UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA

Program Synthesis Using Dependent Types: An Analysis of Efficiency and Search Methodologies

Su Lishun

Mestrado em Engenharia Informática

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

2025

Agradecimentos

O final desta dissertação representa o culminar de um ano de trabalho intenso e, ao mesmo tempo, o resultado de toda uma experiência de vida que me ajudou a alargar horizontes e a moldar a minha forma de pensar.

Em primeiro lugar, quero expressar a minha sincera gratidão ao meu orientador, Professor Alcides, que me incentivou a apresentar o meu trabalho, a participar em eventos académicos e a acreditar nas minhas capacidades, dando-me simultaneamente liberdade e espaço para que o desenvolvimento desta tese fosse um processo pessoal e não a ordem do orientador.

Em segundo lugar, agradeço aos meus colegas do grupo *Cool Kids*, com quem partilhei dúvidas, experiências e de quem recebi feedbacks valiosos ao longo deste ano. Um agradecimento especial à Catarina, pelas críticas construtivas que me motivaram a melhorar continuamente; ao Guilherme, que, apesar de não termos falado muito, nunca deixou de procurar formas de me ajudar com os seus conhecimentos; ao João, companheiro na mesma fase académica, cuja partilha de sentimentos e experiências me fez perceber que não estava sozinho; ao Tomás Carreira, pelas ideias inovadoras que contribuíram significativamente para este trabalho; e ao Paulo, sempre disponível para apoiar, partilhar experiências e ajudar-me a crescer e adaptar-me.

Em terceiro lugar, quero agradecer a todos os membros do LASIGE, que me receberam de braços abertos e me mostraram que este centro é mais do que um espaço de investigação, é também um lugar de convívio, diversidade e partilha entre pessoas de diferentes nacionalidades e objetivos. Um agradecimento especial ao Tomás Barreto, pela sua companhia e apoio ao longo do ano.

Por fim, quero agradecer à minha família por todo o apoio incondicional nos momentos fáceis e difíceis. Sem eles, não teria conseguido ultrapassar este desafio. Aos meus pais, que sempre estiveram ao meu lado e fizeram tudo o que estava ao seu alcance para que eu pudesse perseguir as minhas ambições, deixo a minha mais profunda gratidão.

This work was supported by FCT through project RAP, ref. EXPL/CCI-COM/1306/2021, 2025.00002.HPCVLAB.ISTUL, POR011PRE e 2022.15800.CPCA.A1

À minha família, agradeço o apoio constante e a paciência durante todos os momentos desta jornada.

Resumo

A síntese automática de programas é atualmente considerada um dos maiores desafios da Engenharia Informática e da investigação em linguagens de programação. A sua ambição central é gerar automaticamente programas de computador a partir de especificações de alto nível, sejam elas expressas em exemplos de entrada e saída, em especificações formais ou até em linguagem natural. Apesar do seu potencial revolucionário na automatização do desenvolvimento de software, a síntese de programas enfrenta um problema fundamental: o espaço de procura de soluções possíveis é de dimensão praticamente infinita, o que torna a tarefa extremamente dispendiosa do ponto de vista computacional e de difícil aplicabilidade prática em cenários reais.

Uma das estratégias mais relevantes para mitigar este problema passa pela utilização de sistemas de tipos. Os sistemas de tipos fortes garantem que apenas programas com combinações coerentes de variáveis e funções são considerados válidos, reduzindo de imediato o espaço de procura. Tipos refinados acrescentam um nível adicional de restrição ao introduzirem predicados lógicos que limitam os valores possíveis de uma dada variável. Já os tipos dependentes permitem que os próprios tipos dependam dos valores concretos manipulados pelo programa, oferecendo uma expressividade e precisão superiores. Ao restringirem de forma mais incisiva o espaço de soluções admissíveis, estes mecanismos têm o potencial de acelerar significativamente a síntese de programas.

Em paralelo, diferentes métodos de procura foram propostos para navegar o espaço de soluções. Três famílias principais têm dominado a investigação recente: (i) métodos enumerativos, que exploram sistematicamente todas as possibilidades, garantindo completude mas sacrificando eficiência; (ii) métodos heurísticos, que recorrem a estratégias guiadas por aleatoriedade e heurísticas para acelerar a procura, sem assegurar contudo a obtenção da solução ótima; e (iii) métodos baseados em Large Language models (LLMs), que exploram as capacidades gerativas destes modelos para propor programas candidatos ou para interagir com técnicas tradicionais de síntese.

Esta dissertação insere-se neste cruzamento entre a utilização de tipos dependentes e o estudo comparativo das diferentes abordagens de procura. O trabalho parte da implementação e extensão de um motor de síntese de programas com suporte a tipos refinados e dependentes, o Genetic Engine (GE), cujo funcionamento é amplamente regulado por metahandlers mecanismos responsáveis por impor restrições estruturais e semânticas durante a geração de programas. Entre os metahandlers atualmente existentes destacam-se o IntRange, que limita valores inteiros a um intervalo; e o Dependent, que permite expressar dependências entre atributos dentro da mesma

classe. Apesar da sua utilidade, estes metahandlers apresentam limitações quando é necessário capturar relações contextuais entre diferentes classes ou níveis da árvore sintática. Para ultrapassar essas limitações, esta dissertação introduz um novo metahandler que permitem uma maior expressividade na definição de dependências entre componentes. Entre as contribuições desenvolvidas, destaca-se o Parent Metahandler, que amplia as capacidades dos metahandlers existentes, permitindo que atributos de diferentes classes se relacionem entre si e, assim, que o sistema explore dependências contextuais de forma mais eficaz. Esta inovação simplifica a definição de benchmarks, melhora a legibilidade do código e aumenta a robustez dos resultados obtidos.

Outra contribuição fundamental foi a proposta de um método híbrido de procura, que combina a geração inicial de candidatos por modelos de linguagem de larga escala com a subsequente evolução e otimização desses candidatos por programação genética. A motivação para este método reside na constatação de que os LLMs, apesar de capazes de produzir soluções plausíveis, nem sempre cumprem integralmente as restrições dos sistemas de tipos e metahandlers. Ao conjugar esta geração inicial com a capacidade evolutiva da programação genética, foi possível explorar mais eficientemente o espaço de procura e alcançar melhores taxas de sucesso. Para avaliar este método híbrido, foram conduzidas experiências com diferentes LLMs de código aberto, incluindo versões do Llama, Qwen e DeepSeek, testadas em benchmarks simples e em cenários mais complexos, como a geração de níveis de jogos (Mario benchmark). Os resultados mostraram que, embora os LLMs apresentem limitações quanto à diversidade e validade das instâncias geradas, a sua combinação com programação genética permite obter ganhos significativos em termos de eficiência e qualidade das soluções.

No que respeita ao impacto dos tipos dependentes, a dissertação inclui dois estudos empíricos complementares. O primeiro foi realizado em ambientes controlados, com espaço de procura reduzido, onde foi possível observar de forma clara as diferenças entre tipos refinados e tipos dependentes. Os resultados mostraram que, nestes cenários, os tipos dependentes se revelam substancialmente mais eficazes, reduzindo o número de soluções inválidas e acelerando a convergência para soluções válidas. O segundo estudo, mais abrangente, foi conduzido sobre uma linguagem funcional que suporta síntese de programas (Aeon) e recorreu a benchmarks inspirados em problemas clássicos de programação. Neste caso, os benefícios dos tipos dependentes foram menos pronunciados: embora tenham permitido expressar restrições mais complexas, os custos adicionais de procura e verificação tornaram os ganhos de eficiência menos claros.

De uma forma mais detalhada, os resultados experimentais mostraram que os métodos heurísticos, em particular a programação genética com tipos refinados e dependentes, apresentam um desempenho globalmente superior face aos métodos puramente enumerativos e aos métodos hibrido. A introdução de dependências contextuais através do novo metahandler revelou-se decisiva para explorar problemas de maior complexidade estrutural. Já o método híbrido, embora ainda limitado pela capacidade dos modelos de linguagem utilizados, demonstrou um caminho promissor para futuras investigações, ao conjugar a criatividade e diversidade dos LLMs com a robustez e sistematicidade da programação genética.

A primeira contribuição é a extensão do Genetic Engine com o desenvolvimento do Parent Metahandler, que permite expressar dependências entre classes de forma mais natural e expressiva, ultrapassando limitações do metahandler dependente existente.

A segunda contribuição é a implementação de um método de procura híbrido que conjuga LLMs com programação genética, demonstrando empiricamente a sua viabilidade e vantagens em cenários de síntese complexa.

A terceira contribuição é a avaliação comparativa do impacto de tipos dependentes face a tipos de refinamento, tanto em ambientes controlados como em benchmarks de maior escala, fornecendo evidência empírica sobre os ganhos e limitações associados a cada abordagem.

A investigação realizada confirma que os tipos dependentes são um instrumento poderoso para restringir o espaço de procura e assegurar maior correção dos programas sintetizados, mas também evidencia que o seu custo computacional não deve ser subestimado. Em contextos controlados, os ganhos são claros e justificam a sua utilização; em contextos mais amplos, a relação custo-benefício requer uma análise cuidadosa, sendo recomendável uma integração seletiva destas técnicas. Por outro lado, os métodos de procura híbridos emergem como uma linha de investigação altamente promissora, sobretudo considerando a rápida evolução dos modelos de linguagem de larga escala. Acredita-se que, à medida que estes modelos se tornem mais eficientes e especializados para tarefas de síntese, o papel desta abordagem híbrida ganhará ainda maior relevância.

Em conclusão, esta dissertação contribui para a compreensão do papel dos tipos dependentes na síntese automática de programas e para a avaliação comparativa das principais estratégias de procura. O trabalho realizado demonstra que, apesar de persistirem desafios significativos, a combinação de tipos avançados com métodos híbridos de procura oferece um caminho viável para aproximar a síntese automática da sua plena aplicação prática em Engenharia de Software. Além disso, abre novas perspetivas para investigação futura, como a otimização do custo de verificação de tipos dependentes, a adaptação dinâmica das estratégias de procura em função do problema, e a integração mais estreita entre LLMs e sistemas formais de tipos.

Palavras-chave: Síntese de Programas, Tipos Refinados, Tipos Dependentes, LLMs, Programação Genética.

Abstract

Program synthesis is widely regarded as one of the most critical challenges in automating software development. A major obstacle lies in the enormous size of the search space, which severely limits the practical applicability of program synthesis in real-world scenarios.

One approach to constraining the search space is through type systems. Strong typing requires programmers to specify variable and function types explicitly. Refinement types extend this by introducing logical predicates that restrict permissible values, while dependent types go further by allowing types to depend on program values, enabling greater expressive power and precision.

Another line of research focuses on more efficient search strategies. Three main families of search methods have emerged: Enumerative methods, which systematically explore all possible solutions; Heuristic methods, which apply guided randomness to improve efficiency; and LLM-based methods, which leverage LLMs to generate possible programs or combine them with traditional synthesis techniques.

This thesis investigates how dependent types can reduce the search space in program synthesis and evaluates the performance of different search methods. We propose a novel LLM-based search method and benchmark it against existing approaches, finding that heuristic search delivers the strongest overall performance. To assess the impact of dependent types, we conduct two evaluations: first, in a controlled setting with a smaller search space, where dependent types demonstrate significant improvements over refinement types; and second, in a functional programming language, where the benefits of dependent types are less pronounced.

Keywords: Program Synthesis, LLMs, Refinement types, Dependent types, Genetic Programming.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Program synthesis refers to the automatic generation of computer programs from high-level specifications, such as examples, natural language, or a formal specification. However, one of the major challenges of program synthesis lies in efficiently navigating the large search space of possible program implementations.

To address this large search space, researchers have developed various approaches to explore the different ways of navigating it. There are three main families of methods, which are the Enumerative method, the Heuristic method, and the Large Language Models(LLMs) based method, aimed at improving the process and making it more scalable.

Enumerative methods [1, 3, 4, 25, 34, 9] systematically generate programs within a designated search space in a specific order, and check whether each candidate meets the defined synthesis constraints. This family provides the guarantee that all the candidates are verified.

Heuristic methods [29, 16, 10, 22, 21] provides a random navigation in the search space, and following soft guidelines (heuristics) that hope to help the navigation to better solutions.

LLMs-based methods [19, 31] leverages LLMs to generate candidate programs and then applies traditional synthesis techniques to refine results.

Researchers have also explored various methods to constrain the search space in program synthesis, aiming to make the process more efficient and precise. One prominent method involves the use of grammars [21, 10]. A grammar is a structured set of rules defined by terminal symbols, non-terminal symbols, and a starting symbol that collectively describe all valid expressions within a specific context. Grammars ensure that all generated candidates adhere to problem-specific rules, effectively excluding invalid solutions from the search space.

Another approach leverages type systems to enforce type correctness [25, 22, 10]. Type systems define constraints on inputs and outputs, dictating how components can be combined. This prevents incompatible combinations and ensures that only type-consistent structures are synthesized. Refinement types build on this by introducing logical predicates to restrict the values a variable's type can accept [10]. These predicates add an additional layer of constraint, enabling more precise control over program behavior. Programming languages such as LiquidHaskell, Lean, and

Aeon implement **Refinement types**. Listing 1.1 illustrates a synthesis problem where refinement types are employed to reduce the number of candidate solutions. In this example, the function `multiplyPositive` receives an argument, `i`, that corresponds to the number you want to multiply. The first refinement is to minimize the result of `multiplyPositive 7 + multiplyPositive 5`. Then `?hole` represents the portion that needs to be synthesized, and the refinement type `x > 0` serves to constrain the search space effectively.

```
@minimize(multiplyPositive 7 + multiplyPositive 5)
def multiplyPositive (i:Int) : Int  {
    (?hole: {x:Int | x > 0} ) * i
}
```

Listing 1.1: Program synthesis problem in Aeon using Refinement type

Similarly, dependent types extend this idea by allowing types to depend on values. Dependent types enable the specification of program behavior directly within the type system, ensuring correctness at a granular level. Languages that support dependent types include Aeon, Lean, Agda, and Idris.

Listing 1.2 presents a synthesis problem where dependent types assist the valid solution. The function `division` receives an argument, `i`, that corresponds to the number you want to be divided. The initial refinement is to maximize the result of `division 1 + division 2`. Then, the dependent type `y < 0` serves to restrict the possible solutions, and `y != 0` because it is a division.

```
@maximize(division 1 + division 2)
def division (i:Int) : Int  {
    x : Int = i * 2
    i / (?hole: {y:Int | y < x && y != 0} )
}
```

Listing 1.2: Program synthesis problem in Aeon using Dependent type

1.2 Problem

Today, numerous approaches aim to navigate the search space efficiently in program synthesis, yielding promising results, each with its strengths and limitations. However, it remains unclear which approach best navigates the search space effectively and consistently across diverse synthesis tasks. The optimal method for constraining the search space is still an open question, and further research is required to determine which strategies yield the most reliable and efficient outcomes in varying contexts.

Dependent types appear promising for reducing the search space in the program synthesis process by introducing value-based constraints that eliminate invalid solutions early in the search. However, the extent to which they effectively enhance the efficiency and success of the synthesis process remains an open question, requiring further empirical evaluation and theoretical analysis to substantiate their impact.

1.3 Objectives

The primary objective of this thesis is to investigate the efficiency of different search methods in navigating the search space during program synthesis with dependent types. In particular, we aim to evaluate how dependent types influence the synthesis process compared to refinement types. Specifically, the thesis seeks to:

- Identify which search method achieves the best balance between search space exploration and solution quality when using dependent types.
- Analyze the extent to which dependent types improve the synthesis process in comparison to refinement types.
- Evaluate the trade-offs introduced by dependent types, such as increased expressiveness versus higher computational cost during generation.

1.4 Contributions

This thesis has the following contributions:

- Development of a type annotation that constrains the generation (and mutation) of a grammar element, based on its parent nodes.
- The second contribution is the implementation of the novel search method. The new search method combines the LLM with Genetic Programming search.
- The third contribution is development of Dependently-Typed synthesis of random and exhaustive programs.

