UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA

Towards the Conceptualization of Refinement Typed Genetic Programming

Paulo Alexandre Canelas dos Santos

Mestrado em Engenharia Informática
Especialização em Engenharia de Software

Dissertação orientada por:
Professor Doutor Alcides Miguel Cachulo Aguiar Fonseca

2020

Agradecimentos

O final de uma dissertação não é o culminar de um ano de trabalho, mas sim o acumular de toda uma experiência de vida. Por isso, gostaria de começar por agradecer a todos, aqueles que por bem ou mesmo por mal, me trouxeram onde estou hoje, orgulhoso e agradecido, pela presença que tiveram no meu Passado.

O primeiro agradecimento pessoal é à minha família, especialmente à minha mãe. Tem sido desde sempre um grande apoio, e, sempre com os grandes sacrifícios que tivemos, tentou ao máximo permitir que eu alcançasse as minhas ambições. Gostaria também de agradecer ao Henrique pela presença dele nestes últimos anos, que como meu pai fosse, me ensinou várias lições de vida. Finalmente um grande obrigado pelo vosso apoio, à minha avó, e aos meus irmãos, José, Tiago, Soraia, Gabriel e Bruno.

Para alguns, era este o parágrafo que estavam à espera. Falo de vocês, que durante muitas horas de trabalho, grande esforço (e ansiedade) permitiram-me ter, ainda assim, uns dos melhores anos da minha vida. Gostaria de deixar uma mensagem de agradecimento a cada um de vós: Duarte, nunca me esquecerei daquela noite, nem de uma única palavra que me disseste, irmão; Guilherme, foste uma das primeiras pessoas que conheci, e uma daquelas que sempre acreditou em mim e no meu potencial; João Lobo, a pessoa que mais mudou nestes últimos anos, e me mostrou que todos conseguimos alcançar os nossos objetivos com determinação, tens sido o meu modelo a seguir; João Gil, agradeço-te todas as experiências que passei contigo, e, mesmo quando saltei de um precipício para a morte certa, estiveste lá para me ajudar; Miguel, normalmente o melhor fica para o fim, neste caso não, ficas só tu, que mesmo sendo como és, nunca perdeste aquilo que eu mais valorizo, confiança, obrigado.

Finalmente, gostaria de agradecer ao meu orientador, Professor Alcides. Foi o Professor que me incentivou a sair da zona de conforto, a confiar nas minhas capacidades, candidatar-me a vários projetos, e, apesar de rejeitado em vários deles, continuar em frente. Mesmo quando a meio, perdi a motivação, o Professor fez os possíveis para me ajudar e orientar até ao final deste trabalho. Gostaria também de agradecer ao Professor Vasco, à Professora Andreia e à Sara, pela paciência que tiveram comigo no início do projeto e durante todo o seu prosseguimento, e, apesar de não ter tido oportunidade, espero vos conseguir surpreender no futuro!

Dedico-te a ti pai, pela presença que pudeste ter

Resumo

Turing apresentou pela primeira vez a evolução de programas através da descrição de operadores evolutivos bastante básicos, como a mutação, seleção, e material genético. Forsyth foi o pioneiro que desenvolveu o primeiro passo em direção à computação evolutiva através da criação de uma abordagem evolutiva para induzir regras de decisão.

A Programação Genética (PG) é um método de resolução de problemas através da geração e evolução de programas. A PG tem sido bastante usada em diferentes áreas, desde reparação automática de programas até à optimização de *hyper-features*. O seu objetivo é explorar o espaço de procura, avaliando os indivíduos de uma população, em busca daquele que melhor se adequa a um determinado problema.

A programação genética, aplicada a síntese de programas, apresenta alguns desafios a serem resolvidos. O primeiro desafio é a pesquisa num largo espaço de procura. Como estamos à procura de um programa que resolva um determinado problema, encontrar esse programa, ou outro computacionalmente equivalente, no infinito espaço de programas que podem ser gerados, é uma tarefa quase impossível sem qualquer tipo de auxílio. A Programação Genética Fortemente Tipada (PGFT) dá um primeiro passo para a resolução deste problema. Esta abordagem considera o sistema de tipos da linguagem para limitar a quantidade de programas gerados, descartando os programas que são inválidos. A utilização do sistema de tipos permite guiar com melhor precisão o processo evolutivo, mas continua a ser insuficiente. A introdução de tipos refinados, tipos que apresentam um predicado que restringem o seu domínio, permitem reduzir mais ainda o espaço de procura de programas válidos. Quando existe a geração de um programa, típicamente é necessário avaliar a correção desse mesmo programa, de onde podem ser utilizadas uma das seguintes abordagens: verificação através de um conjunto de pares input/output, ou um conjunto de restrições lógicas indicativas do comportamento do programa.

O segundo obstáculo para a geração automático de código é a fraca capacidade de generalização. Generalizar um determinado tipo de problema para qualquer argumento e devolvendo um resultado válido é até hoje um problema em aberto. Por exemplo, a programação guiada por exemplos pega nos argumentos de entrada e tenta gerar código de modo a cumprir com o valor de retorno esperado. Um problema desta abordagem encontrasse na dependência dos inputs para gerar o código, ou seja, considerando os valores de entrada e saída input=1, output=2 e input=2, output=3, um código gerado e válido para o input dado seria if input == 1 then 2 else 3, o que claramente não seria generalizado para outro conjunto de testes. A síntese de programas recorrendo a uma especificação retira este tipo de limitação. A utilização dos tipos refinados e dependentes define o comportamento que o código sintetizado tem de cumprir permitindo que o código sintetizado mais facilmente generalize para qualquer tipo de parâmetro.

O último problema na síntese de programas é conseguir especificar convenientemente a intenção do programador. O programador necessita de apresentar uma especificação parcial ou completa do problema

de modo a que o código possa ser gerado automaticamente. A especificação permite que o programa seja gerado de acordo com a intenção do utilizador. Todavia, vários fatores como a experiência de programação do utilizador e a sua capacidade de expressar especificações representam um entrave. Várias abordagens têm sido apresentadas para facilitar ao programador comum expressar a sua intenção de programação. A síntese de programas guiada por exemplos facilita ao desenvolvedor de software indicar a sua intenção, visto que necessita apenas de indicar os inputs/outputs. Abordagens recorrendo a especificações tentam utilizar, por exemplo, contratos para definir a especificação ou esboços dos programas a serem gerados. Porém, este tipo de abordagens basead em contratos ou esboços dos programas podem comprometer a usabilidade do próprio sistema de síntese.

Para resolver estes desafios, propomos a linguagem de programação ÆON que permite a síntese total ou parcial de programas. Tal como outras linguagens de programação funcionais, como o Haskell e Scheme, o ÆON tem funções nativamente implementadas, abstrações, criação de tipos e criação de abstrações de tipos, polimorfismo. A criação de novos tipos e o polimorfismo enriquecem a expressividade da linguagem ÆON, permitindo que esta ataque diferente áreas da síntese de programas.

O ÆON oferece aos programadores a capacidade de criar especificações em funções a partir do próprio sistema de tipos. Estas especificações são usadas não só para garantir a correção de uma função, mas também para sintetizar funções completas ou parciais. A síntese de programas generalizada é o que diferencia a linguagem de programação ÆON das restantes abordagens no estado da arte. O programador comum consegue rapidamente gerar a sua função ao introduzir a especificação necessária, através do tipo de retorno esperado, e o operador buraco, ■, que indica ao sistema que pretende sintetizar código que substitua o local sem implementação. A linguagem identifica que existe um buraco e inicia o procedimento de síntese de programas. Neste procedimento, uma população inicial é gerada recorrendo aos tipos refinados e a um sintetizador de tipos polimórficos que garante a correção dos programas. Os programas são avaliados de acordo do quão próximos se encontram do programa pretendido pelo programador, e aqueles com melhor desempenho são então escolhidos para a próxima etapa evolucionária através da sua reprodução e mutação. Este processo repete-se iterativamente até se chegar a uma solução compilável e que respeita a especificação. No final, a solução encontrada é injetada no código original, substituindo todos os buracos na implementação pela expressão sintetizada respetiva. A linguagem de programação ÆON foi desenvolvida em Python e encontra-se interpretada.

A primeira contribuição deste trabalho é a síntese não deterministica de tipos refinados estaticamente verificáveis. O sintetizador que foi criado permite a geração de expressões aleatórias com uma determinada profundidade máxima para um determinado tipo. O ÆONCORE é uma linguagem de programação sobre o qual o ÆON age como uma façada sintática, e que faz uso de tipos refinados verificaveis estaticamente para sintetizar programas válidos através de combinações dos componentes de síntese que se encontram disponíveis no contexto do problema.

A segunda contribuição é a linguagem de programação ÆON como uma linguagem de síntese de programas num contexto geral da programação. A sintaxe simples da linguagem ÆON permite ao programador facilmente especificar predicados que restringem o comportamento de uma função. Estas restrições são utilizadas para não só verificar a correção do programa mas também realizar a geração automática de programas aquando na presença de um buraco.

A terceira e quarta contribuições são, respetivamente, a utilização de Programação Genética (PG)

para gerar de forma generalizada e válida programas e a utilização de uma forma mais expressiva os predicados lógicos como funções de avaliação no procedimento evolucionário.

A última contribuição é a prova de versatilidade da Programação Genética com Tipos Refinados (RTGP) através da criação de uma ferramenta de testes baseada em propriedades em Python, o pyCheck. Esta ferramenta permite ao utilizador criar com maior expressividade restrições sobre os parâmetros de entrada de uma função gerando assim testes aleatórios que permitam verificar a correção do programa. A execução da ferramenta produz um relatório com a indicação dos testes passados e falhados e uma pequena análise do quão próximo o programa está de se encontrar correto.

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

Abstract

The Genetic Programming (GP) approaches typically have difficulties dealing with the large search space as the number of language components grows. The increasing number of components leads to a more extensive search space and lengthens the time required to find a fitting solution. Strongly Typed Genetic Programming (STGP) tries to reduce the search space using the programming language type system, only allowing type-safe programs to be generated. Grammar Guided Genetic Programming (GGGP) allows the user to specify the program's structure through grammar, reducing the number of combinations between the language components. However, the STGP restriction of the search space is still not capable of holding the increasing number of synthesis components, and the GGGP approach is arguably usable since it requires the user to create not only a parser and interpreter for the generated expressions from the grammar, but also all the functions existing in the grammar.

This work proposes Refinement Typed Genetic Programming (RTGP), a hybrid approach between STGP and RTGP, which uses refinement types to reduce the search space while maintaining the language usability properties. This work introduces the Æon programming language, which allows the partial or total synthesis of refinement typed programs using genetic programming. The potential of RTGP is presented with the usability arguments on two use cases against GGGP and the creation of a prototype property-based verification tool, pyCheck, proof of RTGPs components versatility.

Keywords: Genetic Programming, Refined Types, Program Synthesis, Refined Typed Genetic Programming

Contents

List of Figures

List of Tables

Listings

Acronyms

AST Abstract Syntax Tree.

CEGIS Counterexample Guided Synthesis.

GGGP Grammar-Guided Genetic Programming.

GP Genetic Programming.

PBT Property Based Testing.

PS Program Synthesis.

RTGP Refined Typed Genetic Programming.

STGP Strongly Typed Genetic Programming.

Chapter 1

Introduction

1.1 Motivation

Turing introduced the first evidence on program evolution with a glance to basic evolutionary operators [54] (mutation, natural selection, and hereditary material). Forsyth pioneered the first step towards evolutionary computation by presenting an evolutionary approach for decision-rules by induction [21] using a population, classifying it, eliminating unfit individuals and mating and mutating fit individuals. However, it was only in 1992 that Holland defined the foundations of Genetic Programming [29].

Genetic Programming (GP) is a problem-solving heuristic method that works by generating and evolving programs. GP has been widely used in different areas, from automatic program repair [22] to hyper-feature optimization [5], whose objective is to explore the space of possible programs by searching for the individual in a population which might be capable of solving a given problem.

One of the main challenges in GP is being able to find a solution in a large search space. The amount of operation combinations makes it hard for simple GP approaches to converge and find a solution quickly. In order to restrict the search space to programs that are type safe, Strongly Typed Genetic Programming (STGP) [39] was created to consider the type system of the programming language, allowing the process of type-checking inside the GP language.

The STGP approach presented a first step towards search space optimization using the types. However, this approach was not expressive enough and was later enhanced with type inheritance [24] and polymorphism [59]. Such improvements allowed STGP to tackle a different kind of problems but did not necessarily reduce the search space.

McKay et al. later introduced Grammar-Guided Genetic Programming (GGGP) [37] as a way to further reduce the number of available programs by providing a grammar, and explicitly declaring the combination of the synthesis components. This approach, however, requires the user not only to learn the concepts behind grammars, namely the Backus Normal Form (BNF), but also to create all the components for the synthesis and the genetic programming algorithm in a familiar language.

1.2 Objectives and Contributions

The objective of this thesis is to explore the viability of using a type system based techniques for program synthesis within the nature-inspired non-deterministic search-based techniques. This work aims to

evaluate how we can improve the low response time of the evolutionary algorithm, widely known for their long running times, with the nimbleness of algorithms techniques based on SMT/SAT Solvers. In the end, we plan to have the ability to work with the code provided by the evolutionary procedure at a faster speed because of the SAT solver.

This objective was accomplished with the following contributions:

- The first contribution is the *Æon* user-facing programming language as a programming language that allows program synthesis through the use of refined and dependent types. *Æon* type system lets the regular programmer to easily specify predicates within the type system that restrict the behaviour of the expressions.
- The second contribution is the non-deterministic synthesis of polymorphic refined types. The synthesizer allows the creation of random expressions with a set depth from a given type and typing context.
- The third contribution is the implementation of a Genetic Programming (GP) algorithm for the hole synthesis in *Æon*. The programmer can provide holes in the implementation of a function, informing the compiler to automatically evolve and create expressions that may be filled in the holes.
- The fourth contribution is a more expressive conversion of logical predicates into continuous fitness functions used to evaluate individual programs in the genetic programming procedure.
- The fifth and final contribution is the application of Refinement Typed Genetic Programming (RTGP) for property-based testing in Python. We prove the versatility of the concepts described in RTGP, with the creation of pyCheck, a property-based prototype testing tool, which not only allows the user to provide more expressive restrictions over the functions but also produces extra information on program correctness, like the test accuracy.

This work has been recognized in the publication of a poster and a full paper in peer-reviewed venues, respectively:

Paulo Santos, Sara Silva, and Alcides Fonseca. 2020. Refined typed genetic programming as a user interface for genetic programming. In Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion (GECCO '20) [48];

Alcides Fonseca, Paulo Santos, and Sara Silva. 2020. The Usability Argument for Refinement Typed Genetic Programming. In: Bäck T. et al. (eds) Parallel Problem Solving from Nature – PPSN XVI. PPSN 2020 [20].

1.3 Context

This work was carried at the line of excellence of Reliable Software Systems at the Laboratory of Large-Scale Systems (LASIGE). This work could not have been done without the help of the research team composed by Alcides Fonseca, Andreia Mordido, Sara Silva, and Vasco Vasconcelos. The work was also partially developed under the guidance of Chris Timperley of Carnegie Mellon University (who help I genuinely thank for).

Fundação para a Ciência e Tecnologia (FCT) partially funded this project through the CONFIDENT project (PTDC/EEI-CTP/4503/2014) and with the CMU-Portugal project CAMELOT (POCI-01-0247-FEDER-045915). This work was carried at LASIGE (UIDB/00408/2020), Faculty of Sciences, University of Lisbon.

1.4 Structure of the document

The document is organized as follows:

Chapter 2 briefly describes the main topics essential for the understanding of this thesis, related to genetic programming, type theory, and program synthesis.

Chapter 3 presents the different approaches of program synthesis, namely on the deductive and inductive synthesis, and their correlation to the state of the art approaches.

Chapter 4 introduces the ÆON programming language concept and examples. Formally details the language syntax and displays an explanation of the ÆON compiler.

Chapter 5 contains the ÆONCORE programming language. It motivates the conversion from ÆON to ÆONCORE, the language syntax, the type system and formation, and the deduction of the hole types.

Chapter 6 describes the non-deterministic synthesis rules required to automatically generate expressions well-formed and type safe, and the synthesizer optimizations to improve its performance.

Chapter 7 presents the genetic programming algorithm for the program synthesis procedure. It describes the different aspects of the genetic programming components and implementation details, and the code optimization after synthesis.

Chapter 8 evaluates the proposed approach in terms of usability and versatility of the concepts. It also provides a brief description of the limitations of the current approach.

Chapter 9 describes the intended future work of the refinement typed program synthesis.

Chapter 10 concludes the current work by extolling its contributions.

Chapter 2

Background

This chapter presents the main concepts needed for better understanding this work: genetic programming, type theory and program synthesis.

2.1 Genetic Programming

As previously mentioned, Genetic Programming (GP) is a problem-solving method which search for a solution by generating and evolving programs. It generates a random population of programs and evolves them throughout generations, searching for one that is able to achieve the provided goal. The essential components of a genetic algorithm, and typically used in genetic programming [11, 39] are representation, initialization procedure, evaluation function, genetic operators (mutation and crossover), and parameters. Figure 2.1 is a simple example of the evolutionary procedure, with each component explained posteriorly. The genetic programming procedure starts by generating and evaluating a population of individuals. For each generation, a new offspring population is created from the selection, recombination, and mutation from the previous one. The new population is evaluated according to a fitness function, and the genetic programming algorithm terminates if an individual is found to solve a specific problem.

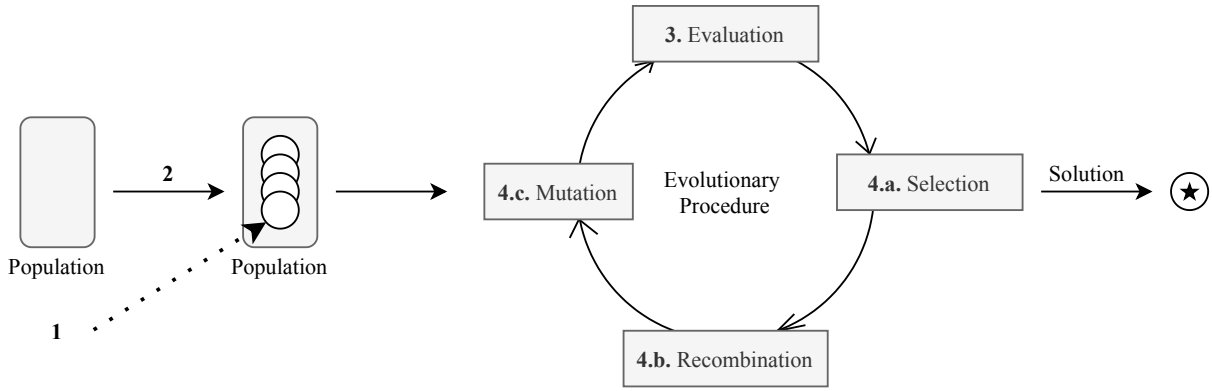


Figure 2.1: Simple Evolutionary procedure diagram.

1. Representation: In genetic programming, individuals correspond to programs and are typically represented as abstract syntax trees. Abstract syntax tree (AST) is a tree-structure composed by the combination of two kinds of nodes: functions and terminals. Procedures and functions (non-terminal

nodes) receive arguments for their computation while terminal nodes, such as variables and constants values, do not receive any arguments, being leaves of the tree.

The execution of an abstract syntax tree starts on its root. It delegates the execution to the subtrees, following a post-order transversal, until each subtree returns the value of its computation.

Consider as an example the following code (2.1) of a function written in ÆON, which computes the hypotenuse of a right-angled triangle using the Pythagorean theorem. This function receives two parameters, a and b, and returns an **Integer**, result of the computation.

Listing 2.1: Pythagorean theorem in ÆON.

```
1 hypotenuse(a:Integer, b:Integer) -> c:Integer {
2     c = sqrt(pow(a, 2) + pow(b, 2));
3 }
```

Figure 2.2 shows an example of representation of the AST generated by the given program. The tree is read as follows: assign to the variable c the result of computing the square root of the sum of a to the power of two, and b to the power of two.

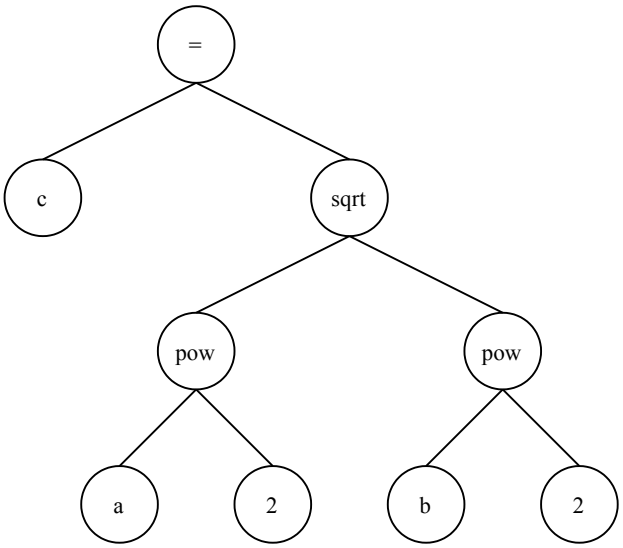


Figure 2.2: Parse tree of the hypotenuse function using the Pythagorean theorem.

2. Initialization procedure: The first step towards our genetic programming algorithm is to generate a random initial population. Koza [33] describes three techniques, grow, full, and ramped half-and-half, to create random individuals of a population.

(a) **Grow**: A random node (function or terminal) is selected as the root of the tree. When generating a function, the same procedure is applied for each n-arity son of the current node. Once the tree reaches a maximum depth m, only terminals are generated.

(b) **Full**: The full approach takes into consideration a final depth m. Starting on the root and for all its sons until depth $m - 1$, it generates function nodes. Once it reaches depth m, it only generates terminals.

(c) **Ramped half-and-half**: Combination of both grow and full approaches. The population is evenly split into $m-1$ (maximum depth) parts. Half of each part uses the growing approach with maximum depth m. The remaining half uses the full approach with an increasing final depth of $1, 2, ..., m-1$ for each $m-1$ parts.

3. Evaluation function: An evaluation function is used to assess how close the execution of a given parse tree is to a solution. This evaluation function, or fitness function, is used to guide the evolution of individuals in a population towards optimal solutions.

For instance, since the program is correct, the execution of the parse tree 2.2 with the input $a = 1$ and $b = 2$ will assign to c the hypotenuse of the triangle.

4. Genetic operators: The genetic operators are essential in the evolutionary procedure. By selecting, combining and mutating individuals, it is possible to maintain, exclude or create individuals in the population. There are three leading genetic operators: selection, crossover, and mutation.

(a) **Selection**: The selection process chooses the best individuals from a population to propagate their genes. By selecting the best individuals, we try to ensure that throughout the generations, the overall fitness of the population keeps improving. A selection strategy typically used to ensure that the evolution converges quickly towards a solution is **elitism** [1], where, by the end of each generation, the best individuals transition, without changes, to the following one.

(b) **Crossover**: Crossover is the process of crossing two or more individuals in one point or multiple points [16] of their genetic information. In genetic programming, two random nodes from each tree are chosen and swapped, generating two new offspring. Figure 2.3 demonstrates a simple crossover where the left subtree of the first function is exchanged with the right subtree of the second function generating one of the two possible offspring on the combination of these two individuals.

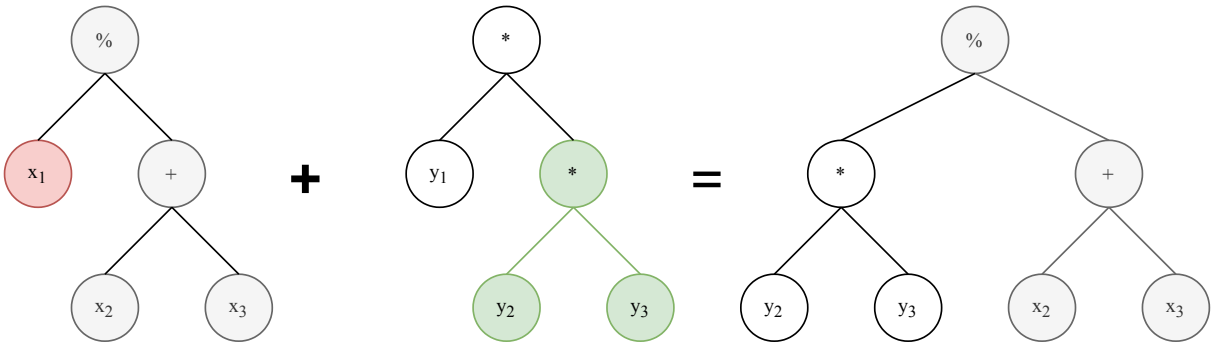


Figure 2.3: One point crossover between two Abstract Syntax Trees.

(c) **Mutation**: Mutation is an operation where a gene from an individual is chosen and replaced with another one, keeping the tree valid. In genetic programming, a random node from the tree is chosen, and a random subtree is generated. Figure 2.4 presents a simple mutation where the yellow subtree was selected to be replaced by a randomly generated subtree (the tree in green).

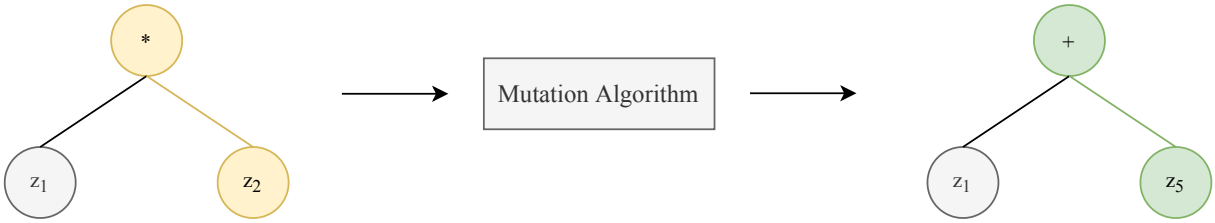


Figure 2.4: One point mutation over the Abstract Syntax Tree.

Figure 2.5, on the other hand, presents a partial mutation where part of the genetic information is reused when generating the replacement for the yellow nodes. The variable z_2 was kept, while its operation was replaced with a sum.

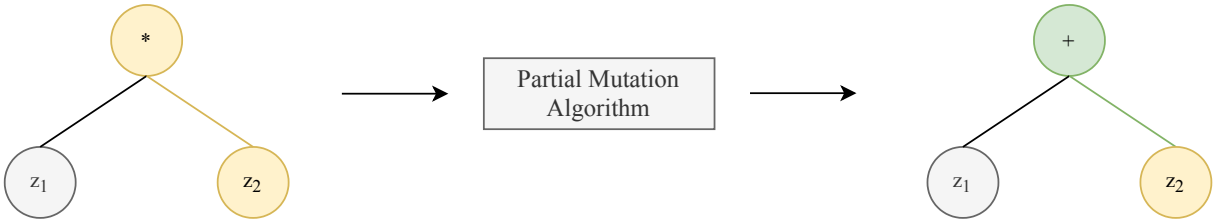


Figure 2.5: One point partial mutation with reused genetic material.

5. Parameters: A wide range of decisions made by the user are essential for the execution of the genetic algorithm. The choice of the control parameters has a massive impact on algorithm performance [17, 12]. The most used control parameters are the following: **Population size**: Amount of individuals in the population; **Amount of generations**: Maximum amount of iterations the genetic algorithm runs in the search for an optimal solution; **Maximum Depth**: Maximum depth allowed for a tree to grow during initialization, crossover, and mutation; **Mutation rate**: Probability of an individual to suffer a mutation; and **Ofsspring size**: Represents the size of the offspring population, which can be equal to the population size.

Listing 2.2 presents the pseudo-code for the evolutionary procedure previously showed.

Listing 2.2: Pseudo-code of genetic algorithm.

```
1. generate the initial population with size MAX_POPULATION
2. evaluate each individual
REPEAT
    3.1. create an empty offspring population
    3.2. apply selection from population to offspring
    REPEAT
        3.3.1. apply crossover to the population
        3.3.2. apply mutations to the offspring
        3.3.3. evaluate the offspring
        3.3.4. add offspring to the offspring population
    WHILE offspring is not complete
    3.4. replace the population with the new offspring population
WHILE current generation < MAX_GENERATIONS
4. return the fittest individual
```

2.2 Type Theory

Strongly typed programming languages force the regular programmer to define the type of variables or functions explicitly. Explicitly declaring the type improves legibility and software quality in overall [47]. It also allows type errors to be caught earlier during the type checking process.

The type of a type is called a **Kind**. Kinds are used to specify the arity of a type. For instance, the type `Integer` has the kind $*$, and the type `List` has the kind $* \rightarrow *$, `List[Integer]` is the type `List` applied to `Integer`, which has the kind $*$. Kinds are useful in several languages, like Haskell and Scala, that allow type constructors with parametric polymorphism. The parametric polymorphism increases the expressiveness of the programming language, allowing it to tackle a different kind of problems.

Parametric polymorphism allows variables, functions and data types to be written with generics so they can handle different types the same way independently on their type. Sometimes, types by themselves may not be expressive enough to restrict other behaviours [6], and so, as a way to tackle this problem, the refined types were introduced.

Refined types allow the introduction of logical predicates which restrict the values accepted by the type of a variable. Different kinds of programming languages, like Haskell [56], TypeScript [57] and SafeRestScript [8], have already studied the importance of refined types.

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 [13], a tool capable of verifying the truth value of logical predicates. The **non-liquid** refinement types cannot be statically verifiable but can be used for runtime correctness verification. When programming in Æon, the regular programmer does not need to worry about this distinction. Other than Æon, the language presented in this work, these concepts exist in other languages such as LiquidHaskell [56], Lean [14], and Coq [10].

For better understanding, let us take as an example a slightly safer version of the previous function 2.1 written in Æon.

Listing 2.3: Slightly safer hypotenuse using Pythagorean theorem in Æon.

```
1 hypotenuse({a:Integer | a > 0}, {b:Integer | is_nat(b)}) -> c:Integer {
2     c = sqrt(pow(a, 2) + pow(b, 2));
3 }
```

Listing 2.3 presents a binary function that computes the hypotenuse of a triangle using the Pythagorean theorem. The function receives two arguments, a and b that correspond to the sides of a triangle and, therefore, should not be negative. The type of argument a, a liquid 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, `hypotenuse(-1, 10)` raises a compile-time error. The argument b expresses the same condition of a, but with a non-liquid refinement. The `is_nat` function, defined elsewhere in the program or the standard library, can only be verified at runtime, when the `hypotenuse` function is called with a negative b value, for example, `hypotenuse(1, -10)`.

Although function calls may provide a hint on how to distinguish between liquid and non-liquid refinement types, this may not be entirely true. Uninterpreted functions, available in Æon, allow the user

to freely describe the composition of values, which are, à posteriori verifiable with the SMT-Solver. For instance, if we create an empty list, l, we can state its size is 0. By appending an element to the list, its size is (0 + 1), by appending a second element, the size of l is ((0 + 1) + 1). By providing this expression to an SMT-Solver, it is possible to deduce that the size of the list l is 2. Listing 2.4 presents the previous example in pseudocode by creating an uninterpreted function, `size`, and the two functions, possibly implemented in a foreign language: `empty` and `append`. Since the list type has no attributes in this pseudocode language, obtaining the size value of the previous example, `append(x2, append(x1, empty()))`, depends on calling the SMT-Solver to reduce the expressions refinement.

Listing 2.4: Example of uninterpreted functions (pseudocode).

```
size(l:list) -> int = uninterpreted;

empty() -> {l:list where size(l) == 0} {...}
append(e, l) -> {l2:list where size(l2) == size(l) + 1} {...}
```

Dependent types are types whose truth value depend on its arguments. Similarly to refinement types, dependent types help programmers improve the quality of their code by specifying its behaviour. Existing functional programming languages, like Agda, have implemented this kind of types. Listing 2.5 presents an example of a dependent type, where the return type is refined to ensure it is greater than any of the function arguments.

Listing 2.5: Dependent refinement type example in Æon.

```
hypotenuse(a : Integer, b : Integer) -> {c:Integer | c > a && c > b} {
    c = sqrt(pow(a, 2) + pow(b, 2));
}
```

Another example of dependent types can be found in Idris. Idris allows the user to create a specification where the output can be refined to consider the input parameters. Listing 2.6 presents a typical example of the Idris language where we try to concatenate two vectors [23]. By providing two vectors, each with size n and m, respectively, the function outputs a new vector whose size is the sum of its inputs vectors sizes.

Listing 2.6: Dependent refinement type of the append function in Idris.

```
append : Vect n a -> Vect m a -> Vect (n + m) a
append []        ys = ys
append (x :: xs) ys = x :: append xs ys
```

2.3 Program Synthesis

Program synthesis consists in automatically generating code compliant with a given set of input-output examples, or a given specification. Currently, there are two main strategies on program synthesis: **Deductive synthesis**, based on a formal specification; and **Inductive synthesis**, which makes use of pairs of input/output as user intention (further explained in Chapter 3).

The first challenge on program synthesis is the large search space. The available functions and terminals allow the generation of an infinite amount of combinations. In program synthesis by specification, the first step 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 `(true + 1)`. Refinement types, introducing a predicate that refines a specific type, also provide a further restriction to the space of valid generated programs. Valid Programs correctness is checked by using distinct approaches: by using examples of input/output pairs, or by providing restrictions using non-liquid refined types and testing these with random tests.

The second obstacle on program synthesis is the poor generalization performance. The ability to generalize the problem for any input and return the valid output presents a current open issue. For example, synthesis by example takes inputs and tries to generate code that respects the output. As seen previously, the overfit to introduced arguments and returned results does not generalize well to foreign arguments from the dataset. Synthesis by specification using refined and dependent types allow an easy way to generalize the behaviour of a program.

The final problem on program synthesis is the intention of the user specification. The regular programmer must provide a complete or partial specification of the problem in order to generate the code automatically. Specifications allow the program to be synthesized according to user intentions. However, the user programming experiences and its ability to express specifications represent a bottleneck to these approaches.

Chapter 3

Related Work

This chapter presents two different types of program synthesis: deductive synthesis and inductive synthesis. We present a more detailed introduction to deductive synthesis due to similarities to the current presented approach. This section also presents other types of approaches based on partial synthesis using sketches and approaches using neural networks and evolutionary computation.

3.1 Deductive Synthesis

Deductive synthesis or specification based synthesis requires the introduction of a formal specification to declare user intention. From contracts [2, 42, 43] to specification languages [34] and even using types [39], different ways have been introduced to ensure the program correctness from the specification. The study of the current main approaches is described below.

SYNQUID [45] is a framework developed by Polikarpova et al. which allows a specification based synthesis of functions. The framework primarily uses refined polymorphic types to restrict the search space of valid programs. Its unique synthesis approach allows incomplete programs to be type-checked during synthesis, and, if an incomplete program fails, then all following subprograms generated from the current program are not synthesized. However, the tool requires complete and correct refinements and the restriction of components, functions and variables that the function is allowed to use, in order to synthesize full programs. This kind of restrictions does not scale in the general-purpose programming paradigm, as it might be harder to catalogue every single required component that can be used to generate a program. This framework was later extended by Knoth et al. with the new framework **RESYN** [32], allowing user-provided resources to guide the search. Synthesized programs that do not comply with the resource bounds are rejected during synthesis.

NEO [18] was presented by Feng et al. on a new synthesis approach based on conflict-driven learning and automated reasoning. Their approach can generate new lemmas to prevent mistakes made in the previous synthesis. The synthesis algorithm was split into three components, the Decide that given a partial program with multiple holes, decides the hole and expression to be filled in, the Deduce that creates new knowledge based on the semantics and syntax and finally the conflict analyzer that detects defects on a program and identifies the root cause of the failure. As proof of concept, the synthesis

tool NEO was created. The tool was tested against Morpheus and DeepCoder on data wrangling tasks and outperformed the state-of-the-art synthesizers. Even though the surprising results, similarly to ÆON, NEO is restricted to the quality of its specifications. A second limitation of NEO is the inability to synthesize recursive functions, restraining the number of problems it can tackle. **TRINITY** [36] is a second-generation framework from NEO and developed by Martins et al. that not only performs with ease the work in data wrangling tasks but also allows expert users to extend the behaviour of the framework.

SYGUS [3] is an approach on program synthesis developed by Alur et al. which uses logical constraints and syntactic templates to restrict the space of implementations. The authors studied different kinds of inductive synthesis procedures on the search for correct programs. *Active learning* uses a query-based model to control the selection of examples that it generalizes from and can query one or more oracles to obtain both examples and labels for those examples. *Counterexample-guided inductive synthesis* (CEGIS) starts by choosing a candidate from the space of valid candidate concepts. This candidate c is presented to the verification oracle, checking for program correctness. If the candidate is correct, the synthesizer terminates; otherwise, it generates a counterexample that is added to the learning algorithm. If, after some iterations, the learning algorithm is not able to find a concept which respects the restrictions, the CEGIS procedure fails. Listing 3.1 presents an example, adapted from SyGuS [1], of a synthesized function that computes the minimum of two variables x and y. This approach requires the user to declare the syntax and operations in order to generate the code. The necessity of specifying every operation and terminal may prove to be harder than programming the function itself, limiting this approach in terms of usability.

Listing 3.1: Minimum of two variables in SYGUS.

```
;; the background theory is linear integer arithmetic
(set-logic LIA)

;; name and signature of the function to be synthesized
(synth-fun min ((x Int) (y Int)) Int

    ;; non-terminals that would be used in the grammar
    ((I Int) (B Bool))

    ;; define the grammar for allowed implementations of min
    ((I Int (x y 0 1 (+ I I) (- I I) (ite B I I)))
        (B Bool ((and B B) (or B B) (not B)
                (= I I) (<= I I) (>= I I))))
)

(declare-var x Int)
(declare-var y Int)

;; define the semantic constraints on the function
(constraint (<= (min x y) x))
(constraint (<= (min x y) y))
(constraint (or (= x (min x y)) (= y (min x y))))
```

[1] https://sygus.org/language/

```
23
24 (check-synth)
```

Genetic Programming on Deductive Synthesis Montana [39] was the first to research on the topic of program synthesis using genetic programming. In this work, Montana presents Strongly Typed Genetic Programming [37] (STGP) that uses types to constrain the search space of correct synthesized programs. This method also supports generic data structures and operations. Initially, the synthesis was tested on simple matrix operations, providing a brief insight into its potential. STGP was later capable of tackling different classes of problems with extensions to support more generic program patterns with type inheritance [24] and polymorphism [59].

On the original work, for each problem, Montana strictly follows the typical genetic programming procedure. He defines the fitness function, manually chooses the terminals and non-terminals for the evolutionary procedure, the genetic parameters and the provides the results, in case of linear regression. One of the main challenges in Genetic Programming is the large search space. The amount of operators combinations can make it difficult to to find a proper solution. So, with this insight, