user's program context. This includes only constructs explicitly defined or imported in the specification, such as variable names, types, and available function definitions. By leveraging this localized and minimal grammar, the system dramatically reduces the synthesis search space while maintaining semantic correctness and contextual relevance.

This context-aware grammar extraction not only improves the efficiency of the search but also ensures that the synthesized expressions are type-safe and aligned with the user's original intent. It is a key enabler for scalable refinement typed program synthesis.

Hole Identification and Context Capture

Each hole's location is determined via a pre-order traversal of the program's abstract syntax tree (AST). During this traversal, the system records the typing context surrounding each hole, including variable declarations, function definitions, and all type information visible within the lexical scope. Simultaneously, the type of the hole is inferred and used as the root of the grammar.

For instance, in Listing 5.9, the hole named `"hole"` is inferred to have the type `Int`. The local context available at this point includes both built-in arithmetic and comparison operators, such as `==`, `!=`, `<`, `<=`, `>`, `>=`, `+`, `+.`, `-`, `-.`, `*`, `*.`, `/`, `/.`, `%`, and `%.`, as well as user-defined elements like the function `id_function` and the variable `i`.

This context reflects both the built-in capabilities of the language and the user-defined elements available within the scope of the function `fun`. By tightly scoping the grammar to what is locally available, Æon ensures that the synthesizer operates on a semantically valid and highly focused search space, significantly improving synthesis performance and correctness.

```
import "Math.ae";

def id_function (x:Int) : Int { x }

@minimize_int(Math_abs 2023 - fun 289)
def fun (i:Int) : Int {
    (?hole: Int) * i
}
```

Listing 5.9: Æon example of a simple synthesis task.

Dynamic Grammar Generation

The extracted context and hole type serve as inputs to the grammar generation process. This information is translated into a set of production rules, programmatically encoded as Python classes that define the synthesis grammar.

In Section 5.2.7, we describe how these context-sensitive grammars are represented, including the encoding of terminal and non-terminal nodes, function applications, and control-flow constructs.

5.2.7 Grammar Representation

The grammar extracted from an Æon program is compiled into a class-based representation compatible with the GeneticEngine framework. This grammar reflects a context-free grammar (CFG) structure and is composed of *non-terminal* and *terminal* nodes. These nodes represent the core syntactic and semantic elements of the language, including basic data types, control flow constructs, operators, and user-defined functions.

Non-Terminal Nodes

By definition, non-terminal nodes represent grammatical categories or abstract syntactic elements that can be further decomposed into sequences of non-terminal and terminal nodes. In Æon, data types and function signatures are conceptualised as non-terminal nodes. They act as abstract classes in grammar, which do not directly correspond to concrete implementations but rather to a set of permissible expressions. We represent these non-terminal nodes by the use of Python Abstract classes.

These data types are the ones predefined in Æon or types defined by the user. Æon has four data types by default `Int`, `Float`, `String` and `Bool`. These types are represented as abstract Python classes, named `t_Int`, `t_Float`, `t_String` and `t_Bool` respectively. Listing 5.10 shows the grammar nodes generated for these types

For types defined by the user, it is the same procedure. For example, in Listing 5.11 we can see a type defined by the user called `List` and in the Listing 5.12 is the corresponding grammar node. These abstract classes do not implement functionality directly; they serve as scaffolds defining the types of values or expressions that terminal nodes may instantiate.

Similarly to how ÆON handles user-defined types, it also generates corresponding grammar nodes for function signatures, as we can see in Listing 5.14 for the ÆON `square` function in Listing 5.13.

```
class t_Int(ABC):
        pass

class t_Float(ABC):
        pass

class t_String(ABC):
        pass

class t_Bool(ABC):
        pass
```

Listing 5.10: Python Abstract classes representing the non-terminal nodes of a grammar that are generated from the ÆON data types.

```
type List;
```

Listing 5.11: Type defined by the user in ÆON.

```
class t_List(ABC):
        pass
```

Listing 5.12: Python Abstract class representing the non-terminal nodes of a grammar that are generated from a ÆON data type defined by the user in Listing 5.11.

```
def square : ( n:Int ) -> Int = \x -> x * x ;
```

Listing 5.13: Function in ÆON that calculates the square of an integer.

```
class t_Int_t_Int(ABC):
        pass
```

Listing 5.14: Python Abstract class representing the non-terminal nodes of a grammar that are generated from a ÆON data type defined by the user in Listing 5.13.

Terminal Nodes

Terminal nodes, on the other hand, are the actual literal values. Terminals are the concrete characters or sequences of characters that constitute the actual program text, such as literals, variables, and operators.

InÆon, a terminal node might be a specific integer or float value, an arithmetic operator, a comparison operator, user-defined functions, and control flow operations.

Literal values, such as `literal_Int` in Listing 5.15, are a fundamental type of terminal node, representing an integer literal in Æon. This terminal node has an attribute, `value`, that holds the concrete integer value, and it extends the `t_Int` class, tying the literal value to the integer type within Æon's type system.

```
class literal_Int(t_Int):
    value: int
```

Listing 5.15: Terminal node representing a literal integer in Æon.

Function Application and Variable Nodes

Æon has a variety of predefined functions, including **arithmetic and comparison operators** essential for mathematical operations and relational expressions. Arithmetic operations support both integer and floating-point calculations. The comparison operators allow for the implementation of conditional logic, enabling the development of algorithms and making decisions based on comparative evaluations. Listing 5.16 showcases the essential arithmetic and comparison operations available in Æon, along with their respective type signatures.

```
    "+"  : (x:Int) -> (y:Int) -> Int
    "+." : (x:Float) -> (y:Float) -> Float
    "-"  : (x:Int) -> (y:Int) -> Int
    "-." : (x:Float) -> (y:Float) -> Float
    "*"  : (x:Int) -> (y:Int) -> Int
    "*." : (x:Float) -> (y:Float) -> Float
    "/"  : (x:Int) -> (y:Int) -> Int
    "/." : (x:Float) -> (y:Float) -> Float
    "%". : (x:Int) -> (y:Int) -> Int
    "%." : (x:Float) -> (y:Float) -> Float
    "==" : (x:Int) -> (y:Int) -> Bool
    "!=" : (x:Int) -> (y:Int) -> Bool
    "<"  : (x:Int) -> (y:Int) -> Bool
    "<=" : (x:Int) -> (y:Int) -> Bool
    ">"  : (x:Int) -> (y:Int) -> Bool
    ">=" : (x:Int) -> (y:Int) -> Bool
```

Listing 5.16: Predefined Integer and Float operations in Æon.

Each of these operations is a terminal node within Æon's grammar. For instance, the `+` operator, which represents integer addition, has a type signature `(x:Int)-> (y:Int)-> Int`, indicating it takes two integers and produces an integer result. Similarly, the floating-point addition operator `+.` takes two floats and returns a float. These operators are terminal nodes because they represent final, executable actions within the language and are not expanded further in the grammar.

User-defined functions and **predefined functions** are treated similarly during the grammar generation; both involve creating two distinct types of grammar terminal nodes. These nodes are specifically designed to represent the dual aspects of a function, its application, and variable form. This approach ensures seamless integration of functions into the grammar, whether they are built

into the language or defined by the user. For Instance, consider the `id_function` defined in the earlier Listing 5.9. From this function, Æon generates two separate grammar nodes:

Application node: The node `app_id_function`, represents the application of the function named `id_function`. It extends the data type corresponding to the function's return type. In the case of `id_function`, which returns an `Int`, the `app_id_function` class extends `t_Int` and includes information about the function's argument, both the name and type, as we can see the Listing 5.17.

```
class app_id_function (t_Int):
        x: t_Int
```

Listing 5.17: Application grammar node for the user-defined function, `id_function`, in Listing 5.9.

Variable Node: The second node, `var_id_function`, conceptualizes the function as a variable. Unlike the application node, it extends to the data type representing the function's signature, `Int -> Int` in this case, and does not contain information about the function's arguments, as we can see the Listing 5.18.

```
class var_id_function (t_Int_t_Int):
        pass
```

Listing 5.18: Variable grammar node for the user-defined function, `id_function`, in Listing 5.9.

Similarly, two nodes are generated for the predefined addition operation that we saw in Listing 5.16, the `app_add` class serving as an application node showing how the operation is executed with specific operands and the `var_add` class representing the addition operation as a variable entity extending the type that represents its signature. We can visualise the generated classes in Listing 5.19.

```
class app_add(t_Int):
    x: t_Int
    y: t_Int

class var_add(t_Int_t_Int):
    pass
```

Listing 5.19: Application and Variable grammar nodes for the Æon predefined `+` arithmetic function.

Control Flow Constructs

Lastly, **control flow operations** like the `if - then - else` block are also represented as terminal nodes. To encapsulate the `if - then - else` logic in the extracted grammar, a series of nodes have been created, and each of these nodes corresponds to different return type of the `if - then - else` block, allowing the language to maintain type safety. The structure of the nodes can be seen in Listing 5.20. This example represents the control flow nodes specifically for basic Æon's types, but the grammar generation extends beyond these. A corresponding control flow

node is dynamically generated for every non-terminal node in Æon, which may represent a predefined data type, a user-defined type, or a function signature. This ensures that every data type and function signature in the program, whether an inherent part of Æon or defined by the user, has an associated control flow node, thus maintaining comprehensive coverage and type safety across all constructs within Æon 's grammar.

```
class If_t_Float (t_Float):
        cond: t_Bool
        then: t_Float
        otherwise: t_Float

class If_t_Int (t_Int):
        cond: t_Bool
        then: t_Int
        otherwise: t_Int

class If_t_Bool (t_Bool):
        cond: t_Bool
        then: t_Bool
        otherwise: t_Bool

class If_t_String (t_String):
        cond: t_Bool
        then: t_String
        otherwise: t_String
```

Listing 5.20: Generated control Flow Grammar nodes for the Æon predefined types.

Example of Context-Driven Grammar Node Generation in Æon

The example provided in Listing 5.9, featuring user defined types and functions in Æon, vividly illustrates the dynamic node generation process of the language, particularly from the context of a specific program element, `?hole: Int`. From this single example, Æon dynamically generates a set of over 50 grammar nodes, each tailored to the particular constructs encountered in the context of the hole. This extensive generation underscores the ability of the language to adapt its grammar to the precise needs of any given program segment. These nodes cover a wide spectrum of capabilities of Æon, ranging from basic data types like `t_Int`, `t_Float`, and `t_Bool` to more complex constructs such as `If_t_Int` and `If_t_Float` for control flow operations. The generation process includes nodes for literal values (e.g., `literal_Int`, `literal_Float`), arithmetic operations (e.g., `app_add`, `app_sub`), and comparison operations (e.g., `app_equal`, `app_greater_than`). Notably, Æon creates both application nodes (e.g., `app_id_function`, `app_add`) and variable nodes (e.g., `var_id_function`, `var_add`) for each operation and function, reflecting their use in both functional application and as first-class values.

The comprehensive nature of this node generation from just the context surrounding `?hole: Int` highlights the capability of Æon to synthesize contextually relevant and type-safe grammar structures. It demonstrates the proficiency of the language in understanding the current programming context and preparing a grammar that can effectively handle many potential operations, con-

ditions, and data manipulations.

The starting node of every generated grammar is determined by the hole type, which in this case is `Int`. Therefore, the starting node in the generated grammar is `t_Int`. The grammar productions derived from this context are focused and precise, including only those nodes that are pertinent to operations, conditions, and data types related to an integer type hole. The productions are shown in Listing 5.21. Out of the 52 grammar nodes generated in this example, only a relevant subset is actively used in the grammar productions, ensuring focus and efficiency in the synthesis process.

```
Starting node = t_Int

Productions =
    t_Int  -> literal_Int(value: int)
            | app_i()
            | app_id_function(x: t_Int)
            | app_mod(x: t_Int, y: t_Int)
            | app_div(x: t_Int, y: t_Int)
            | app_mult(x: t_Int, y: t_Int)
            | app_sub(x: t_Int, y: t_Int)
            | app_add(x: t_Int, y: t_Int)
            | If_t_Int(cond: t_Bool,
                       then: t_Int,
                       otherwise: t_Int)

    t_Bool -> literal_Bool(value: bool)
            | app_greater_equal(x: t_Int, y: t_Int)
            | app_greater_than(x: t_Int, y: t_Int)
            | app_less_equal(x: t_Int, y: t_Int)
            | app_less_than(x: t_Int, y: t_Int)
            | app_not_equal(x: t_Int, y: t_Int)
            | app_equal(x: t_Int, y: t_Int)
            | If_t_Bool(cond: t_Bool,
                        then: t_Bool,
                        otherwise: t_Bool)
```

Listing 5.21: Grammar created from all the grammar nodes generated and the starting node.

This approach ensures that Æon's grammar remains expressive yet constrained, guiding the synthesizer toward correct and efficient solutions. By abstracting away the complexity of grammar design, Æon allows users to focus on writing expressive and complete specifications, rather than managing low-level grammar rules.

Integrating Refinement Types via Meta-Handlers

The integration of refinement types in Æon with GeneticEngine's Meta-Handlers represents a key advancement in the synthesis process. Refinement types allow developers to specify fine-grained constraints on program variables, inputs, and outputs, ensuring that synthesized programs adhere not only to type correctness but also to logical and domain-specific requirements.

To leverage this expressiveness during synthesis, refinement types defined in the Æon specification are extracted and translated into constraints that can be consumed by Meta-Handlers. These constraints are then used to guide the search, pruning semantically invalid candidates and improving both the precision and efficiency of the synthesis.

The translation process proceeds as follows: hard refinement types are parsed and transformed into symbolic expressions using the *sympy* library. These expressions are simplified and converted into Disjunctive Normal Form (DNF), a canonical representation of constraints as a disjunction of conjunctions. From the DNF form, bounded intervals are extracted and mapped into a list of conjunctions. Each conjunction is then translated into a dedicated Meta-Handler. When multiple handlers are present, a Union Meta-Handler is created to aggregate them.

Once this transformation is complete, the resulting Meta-Handlers are attached to the appropriate grammar nodes in the synthesis grammar. During the search, these handlers act as runtime filters, eliminating any candidate that violates the specified refinement constraints.

This integration enables the synthesis of programs that satisfy complex logical specifications. By combining the expressive power of refinement types with the constraint-enforcing capabilities of Meta-Handlers, Æon and GeneticEngine can generate high-quality, semantically valid programs more efficiently.

Example Translation

To illustrate this process, consider Listing 5.22, which defines a variable `i` with a refined type that enforces the value of `i` to be either within the range $(0, 10)$ or $(20, 30)$.

```
(i:Int | (i > 0 && i < 10) || (i > 20 && i < 30))
```

Listing 5.22: A simple variable with a refined type in Æon.

This refinement is converted into the following *sympy* expression:

```
Or(And(Gt(i, 0), Lt(i, 10)), And(Gt(i, 20), Lt(i, 30)))
```

This expression is then simplified into Disjunctive Normal Form (DNF), from which bounded intervals are extracted:

```
[(LOpen(0), ROpen(10)), (LOpen(20), ROpen(30))]
```

Each interval is translated into a corresponding Meta-Handler:

```
Annotated(int, IntRange(1, 9))
Annotated(int, IntRange(21, 29))
```

Finally, these are combined into a Union Meta-Handler:

```
Union([
  Annotated(int, IntRange(1, 9)),
  Annotated(int, IntRange(21, 29))
])
```

This Meta-Handler is attached to the grammar node responsible for generating integer values at the hole position, ensuring that only values satisfying the refinement are considered during synthesis.

Based on this refinement, Æon generates grammar classes for the abstract type, literals, and the variable. Listing 5.23 shows the grammar nodes derived from the disjunctive refinement of `i`.

These nodes allow the synthesizer to explore only type-safe and semantically valid candidates. By distinguishing base from refined types in the grammar, Æon keeps the search focused on solutions that respect the specified logical constraints.

```
class t_RefinedInt_Gt0Lt10_Or_Gt20Lt30(ABC):
    pass

class literal_RefinedInt_Gt0Lt10_Or_Gt20Lt30(t_RefinedInt_Gt0Lt10_Or_Gt20Lt30):
    value: Union([
        Annotated(int, IntRange(1, 9)),
        Annotated(int, IntRange(21, 29))
    ])

class refined_app_i(t_RefinedInt_Gt0Lt10_Or_Gt20Lt30):
    x: t_RefinedInt_Gt0Lt10_Or_Gt20Lt30

class app_i(t_Int):
    x: t_RefinedInt_Gt0Lt10_Or_Gt20Lt30
```

Listing 5.23: Grammar nodes generated for a refined variable `i` with disjunctive refinement ranges.

5.2.8 Synthesis

The final stage of Æon's pipeline is the synthesis of programs that satisfy a given specification, combining the refined grammar extraction and type system with the GeneticEngine framework. To enable seamless integration, Æon provides two essential components: a **grammar** and a **fitness function**. These are the only components required by GeneticEngine to perform a search task. The grammar is dynamically generated from the user's program context and the extracted refinement types, ensuring that only syntactically and semantically valid candidates are considered. The fitness function, defined through user-facing annotations such as `@minimize_int`, encodes the objective of the synthesis task. These annotations are compiled into evaluators that guide the search, whether by optimizing numerical values, satisfying Boolean predicates, or meeting structural constraints.

Modular Synthesizer Design

The synthesizer in Æon was designed with modularity in mind. Although the current implementation uses GeneticEngine for grammar-guided genetic programming, the system is designed to support alternative strategies such as enumerative search, constraint solving, or stochastic and hybrid approaches. By decoupling grammar generation, fitness evaluation, and the underlying search engine, Æon enables researchers to swap or extend components with minimal effort.

Search Configuration

To allow fine-tuned control over the synthesis process, GeneticEngine exposes a range of configuration parameters that can be customized directly from within Æon. These parameters

govern the behavior of the underlying evolutionary algorithm, enabling users to adjust aspects such as population size, search depth, stopping criteria, and representation strategy. This configurability provides practical flexibility when running different benchmarks or exploring varying synthesis objectives. Table 5.2 summarizes the available parameters and their typical usage.

Parameter	**Description**
Seed	Fixes randomness for reproducibility (`seed = 123`).
Maximum Depth	Limits the depth of generated program trees (`max_depth = 8`).
Population Size	Sets the population size (`population_size = 100`).
Elitism	Number of top individuals preserved between generations (`n_elites = 1`).
Verbosity	Controls logging and output verbosity (`verbose = 2`).
Target Fitness	Enables early stopping once a threshold is reached (`target_fitness = 0`).
Mutation/Crossover	Mutation and crossover probabilities (`probability_mutation = 0.1`, `probability_crossover = 0.7`).
Timeouts	Time-based stopping conditions (`timer_stop_criteria = True`, `timer_limit = 60`) (seconds).
Representation	Type of encoding used (`representation = "tree"`).

Table 5.2: Search Configuration Parameters in GeneticEngine and default values.

Design Trade-offs and Benefits

Unlike traditional synthesis approaches that rely on fixed or handcrafted grammars, Æon builds its grammar dynamically based on the user's code and refinements. This approach, combined with refinement types and Meta-Handlers ensures that the search space is both type-safe and semantically guided. As a result, the system can prune invalid solutions early and focus computation on meaningful candidates. By using GeneticEngine as the synthesis engine, Æon allows users to control *what* is being generated (through types, refinements, and annotations) while remaining agnostic to *how* it is generated. The integration of Æon with GeneticEngine centers on the automatic generation of a valid, context-aware grammar and an expressive fitness function. These components enable the synthesis of programs that match user-defined specifications, while offering an extensible platform for incorporating alternative search strategies. This design makes Æon a solid foundation for future research and practical applications in program synthesis.

Chapter 6

Evaluation

This chapter presents a two-part experimental evaluation of the synthesis framework. First, we analyze the behavior of the base ÆON synthesizer on the Program Synthesis Benchmark Suite (PSB2), treated as one benchmark suite with 22 exercises, focusing on solution validity, aggregated fitness, and how type structure influences performance. Second, we evaluate our RTGP implementation in a *Super Mario* level generation case study by comparing grammars with refinement types against a non-refined baseline, to assess how refinement constraints affect validity and search quality.

6.1 Experimental Setup

The evaluation comprises two parts. Part I studies PSB2 exercises to characterize the behavior of the base ÆON synthesizer across problem types. Part II evaluates our RTGP implementation in a *Super Mario* level generation case study by comparing refined and non-refined grammars.

Across both parts, runs were executed on the same hardware platform[1], and results were analyzed using two main outcomes: solution validity and aggregated fitness. A solution was considered valid if it executed without runtime exceptions and passed the type checker. Detailed search settings for each part are reported in Table 6.1 and Table 6.3.

Since all experiments used Lexicase Selection [21], which evaluates individuals across multiple test cases, we report *aggregated fitness* per run, defined as the average fitness across the elite individuals selected during evolution.

6.2 Part I: Program Synthesis Benchmark Suite 2 (PSB2)

This section reports results for the 22 PSB2 exercises, focusing on solution validity, aggregated fitness, and the relation between exercise type structure and synthesis performance.

6.2.1 Benchmark Setup and Search Configuration

Part I uses one benchmark suite with 22 exercises, the PSB2 [20]. For each exercise, 30 independent synthesis runs were performed per representation (Tree-Based and Grammatical Evolution).

[1] All runs were executed on a standard workstation equipped with an AMD Ryzen 9 7950X CPU and 128GB of RAM.

Before comparing fitness scores and solution counts, we conducted an initial validation step to ensure that every exercise could be solved within capabilities of Æon. To that end, we manually implemented a reference solution for each exercise and verified its execution.

Table 6.1 summarizes the search configuration used in GeneticEngine for the PSB2 runs.

Parameter	Value
`seed`	123
`max_depth`	12
`population_size`	1000
`n_elites`	1
`verbose`	2
`novelty`	1
`target_fitness`	0
`probability_mutation`	0.01
`probability_crossover`	0.9
`timer_stop_criteria`	`True`
`timer_limit`	7200 (seconds)
`representation`	`"tree"` (Tree-Based), `"ge"` (Grammatical Evolution)

Table 6.1: Search Configuration Parameters in GeneticEngine for PSB2 runs.

6.2.2 Solution Validity

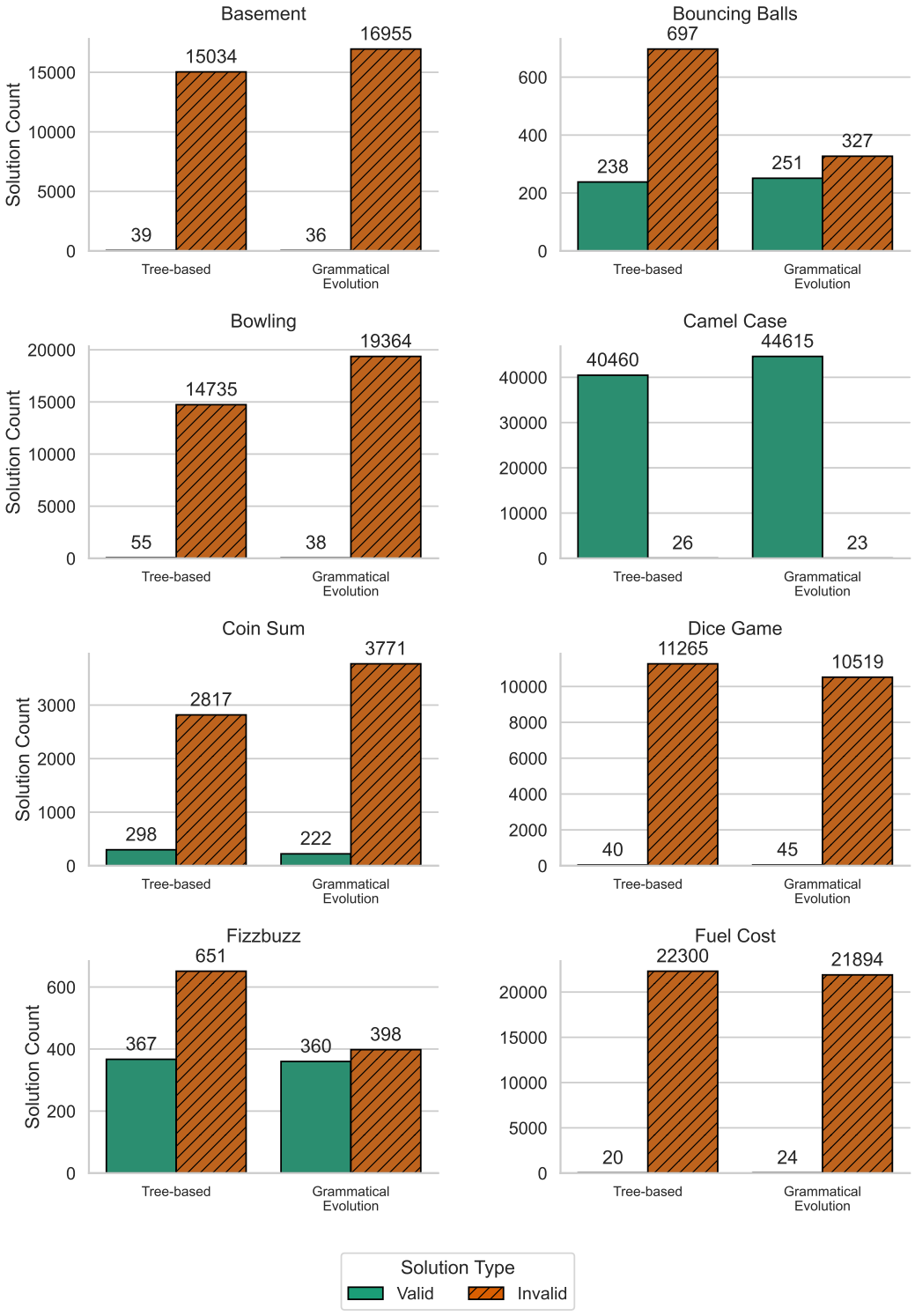
Figure 6.1 presents the count of valid and invalid solutions for each exercise. While both representations achieve comparable totals in many exercises, some differences stand out. The Tree-Based approach shows higher validity in exercises such as *GCD*, *Snow Day*, and *Spin Words*, suggesting improved syntactic and semantic coherence. Conversely, Grammatical Evolution tends to explore a wider but less constrained solution space, leading to more invalid individuals in problems like *Leaders*, *Camel Case*, and *Solve Boolean*. Beyond representation effects, the results indicate a clear relationship between type complexity and solution validity. Exercises involving at most two primitive types (e.g., *GCD*, *Spin Words*) generally produced a higher proportion of valid solutions, whereas exercises involving more than two primitive types (e.g., *Mastermind*, *Dice Game*) tended to produce more invalid candidates. A similar trend is observed for exercises with more complex data structures (e.g., *Shopping List*, *Indices of Substrings*), which also exhibited higher invalidity rates. Overall, these findings show that more complex types and data structures reduce validity. As the number of types and data structures in an exercise increases, invalid candidates become more frequent, likely because the corresponding grammar is harder to navigate during search. In these cases, invalidity was mainly due to runtime exceptions (e.g., out-of-bounds access and division by zero). A potential improvement is to incorporate dependent types, which could express stricter input/output constraints and further reduce invalid candidates. However, dependent types are not currently supported in Æon and remain a direction for future work.

6.2.3 Fitness Results

In terms of solution quality, Figure 6.2 shows the distribution of aggregated fitness values. Lower scores indicate better performance. The Tree-Based representation outperforms Grammatical Evolution in problems such as *Basement*, *Paired Digits*, and *Mastermind*, where type-aware expansion and structural constraints seem to help the search converge. On the other hand, Grammatical Evolution matches or outperforms Tree-Based in exercises like *Bowling* and *Solve Boolean*, where grammatical flexibility and syntactic variety are advantageous.

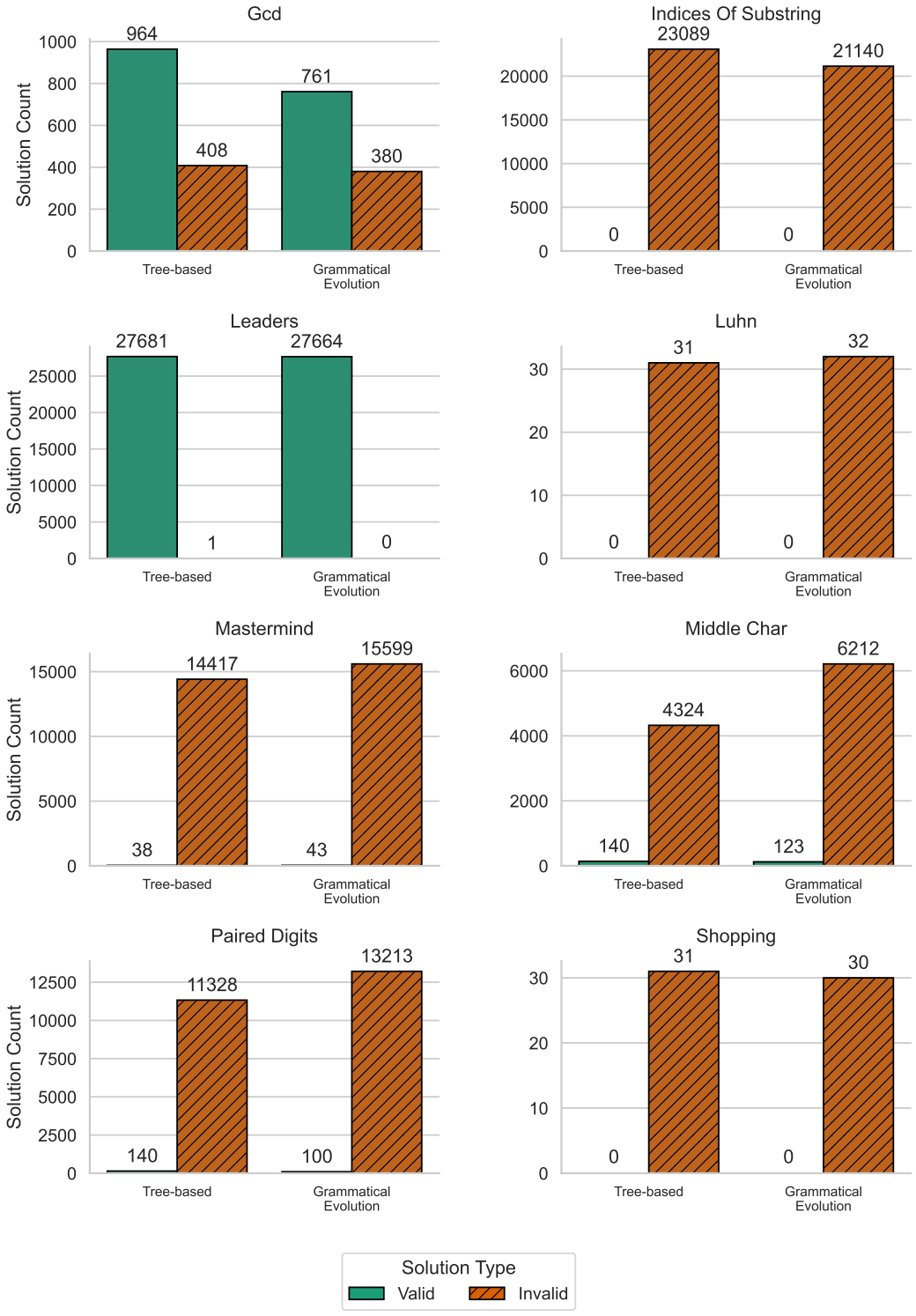
Problem	integers	floats	booleans	char	string	vector of integers	vector of floats
Basement	x		x			x	
Bouncing Balls	x		x				
Bowling	x		x	x			
Camel Case	x		x	x	x		
Coin Sum	x		x				
Dice Game	x	x	x				
Fizz Buzz	x		x				
Fuel Cost	x		x		x		
GCD	x		x				
Indices of Substrings	x		x		x	x	
Leaders	x					x	
Luhn	x		x				
Mastermind	x		x	x			
Middle Character	x		x				
Paired Digits	x		x	x			
Shopping List	x	x	x	x	x		x
Snow Day	x		x				
Solve Boolean	x		x		x		
Spin Words	x				x		
Square Digits	x		x				
Substitution Cipher	x		x				
Twitter	x		x		x		

Table 6.2: Types and Data Structures Used per Exercise.



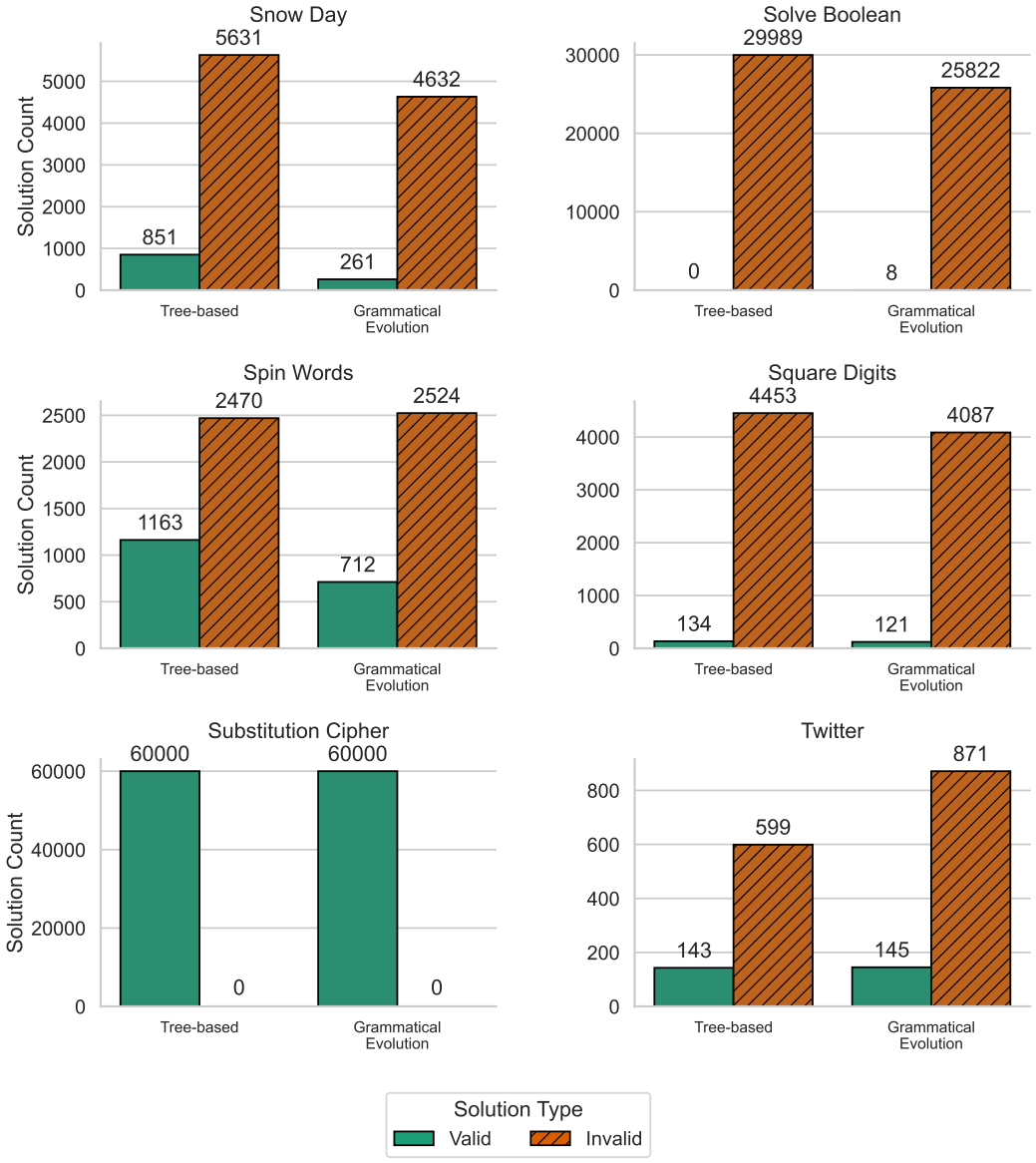
(a)

Figure 6.1: Valid and Invalid Solutions Across PSB2 Exercises.



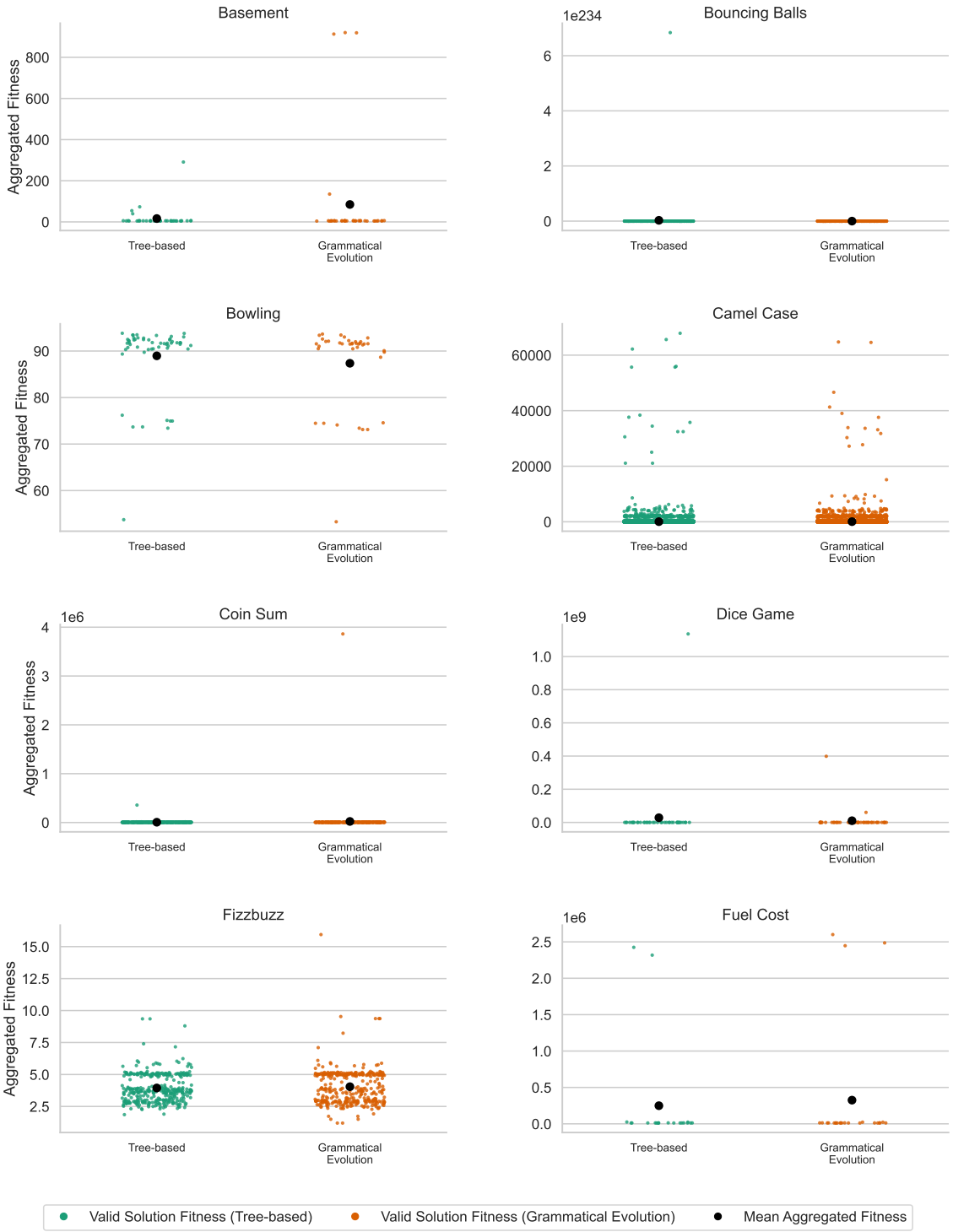
(b)

Figure 6.1: Valid and Invalid Solutions Across PSB2 Exercises (continued).



(c)

Figure 6.1: Valid and Invalid Solutions Across PSB2 Exercises (continued).

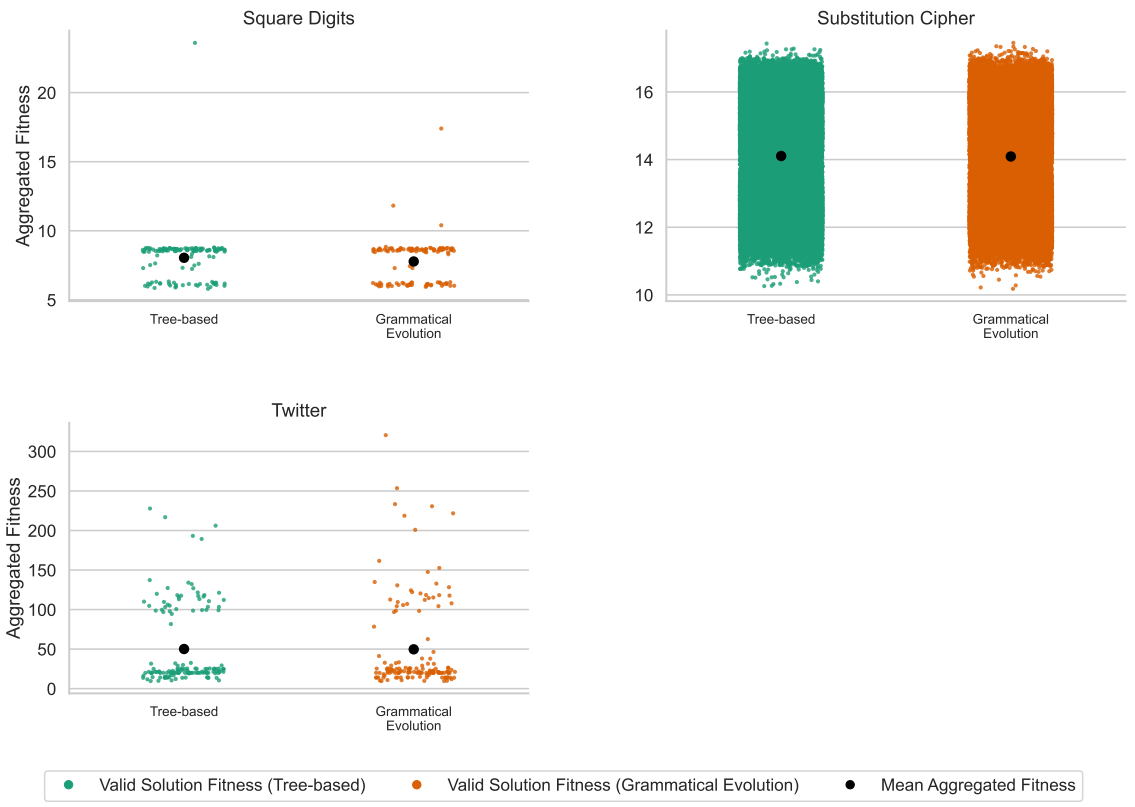


(a)

Figure 6.2: Aggregated fitness values across PSB2 exercises for both representations. Each coloured point corresponds to the aggregated fitness obtained in every valid solution, while the bigger black point represents the mean fitness for each representation.

(b)

Figure 6.2: Aggregated fitness values across PSB2 exercises for both representations (continued).



(c)

Figure 6.2: Aggregated fitness values across PSB2 exercises for both representations (continued).

6.2.4 Relation Between Exercise Types and Performance

To further understand performance differences, we analyzed the relationship between the data structures used in each PSB2 exercise and the observed synthesis outcomes. Table 6.2 summarizes the primitive and composite types involved in each exercise.

Exercises involving numerical inputs and arithmetic operations (e.g., *GCD*, *Coin Sum*) generally performed better with the Tree-Based representation, benefiting from strong typing and flexible numeric literals. In contrast, problems centered on string or character manipulation (e.g., *Camel Case*, *Middle Char*) often yielded similar or slightly better results under Grammatical Evolution, likely due to its greater syntactic variability. Exercises such as *Snow Day* and *Paired Digits*, which involve vectors or more structured inputs, produced a higher number of invalid candidates, particularly under Grammatical Evolution. These findings suggest that problems requiring coordination among multiple types benefit more from Tree-Based representations, where structural constraints are enforced more strictly.

Overall, this analysis highlights that the effectiveness of each representation is closely tied to the data types and structures involved in each exercise. Problems dominated by simple numerical domains and clear algebraic operations benefit from the structured, strongly-typed nature of the Tree-Based approach. Conversely, exercises relying on symbolic structure, pattern variation, or text manipulation appear better suited to the flexibility of Grammatical Evolution. Problems in-

volving heterogeneous or composite data types further expose limitations of purely grammar-based constraints, reinforcing the value of explicit structural and semantic guidance. Together, these results indicate that both type structure and representation influence synthesis outcomes.

6.3 Part II: Super Mario Level Generation Case Study

This section reports results for the *Super Mario* level generation case study, evaluating the efficiency of RTGP in reducing the search space through refinement type constraints. In particular, we assess how these constraints affect solution validity and fitness by comparing the RTGP configuration against a non-refined baseline.

6.3.1 Setup and Grammar

To further assess the effect of grammars with refinement types on synthesis, we conducted a dedicated experiment on procedural content generation for *Super Mario*, inspired by the paper *Evolving levels for Super Mario Bros using grammatical evolution* [43]. This case study differs from the other exercises in our evaluation primarily in how fitness is defined. While all other benchmark compute fitness using input–output examples, here fitness was computed via two auxiliary objectives: minimizing conflicts (such as overlapping), and maximizing level occupancy. This setup provides a direct assessment of how refinement types constrain search and improve validity of generated solutions. For this case study, we evaluated two configurations: (i) a baseline without refinement types (not RTGP), and (ii) our RTGP approach, which uses a refinement-typed grammar. We performed 30 independent runs for the non-refined baseline and 30 independent runs for the refined (RTGP) configuration. Table 6.3 summarizes the search configuration used in GeneticEngine for the *Super Mario* runs.

Parameter	Value
`seed`	123
`max_depth`	100
`population_size`	1000
`n_elites`	1
`verbose`	2
`novelty`	1
`target_fitness`	0
`probability_mutation`	0.1
`probability_crossover`	0.7
`timer_stop_criteria`	`True`
`timer_limit`	7200 (seconds)
`representation`	`"tree"` (Tree-Based)
`only_record_best_inds`	`True`

Table 6.3: Search Configuration Parameters in GeneticEngine for *Super Mario* runs.

The grammar used for level generation defines high-level constructs such as chunks, platforms, hills, cannons, tubes, boxes, and enemies. A simplified excerpt is shown in Listing 6.1:

```
<level>    ::= <chunks> <enemy>
<chunks>   ::= <chunk> | <chunk> <chunks>
<chunk>    ::= gap(<x>,<y>, <wg>,<wbefore>,<wafter>)
             | platform(<x>,<y>,<w>)
             | hill(<x>,<y>,<w>)
             | cannon_hill(<x>,<y>,<h>,<wbefore>,<wafter>)
             | tube_hill(<x>,<y>,<h>,<wbefore>,<wafter>)
             | coin(<x>,<y>,<wc>)
             | cannon(<x>,<y>,<h>,<wbefore>,<wafter>)
             | tube(<x>,<y>,<h>,<wbefore>,<wafter>)
             | <boxes>
<boxes>    ::= <box_type>(<x>,<y>)2 | ... | <box_type>(<x>,<y>)6
<box_type> ::= blockcoin | blockpowerup | rockcoin | rockempty
<enemy>    ::= (koopa | goompa)(<x>)2 | ... | (koopa | goompa)(<x>)10
<x>        ::= [5..95]            <wg>       ::= [2..5]
<y>        ::= [3..5]             <wBefore>  ::= [2..7]
<w>        ::= [3..15]            <wAfter>   ::= [2..7]
<h>        ::= [2..3]             <wc>       ::= [2..15]
```

Listing 6.1: Excerpt of the grammar extracted from ÆON used for Super Mario level generation.

We ran two experiments: (i) our RTGP implementation, using the refinement-typed grammar shown in Listing 6.1, and (ii) a baseline configuration using the same grammar structure but without refinement types (non-refined). In the RTGP configuration, each terminal in the grammar, such as $\langle x \rangle$, $\langle y \rangle$, $\langle w \rangle$, $\langle h \rangle$, is annotated with bounded interval refinements that are translated into Meta-Handlers and enforced during grammar expansion. This means that every individual generated by the GP engine is guaranteed to lie within the valid parameter ranges before any fitness evaluation takes place. In the non-refined baseline, all refinement annotations are not translated into the grammar, leaving terminal values unconstrained: the GP engine can produce coordinates, widths, and heights that are structurally arbitrary and semantically meaningless in the context of Super Mario level layout.

6.3.2 Validity and Fitness Results

As shown in Figure 6.3, this difference had a decisive effect on validity. The RTGP configuration produced exclusively valid individuals across all 30 runs: every candidate passed the type checker and satisfied the syntactic constraints encoded in the grammar. In contrast, the non-refined baseline generated only invalid candidates, failing type checks in all runs.

Figure 6.4 shows the aggregated fitness for the valid solutions using the Tree-Based representation. Fitness was computed using two complementary objectives: minimizing conflicts (e.g., overlapping level elements) and maximizing level occupancy, with a score of zero corresponding to a level that fully occupies the map without any conflicts. Fitness values converged toward this optimal range, which is a direct consequence of how RTGP operates: refinement annotations on positional and dimensional parameters, such as $x \in [5, 95]$, $y \in [3, 5]$, $w \in [3, 15]$, and $h \in [2, 3]$, prevent out-of-bounds placements and structurally ill-formed constructs from entering the popula-

tion. This eliminates a large class of runtime errors that would otherwise inflate fitness scores or waste evaluation resources.

While these results demonstrate the correctness and search-efficiency advantages of refinement types, future work should also examine their impact on the diversity and novelty of generated content, as tighter constraints may reduce phenotypic variety even when they improve validity and convergence.

This outcome directly illustrates the central claim of this thesis: *Refinement-Typed Genetic Programming (RTGP) narrows the search space, prevents the generation of semantically invalid candidates, and guides evolution toward more desirable solutions*.

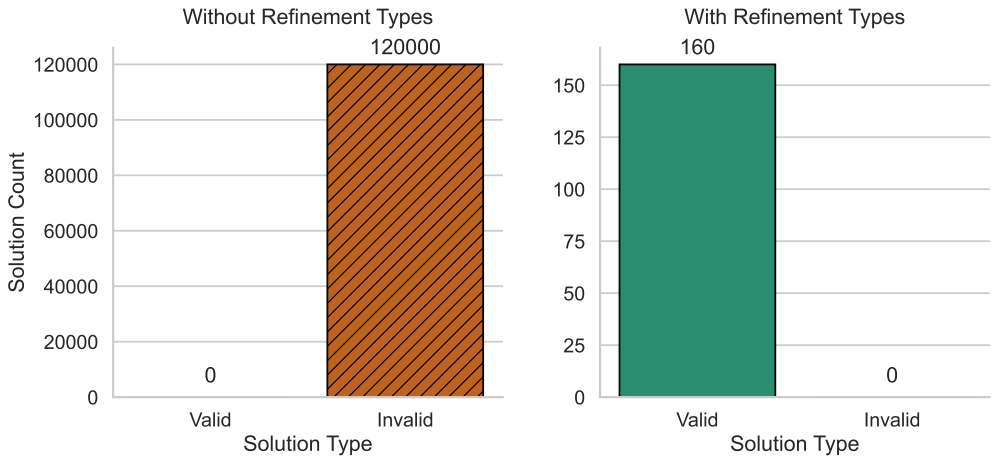


Figure 6.3: Valid and Invalid Solutions for Super Mario

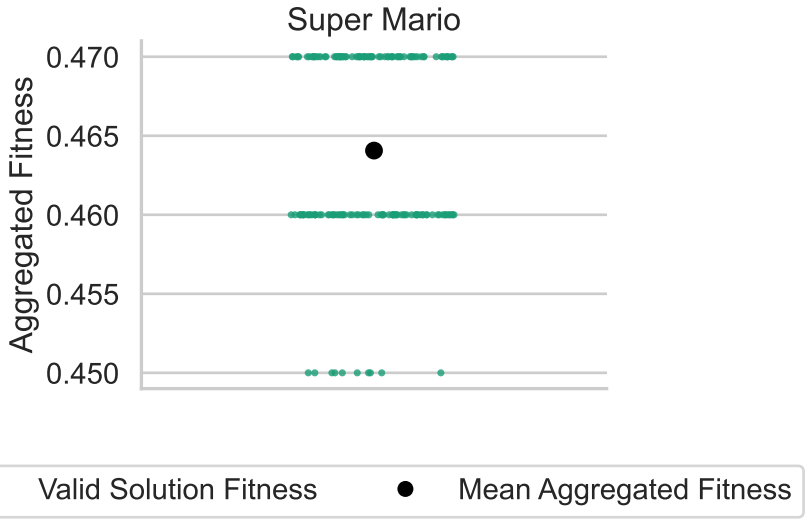


Figure 6.4: Aggregated fitness values for the *Super Mario* case study. Each coloured point corresponds to the aggregated fitness obtained in every valid solution, while the bigger black point represents the mean fitness across all valid solutions.

6.4 Summary of Findings

Overall, PSB2 results show that the type and data-structure profile of each exercise is a central factor in synthesis behavior. Exercises with simpler type combinations tended to produce more valid candidates, while exercises with richer type mixtures and composite structures showed higher invalidity and reduced search efficiency.

In this context, representation effects appear secondary to exercise structure: Tree-Based search was generally more robust on structured, type-rich exercises, whereas Grammatical Evolution remained competitive in exercises with higher syntactic variability. This suggests that synthesis performance is primarily shaped by the underlying type/data profile, with representation acting as an adaptation to those task characteristics.

The *Super Mario* case study supports this interpretation. Comparing our RTGP configuration with the non-refined baseline showed that refinement constraints decisively eliminated semantically invalid candidates and narrowed the search space. The fitness results further confirm that these constraints focus search on feasible regions of the design space, improving convergence without requiring additional evaluation budget.

In summary, these results demonstrate that RTGP is an effective strategy for constraining and guiding synthesis, improving both validity and search efficiency in complex domains.

Chapter 7

Conclusion

This thesis presented the implementation of a framework for *Refinement-Typed Genetic Programming* (RTGP), combining the expressive power of refinement types with the exploratory capabilities of genetic programming. The proposed system integrates the GeneticEngine framework into Æon to enable the synthesis of semantically valid and contextaware programs. Through this integration, RTGP constrains the search space using logical and type-based constraints while maintaining flexibility for multi-objective optimization and stochastic search.

The main goal of this work was to investigate how refinement types can guide and constrain the synthesis process, improving both correctness and efficiency. To this end, several technical contributions were made:

- **Language and type system extension:** Æon was redesigned to support both hard and soft refinement constraints, allowing users to encode strict correctness requirements and optimization preferences in a unified specification.
- **Integration with Genetic Programming:** The framework integrates dynamic grammar generation, refinement-type validation, and fitness extraction to produce type-safe grammars that ensure only valid candidates are synthesized.
- **Framework improvements:** GeneticEngine was extended to support multi-objective optimization via NSGA-II, unions of Meta-Handlers for composable constraints, and a wider numeric initialization range to improve diversity in arithmetic-heavy domains.
- **Empirical validation:** The system was evaluated on the PSB2 benchmark suite (22 exercises) and a procedural content generation case study in *Super Mario*, demonstrating the effectiveness of RTGP in improving validity and narrowing the search space.

The evaluation results show that RTGP significantly improves program validity and synthesis efficiency. The *Super Mario* case study provided the clearest evidence of this: refinement-typed grammars completely eliminated semantically invalid candidates across all 30 runs, whereas the non-refined baseline produced exclusively invalid candidates. Fitness values also converged toward optimal, confirming that refinement constraints effectively narrow the search space and guide

evolution toward feasible solutions. The PSB2 results further showed that type and data-structure complexity is the primary factor shaping synthesis difficulty, with both representations serving as secondary adaptations to those task characteristics. Importantly, even in exercises where invalid solutions were observed, all generated candidates were syntactically valid, confirming that the dynamic grammar generation implemented in this thesis behaves as expected, with invalidity arising solely from runtime semantic errors rather than structural malformation. Together, these findings validate the central hypothesis of this thesis: that Refinement-Typed Genetic Programming (RTGP) narrows the search space, prevents the generation of semantically invalid candidates, and guides evolution toward more desirable solutions.

Overall, this thesis showed that RTGP provides a concrete and effective way to guide synthesis toward semantically correct programs. The approach reduces invalid candidates, improves synthesis efficiency, and demonstrates that formal reasoning can be integrated into stochastic search without sacrificing flexibility. While there is still room for optimization and broader applicability, the work establishes a strong foundation for further research into practical program synthesis.

Chapter 8

Future Work

Although the results are encouraging, several practical challenges remain. The current implementation focuses primarily on small and medium-scale synthesis problems. Extending RTGP to larger specifications or real-world programs will require further optimization of both the type-checking and evolutionary stages, particularly with respect to runtime performance and memory usage. Another direction is improving the expressiveness of the Æon language to better support user-defined data types and richer refinement logic. These extensions would broaden the range of synthesis tasks that can be addressed while preserving the guaranteed type-safety.

It would also be valuable to investigate how refinement-type guidance interacts with different search strategies. Alternative evolutionary operators or hybrid approaches combining static analysis with search-based synthesis could further improve convergence and solution diversity. Expanding the case study evaluation to additional procedural generation domains, as well as applied settings such as automated test generation or domain-specific configuration synthesis, would help assess the generality of RTGP beyond the benchmarks explored in this thesis.

Another promising extension to the type system is the addition of dependent types to Æon. Unlike refinement types, which constrain individual values, dependent types allow types to depend on values, enabling relational invariants between inputs and outputs (e.g., ensuring a sorted output has the same length as its input). This would allow stricter specifications to be expressed directly in the type system, further reducing the search space of semantically invalid candidates.

Finally, a promising direction is integrating Æon with grammar-guided Large Language Model (LLM) generation frameworks such as SynCode [46], which constrain LLM output to follow a given context-free grammar. Like GeneticEngine, SynCode guarantees syntactic validity using a grammar, but instead of guiding search through a fitness function, it steers an LLM through a natural language prompt. Since Æon already generates grammars dynamically from type annotations, a natural extension would be to export these grammars in the Extended Backus–Naur Form (EBNF) format supported by SynCode and introduce a new Æon decorator allowing users to specify a synthesis prompt in place of the fitness function. This would enable a direct comparison between evolutionary search in GeneticEngine and LLM-guided synthesis under the same grammar, offering insight into the relative strengths of stochastic and neural approaches within a unified framework.

Bibliography

[1] Milad Taleby Ahvanooey et al. "A Survey of Genetic Programming and Its Applications". In: *KSII Trans. Internet Inf. Syst.* 13.4 (2019), pp. 1765–1794. DOI: 10.3837/tiis.2019.04.002. URL: https://doi.org/10.3837/tiis.2019.04.002.

[2] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. "Scaling Enumerative Program Synthesis via Divide and Conquer". In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*. Ed. by Axel Legay and Tiziana Margaria. Vol. 10205. Lecture Notes in Computer Science. 2017, pp. 319–336. DOI: 10.1007/978-3-662-54577-5_18. URL: https://doi.org/10.1007/978-3-662-54577-5_18.

[3] Rajeev Alur et al. "Syntax-guided synthesis". In: *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013, pp. 1–8. URL: https://ieeexplore.ieee.org/document/6679385/.

[4] Ishwa Anadani et al. "A Genetic Algorithm Approach for Portfolio Optimization". In: *Data Science and Applications*. Ed. by Satyasai Jagannath Nanda et al. Singapore: Springer Nature Singapore, 2024, pp. 113–124.

[5] Jacob Austin et al. "Program Synthesis with Large Language Models". In: *CoRR* abs/2108.07732 (2021). arXiv: 2108.07732. URL: https://arxiv.org/abs/2108.07732.

[6] Clark W. Barrett and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Springer, 2018, pp. 305–343. DOI: 10.1007/978-3-319-10575-8_11. URL: https://doi.org/10.1007/978-3-319-10575-8_11.

[7] Christopher M. Bishop and Nasser M. Nasrabadi. "*Pattern Recognition and Machine Learning*". In: *J. Electronic Imaging* 16.4 (2007), p. 049901. DOI: 10.1117/1.2819119. URL: https://doi.org/10.1117/1.2819119.

[8] Iwo Bladek and Krzysztof Krawiec. "Counterexample-Driven Genetic Programming for Symbolic Regression With Formal Constraints". In: *IEEE Trans. Evol. Comput.* 27.5 (2023), pp. 1327–1339. DOI: 10.1109/TEVC.2022.3205286. URL: https://doi.org/10.1109/TEVC.2022.3205286.

[9] Mark Chen et al. "Evaluating Large Language Models Trained on Code". In: *CoRR* abs/2107.03374 (2021). arXiv: 2107.03374. URL: https://arxiv.org/abs/2107.03374.

[10] Kalyanmoy Deb et al. "A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimisation: NSGA-II". In: *Parallel Problem Solving from Nature - PPSN VI, 6th International Conference, Paris, France, September 18-20, 2000, Proceedings*. Ed. by Marc Schoenauer et al. Vol. 1917. Lecture Notes in Computer Science. Springer, 2000, pp. 849–858. DOI: 10.1007/3-540-45356-3_83. URL: https://doi.org/10.1007/3-540-45356-3_83.

[11] Guilherme Espada et al. “Data Types as a More Ergonomic Frontend for Grammar-Guided Genetic Programming”. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2022, Auckland, New Zealand, December 6-7, 2022*. Ed. by Bernhard Scholz and Yukiyoshi Kameyama. ACM, 2022, pp. 86–94. DOI: 10.1145/3564719.3568697. URL: https://doi.org/10.1145/3564719.3568697.

[12] John K. Feser, Swarat Chaudhuri, and Isil Dillig. “Synthesizing data structure transformations from input-output examples”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Stephen M. Blackburn. ACM, 2015, pp. 229–239. DOI: 10.1145/2737924.2737977. URL: https://doi.org/10.1145/2737924.2737977.

[13] Cormac Flanagan. “Hybrid type checking”. In: *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. Ed. by J. Gregory Morrisett and Simon L. Peyton Jones. ACM, 2006, pp. 245–256. DOI: 10.1145/1111037.1111059. URL: https://doi.org/10.1145/1111037.1111059.

[14] Alcides Fonseca, Paulo Santos, and Sara Silva. “The Usability Argument for Refinement Typed Genetic Programming”. In: *Parallel Problem Solving from Nature - PPSN XVI - 16th International Conference, PPSN 2020, Leiden, The Netherlands, September 5-9, 2020, Proceedings, Part II*. Ed. by Thomas Bäck et al. Vol. 12270. Lecture Notes in Computer Science. Springer, 2020, pp. 18–32. DOI: 10.1007/978-3-030-58115-2_2. URL: https://doi.org/10.1007/978-3-030-58115-2_2.

[15] Timothy S. Freeman and Frank Pfenning. “Refinement Types for ML”. In: *Proceedings of the ACM SIGPLAN’91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*. Ed. by David S. Wise. ACM, 1991, pp. 268–277. DOI: 10.1145/113445.113468. URL: https://doi.org/10.1145/113445.113468.

[16] Catarina Gamboa et al. “User-driven Design and Evaluation of Liquid Types in Java”. In: *CoRR* abs/2110.05444 (2021). arXiv: 2110.05444. URL: https://arxiv.org/abs/2110.05444.

[17] Alberto Goffi et al. “Search-based synthesis of equivalent method sequences”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. Ed. by Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey. ACM, 2014, pp. 366–376. DOI: 10.1145/2635868.2635888. URL: https://doi.org/10.1145/2635868.2635888.

[18] Justin Gottschlich et al. “The three pillars of machine programming”. In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. Ed. by Justin Gottschlich and Alvin Cheung. ACM, 2018, pp. 69–80. DOI: 10.1145/3211346.3211355. URL: https://doi.org/10.1145/3211346.3211355.

[19] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. “Program Synthesis”. In: *Found. Trends Program. Lang.* 4.1-2 (2017), pp. 1–119. DOI: 10.1561/2500000010. URL: https://doi.org/10.1561/2500000010.

[20] Thomas Helmuth and Peter Kelly. “PSB2: the second program synthesis benchmark suite”. In: *GECCO '21: Genetic and Evolutionary Computation Conference, Lille, France, July 10-14, 2021*. Ed. by Francisco Chicano and Krzysztof Krawiec. ACM, 2021, pp. 785–794. DOI: 10.1145/3449639.3459285. URL: https://doi.org/10.1145/3449639.3459285.

[21] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. “Lexicase Selection for Program Synthesis: A Diversity Analysis”. In: *Genetic Programming Theory and Practice XIII [GPTP 2015]*. Ed. by Rick L. Riolo et al. Springer, 2015, pp. 151–167. DOI: 10.1007/978-3-319-34223-8_9. URL: https://doi.org/10.1007/978-3-319-34223-8_9.

[22] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992. ISBN: 9780262275552. DOI: 10.7551/MITPRESS/1090.001.0001. URL: https://doi.org/10.7551/mitpress/1090.001.0001.

[23] Ranjit Jhala and Niki Vazou. “Refinement Types: A Tutorial”. In: *Found. Trends Program. Lang.* 6.3-4 (2021), pp. 159–317. DOI: 10.1561/2500000032. URL: https://doi.org/10.1561/2500000032.

[24] Donald E. Knuth. “backus normal form vs. Backus Naur form”. In: *Commun. ACM* 7.12 (1964), pp. 735–736. DOI: 10.1145/355588.365140. URL: https://doi.org/10.1145/355588.365140.

[25] William B. Langdon and Mark Harman. “Optimizing Existing Software With Genetic Programming”. In: *IEEE Trans. Evol. Comput.* 19.1 (2015), pp. 118–135. DOI: 10.1109/TEVC.2013.2281544. URL: https://doi.org/10.1109/TEVC.2013.2281544.

[26] Nico Lehmann et al. “Flux: Liquid Types for Rust”. In: *Proc. ACM Program. Lang.* 7.PLDI (2023), pp. 1533–1557. DOI: 10.1145/3591283. URL: https://doi.org/10.1145/3591283.

[27] Zohar Manna and Richard J. Waldinger. “A Deductive Approach to Program Synthesis”. In: *ACM Trans. Program. Lang. Syst.* 2.1 (1980), pp. 90–121. DOI: 10.1145/357084.357090. URL: https://doi.org/10.1145/357084.357090.

[28] Daniel Manrique, Juan Rios, and Alfonso Rodríguez-Patón. “Grammar-Guided Genetic Programming”. In: *Encyclopedia of Artificial Intelligence (3 Volumes)*. Ed. by Juan R. Rabuñal, Julian Dorado, and Alejandro Pazos. IGI Global, 2009, pp. 767–773. URL: http://www.igi-global.com/Bookstore/Chapter.aspx?TitleId=10331.

[29] David J. Montana. “Strongly Typed Genetic Programming”. In: *Evol. Comput.* 3.2 (1995), pp. 199–230. DOI: 10.1162/evco.1995.3.2.199. URL: https://doi.org/10.1162/evco.1995.3.2.199.

[30] ThanhVu Nguyen et al. “Connecting Program Synthesis and Reachability: Automatic Program Repair Using Test-Input Generation”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*. Ed. by Axel Legay and Tiziana Margaria. Vol. 10205. Lecture Notes in Computer Science. 2017, pp. 301–318. DOI: 10.1007/978-3-662-54577-5_17. URL: https://doi.org/10.1007/978-3-662-54577-5_17.

[31] Patryk Orzechowski, William G. La Cava, and Jason H. Moore. “Where are we now?: a large benchmark study of recent symbolic regression methods”. In: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2018, Kyoto, Japan, July 15-19, 2018*. Ed. by Hernán E. Aguirre and Keiki Takadama. ACM, 2018, pp. 1183–1190. DOI: 10.1145/3205455.3205539. URL: https://doi.org/10.1145/3205455.3205539.

[32] Peter-Michael Osera and Steve Zdancewic. “Type-and-example-directed program synthesis”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Stephen M. Blackburn. ACM, 2015, pp. 619–630. DOI: 10.1145/2737924.2738007. URL: https://doi.org/10.1145/2737924.2738007.

[33] Edward R. Pantridge and Thomas Helmuth. “Solving Novel Program Synthesis Problems with Genetic Programming using Parametric Polymorphism”. In: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2023, Lisbon, Portugal, July 15-19, 2023*. Ed. by Sara Silva and Luís Paquete. ACM, 2023, pp. 1175–1183. DOI: 10.1145/3583131.3590502. URL: https://doi.org/10.1145/3583131.3590502.

[34] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. lulu.com, 2008. ISBN: 978-1-4092-0073-4. URL: http://www.gp-field-guide.org.uk/.

[35] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. “Program synthesis from polymorphic refinement types”. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. Ed. by Chandra Krintz and Emery D. Berger. ACM, 2016, pp. 522–538. DOI: 10.1145/2908080.2908093. URL: https://doi.org/10.1145/2908080.2908093.

[36] J. Ross Quinlan. “Induction of Decision Trees”. In: *Mach. Learn.* 1.1 (1986), pp. 81–106. DOI: 10.1023/A:1022643204877. URL: https://doi.org/10.1023/A:1022643204877.

[37] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. “Liquid types”. In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. Ed. by Rajiv Gupta and Saman P. Amarasinghe. ACM, 2008, pp. 159–169. DOI: 10.1145/1375581.1375602. URL: https://doi.org/10.1145/1375581.1375602.

[38] Conor Ryan, JJ Collins, and Michael O. Neill. “Grammatical evolution: Evolving programs for an arbitrary language”. In: *Genetic Programming*. Ed. by Wolfgang Banzhaf et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 83–96. ISBN: 978-3-540-69758-9.

[39] Amr Sabry and Matthias Felleisen. “Reasoning About Programs in Continuation-Passing Style”. In: *Proceedings of the Conference on Lisp and Functional Programming, LFP 1992, San Francisco, California, USA, 22-24 June 1992*. Ed. by Jon L. White. ACM, 1992, pp. 288–298. DOI: 10.1145/141471.141563. URL: https://doi.org/10.1145/141471.141563.

[40] Ángel Juan Sánchez-García et al. “Genetic Programming in Software Engineering: A Systematic Literature Review”. In: *Int. J. Comb. Optim. Probl. Informatics* 14.3 (2023), pp. 61–70. DOI: 10.61467/2007.1558.2023.V14I3.408. URL: https://doi.org/10.61467/2007.1558.2023.v14i3.408.

[41] Paulo Santos, Sara Silva, and Alcides Fonseca. "Refined typed genetic programming as a user interface for genetic programming". In: *GECCO '20: Genetic and Evolutionary Computation Conference, Companion Volume, Cancún, Mexico, July 8-12, 2020*. Ed. by Carlos Artemio Coello Coello. ACM, 2020, pp. 251–252. DOI: 10.1145/3377929.3390042. URL: https://doi.org/10.1145/3377929.3390042.

[42] Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stochastic superoptimization". In: *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*. Ed. by Vivek Sarkar and Rastislav Bodík. ACM, 2013, pp. 305–316. DOI: 10.1145/2451116.2451150. URL: https://doi.org/10.1145/2451116.2451150.

[43] Noor Shaker et al. "Evolving levels for Super Mario Bros using grammatical evolution". In: *2012 IEEE Conference on Computational Intelligence and Games, CIG 2012, Granada, Spain, September 11-14, 2012*. IEEE, 2012, pp. 304–311. DOI: 10.1109/CIG.2012.6374170. URL: https://doi.org/10.1109/CIG.2012.6374170.

[44] Ehud Y. Shapiro. "Inductive Program Synthesis". In: *Algorithmic Program Debugging*. 2004, pp. 81–137.

[45] Sabrina Tseng, Erik Hemberg, and Una-May O'Reilly. "Synthesizing Programs from Program Pieces Using Genetic Programming and Refinement Type Checking". In: *Genetic Programming - 25th European Conference, EuroGP 2022, Held as Part of EvoStar 2022, Madrid, Spain, April 20-22, 2022, Proceedings*. Ed. by Eric Medvet, Gisele L. Pappa, and Bing Xue. Vol. 13223. Lecture Notes in Computer Science. Springer, 2022, pp. 197–211. DOI: 10.1007/978-3-031-02056-8_13. URL: https://doi.org/10.1007/978-3-031-02056-8_13.

[46] Shubham Ugare et al. "SynCode: LLM Generation with Grammar Augmentation". In: *Trans. Mach. Learn. Res.* 2025 (2025). URL: https://openreview.net/forum?id=HiUZtgAPoH.

[47] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. "LiquidHaskell: experience with refinement types in the real world". In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*. Ed. by Wouter Swierstra. ACM, 2014, pp. 39–51. DOI: 10.1145/2633357.2633366. URL: https://doi.org/10.1145/2633357.2633366.

[48] Niki Vazou et al. "Refinement types for Haskell". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. Ed. by Johan Jeuring and Manuel M. T. Chakravarty. ACM, 2014, pp. 269–282. DOI: 10.1145/2628136.2628161. URL: https://doi.org/10.1145/2628136.2628161.

[49] Peter A. Whigham. "Grammatically-based Genetic Programming". In: 1995. URL: https://api.semanticscholar.org/CorpusID:14127572.

[50] Bing Xue et al. "A Survey on Evolutionary Computation Approaches to Feature Selection". In: *IEEE Trans. Evol. Comput.* 20.4 (2016), pp. 606–626. DOI: 10.1109/TEVC.2015.2504420. URL: https://doi.org/10.1109/TEVC.2015.2504420.

[51] Geunseok Yang et al. "Applying Genetic Programming with Similar Bug Fix Information to Automatic Fault Repair". In: *Symmetry* 10.4 (2018), p. 92. DOI: 10.3390/SYM10040092. URL: https://doi.org/10.3390/sym10040092.