1.5 Structure of the document

The structure of this thesis is organized as follows. Chapter 2 provides the necessary background by introducing the core topics required to understand this work, including type theory, program synthesis, Genetic Engine, and the Aeon programming language. Chapter 3 reviews related work by presenting existing benchmark suites for program synthesis, various synthesis approaches, and their relationship to state-of-the-art methods. Chapter 4 introduces a parent node-dependent synthesis approach through a novel metahandler. Chapter 5 presents the proposed LLM-GP hybrid method, detailing the novel search strategy, model selection, and its evaluation against other search methods. Chapter 6 compares dependent and independent refinement types by evaluating their performance in a low-level environment with a reduced search space. Chapter 7 further evaluates the performance of dependent and refinement types in a functional programming language setting. Finally, Chapter 8 outlines directions for future work, and Chapter 9 concludes the thesis.

Chapter 2

Background

This chapter presents the main concepts necessary for a better understanding of this work: type theory, program synthesis, the Genetic Engine synthesis library, and the Aeon Language.

2.1 Type Theory

Strongly typed programming languages require programmers to define the types of variables and functions explicitly. This ensures that type errors are caught early during the type-checking process, improving code reliability. These languages are typically divided into two categories: Statically Typed Languages, and Dynamically Typed Languages. In these languages where Static Type check is adopted, type checking occurs at compile time, allowing errors to be detected before execution. Examples include Java [24], Haskell [13, 20], and C [18, 17]; in Dynamically Typed Languages, type checking occurs at runtime, providing greater flexibility but potentially delaying the detection of type errors. Examples include Python [26] and Ruby [28].

To enhance flexibility, some languages introduce polymorphic types, which are particularly valuable in languages like Haskell and Scala [38]. Polymorphism, especially parametric polymorphism, increases the expressiveness of the language by allowing type constructors to work generically across various types. This capability enables variables, functions, and data structures to handle different types in a uniform way, making it easier to address a wide range of programming problems. Many times, while types in strongly typed programming languages enforce constraints and catch errors early, they may not always be expressive enough to restrict certain behaviors or enforce more complex invariants [5]. To address these limitations, advanced type systems such as refinement types and dependent types have been introduced.

Refinement types enhance the expressiveness of type systems by introducing logical predicates that restrict the values a variable can accept. This added layer of constraint helps ensure that programs adhere to specific correctness properties. Languages like LiquidHaskell [33] and LiquidJava [11] support refinement types, enabling more precise control over program behavior.

There are two classes of refinements: liquid types and non-liquid types. Liquid refinement types are those that can be statically verified with the help of a Satisfiability Modulo Theories (SMT) solver, which is a tool capable of evaluating the truth of logical predicates. The non-

liquid refinement types cannot be statically verifiable, but can still be used for runtime correctness checks.

```
def division (i:Int | i > 0) : {result:Int} {
    1 / i
}
```

Listing 2.1: Refinement type in Aeon

Listing 2.1 presents an unary function that receive a number `i` and returns `1` divided by `i`. The type of argument `i`, a refinement type, states that it must be an `Integer` with a value greater than 0. The invocation of the function with a value that does not comply with the condition, for instance, `division -1` raises a compile-time error, avoid runtime division-by-zero errors.

Dependent types are types whose definitions depend on the values of their arguments. Like refinement types, dependent types enable programmers to improve code quality by embedding precise specifications about a program's behavior directly into its type system. Functional programming languages such as Agda [23] and Coq [6] implement **Full Dependent types**, embracing them comprehensively at every level. In these languages, types and values are treated uniformly, allowing types to be first-class citizens that can depend on values in arbitrary ways.

Other languages, like Aeon, adopt **Value-dependent types** or **Dependent Refinement types**, where a type can depend on a value, but this value must have the same type. This allows the type of a function or variable to be parameterized by specific values. Listing 2.2 illustrates an example of dependent types in Aeon, where the return type of a function is refined to guarantee that it remains greater than zero and smaller than the input argument.

```
def division (i:Int | i > 0) : (result:Int | 0 < result && result <= i )  {
    result = 1 / i
}
```

Listing 2.2: Dependent type in Aeon

2.2 Program synthesis

2.2.1 Overview of Program Synthesis

Program synthesis involves the automatic generation of software programs based on user intent, which is expressed through high-level specifications or input-output examples. This technique has broad applications in software engineering, including optimizing compilers, identifying bugs, and refactoring code.

Currently, there are three prominent families of search method in program synthesis: **Enumerative Methods**, systematically explores all possible solutions, ensuring completeness but often being computationally expensive; **Heuristic Methods**, by applying guiding rules or strategies, this method enhances search efficiency but does not guarantee finding the optimal solution; and **LLMs-based Methods**, LLMs are leveraged to generate candidate programs. These models can

either provide an initial population for refinement using other methods or directly generate complete solutions (further explained in Chapter 3).

2.2.2 Challenges in Program Synthesis

The main challenge in the program synthesis is the infinite search space. The available functions and terminals allow the generation of an infinite number of combinations. Currently, one of the solutions to reduce the search space is by considering the type system. Synthesizing only correct programs, programs whose types are properly type checked, lessens the search space. For instance, the synthesizer does not generate programs like (”s” + 0). Refinement types, introducing a predicate that refines a specific type, also provide a further restriction to the space of valid generated programs.

2.3 Genetic Engine

The Genetic Engine (GE) is a Refinement Type Genetic Programming library implemented in Python to realize program synthesis. In this framework, the Grammars like listing 2.3 are mapped as Data Classes listing 2.4, and then you can use Refinement types like the example where you use *IntRange* where the variable can only be in the range.

```
S      ::= <expr>
<expr> ::= <num> | <expr> <operator> <expr>
<num>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<op>   ::= + | -
```

Listing 2.3: Simple Grammar Example

```
class Expr(ABC):
    pass

@dataclass
class num(Expr):
    num: Annotated[int, IntRange(0, 9)]

@dataclass
class Add(Expr):
    left: Expr
    right: Expr

@dataclass
class Minus(Expr):
    left: Expr
    right: Expr
```

Listing 2.4: Simple Grammar Example in Genetic Engine

An important feature of GE is currently that it supports Dependent Types, as shown in listing 2.5. The attributes can be dependent on the attributes in the same class.

```
@dataclass
class Numbers:
```

```
    n1: Annotated[int, IntRange(0, 9)]
    n2: Annotated[int, Dependent("n1", lambda x: IntRange(x, x+10))]
```

Listing 2.5: Dependent MetaHandler

2.3.1 MetaHandler

MetaHandlers are type refinements that extend beyond the conventional role of restricting values to a subset of a type, as is the case with traditional refinement types. While a standard refinement type might, for example, constrain an integer to values between 0 and 3, MetaHandlers additionally influence the way abstract syntax trees (ASTs) are generated during program synthesis. In this way, they serve a dual purpose: both restricting the admissible values of a type and guiding the structural construction of synthesized programs.

Currently, the system provides the following MetaHandlers:

IntRange As illustrated in Listing 2.4, the `IntRange(0, 9)` MetaHandler restricts an attribute to take integer values between 0 and 9. The first parameter specifies the minimum value and the second the maximum value. During generation, the GE produces values within this range.

FloatRange The `FloatRange` MetaHandler, shown in Listing 2.6, behaves analogously to `IntRange` but applies to floating point numbers. It restricts a floating-point attribute to lie within the specified range.

```
@dataclass
class num():
    num: Annotated[int, FloatRange(1.5, 4.1)]
```

Listing 2.6: Float MetaHandler

ListSizeBetween The `ListSizeBetween` MetaHandler, illustrated in Listing 2.7, constrains the size of lists. For example, `Annotated[list[int], ListSizeBetween(1,100)]` specifies that the GE will generate a list of integers with a size between 1 and 100. If a fixed-size list is required, both parameters can be set to the same value, e.g., `ListSizeBetween(10,10)`, in which case the GE will always generate lists of exactly size 10.

```
@dataclass
class Numbers:
    ns: Annotated[list[int], ListSizeBetween(1,100)]
```

Listing 2.7: MetaHandler for lists

VarRange This MetaHandler, illustrated in Listing 2.8, is used when the generated values should be restricted to a specific set. For example, if the target values are prime numbers, the list of prime numbers can be provided, and the GE will generate values exclusively from this list.

```
@dataclass
class Number:
    n: Annotated[int, VarRange([1,3,5,7,11])]
```

Listing 2.8: VarRange MetaHandler

Dependent The `Dependent` MetaHandler allows attributes within the same class to access previously generated attributes and refine their own values based on these dependencies. In Listing 2.5, for example, the attribute `n2` uses the MetaHandler by specifying the name of the attribute it depends on as the first parameter. Multiple attributes can be declared by separating them with commas. The second parameter receives a lambda function that applies the refinement using the referenced attribute(s). In this case, after obtaining the value of `n1`, the attribute `n2` generates a value within the range `[n1, n1 + 10]`.

A second example is shown in Listing 2.9, where an integer is made dependent on a list. Here, the attribute `ns` generates a list of integers with a size between 1 and 20, while `n2` uses the MetaHandler to depend on the length of the list. If the list size is even, `n2` is generated within one range; if it is odd, a different range is used.

```
@dataclass
class Numbers:
    ns: Annotated[list[int], ListSizeBetween(1, 20)]
    n2: Annotated[int, Dependent("ns",
    lambda x: IntRange(50, 100) if len(x) % 2 == 0 else IntRange(0, 20))]
```

Listing 2.9: Dependent MetaHandler

2.4 Aeon Language

Aeon is a general-purpose functional programming language that utilizes refined and dependent types, as well as program synthesis, which leverages GE as its framework for synthesizing programs.

Similar to other functional programming languages, like Haskell and Scheme [27], Aeon contains native functions, lambda expressions, type creation, type abstractions, and applications.

```
@minimize_int( if (fun 3) == 4 then 0 else 1)
def nextNumber (x:Int) : {y:Int | y > x} { ?hole }
```

Listing 2.10: Program synthesis problem in Aeon

listing 2.10 illustrates the program synthesis problem in Aeon, specifically highlighting the part labeled *?hole* as the target for synthesis. The function *nextNumber* takes an integer x and returns an integer y, with the dependent type constraint that y must be greater than x. The annotation *minimize int* adds an additional synthesis constraint by instructing the synthesizer to choose the smallest valid result. For example, when the input $x = 3$, the minimal valid y is 4. This restriction is straightforward and leaves little room for variation. However, in more complex problems, where

the quality of the function's output can be evaluated incrementally, such annotations help guide the synthesizer toward increasingly optimized solutions.

Chapter 3

Related Work

3.1 Benchmark Suites

This section presents the alternative benchmark suites that are used in program synthesis.

3.1.1 SyGuS's Benchmark Suite

This SyGuS's Benchmark Suite [1, 2, 3, 4] is a benchmark suite of synthesis problems to provide a basis for side-by-side comparisons of different solution strategies. The suite includes several specialized tracks to evaluate synthesis solvers across diverse tasks. The General Track covers a broad range of synthesis problems, such as cryptographic circuit design and program repair, allowing solvers to demonstrate versatility. The Conditional Linear Integer Arithmetic (CLIA) Track focuses on problems involving conditional linear integer expressions, requiring solvers to handle arithmetic conditions with precision. The Invariant Generation Track tasks solvers with creating loop invariant logical statements that hold true within loops, essential for program verification. Lastly, the Programming by Examples (PBE) Track includes two subcategories: PBE-Bitvectors for bitwise operation synthesis, often applied in low-level code optimization, and PBE-Strings for generating string manipulation functions based on input-output examples, aiding in tasks like text processing and data cleaning. These tracks collectively assess the solvers' efficiency and adaptability across different synthesis requirements. Although this suite has a large number of benchmarks, it does not include tests related to types or dependent types.

3.1.2 Super Mario Bros Level Benchmark

The Super Mario Bros Level Benchmark [30] is a structured benchmark that the author employs to evaluate the capacity of grammatical evolution, a form of program synthesis, to generate playable game levels automatically. Within this context, level generation is treated as a synthesis problem: programs, expressed as derivations from a context-free grammar, are evolved to produce structured outputs (levels) that satisfy both syntactic constraints and semantic requirements of playability. The benchmark, therefore, illustrates how program synthesis techniques can be applied to domains traditionally considered creative or design-oriented.

3.1.3 PSB

The General Program Synthesis Benchmark Suite [15] (PSB) consists of 29 problems designed to evaluate automated program synthesis systems by mimicking typical programming challenges. These problems cover various programming constructs, including loops, conditionals, recursion, text processing, and mathematical computations. They range from simple tasks, like calculating sums or counting specific elements, to more complex string transformations, pattern recognition, and text analysis. Each problem aims to test different aspects of a system's capability to handle general programming tasks, such as handling arrays, control flow, data type manipulation, and algorithmic thinking, providing a comprehensive test bed for assessing progress in program synthesis. However, PSB offers a diversity of benchmarks, but these tests are not related to real-world problems; that is, they are not complex problems, and PSB is not focused on data structure.

3.1.4 SYNQUID's Benchmark suite

The SYNQUID's Benchmark suite [25] is a collection designed to evaluate SYNQUID, focusing on assessing its effectiveness in synthesizing complex, correct-by-construction programs using polymorphic refinement types. However, it is versatile and can also be used to evaluate other synthesis programs. In this regard, this benchmark suite comprises 64 synthesis challenges organized by problem type to assess the performance and expressiveness of synthesis tools. These benchmarks span categories that demonstrate a tool's capacity to handle various data structures, including lists, binary search trees, AVL trees, red-black trees, and sorting algorithms. Specific categories include List Manipulations, which include operations such as checking for empty lists, duplicating elements, and removing duplicates, requiring recursive reasoning and maintaining universal properties like uniqueness and order. Unique Lists tests focus on preserving uniqueness within lists through operations like insertions and deletions, while Sorted Lists and Strictly Ordered Lists challenges require handling ordering constraints in operations like merging sorted lists and finding intersections. The Sorting Algorithms category, with algorithms such as insertion sort and quicksort, tests the tool's ability to enforce order through recursive synthesis. The Binary Search Trees (BSTs) category includes creating, inserting, and deleting nodes while maintaining BST invariants. More advanced capabilities are tested with AVL and Red-Black Trees, which include balancing operations and involve complex recursive reasoning. Lastly, Custom Data Structures benchmarks, like address book manipulation, represent real-world tasks involving user-defined data and type constraints. However, this benchmark focuses on tests involving classic data structures and refinement types, but it primarily evaluates small-scale implementations.

3.1.5 PSB2

The Second Program Synthesis Benchmark Suite [14] (PSB2) includes 25 varied program synthesis challenges designed to advance automatic programming research. The PSB2 focuses on evaluating and advancing general program synthesis, the ability of automated systems to generate

programs that resemble those written by human programmers. By providing complex and realistic tasks, PSB2 emphasizes the need for systems to handle diverse data types, control flows, and problem-solving techniques. Tasks range from mathematical computations, like finding the factorial digit sum or calculating an Euclidean distance, to practical algorithms for list manipulation, such as splitting vectors or formatting strings. Some problems emulate real-world tasks, like calculating bowling scores, encoding messages using substitution ciphers, and determining Twitter character limits. Many require handling complex control flows and diverse data types—integers, floats, strings, Booleans, and vectors—offering a broad spectrum of programming challenges aimed at testing and improving synthesis systems. This diverse suite drives research toward realistic and generalizable solutions in automated coding. Although this suite is designed to address real-world problems, it does not include tests that involve high collision scenarios with dependent types.

3.2 Program Synthesis Family of Methods

3.2.1 Enumerative Methods

Enumerative methods are a straightforward and often highly effective strategy for program synthesis. They systematically enumerate programs within the underlying search space in a specific order and check each program to determine whether it satisfies the given synthesis constraints. The study of the current main approaches is described below.

SyGuS

Syntax-Guided Synthesis [1] (SyGuS) is an approach in which users provide both a logical specification of a program's behavior and a syntactic template to constrain the search space of potential implementations. The authors explore various synthesis procedures in the pursuit of correct programs, one of which is Enumerative Learning. This method systematically generates candidate programs by incrementally increasing their complexity, whether in terms of size or expression depth, and verifying each candidate against the given specification. While straightforward, this approach starts by enumerating all possible expressions of minimal size based on the provided grammar (syntactic template). It then progressively increases the size of these expressions, generating more complex programs. However, this approach struggles to scale effectively, as the number of possible candidate programs rapidly becomes unmanageable for larger or more intricate tasks.

Synquid

Synquid [25] is a program synthesis framework that automatically generates correct-by-construction recursive functions using polymorphic refinement types to guide the synthesis process. It utilizes these refined polymorphic types to constrain the search space of valid programs. Synquid's key features include its correct-by-construction mechanism, where incomplete programs are type-checked during the synthesis process. If an incomplete program fails the type check, all subsequent

subprograms generated from it are discarded, which significantly reduces the search space. The use of polymorphic refinement types, which combine the flexibility of polymorphic types with the precision of refinement types, provides additional information to narrow down the search space during synthesis further.

However, Synquid has limitations. It requires complete and correct refinement types, along with explicit restrictions on components, functions, and variables that the synthesized function is allowed to use. These constraints make it challenging to scale the framework for real-world applications, as identifying all the correct refinements can be a time-consuming and challenging task.

Finite Tree Automata

Finite Tree Automata [34] (FTA) was developed to automate data completion tasks in tabular formats such as spreadsheets, dataframes, and relational databases, using a program synthesis technique that combines Programming-by-Example (PBE) and program sketching. This method allows users to provide partial programs and input-output examples to guide the system in automating data completion. It employs a combination of Finite Tree Automata and enumerative search to explore the space of all possible programs that fit the examples. The FTA efficiently represents and shares standard parts between similar programs, making the search more efficient. The FTA-based unification algorithm ensures that the final program satisfies all provided examples, while a heuristic search ranks and selects the simplest and most general solution. This hybrid of enumerative and heuristic search ensures both correctness and efficiency in completing missing or incomplete data in tabular formats.

Despite its promising results, this approach faces challenges related to scalability. As the dataset size or table complexity increases, the synthesis process can become less efficient. Although the FTA-based method reduces some computational overhead, the search and unification process may still struggle to scale effectively with extensive datasets or highly complex tasks.

Later, the Metric program synthesis [9] combined bottom-up enumerative search with local repair guided by a distance metric to synthesize programs efficiently. It starts by building an approximate finite tree automaton (XFTA), which represents all candidate programs up to a specific size. During this phase, programs with similar outputs (as measured by a domain-specific distance metric) are clustered into equivalence classes, and only a representative from each cluster is retained. This reduces the search space by focusing on approximate solutions instead of requiring exact matches, which are often rare or computationally expensive to find. Once the global search identifies a program that is "close enough" to the desired behavior, a local search phase refines the program through a series of syntactic adjustments (using domain-specific rewrite rules). These rules incrementally modify the program, guided by the distance metric, to minimize the gap between the current and target outputs. By combining the efficiency of coarse-grained clustering in the global search with the precision of local repair, this approach balances scalability and accuracy, making it suitable for domains where exact matches are hard to achieve directly. Although

this approach has those improvements, it has limitations, such as in the search phase, using rewrite rules does not guarantee finding a globally optimal solution.

3.2.2 Heuristic Methods

In this approach, the heuristics serve as guiding rules or strategies to improve the search performance and reduce the search space. Heuristics can prioritize specific paths, avoid inefficient or incorrect solutions, and speed up the synthesis process by exploiting domain-specific knowledge, patterns, or shortcuts.

Random Search

STOKE [29], automatically generates x86-64 assembly code using Random Search (RS). In the synthesis phase, STOKE leverages a random search driven by Markov Chain Monte Carlo (MCMC) sampling to generate and refine code sequences towards correctness, guided by a correctness-focused cost function. Once a correct solution is synthesized, the system shifts to performance optimization. A key advantage of RS is that it allows STOKE to explore a vast range of potential code sequences without being constrained by predefined rules or patterns.

However, STOKE faces notable limitations. One significant issue is its inability to handle loops, as it only generates code without loops, restricting the complexity of the code it can produce. Additionally, while the stochastic approach is more adept at navigating large search spaces than traditional superoptimizers, it still struggles with scalability, particularly when dealing with larger programs or more complex optimizations.

Simulated Annealing

Simulated Annealing [16] (SA) is a heuristic commonly used in program synthesis that iteratively refines a single candidate program through random mutations, aiming to optimize program fitness. At each iteration, the algorithm explores neighboring solutions, accepting improvements while occasionally accepting worse solutions to avoid getting trapped in local optima. This probabilistic acceptance is controlled by a cooling schedule, which allows SA to balance exploration and exploitation during the search for a correct program. While SA is a powerful technique, its performance is highly dependent on the cooling schedule, which dictates how the temperature decreases over iterations. If the cooling occurs too quickly, the search may get stuck in local optima; if it is too slow, the search can become computationally expensive. Finding an optimal cooling schedule is challenging, and even with penalties applied to longer programs, SA tends to generate increasingly lengthy programs over time, which raises computational costs when evaluating the fitness of these longer solutions.

Genetic Programming

[22] introduced Strongly Typed Genetic Programming (STGP), which uses types to constrain the search space and prevent the generation of invalid programs by ensuring operations are applied

only to appropriate data types. STGP also supports generic data structures and operations, offering more flexible and reusable solutions. While this reduces the search space by filtering out candidates that fail type checking, the evaluation process remains slow, as significant time is spent on evaluating trees that ultimately violate type constraints.

One of the key challenges in Genetic Programming is the vast search space, where the sheer number of operator combinations makes finding a valid solution difficult. Grammar-Guided Genetic Programming [21] (GGGP) was developed to mitigate this issue by reducing the number of possible operation combinations, enabling faster program synthesis. In GGGP, the solution's structure is defined using a grammar, which reduces the search space and guides the selection of sub-functions within the target function. Despite these improvements, GGGP has limitations, particularly when dealing with problems that require more complex, context-sensitive dependencies, as it relies on Context-Free Grammar (CFG), which struggles to represent such dependencies.

A new approach, Refinement Typed Genetic Programming [10] (RTGP), combines aspects of STGP and GGGP. The core concept of RTGP lies in refinement types, which introduce constraints to types that guide and validate solutions. Like STGP, RTGP enforces type safety, ensuring that all solutions conform to specified type rules. However, RTGP takes this further by using dependent types that allow constraints based on values, ensuring only valid solutions are generated.

While it shares GGGP's goal of restricting the search space, RTGP achieves this through types rather than grammar rules, offering more flexibility for complex problems. Refinements are used to select Metahandlers, and it will be based on the Metahandlers to generate trees that adhere to specified refinement types. This integration enables a robust synthesis process, ensuring that only type-safe, constraint-satisfying solutions are generated and evolved. Although this approach introduces structure and efficiency to GP, it can be complex to implement and computationally demanding, balancing precision with resource requirements.

3.2.3 Large Language Models

Program synthesis with LLMs involves leveraging pre-trained large language models to generate code by interpreting natural language instructions, translating them into executable programs, and automating tasks like code completion, bug fixing, or entire program generation based on patterns learned from vast code datasets.

Enumerative with Large Language Models

Enumerative with LLMs [19] is a novel approach to program synthesis that combines LLMs with traditional enumerative synthesis algorithms. The core idea is to leverage the LLMs to guide the enumerative search rather than relying entirely on the LLMs to solve the synthesis problem. Initially, the LLMs are prompted with the formal synthesis task and generate candidate programs. If these candidates are incorrect, they serve as feedback to direct the enumerative search towards more promising areas where the correct solution is likely to be found. However, the effectiveness of this approach heavily depends on the quality of the dataset used to train the LLMs, and

overfitting can negatively impact the synthesis results.

Genetic Programming with Large Language Models

Genetic Programming with LLMs [31] is a hybrid approach that combines Large Language Models (LLMs) with GGGP for program synthesis, seeking to harness the strengths of both methods. In this approach, LLM-generated code is used as a seed in GGGP's initial population, and the evolutionary process is guided by many-objective optimization with similarity measures. While this combination of techniques offers potential, it faces limitations. One challenge is that the seeded information from the LLMs is often quickly lost during the evolutionary process, which hampers the hybrid approach's performance. Additionally, conflicts between the LLM-generated code and GGGP's grammar rules further complicate the synthesis process.

Chapter 4

Parent Metahandler

4.1 Motivation

The current support for dependent types in GE is provided through the Dependent Metahandler, which expresses dependencies only inside the class, e.g., if we had two classes we can't express dependency between two classes. However, in the real-world problem of program synthesis, it is uncommon to work with just one class or to encapsulate all information within a single class. This limitation makes it significantly more challenging to design problems that leverage dependent types.

In Listing 4.1, we aim to have GE generate a random integer and instantiate an object of class B, such that the attribute `elem3` in class B has a lower value than `elem1` in class A. While the Dependent metahandler offers some related functionality, it does not allow attributes of one class to access attributes of another. This restriction can lead to design issues, as illustrated in Listing 4.2, where we are forced to modify our original structure to accommodate the limitations of the Dependent metahandler.

```
@dataclass
class A:
    elem1: Int
    elem2: B

@dataclass
class B:
    elem3: Int #we want it to have a value lower than the value of elem1 of
                    the class A
```

Listing 4.1: Initial structure of the classes

```
@dataclass
class A:
    elem1: Int
    elem2: Annotated[int,
    Dependent("elem1", lambda n: IntRange(0, n))]
```

Listing 4.2: Problem using Dependent Metahandler

Listing 4.3 illustrates how the Dependent Metahandler can be used to generate an ordered

tuple. In this example, three ordered elements are generated: `elem1` is a random number; elem2 is constrained to be greater than `elem1`; and `elem3` must be greater than `elem2`. While this approach works for small or fixed-size objects, such as tuples, it becomes impractical when we want to implement something mutable, like a list, as each new element requires manually adding additional attributes. This limitation makes the solution unsuitable for dynamic or scalable scenarios.

```
@dataclass
class OrderTuple:
  elem1: Annotated[int,IntRange(0, 100)]
  elem2: Annotated[int,
  Dependent("elem1", lambda n: IntRange(n, 100))]
  elem3: Annotated[int,
    Dependent("elem2", lambda n: IntRange(n, 100))]
```

Listing 4.3: Order Tuple with Dependent Metahandler

4.2 Solution

To address the limitation, we introduce a new metahandler called Parent. The Parent metahandler enables a class to access the context of the first matched attribute found in its ancestor nodes, thereby extending contextual capabilities across related classes. This enhancement resolves the earlier issue by ensuring that attributes and behaviors defined in parent classes are accessible to dependent child classes, improving both the flexibility and correctness of the synthesis process.

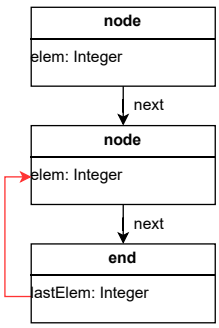

Figure 4.1: Diagram of Parent search flow, the red arrow refers to the target which `lastElem` will depend on.

Figure 4.1 illustrates the operation of the Parent metahandler. In this flow, the objective is to assign a value to `lastElem` such that it is greater than the value of `elem`. To achieve this, we must first retrieve the value of `elem`. The Parent metahandler initiates its search from the current context, specifically, the `end` class and traverses upward through the hierarchy of parent nodes until it finds the first occurrence of an attribute named `elem`. The red line in Figure 4.1 marks the node where this attribute is located.

Additionally, we implement a second metahandler called Parents. Unlike the Parent metahandler, which retrieves only the first matched attribute, the Parents metahandler collects all matched attributes from the ancestor nodes. This behavior is illustrated in Figure 4.2, while the remainder of the search logic remains identical to that of the Parent metahandler.

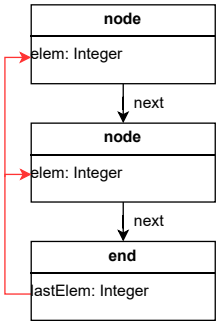

Figure 4.2: Diagram of Parents search flow, the red arrows refer to the targets which `lastElem` will depend on.

With the introduction of these two new MetaHandlers, future evaluations of dependent types in GE will be simplified. In particular, the implementation of benchmarks will become more straightforward, and the readability of the code in complex problems will be significantly improved.

4.2.1 Demonstration

The Parent Metahandler receives two parameters: the first is a string representing the names of attributes(separated by commas), and the second is a function that takes the attribute's value as input and uses this value to do something. Listing 4.4 illustrates how the Parent Metahandler can be applied to solve the problem defined in Listing 4.1 while preserving the original structure. In this example, the Parent Metahandler retrieves the value of `elem1`, passes it to the function, and the function returns a result constrained between 0 and the value of `elem1`.

```
@dataclass
class A:
    elem1: Int
    elem2: B

@dataclass
class B:
    elem3: Annotated[int,
    Parent("elem1", lambda element: IntRange(0, element))]
```

Listing 4.4: Solving the first problem

Listing 4.5 demonstrates the use of the Parent Metahandler to address the issue. Using dependent types and contextual information between nodes, the Parent Metahandler ensures that each

element maintains order by accessing the context of its parent node. This approach not only enforces the ordering constraint dynamically but also allows for the generation of lists of any size without manual intervention. This makes it a more flexible and scalable solution.

```
@dataclass
class LinkedListStop(LinkedList):
  elem: Annotated[int,
  Parent("elem", lambda element: IntRange(element if element != None else 0,
    100))]

@dataclass
class LinkedListGenration(LinkedList):
  elem: Annotated[int,
    Parent("elem", lambda element: IntRange(element if element != None else
    0, 100))]
  next_elem: LinkedList
```

Listing 4.5: Order List with Parent Metahandler

Chapter 5

LLM-GP Hybrid Method

This chapter presents the novel hybrid search method that combines Genetic Programming with LLM, prompt setting for the LLM, LLM selection, and the evaluation of this search method with other search methods.

5.1 Overview

In order to understand the performance of each family of search methods, we need to evaluate them in GE, and the Enumerative search (Enumerative method) and Genetic Programming search (Heuristic method) have already been implemented in the GE [10]. However, the LLMs method was not.

To evaluate the three families, we need to implement the LLM method in GE. Then we implement this method, named the Hybrid Method. The main concept of this method is inspired by the approach detailed in [31], which leverages the capabilities of LLMs to generate the initial population. Afterward, the robust evaluation and optimization mechanisms of GP are employed to guide the process towards better solutions.

Figure 5.1 provides a simple and illustrative example of the workflow for this method. The core idea is to combine the strengths of LLMs for generating diverse, high-quality initial candidates with the structured refinement capabilities of GP to navigate the search space effectively.

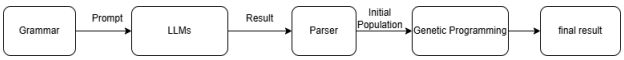

Figure 5.1: Flow Of Hybrid method

To achieve this, the design utilizes prompts to guide LLMs in generating meaningful and relevant initial candidates based on the problem specification. Additionally, it involves developing a parser to convert the LLMs' output into a format compatible with the existing population structure in GE, thereby ensuring compatibility.

5.1.1 Prompt setting

An important aspect of leveraging LLMs in this approach is practical prompt design. The prompt plays a critical role in guiding the LLMs to generate meaningful and diverse initial populations. To facilitate this, we have chosen a zero-shot prompting technique to accelerate the process. The prompt is divided into two key components and an example Listing 5.1:

- **Request part**, contains the task for the LLM, such as generating an initial population. Additional requirements can also be included, such as requesting more diverse candidates or asking the LLM to follow specific logic during generation.
- **Code part**, includes the structural details of the grammar, obtained from the GE classes, to provide the LLM with a clear understanding of the rules and format for generation.

````
"""Given the following Python code, that uses dataclasses and custom
annotations, please create 100 instantiations of class
<class '__main__.Expr'>, one per line and with diversity.

```
@dataclass
class Expr(ABC):
    pass
@dataclass
class Sum(Expr):
    left: Expr
    right: Expr
@dataclass
class Num(Expr):
    value: Annotated[int, IntRange(0,9)]
```
"""
````

Listing 5.1: Example of prompt

5.1.2 Parser

After receiving the result from the LLMs, the output must be parsed from a raw string into structured populations. To simplify this step, the prompt is carefully designed to instruct the LLMs to generate instances without using keyword arguments and to focus strictly on producing clean, population-only data. This ensures that the output is free of unnecessary metadata or formatting, making it easier to parse.

Following the example in Listing 5.1, the output will be in this format *"Sum(Num(50), Sum(Num(75), Num(90)))"*, and then it can be parsed into a tree structure, as illustrated in Figure 5.2. Finally, convert the tree structure into a format suitable for further processing in the Genetic Engine.

5.2 Model Selection and Comparison

The first step is to choose LLMs. The first criterion is that the model must be open source; then, it is considered whether the model has a lightweight version. If not, we will not choose it. Finally,

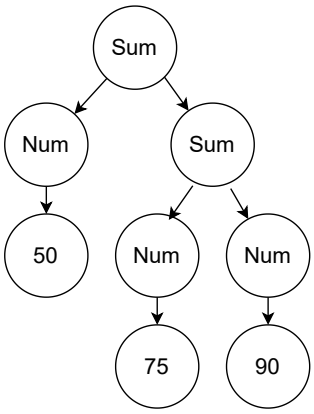

Figure 5.2: Tree structure

we will consider the performance of the model, which includes the execution time and the quality of the response. We chose those LLMs and theirs versions:

- Llama3.1:8B [8], a Multilingual LLM deployed by Meta, is a very used open-source LLM and also offers a lightweight version of the model;
- Llama3.3:70B [8], an optimized version of Llama 3.1 that only has 70B of parameters and has similar performance with Llama3.1 405B;
- Qwen2.5:14B, 32B [36, 37], offers lightweight models and powerful performance.
- DeepSeek-R1:8B, 14B, 32B [7] distill model, as a recent model, and received much discussion, so that it might yield some interesting results. The 8B version is distilled Llama, and the 14B and 32B versions are distilled Qwen.

5.2.1 Benchmark setting

To evaluate the performance, accuracy, and diversity of the models, we have chosen two benchmarks:

- Simple benchmark, a problem with a class that has five attributes, aim to evaluate the basic performance of LLMs generating objects;
- Mario Level Benchmark, contains various classes to evaluate the performance of generating multiple classes;

Simple Benchmark

The simple benchmark is a problem with only one class, containing two different Metahandlers from GE: `IntRange` and `ListSizeBetween`, as shown in Listing 5.2.

This benchmark aims to evaluate the understandability of LLMs and assess whether they correctly interpret and adhere to the restrictions imposed by the Metahandlers. The evaluation is based on execution time, success rate, and the total number of instances generated.

```
@dataclass
class A:
    nums: Annotated[int, IntRange(350,800)]
    num2: Annotated[int, IntRange(30,50)]
    num3: Annotated[int, IntRange(136,190)]
    num4: Annotated[list[int], ListSizeBetween(2,6)]
    num5: Annotated[int, IntRange(-100,0)]
```

Listing 5.2: The simple benchmark

Mario Level Benchmark

This benchmark evaluates the performance of LLMs in generating game levels. The objective is for the LLM to generate a Level, which consists of two key parameters: `chunks` and `enemies`.

The chunks parameter is a list of instances from the Chunk class. Chunk is an abstract class, and its subclasses are shown in Figure 5.3.

The enemies parameter consists of a collection of Enemy objects, each defined by their x position and kind.

This benchmark assesses an LLM's ability to generate a structured game level by correctly handling multiple interrelated classes and ensuring that constraints and relationships are properly followed.

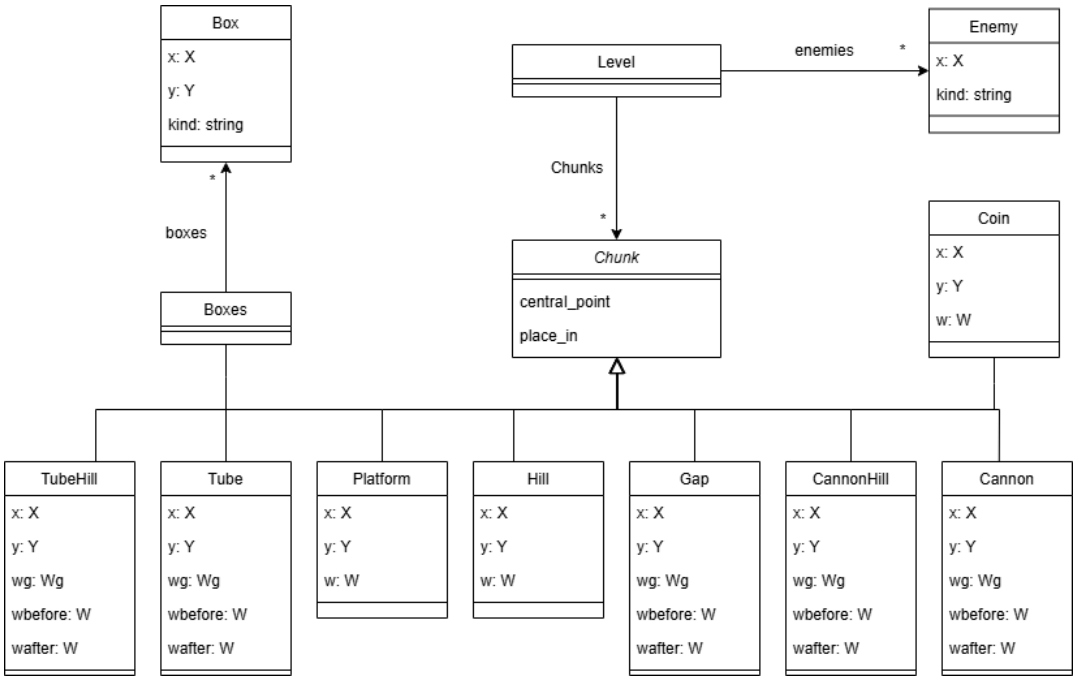

Figure 5.3: the UML of the Mario Level Benchmark

```
X = Annotated[int, IntRange(5, 95)]
Y = Annotated[int, IntRange(3, 5)]
W = Annotated[int, IntRange(3, 15)]
Wg = Annotated[int, IntRange(2, 5)]
```

Listing 5.3: Metahandler specification

5.2.2 Result

The benchmarking environment is running on an AMD Ryzen Threadripper 3960X 24-Core Processor with 98 GB of RAM and a Nvidia GeForce RTX 3090 Ti 24 GB graphics card. The LLMs are executed using Ollama, which uses the Ollama Python Library to communicate with LLMs.

For each model, we perform 100 executions, prompting it to generate 10, 25, 50, and 100 instances. The results are presented in Figure 5.4, Figure 5.5, and Figure 5.6, where we analyze the time required to generate, the total number of generated instances, and the success rate of the generation. An instance is considered successful if it can be correctly compiled by Python. However, we do not evaluate whether the result adheres to the metahandler's constraints.

Llama3.1 By analyzing the execution time results in Figure 5.4, a consistent increase in runtime can be observed for the LLM: as the number of requested generations increases, execution time grows proportionally in both the Simple Benchmark and the Mario Benchmark. As expected, the Mario Benchmark requires more time due to the greater complexity of the task.

When examining the number of generated instances (Figure 5.5), the model produces the requested number with only minor variation for 10 and 25 instances. However, for larger requests (50 and 100 instances), the outputs no longer match the requested values, yielding approximately 40 instances in the 50-instance case and about 80 in the 100-instance case. The discrepancy is even more pronounced in the Mario Benchmark.

Regarding the success rate (Figure 5.6), the model achieves a moderate rate for the Simple Benchmark and approximately 80% for the more complex Mario Benchmark. Interestingly, performance is higher on the more complex task. This outcome may be explained by two factors: (i) complex problems are closer to real-world scenarios and may therefore be overrepresented in the model's training set, or (ii) in the Mario Benchmark, the class names provide informative cues that facilitate the generation of valid instances, whereas in the Simple Benchmark, the class names are less informative.

Overall, these results establish this model as a baseline for evaluating subsequent LLMs. Future models that do not achieve equal or superior performance will not be considered for further experimentation.

Llama3.3 By observing Figure 5.4, when we asked the LLM to generate 10 instances, it failed to return the results within a reasonable time. This can be explained by the fact that our benchmarking platform does not have enough VRAM to execute Llama 3.3 efficiently. When running the 3.3 model, it reaches a bottleneck, utilizing both the GPU and CPU; half of the model runs

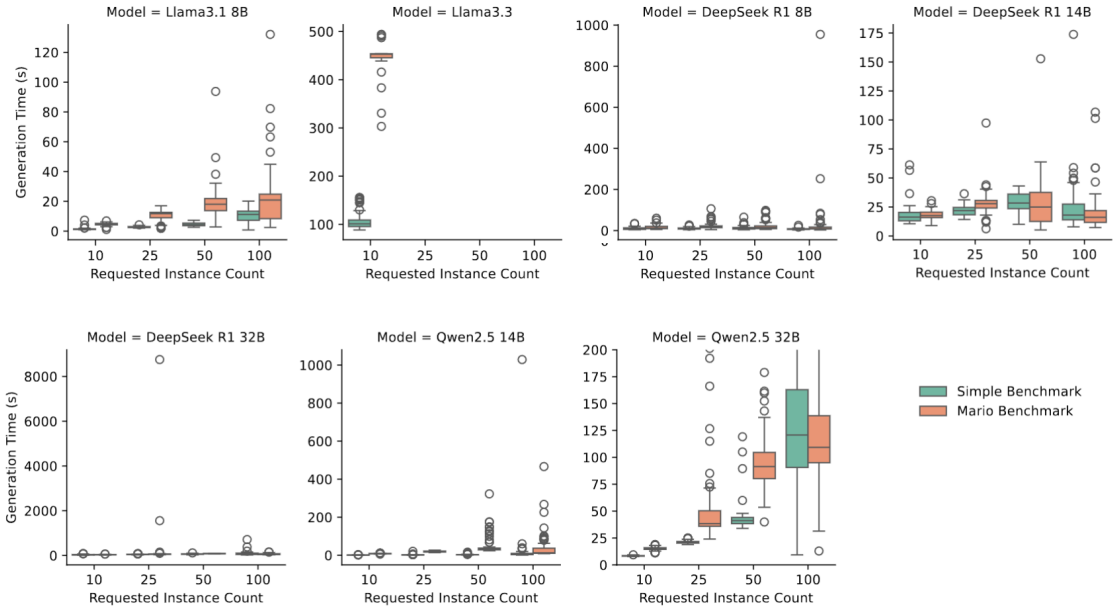

Figure 5.4: The generation time of each LLM based on the number of instances count. The request instance count represents the number of instances requested in the prompt for the LLM to generate.

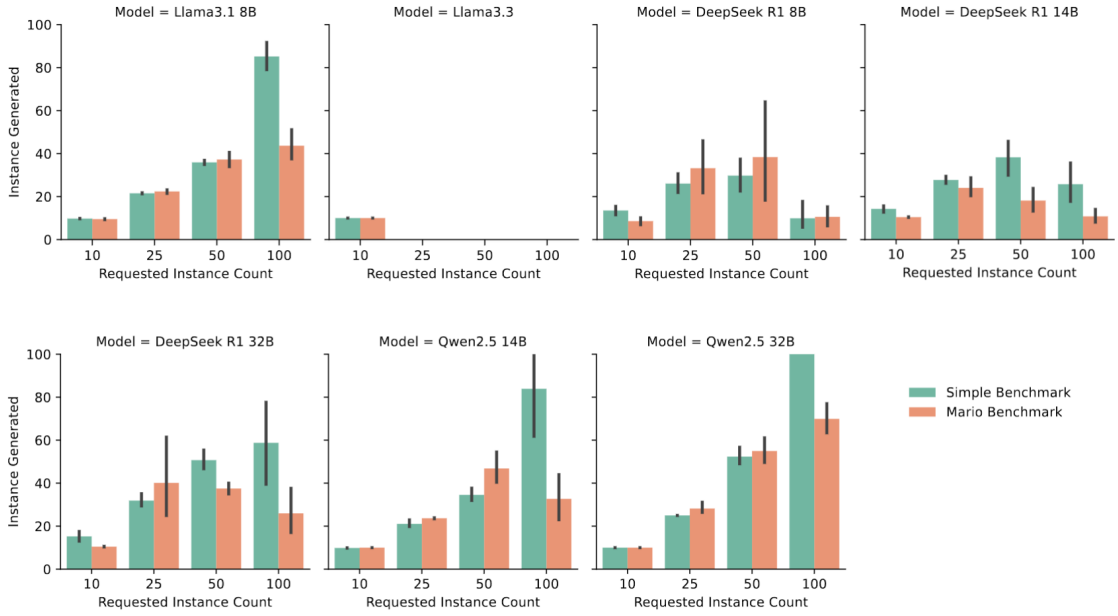

Figure 5.5: The instances generated by LLM based on the number of instances count. The request instance count represents the number of instances requested in the prompt for the LLM to generate.

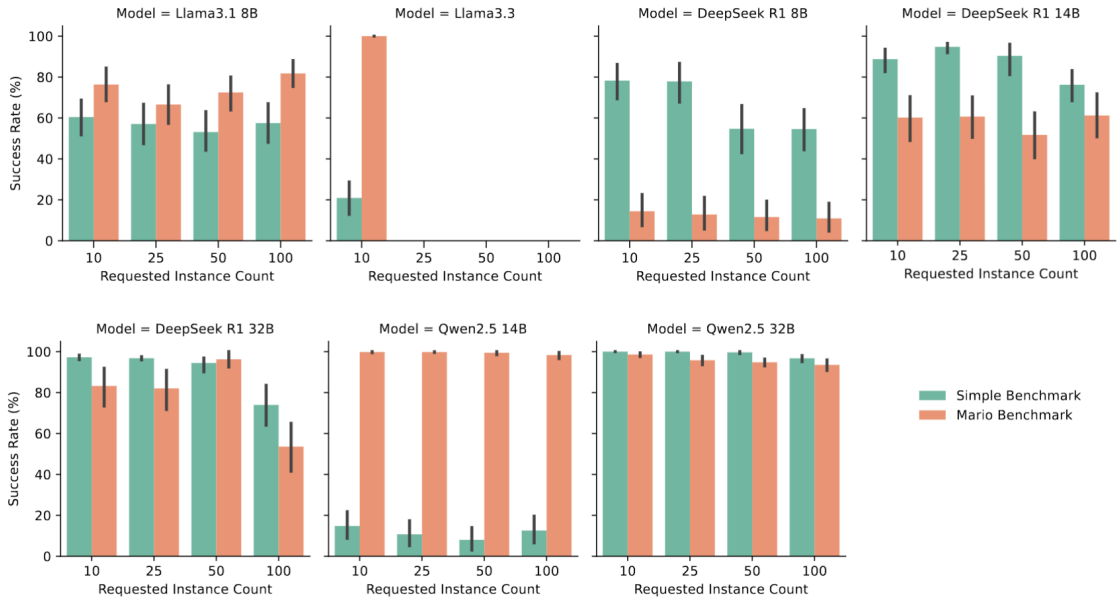

Figure 5.6: The Success Rate of instances generated by LLM based on the number of instances count. It is counted as a success when the generated instance is compilable by Python; the request instance count represents the number of instances requested in the prompt for the LLM to generate.

on the GPU, while the other half runs on the CPU, leading to inefficiency. As a result, we decided to abandon the use of Llama 3.3 for generating more instances and only obtained results for generating 10 instances. The number of instances generated was satisfactory, but the success rate was unacceptable. Given its size, it should have outperformed Llama 3.1. Upon further analysis, we found that most generations only produced a class with four parameters, while our class requires five parameters. The reason for this discrepancy remains unclear. However, in the Mario Benchmark, the model achieved a full success rate, which was expected, as Llama 3.3 is a new version of Llama 3.1 and was trained with the same dataset, making this outcome understandable. Consequently, we decided to eliminate Llama 3.3 from further steps due to the excessive time it took.

DeepSeek By observing the DeepSeek R1 models, we note that their execution times of the 8B and 14B versions are similar to those of Llama 3.1. In the DeepSeek models, a pattern emerges: the execution time increases until 50 instances are requested, but then decreases when 100 instances are requested. For the 8B and 14B versions, the execution times for different requests are comparable. One reason for this may be that generating the result is not the primary challenge of this model; instead, generating the reasoning part is. This is why the execution times for the 8B and 14B models are similar.

Although the execution time seems strange, the number of instances generated is even more. For requests of 10 and 25 instances, the model generates the correct number. However, when asked to generate more instances, the number does not increase proportionally, as observed with 50 and

100 instances. This effect explains the reduction in time when generating 100 instances. Up to 50 instances, the execution time either increases or remains the same. However, when asked to generate 100 instances, all of the DeepSeek R1 models show a decrease in execution time, which corresponds to a decrease in the number of instances generated.

To understand why this happens, we analyzed the output and identified a characteristic of the DeepSeek model that can be described as ”lazy and smart.” This means it does not attempt to generate all the requested instances, as shown in Listing 5.4. Instead, it appears to ”pretend” that it has generated all the instances, whereas in reality, it generates fewer. Meanwhile, other models attempt to generate all the requested instances.

The success rate results show a consistent pattern, with better rates for simpler problems and lower rates for more complex problems. However, a downward trend occurs again when generating 100 instances, where there is a decrease in the success rate for the 14B and 32B models.”

```
A(350, 30, 136, [1,2], -100)
A(400, 40, 150, [3,4,5], -50)
A(500, 50, 170, [6,7,8,9], 0)
A(600, 35, 180, [10,11], -25)
A(700, 45, 190, [12,13,14,15,16], -75)
A(351, 31, 137, [17,18], -99)
A(352, 32, 138, [19,20,21], -98)
A(353, 33, 139, [22,23,24,25], -97)
A(354, 34, 140, [26,27,28,29,30], -96)
A(355, 35, 141, [31,32,33,34,35,36], -95)
... (continue until 100 instances)
```

Listing 5.4: An Illustration of answer of the DeepSeek

Qwen The Qwen2.5 14B model demonstrated extreme speed in the Simple Benchmark but was significantly slower in the Mario Benchmark, with several extreme outliers in execution time when generating 50 and 100 instances. Regarding the number of instances generated, the results are as expected for 10 and 25 instances, where the model performs well. However, for larger requests, the number of instances generated is incorrect in both benchmarks. As for the success rate, the model’s low success rate in the Simple Benchmark and high success rate in the Mario Benchmark are surprising. However, this pattern may share the same explanation as the Llama models—both may have been trained on similar problem sets, leading to these results.

The Qwen2.5 32B model follows a linear pattern, where execution time increases with the number of requested instances. While the model maintains stable time performance for 10-instance requests, higher requests exhibit significant variability and a sharp increase in execution time. In terms of instance generation, the model tends to produce more instances than requested, as observed when generating 100 instances, where the average number of instances generated exceeded the expected limit in the Simple Benchmark. Finally, the success rate results are impressive, with consistently high performance, making this model a strong candidate for the next phase of experimentation.

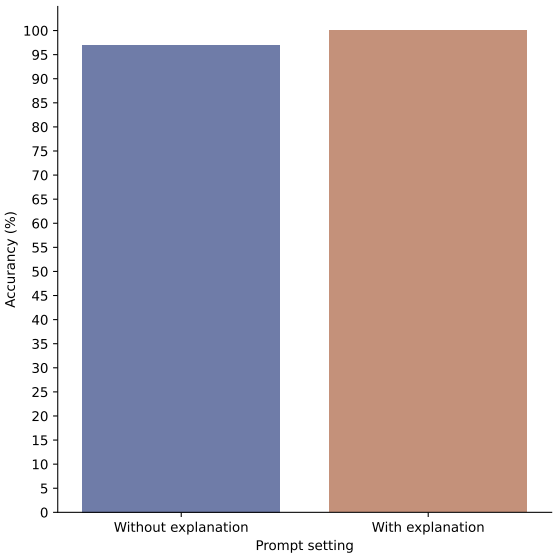

Figure 5.7: The accuracy of the 100 instances generated by Qwen with explanation of the meta-handlers and without

5.3 Comparison with other methods

In this comparison, our primary objectives are to determine whether the Hybrid method can outperform the standalone GP or the Enumerative method and whether LLMs can generate more diverse or higher-quality populations than random generation.

For benchmarking, we continue to use the Mario Benchmark. However, this time, our focus is on two key metrics: fitness score, where a higher value indicates better performance, and entropy score, which evaluates the diversity of the generated instances.

To conduct this analysis, we collected results from 30 executions, each using a different seed and a time limit of 10 minutes per execution.

5.3.1 Final Model & Prompt Configuration

We chose the Qwen2.5 32B model as the final LLM for the Hybrid method due to its strong performance in previous benchmarks.

In the prompt setting, we decided to request 10 instances at a time because, based on the results in Figures 5.4 and 5.6, generating 10 instances is both faster and more precise. For example, if 100 instances are required, we will make 10 separate requests until all instances are obtained.

Additionally, to evaluate the model's ability to understand Metahandlers, that is, whether the generated results adhere to the Metahandler's restrictions, we conducted 100 benchmark executions, comparing its accuracy with and without a Metahandler explanation in the prompt. When a Metahandler explanation is provided, the prompt includes details about what the Metahandler does. Conversely, when no explanation is given, the prompt does not contain this information.

Based on the results in Figure 5.7, we observe that even without an explanation, most instances still follow the Metahandler's restrictions, though some failures occur. In contrast, when an ex-

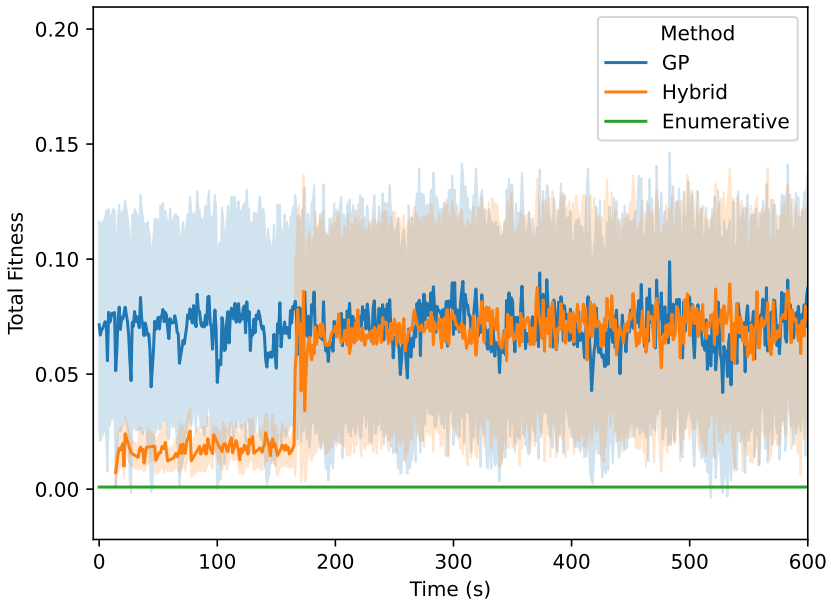

Figure 5.8: The evolution of fitness over time

planation is provided, the results are 100% accurate. Therefore, we have decided to include the Metahandler explanation in the prompt to ensure complete adherence to the restrictions.

5.3.2 Result

Both the GP and Hybrid methods exhibit fluctuating fitness values, Figure 5.8, with GP maintaining relatively stable performance throughout. At the same time, GP demonstrates a steady increase in entropy, Figure 5.9, indicating continuous exploration of the search space. In contrast, the Hybrid method begins with lower fitness during the phase where the LLM contributes, but then experiences a sharp increase around time step 200 when GP takes over, after which its performance stabilizes. A similar effect is observed in Figure 5.9, where the entropy is initially lower because the LLM requires approximately 200 seconds to generate 100 instances and the GP already has generated a lot more instances. However, a significant increase occurs around the 200-second mark when GP takes over, leading to a faster increase in the number of instances.

In contrast, the Enumerative method maintains a nearly constant, low fitness level. This is because only a single instance is recorded as the best instance, whereas in the other methods, multiple instances are recorded as the best instance. This result highlights the limited effectiveness of the Enumerative approach, as expected.

5.4 Discussion & Limitations

The results presented in Section 5.3.2 suggest that the GP method is the most effective approach among those evaluated. In contrast, the Hybrid method does not benefit significantly from the con-

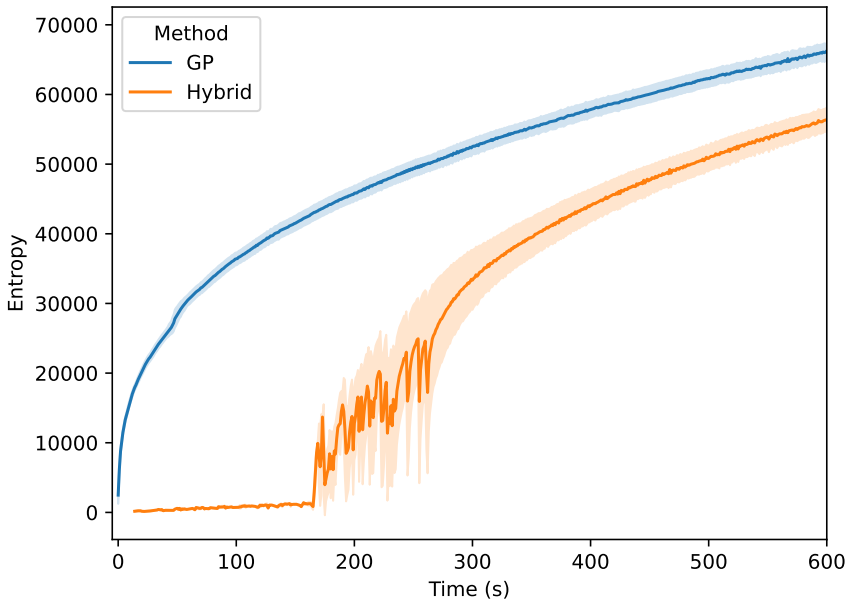

Figure 5.9: The evolution of the diversity of instances generated over time

tribution of LLMs, while the Enumerative method proves to be highly inefficient. Although the current performance of the Hybrid method is not satisfactory, it represents a novel combination of traditional program synthesis with emerging approaches. It thus remains a promising direction for future exploration. Importantly, this method has several limitations that highlight clear opportunities for improvement.

Prompt engineering Prompt design is one of the most critical aspects when working with LLMs. Even minor modifications to the prompt can lead to significant changes in results. In this work, we did not evaluate different prompt engineering techniques. Exploring alternative strategies for prompt construction, therefore, represents an important direction for optimizing the Hybrid method.

Temperature setting Another key factor in LLM performance is the temperature parameter, which controls the randomness of model outputs. Lower temperatures yield more deterministic responses, while higher temperatures increase variability and creativity in the generated text. In this work, temperature settings were not explored, which may have limited the Hybrid method's potential. Future work should investigate how different temperature values influence synthesis quality and consistency.

Chapter 6

Dependent vs Independent Refinement Types

This thesis aims to evaluate the performance gains achieved through the use of Dependent Types in comparison to Refinement Types. To investigate this, we first conducted an evaluation within GE, which provides a low-level environment that has neither wrong types nor infinite search space, and in the next chapter, we extended the evaluation to the functional programming language Aeon.

6.1 Problem Introduction

The problem chosen for this evaluation is the spring mass system [32]. This system consists of two masses, m_1 and m_2, connected in series by two springs with spring constants k_1 and k_2, and their natural lengths L_1 and L_2, respectively. The first spring is attached to a fixed wall at one end and to mass m_1 at the other. The second spring connects m_1 to m_2, as illustrated in Figure 6.1. The masses are restricted to move along the x-axis, and there is no friction. The positions of m_1 and m_2 along the x-axis are denoted by x_1 and x_2, and their velocities by v_1 and v_2, respectively. The goal of this problem is to predict the positions and velocities of both masses based on time.

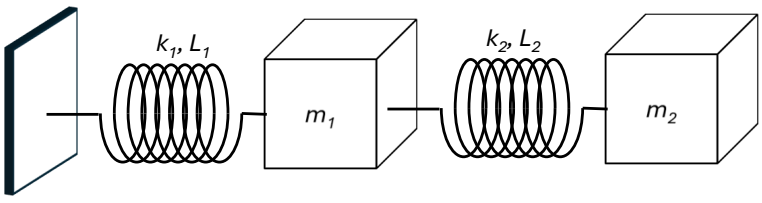

Figure 6.1: illustration of the spring mass system

Since the system is frictionless, its total mechanical energy, E, defined as the sum of the kinetic energy of the masses and the elastic potential energy of the springs, is conserved. This problem serves as a case study to illustrate and compare the performance of Refinement Types and Dependent Types.

6.2 Evaluating Setting

For the evaluation, we set the constant parameters as follows: m_1 = 1 kg, m_2 = 1 kg, k_1 = 5 N/m, k_2 = 2 N/m, L_1 = 0.5 m, L_2 = 0.5 m, and the time interval between two consecutive states to 0.05 s.

After completing the physical parameters of the system, we now present its structure, as shown in Listing 6.1. The implementation defines a class `State`, which represents the system's state at a specific time and stores the parameters to be predicted. Additionally, a class `Status` is defined to contain a list of `State` instances and to predict the subsequent 50 states.

```
@dataclass
class State:
    x1: float
    x2: float
    v1: float
    v2: float

@dataclass
class Status:
    status: Annotated[list[State], ListSizeBetween(50,50)]
```

Listing 6.1: Structure of the system

6.2.1 Refinement Types Structure

The code in Listing 6.2 defines a state that uses refinement to restrict the search space. The refinement values are determined by analyzing the dataset to identify the minimum and maximum values of each attribute, which are then rounded to the first decimal place.

```
@dataclass
class State:
    x1: Annotated[float, FloatRange(-0.7,1.7)]
    x2: Annotated[float, FloatRange(-2.6,2.6)]
    v1: Annotated[float, FloatRange(-0.6,2.5)]
    v2: Annotated[float, FloatRange(-1.7,2.2)]
```

Listing 6.2: Class State using Refinement Type Restrict Case

We also implemented additional refinements, as the configuration in Listing 6.2 is overly restrictive, making it suitable only for our specific dataset and not generalizable. The alternative implementation in Listing 6.3 is more general, allowing the interval bounds to be adjusted. Increasing these bounds makes the refinement applicable to a broader range of data, though at the cost of a longer search time.

```
@dataclass
class State:
    x1: Annotated[float, FloatRange(-3,3)]
    x2: Annotated[float, FloatRange(-3,3)]
    v1: Annotated[float, FloatRange(-3,3)]
    v2: Annotated[float, FloatRange(-3,3)]
```

Listing 6.3: Class State using Refinement Type General Case

6.2.2 Dependent Types Structure

The approach using dependent types leverages the fact that attributes can access information from the previously generated state via the parent Metahandler. This information is then used to compute the attribute values, as illustrated in Listing 6.4.

```
@dataclass
class State:
    x1: Annotated[float, Parent("status", lambda s: classify_position(s, 0))]
    x2: Annotated[float, Parent("status", lambda s: classify_position(s, 1))]
    v1: Annotated[float, Parent("status", lambda s: classify_velocity(s, 0))]
    v2: Annotated[float, Parent("status", lambda s: classify_velocity(s, 1))]
```

Listing 6.4: Class State using Refinement Type Restrict Case

For position prediction, the function `classify_position` uses the previous state and applies the **uniform motion equation**

$$x_{next} = x_{previous} + v_{previous}\Delta t$$

In this work, Δt is set to 0.05. The predicted position is then considered with an uncertainty of 0.1, so that the resulting value lies within the interval $[x_{next} - 0.1, x_{next} + 0.1]$.

Velocity prediction, handled by the `classify_velocity` function, is more involved. First, the acceleration in the previous state is computed according to:

$$\begin{cases} acc_1 = \frac{-k_1(x_1 - L_1) + k_2(x_2 - x_1 - L_2)}{m_1} \\ acc_2 = \frac{-k_2(x_2 - x_1 - L_2)}{m_2} \end{cases} \tag{6.1}$$

Once the acceleration is obtained, the velocity is computed using the **first equation of motion**:

$$v_{next} = v_{previous} + acc_{previous}\Delta t$$

Here, Δt is again 0.05, and an uncertainty of 0.1 is applied, analogous to the position prediction step.

6.2.3 Fitness Criteria

Having established the physical parameters of the system and defined its structure, we now specify the fitness criteria used to assess the performance of the evaluated approaches.

The first fitness criterion, referred to as **Energy Error**, is based on the **total mechanical energy** E. For each predicted state, the total energy is computed and compared with the corresponding energy value from the dataset. The squared difference is then calculated for each of the 50 predicted states, and these values are summed:

$$\sum_{n=1}^{50} (E_{state} - E_{data})^2$$

Here, E_{state} is the energy computed from the predicted state, and E_{data} is the energy from the dataset. This metric measures the squared error in energy, with lower values indicating better performance. By focusing on energy conservation, this criterion is expected to yield accurate results when comparing overall system energy.

The second fitness criterion, termed **Parameter Error**, evaluates accuracy directly in terms of the physical parameters x_1, v_1, x_2, v_2. In this case, the mean squared error across these four parameters is computed for each predicted state, and the results are summed over all 50 states:

$$\sum_{n=1}^{50} \frac{(x_{1(state)} - x_{1(data)})^2 + (v_{1(state)} - v_{1(data)})^2 + (x_{2(state)} - x_{2(data)})^2 + (v_{2(state)} - v_{2(data)})^2}{4}$$

This criterion measures the mean squared error of the predicted positions and velocities relative to the dataset values. Lower values again indicate better performance, with this metric expected to yield accurate predictions when the focus is on matching the system's physical parameters directly.

The third fitness criterion, denoted **Both Criteria**, is not a new metric, but rather a combination of the previous two. It simultaneously considers both the energy-based error and the parameter-based error, thereby averaging performance across energy conservation and parameter accuracy.

6.3 Results

Once the evaluation settings were established, we divided all experiments into three categories: **Refinement Types (Restricted)**(configured in Listing 6.2), **Refinement Types (General)** (configured in Listing 6.3), and **Dependent Types** (configured in Listing 6.4). We employed GP as the search method and executed the program under two different configurations.

In the first configuration, referred to as the 10-minute configuration, the time limit was set to ten minutes, and the program was executed thirty times with different random seeds. This repetition reduces the influence of outliers caused by exceptionally favorable initial populations, which can yield unrealistically high fitness scores that do not reflect actual performance. The results from these runs were then averaged and reported.

In the second configuration, referred to as the 60-minute configuration, the time limit was set to sixty minutes, and the program was executed only once. This setup was designed to observe the performance evolution over a longer period. Due to the extended runtime, multiple repetitions were not feasible.

6.3.1 The Total Number Evaluated

By analyzing Figure 6.2 for the 10-minute configuration, we observe that when using Energy Error as the fitness criterion, both refinement type approaches achieve a similar number of evaluated instances, and both exceed the number achieved by the dependent types. Specifically, the refinement type approaches evaluate approximately 3.3 times more instances than the dependent types. This

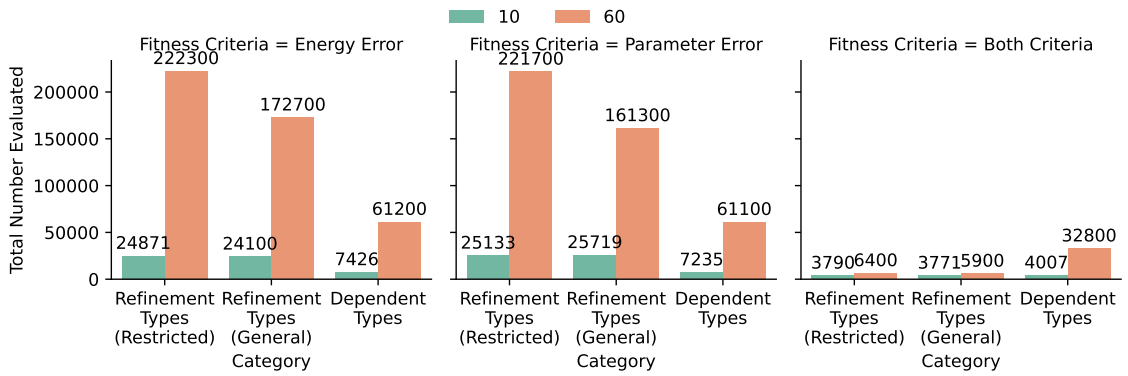

Figure 6.2: **Comparative analysis of the total number of evaluated instances.** The values 10 and 60 correspond to the first and second configurations, respectively. The x-axis categories are: refr(Refinement Types Restricted), refg(Refinement Types General), and dep(Dependent Types)

difference is expected, as dependent types require additional time to generate each instance due to their type dependencies.

When extending the runtime to sixty minutes (60-minute configuration), the Refinement Types (Restricted) approach shows a ninefold increase in the number of evaluated instances compared to the 10-minute run, indicating that it continues to generate new instances consistently over time. In contrast, the Refinement Types (General) approach shows only a sevenfold increase. This smaller growth can be attributed to its less restrictive constraints. At some point, it struggles to generate new instances with improved fitness, shifting GP activity toward mutation and crossover rather than instance generation. The Dependent Types approach evaluates fewer instances overall than either refinement type approach, but its increase from 10 to 60 minutes is also ninefold, similar to the restricted refinement type. This again suggests that more restrictive constraints can sustain a steady flow of improved instances, although the generation rate is likely to decrease if the runtime is extended further.

When switching to Parameter Error as the fitness criterion, the trends are similar to those observed with Energy Error. This indicates that, when treated as a single objective, the two criteria lead to comparable performance in terms of the number of instances generated.

However, when using Both Criteria(minimizing both at the same time), the results change markedly. In the 10-minute configuration, both refinement-type approaches experience a significant drop of around ten times fewer instances compared to the single-criterion runs. This effect is also observed in dependent types, but to a lesser extent. These results suggest that refinement types encounter a performance bottleneck within the first ten minutes. A consistent pattern emerges: the more restrictive the type system, the higher the number of evaluated instances.

In the 60-minute configuration, this bottleneck persists for refinement types. The Refinement Types (Restricted) approach evaluates 6400 instances, whereas the Refinement Types (General) approach evaluates only 5900. In contrast, the Dependent Types approach has not reached the bottleneck yet and continues to generate improved instances throughout the runtime.

6.3.2 The Fitness Evaluation

On the other hand, when examining the fitness function values, we can draw different conclusions.

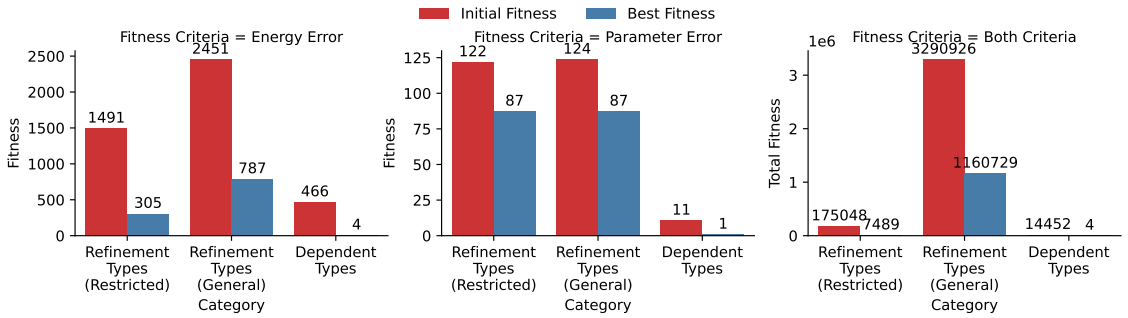

Figure 6.3: **Comparative analysis of the Initial Fitness with the Best Fitness in the 10-minute configuration.** The Initial Fitness corresponds to the first generated instance, averaged over thirty different instances. The Best Fitness represents the best-performing instance, also averaged over thirty different instances.

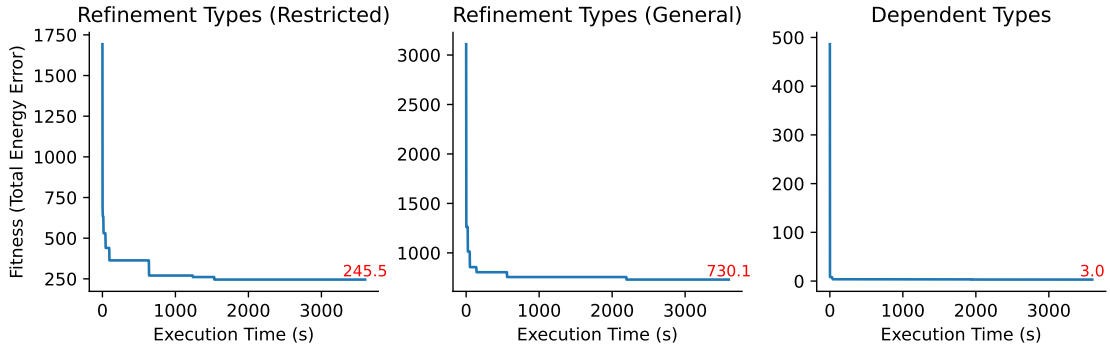

Figure 6.4: **Comparative analysis of the Total Energy Error Fitness by time in the 60-minute configuration.** The performance of different approaches only considering the Total Energy Error

For Energy Error, in the 10-minute configuration, shown in Figure 6.3, the Refinement Types (General) approach performs the worst. It starts with the highest fitness value and, by the end of the run, its best fitness remains the poorest among the three approaches. Extending the runtime to sixty minutes Figure 6.4 does not significantly change the situation; the general refinement type still struggles, showing only minor improvements.

The Refinement Types (Restricted) approach achieves a middle score in Figure 6.3. Here, we can observe a substantial early improvement: the fitness of the first generated instance is already about 60% better than the initial value, and the best result is nearly 140% better than that of the general refinement type. However, in the sixty-minute run Figure 6.4, the restricted refinement type encounters the same difficulty as the general version after approximately 2,000 seconds; no significant decrease in the fitness value is observed.

The Dependent Types approach delivers the strongest performance of the three in Figure 6.3. It starts with the lowest fitness value, and after ten minutes, achieves a best fitness score of 3,

indicating that the applied restrictions are appropriately defined. However, in the sixty-minute configuration Figure 6.4, even though this approach continues to evaluate a substantial number of instances Figure 6.2, the improvement is minimal; after 50 minutes, the fitness decreases by only one unit.

Figure 6.5: **Comparative analysis of the Parameter Error Fitness by time in the 60-minute configuration.** The performance of different approaches only considering the Parameter Error

For Parameter Error, in the 10-minute execution Figure 6.3, the performance of both refinement types is surprisingly identical. From this result, for Parameter Error, the additional restrictions in the Refinement Types (Restricted) approach do not provide a significant advantage, as both approaches achieve precisely the same best fitness score. In contrast, the Dependent Types approach starts with a low fitness value and ends with a score of 1. This outcome is expected, as its dependencies are specifically designed to constrain parameter values.

When moving to the 60-minute configuration Figure 6.5, the Refinement Types (Restricted) approach shows most of its improvement within the first 600 seconds, followed by only one further observable increase in fitness. In the Refinement Types (General) approach, the situation is different: fitness improves steadily until around 2,000 seconds, at which point progress slows, but by then it surpasses the performance of the restricted version. Finally, the Dependent Types approach exhibits substantial improvement during the first 1,000 seconds, after which progress becomes very slow.

Lastly, for Both Criteria, in the 10-minute execution Figure 6.3, the Refinement Types (General) approach appears to struggle significantly, producing a fitness value far higher than the other two approaches. This is likely due to the larger search space and increased complexity of handling two objectives simultaneously. However, by examining its best fitness, we can infer that, with additional runtime, better results might be achievable. The Refinement Types (Restricted) approach also does not achieve powerful results, but its fitness is already much lower than that of the general version, and its best fitness shows gradual improvement over time. The Dependent Types approach again achieves both the lowest starting fitness and the lowest best fitness, with an impressive best score of 4.

In the 60-minute configuration Figure 6.6, we can observe the evolution of the fitness values in more detail. For the Refinement Types (Restricted) approach, total fitness improves continuously.

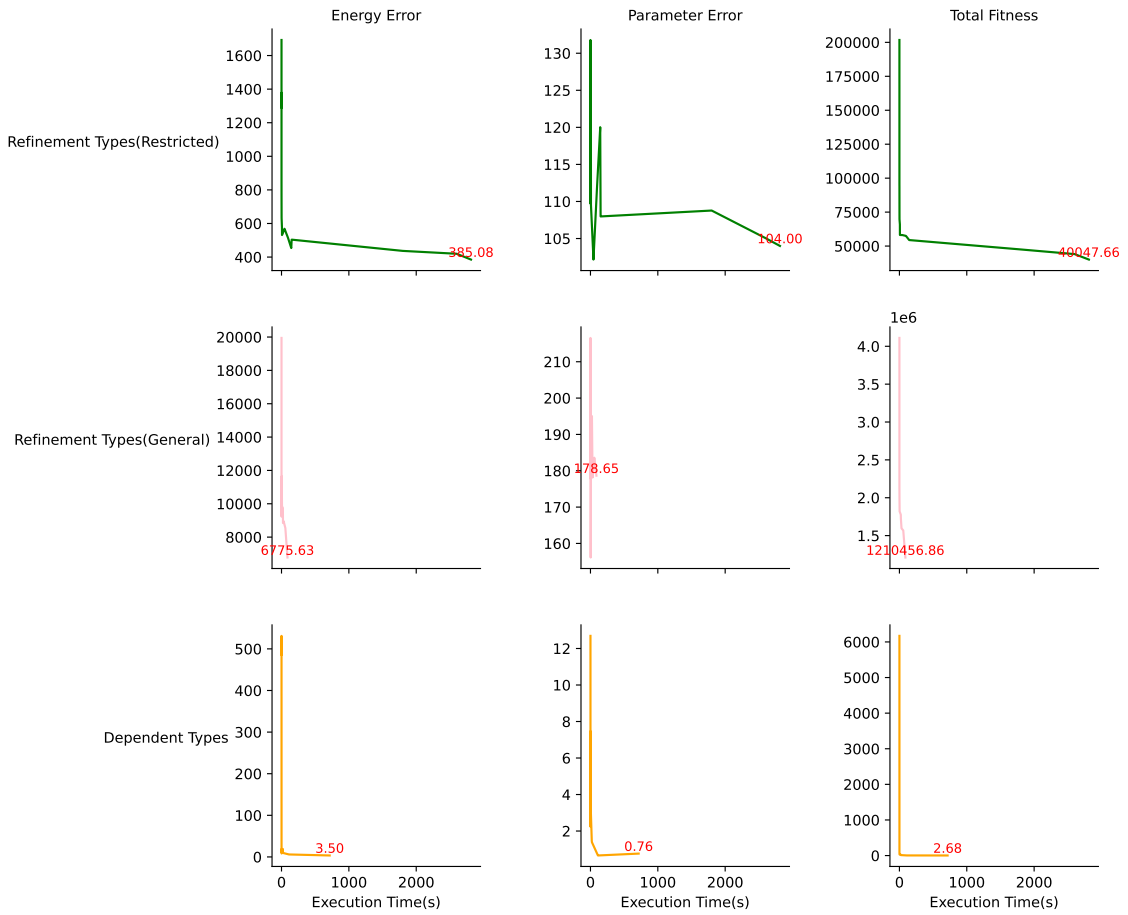

Figure 6.6: **Comparative analysis of Both Criteria over time in the 60-minute configuration.** Total Fitness represents the combined area of the Energy Error and Parameter Error.

Interestingly, improvement in the total fitness does not always require both individual criteria to decrease simultaneously. Initially, the parameter error increases slightly, while the energy error continues to decrease. For the Refinement Types (General) approach, the plotted line records only the very initial data points; after this point, no further improvements in total fitness are registered, highlighting the difficulty it faces in progressing further. Finally, for the Dependent Types approach, data is recorded only during the first 1,000 seconds. In this case, however, the reason is different: the approach quickly reaches a very low fitness value, after which finding better instances becomes extremely difficult. While this demonstrates strong initial performance, it also reveals the challenge of achieving further improvements once the fitness is already near optimal.

6.3.3 The Parameter comparision

Finally, when comparing the best individual (the one with the lowest fitness score) against the dataset values Figure 6.7, we find further evidence that both refinement type approaches deliver lower performance. Specifically, in the Energy Error case, even when energy error is used as the fitness criterion, the results show only slight improvement. Under Both Criteria, performance drops substantially for the refinement types, preventing them from achieving strong results in

either parameters or energy.

In contrast, the Dependent Types approach produces results that are already very close to the dataset values. For Energy Error, we observe only minor differences in the parameters, while energy matches more closely than in any other approach. For Parameter Error, the energy comparison shows some discrepancies, but the parameters match exactly. Most notably, in the Both Criteria case, unlike the refinement types, Dependent Types maintains consistently strong performance, achieving good alignment with the dataset in both parameters and energy.

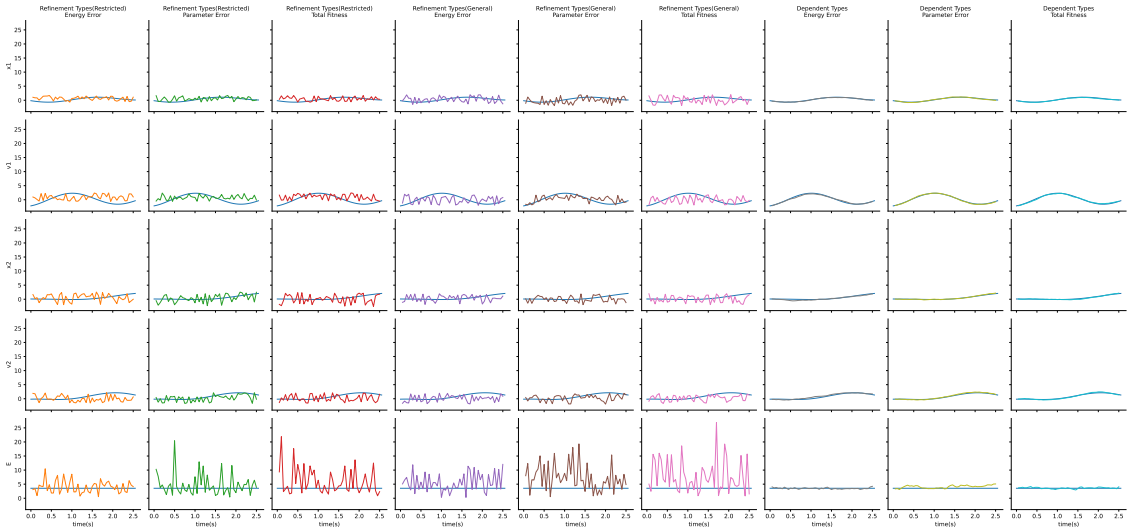

Figure 6.7: **Comparative analysis of parameter values of the best individual with respect to the dataset and total energy in the 60-minute configuration.**

6.4 Conclusion

In this chapter, we have compared Refinement Types with Dependent Types at a low level. Overall, as expected, Dependent Types demonstrate significant improvements over Refinement Types, particularly in terms of initial fitness score improvements when search space restrictions are applied. However, we also identified a limitation of Dependent Types: if the target on which a dependency is based contains an error, that error will propagate through the dependent node and potentially to subsequent nodes. In this experiment, there was a case where the Dependent Types results were worse than those of the Refinement Types. Upon analyzing the code, we discovered that a parameter had been set to an incorrect value. This error existed in both approaches; however, in Refinement Types, the absence of dependencies prevented the error from propagating, effectively confining it within a limited range.

Chapter 7

Evaluation

This thesis aims to investigate the improvements offered by dependent types in program synthesis. In this chapter, we evaluate their performance within a functional programming language, thereby demonstrating their effectiveness in a real-world setting.

7.1 Experimental Setup

7.1.1 Synquid Synthesizer

To evaluate the performance of our synthesizer, we conducted a comparative study. For this purpose, we selected an existing synthesizer, integrated it into Aeon, and assessed its performance. We chose the Synquid Synthesizer [25], which was also discussed in our related works. There are several reasons for this choice. First, Synquid supports refinement types, which directly align with our research objectives. Second, its features are well-documented and clearly explained, making the implementation more straightforward.

Approach

To integrate Synquid into Aeon, we first needed to decide what contextual information should be provided to the synthesizer. We chose to include both terms and types in the context. This means that all built-in functions, custom functions, and types defined in the program are available. Unlike the original Synquid, where the context only contains functions and types explicitly defined in the program (resulting in a relatively small search space), our implementation introduces a much larger and more realistic search space. In practical scenarios, the context typically contains a large number of unrelated terms and types, so our approach better reflects real-world conditions.

The Aeon context consists of the following types:

- **TypeConstructor**: Represents basic and custom types, such as `Integer`, `Float`, `Boolean`, and user-defined types.
- **AbstractionType**: Represents functions and built-in operators, such as the `&&` operator.
- **TypePolymorphism**: Represents polymorphic functions and operators, such as `+` and `-`, which can operate on multiple types.

- **RefinedType**: Represents functions and variables annotated with refinements.

To simplify the implementation, we reused Aeon's existing *validator* and *evaluator*, originally developed during the integration of the GE. The validator checks whether a generated program is valid by performing type checking and refinement validation. For example, given the refinement `x > 1`, the validator will reject any program where `x` is less than or equal to 1. The evaluator executes valid programs and assigns them a score, which can be used in conjunction with the `@minimize_int` annotation to guide the search toward an appropriate solution.

The generation process was implemented in a level-based manner, where the level corresponds to the complexity of the generated solution. At *level 0*, the search space consists of literal values that fit the required type, variables from the context with the same type, and nullary functions (functions that take no parameters). At this stage, only `TypeConstructor` and `AbstractionType` elements with the correct return type are considered. Once all possibilities at a given level have been explored, the level is incremented, expanding the search to include more complex constructs.

For `AbstractionType`, the search requires special handling because both `AbstractionType` and `TypePolymorphism` in Aeon are defined recursively. Accessing the types of a function's parameters and its return type requires traversing this recursive structure. To simplify this process, we apply an **uncurrying** transformation, which converts the recursive structure into a tuple. The first element of the tuple is a list containing all parameter types, and the second element is the return type, as illustrated in Figure 7.1.

Original Structure: (Integer → (Integer → (Integer)))

After Uncurry: ([Integer, Integer], Integer)

Figure 7.1: Illustration of uncurrying a structure

After uncurrying, the return type is checked against the expected type. If it does not match, the search proceeds to the next candidate. Otherwise, the function is recursively called on its parameters with the search level decreased by one. For instance, if the current level is one, the recursive call will be made at level zero. The resulting program candidate is then validated and evaluated.

In contrast, `TypePolymorphism` requires an additional step before uncurrying, namely *monomorphization*. In this step, the polymorphic type is instantiated with all possible `TypeConstructor` substitutions. This produces a set of corresponding `AbstractionType` instances. Once polymorphism has been eliminated, these instances are handled in the same way as ordinary `AbstractionType` : they are uncurried, type-checked, and used for program generation.

To optimize performance, we implemented a memoization strategy. Specifically, we use a combination of the level number, the target type, and the context as the memoization key. With this approach, once a function call has been executed, its result is stored, and subsequent calls with the same key can directly reuse the cached result without re-executing the function.

7.1.2 Benchmark Suite

We collected problems from several sources, including PSB2 [14], the Synquid Benchmark [25], the Mario Level Benchmark [30], and an Arithmetic Benchmark. This combination enhances the diversity of problem types, allowing us to evaluate synthesizer performance across a broad range of tasks.

PSB2

From PSB2, we selected two problems to evaluate the synthesizers. These aim to test the system's ability to solve mathematical problems.

Bouncing Balls (B1) [14] This problem models the motion of a bouncing ball using a simple abstraction. The input consists of the **initial drop height** (h) and the **rebound height** (r) after the first bounce. From these values, we derive the bounciness index, defined as the ratio of rebound height to initial height.

Given this index and a specified **number of bounces** (n), the task is to compute the **total vertical distance** (t) traveled by the ball. This requires accounting for the initial drop and the sequence of upward and downward motions across successive bounces. The problem can be expressed as:

$$bouncing\_balls : (\text{h} : Float,\ \text{r} : Float,\ \text{n} : Integer) \ \mapsto\ \text{t} : Float$$

Dice Game (B2) [14] This problem considers a simple probabilistic game between two players, Peter and Colin. Peter has an n-sided die, while Colin has an m-sided die. Both roll simultaneously, and Peter wins if his outcome is strictly greater than Colin's. The task is to compute the probability that Peter wins:

$$dice\_game : (\text{n} : Integer,\ \text{m} : Integer) \ \mapsto\ \text{probability} : Float$$

Synquid benchmark

From the Synquid benchmark, we selected three list-related problems: `isEmpty`, `replicate`, and `insertAtEnd`.

isEmpty (B3) This function is commonly used in lists to check if a list is empty. If it is, it returns True; otherwise, it returns False.

$$isEmpty : (\text{l} : List) \ \mapsto\ \text{b} : Boolean$$

replicate (B4) This function receives a n, which is the number of times replicas, and $elem$, the element that will be replicated, and it will return a list of the element with size equal to n.

$$replicate : (\text{n} : Integer,\ \text{elem} : Integer) \ \mapsto\ \text{result} : List$$

insertAtEnd (B5) This function inserts the element at the end of the list, so it receives a l, a list, and an $elem$, the element to be inserted in the list.

$$insertAtEnd : (\text{l} : List,\ \text{elem} : Integer) \ \mapsto\ \text{result} : List$$

Mario Level benchmark (B6) [30]

This benchmark, previously introduced in Section 5.2.1, is reused here to evaluate Aeon. While the earlier implementation targeted GE for LLMs, here we consider its counterpart in Aeon to evaluate performance in a problem where there is no single exact answer but rather a higher-level synthesis of functional programs.

To implement this problem, we identified three main classes: **level**, **Chunk**, and **Enemy**. Since a Level contains both a list of Chunk and a list of Enemy, we implemented both Chunk and Enemy recursively. For example, we define `empty\_chunks` (Equation (Def. of empty_chunks)), which produces an empty `Chunks`, and `append\_chunk` (Equation (Def. of append_chunk)), which appends a `Chunk` to a `Chunks` collection. Each `Chunk` type (e.g., Gap, Cannon, Coin) has a constructor function `new\_<type>\_chunk`, with parameters constrained by refinements (see Listing 7.1). In this example, how a Gap is created is shown. Enemies are defined analogously.

$$empty\_chunks : () \ \mapsto\ \text{result} : Chunks \qquad \text{(Def. of empty\_chunks)}$$

$$append\_chunk : (\text{cs} : Chunks,\ \text{c} : Chunk) \ \mapsto\ \text{result} : Chunks \qquad \text{(Def. of append\_chunk)}$$

```
def new_gap_chunk (x: Int | x >= 5 && x <= 95)
                  (y: Int | y >= 3 && y <= 5)
                   (wg: Int | wg >= 2 && wg <= 5)
                   (wBefore: Int | wBefore >= 2 && wBefore <= 7)
                   (wAfter: Int | wAfter >= 2 && wAfter <= 7) :
                   Chunk
```

Listing 7.1: Example of new chunks

The `new\_level` function (Equation (Def. of new_level)) then constructs a `Level` from Chunks and Enemies:

$$new\_level : (\text{cs} : Chunks,\ \text{es} : Enemies) \ \mapsto\ \text{result} : Level \qquad \text{(Def. of new\_level)}$$

Finally, we define the synthesis objective (Listing 7.2). Building on the previously defined functions, we construct a large and complex problem intended to approximate a more realistic scenario. This formulation includes two optimization goals: (1) `number\_of\_chunks`, which measures the size of the level, and (2) `conflicts`, which counts overlapping chunks and enemies.

```
@minimize_float(number_of_chunks 110 10 new_map)
@minimize_float(conflicts 110 10 new_map)
def new_map : Level = ?hole;
```

Listing 7.2: The synthesis of mario with objectives

Arithmetic Benchmarks (B7)

Finally, we designed a simple benchmark to test the synthesizer's core functionality. Here, the target is to synthesize a single integer value.

```
def c : Int = 10;

@minimize_int(fun(25) - c)
def fun (i:Int) : Int  {
    let a : Int = 10*i;
    (?hole: {x:Int | x >= c} ) * i + a
}
```

Listing 7.3: Synthesize an integer

In Listing 7.3, the function receives an integer input. The synthesis target is an integer constrained by the refinement `x >= c`. The synthesized value is then combined with arithmetic expressions, and the result is evaluated under the annotation to minimize the output.

7.2 Results

Figure 7.2 presents the total number of evaluated instances, i.e., the number of possible solutions processed within a given time interval, for each search method under the two search-time configurations across the benchmarks. From these results, we observe that the search methods in GE tend to generate fewer instances than Synquid under the same time configuration. This outcome is expected, as the use of dependent types in GE makes instance generation more complex and time-consuming. An exception is found in the Mario Level Benchmark: due to the high complexity of the problem, both enumerative search methods struggle significantly. For Synquid, this difficulty is even more severe, as the program fails to terminate. In the Insert Benchmark, both GP and Random Search report relatively low numbers of evaluated instances. This behavior arises from a GE setting in which, if no explicit fitness function (objective) is provided, the search terminates immediately upon finding the first valid solution, thereby limiting the total number of generated instances.

Table 7.1 summarizes, for each benchmark, whether a valid solution was found by the respective search method. The results show that both enumerative search approaches perform the worst. In the case of Enumerative Search (Synquid), although it evaluates the most significant number of instances, it successfully solves only two benchmarks, with one remaining incomplete. Even when the search time is extended to 1800 seconds, no additional benchmarks are solved. On the other hand, Enumerative Search (GE) demonstrates mixed behavior: in some benchmarks, it evaluates a number of instances comparable to other GE search methods, while in others, it is closer to Synquid. Within 300 seconds, it solves only two benchmarks slightly worse than Synquid, but when extended to 1800 seconds, its performance matches that of Synquid. Random Search (GE) performs noticeably better, solving four benchmarks within 300 seconds and six when given 1800 seconds. Finally, GP achieves the best results across both time settings: it solves five benchmarks

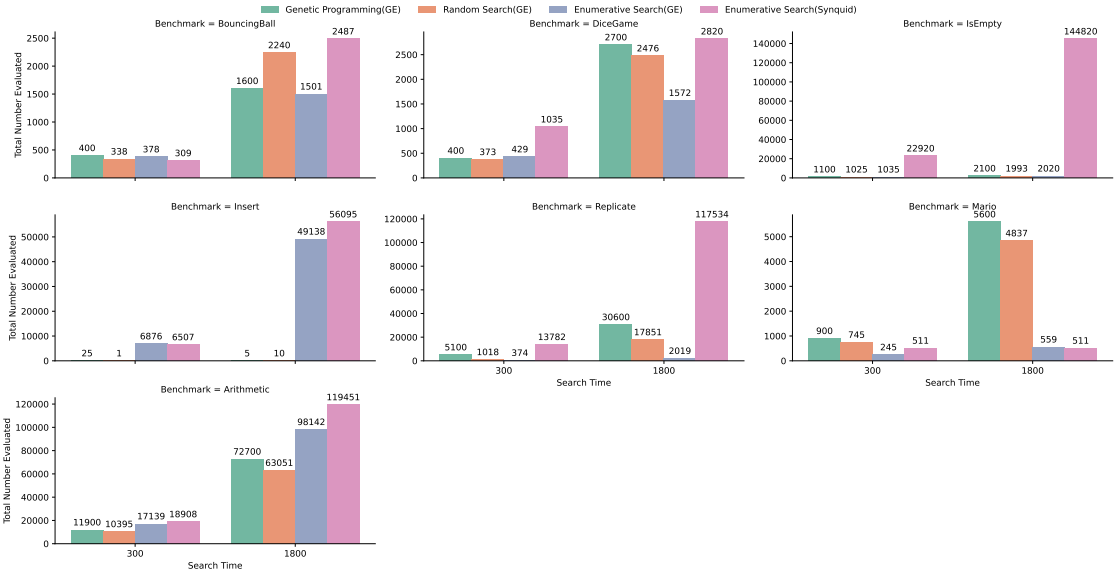

Figure 7.2: Comparative analysis of the Total Number of evaluated instances in two search time settings

within 300 seconds and is able to solve all benchmarks when the runtime is extended to 1800 seconds.

Search time(s)	*Search Method*	*Type*	B1	B2	B3	B4	B5	B6	B7
300	Enumerative Search(Synquid)	RT	✗	✗	✗	✓	✗	✓	✓
	Enumerative Search(GE)	DT	✗	✗	✗	✗	✗	✓	✓
	Random Search(GE)	DT	✗	✓	✗	✓	✗	✓	✓
	Genetic Programming(GE)	DT	✗	✓	✗	✓	✓	✓	✓
1800	Enumerative Search(Synquid)	RT	✗	✗	✗	✓	✗	✓	✓
	Enumerative Search(GE)	DT	✗	✗	✗	✓	✗	✓	✓
	Random Search(GE)	DT	✓	✓	✗	✓	✓	✓	✓
	Genetic Programming(GE)	DT				✓			

Table 7.1: The table presents whether a solution is encountered or not based on the search method and search time. The column *Type* indicates the type of synthesizer used: RT denotes Refinement Types, and DT denotes Dependent Types. For the benchmarks (B1–B8), ✓ indicates that a solution was found and the program terminated normally, ✓ indicates that a solution was found but the program terminated due to a system timeout, and ✗ indicates that no solution was found.

7.3 Discussion

With the results gathered from Section 7.2, we can conclude that, for Enumerative Search, dependent types do not yield a significant improvement in program synthesis. The number of problems solved is essentially the same as with refinement types, suggesting that the additional expressiveness does not translate into greater effectiveness under pure enumeration.

However, the picture changes when considering Random Search and GP. At shorter time set-

tings, the improvements may appear modest. Random Search solves only one more problem, and GP solves two more compared to Enumerative Search. However, when the search time is prolonged, the advantage of dependent types becomes much clearer: Random Search leaves only one benchmark unsolved, and GP successfully solves all benchmarks. This demonstrates how dependent types help search-based methods scale more effectively over time.

Another important observation is that dependent types consistently evaluate fewer instances than refinement types while achieving better results. This indicates that dependent types yield a more restricted and structured search space, guiding the synthesizer toward valid solutions more efficiently. On the other hand, this benefit comes at a cost. As shown in Figure 7.2, the generation step in GE is more computationally expensive than in Synquid. In practice, this means that although the search space is reduced, each instance takes longer to generate.

Overall, dependent types offer a clear advantage in guiding the search process, striking a balance between precision and efficiency, but it is also expected that the improvement in Enumerative search is minimal because the extensive search cannot be reduced significantly by Dependent types. They prune the search space more effectively than refinement types, and when paired with an efficient search method such as GP, they deliver the strongest performance across benchmarks.

Chapter 8

Future work

Hybrid Method optimization The hybrid method we implemented for GE, presented in Chapter 5, is an early prototype that currently functions only in a controlled experimental environment. To be applicable in real-world program synthesis tasks, the implementation requires significant restructuring, both to improve maintainability and to provide meaningful error handling. Writing clear and informative error messages is particularly challenging: at present, the method does not produce any diagnostic messages, leaving users unable to identify or understand the cause of failures.

Another important direction for improvement lies in prompt engineering. Our current implementation employs a zero-shot prompting strategy, but different LLMs often respond quite differently to the same prompt. A systematic study of alternative prompting techniques is therefore necessary to identify the most effective strategies for reliable instance generation.

Finally, the current implementation supports only the LLMs available through Ollama, which limits model choice. A promising solution would be to develop a more general library that enables communication with multiple backends and providers. Such an approach would decouple the synthesizer from a single platform and allow broader experimentation with a variety of models.

Evaluation with more synthesizer The current synthesizers implemented in Aeon, Synquid [25], and GE [10] were sufficient to evaluate the performance of dependent types in our experiments. However, to gain a broader and more comprehensive understanding, it is necessary to extend this comparison to additional synthesizers and search methods. The objective for future work is therefore to implement more synthesizers within Aeon, enabling a deeper evaluation of the effectiveness of Dependent types across diverse synthesis paradigms.

Evaluation on an extensive benchmark suite The evaluation was conducted on a small but diverse set of benchmarks, which was sufficient to assess different kinds of synthesis problems. However, the evaluation could be further strengthened by incorporating more complex benchmarks to illustrate the performance better. We aim to design a more comprehensive benchmark suite that introduces richer problem structures and more sophisticated type systems, allowing for a deeper evaluation of the capabilities and limitations of dependent types.

Error handling Currently, the Aeon does not provide explicit support for error handling. As a result, if an error occurs during synthesis, it is raised using the default settings, which offer little help in understanding the cause or nature of the error.

Synthesis Number The Aeon programming language only supports synthesizing a single problem per execution. This means that if multiple `?hole` placeholders are present, the system will synthesize only the first hole while ignoring the others. Consequently, this limitation restricts users to resolving holes incrementally, one at a time.

Chapter 9

Conclusion

Program Synthesis (PS) is the task of automatically generating programs from a specification. Such specifications are typically provided through input–output examples, natural language descriptions, or formal specifications. A central challenge of program synthesis, particularly in real-world scenarios, is the infinite search space of possible programs. To address this challenge, two main approaches have emerged: designing efficient strategies to navigate the search space and constraining the search space itself.

The first direction, efficient navigation of the search space, can be broadly divided into three families of search methods. Enumerative methods systematically explore all possible solutions, guaranteeing the best solution will eventually be found; however, this is infeasible for real-world problems due to the vastness of the search space. LLM-based methods leverage large language models to directly generate candidate programs or combine LLMs with other strategies to improve performance. These methods are typically fast but highly dependent on model quality and prompt design. Finally, heuristic methods navigate the search space stochastically, using heuristics to guide the search toward better solutions. While effective in practice, these methods cannot guarantee that the final solution is optimal.

The second direction constraining the search space relies on type systems to ensure that only well-typed candidates are generated. This prevents incompatible combinations and enforces structural correctness. Refinement types extend this idea by introducing logical predicates that further restrict the values a type may assume, thus reducing the search space even more. However, even with refinement types, the remaining search space often remains too large for real-world applications.

In this work, we introduced a new MetaHandler into the Genetic Engine (GE) framework, designed to provide greater dependency and flexibility for constructing benchmarks.

To evaluate the three families of search methods, we proposed a novel LLM-based hybrid method, which combines LLMs with Genetic Programming (GP) search within GE. Among the tested LLMs, Qwen achieved the best results and was used in our experiments. Together with the hybrid approach, the existing Enumerative method and GP search were evaluated across benchmarks. Our findings show that GP outperformed the other approaches.

To assess the benefits of Dependent Refinement Types over standard Refinement Types, we

first designed a clean benchmark based on a physics problem. The results confirmed our hypothesis: under the same time constraints, Dependent Types significantly outperformed Refinement Types.

We then extended this evaluation to a real-world setting by implementing the Synquid synthesizer in the functional programming language Aeon, allowing us to compare the two type systems directly. We constructed a benchmark suite spanning diverse synthesis tasks. The results showed that Dependent Types provided little to no benefit when combined with Enumerative search, but demonstrated significant improvements with both Random Search and GP.

In conclusion, despite the limitations of this work and the open opportunities for future research, our results suggest that Dependent Types, particularly when paired with heuristic search methods such as GP, offer a promising direction for more efficient program synthesis. This thesis thus contributes new insights into the interplay between type systems and search strategies, paving the way toward more practical and scalable program synthesis solutions.

Bibliography

[1] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013.

[2] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Results and analysis of sygus-comp'15. In *SYNT*, 2016.

[3] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2016: Results and analysis. In *SYNT@CAV*, 2016.

[4] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2017: Results and analysis. In *SYNT@CAV*, 2017.

[5] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. *2008 21st IEEE Computer Security Foundations Symposium*, pages 17–32, 2008.

[6] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[7] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[8] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony S. Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurélien Rodriguez, Austen Gregerson, Ava Spataru, Bap tiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Cantón Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab A. AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang,

Gabriele Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Grégoire Mialon, Guanglong Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel M. Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Ju-Qing Jia, Kalyan Vasuden Alwala, K. Upasani, Kate Plawiak, Keqian Li, Ken-591 neth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuen ley Chiu, Kunal Bhalla, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melissa Hall Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri S. Chatterji, Olivier Duchenne, Onur cCelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasić, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohit Girdhar, Rohit Patel, Ro main Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Chandra Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whit ney Meers, Xavier Martinet, Xiaodong Wang, Xiaoqing Ellen Tan, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yiqian Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zhengxu Yan, Zhengxing Chen, Zoe Papakipos, Aaditya K. Singh, Aaron Grattafiori, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adi Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alex Vaughan, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Franco, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Ben Leonhardi, Po-Yao (Bernie) Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton,

Catalina Mejia, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Damon Civin, Dana Beaty, Daniel Kreymer, Shang-Wen Li, Danny Wyatt, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Firat Ozgenel, Francesco Caggioni, Francisco Guzm'an, Frank J. Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Govind Thattai, Grant Herman, Grigory G. Sizov, Guangyi Zhang, Guna Lakshminarayanan, Hamid Shojanazeri, Han Zou, Hannah Wang, Han Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Igor Molybog, Igor Tufanov, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kaixing(Kai) Wu, U KamHou, Karan Saxena, Karthik Prasad, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kun Huang, Kunal Chawla, Kushal Lakhotia, Kyle Huang, Lailin Chen, Lakshya Garg, A Lavender, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Maria Tsimpoukelli, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikolay Pavlovich Laptev, Ning Dong, Ning Zhang, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollár, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Rohan Maheswari, Russ Howes, Ruty Rinott, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shiva Shankar, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Sung-Bae Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Kohler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar,

Vishal Mangla, Vlad Ionescu, Vlad Andrei Poenaru, Vlad T. Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xia Tang, Xiaofang Wang, Xiaojian Wu, Xiaolan Wang, Xide Xia, Xilun Wu, Xinbo Gao, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu Wang, Yuchen Hao, Yundi Qian, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, and Zhiwei Zhao. The llama 3 herd of models. *ArXiv*, abs/2407.21783, 2024.

[9] Jack Feser, Işıl Dillig, and Armando Solar-Lezama. Inductive program synthesis guided by observational program similarity. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023.

[10] Alcides Fonseca, Paulo Canelas, and Sara Silva. The usability argument for refinement typed genetic programming. In *Parallel Problem Solving from Nature*, 2020.

[11] Catarina Gamboa, Paulo Canelas, Christopher Timperley, and Alcides Fonseca. Usability-oriented design of liquid types for java. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 1520–1532. IEEE Press, 2023.

[12] Sumit Gulwani, Alex Polozov, and Rishabh Singh. *Program Synthesis*, volume 4. NOW, August 2017.

[13] Haskell.org. The haskell programming language. `https://www.haskell.org`, 2024. Accessed: 2025-03-08.

[14] Thomas Helmuth and Peter M. Kelly. Psb2: the second program synthesis benchmark suite. *Proceedings of the Genetic and Evolutionary Computation Conference*, 2021.

[15] Thomas Helmuth and Lee Spector. General program synthesis benchmark suite. *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, 2015.

[16] Idress Husien and Sven Schewe. Program generation using simulated annealing and model checking. In *IEEE International Conference on Software Engineering and Formal Methods*, 2016.

[17] International Organization for Standardization. *ISO/IEC 9899:2018 — Programming Languages — C*. ISO, 2018.

[18] Brian W. Kernighan and Dennis M. Ritchie. The c programming language, 1988.

[19] Yixuan Li, Julian Parsert, and Elizabeth Polgreen. Guiding enumerative program synthesis with large language models. In *International Conference on Computer Aided Verification*, 2024.

[20] Simon Marlow. *Haskell 2010 Language Report*. Cambridge University Press, 2010.

[21] Robert McKay, Nguyen Hoai, P.A. Whigham, Yin Shan, and Michael O'Neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11:365–396, 09 2010.

[22] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[23] Ulf Norell. Agda—a dependently typed programming language. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation (TLDI)*. ACM, 2009.

[24] Oracle Corporation. Java programming language. `https://www.oracle.com/java/`, 2024. Accessed: 2025-03-08.

[25] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *SIGPLAN Not.*, 51(6):522–538, June 2016.

[26] Python Software Foundation. Python programming language. `https://www.python.org`, 2024. Accessed: 2025-03-08.

[27] Robert Kent Dybvig. *Revised6 Report on the Algorithmic Language Scheme (R6RS)*. MIT Press, 2007.

[28] Ruby Core Team. Ruby programming language. `https://www.ruby-lang.org`, 2024. Accessed: 2025-03-08.

[29] Eric Schkufza, Rahul Sharma, and Alexander Aiken. Stochastic superoptimization. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[30] Noor Shaker, Miguel Nicolau, Georgios N. Yannakakis, Julian Togelius, and Michael O'Neill. Evolving levels for super mario bros using grammatical evolution. *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 304–311, 2012.

[31] Ning Tao, Anthony Ventresque, Vivek Nallur, and Takfarinas Saber. Enhancing program synthesis with large language models using many-objective grammar-guided genetic programming. *Algorithms*, 17:287, 2024.

[32] Matilde Valente, Tiago C. Dias, Vasco Guerra, and Rodrigo Ventura. Physics-consistent machine learning: output projection onto physical manifolds, 2025.

[33] Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. Refinement types for haskell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 269–282. ACM, 2014.

[34] Xinyu Wang, Isil Dillig, and Rishabh Singh. Synthesis of data completion scripts using finite tree automata. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[35] Chao Wen, Jacqueline Staub, and Adish Kumar Singla. Program synthesis benchmark for visual programming in xlogoonline environment. *ArXiv*, abs/2406.11334, 2024.

[36] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zhihao Fan. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024.

[37] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[38] École Polytechnique Fédérale de Lausanne (EPFL). Scala programming language. `https://www.scala-lang.org`, 2024. Accessed: 2025-03-08.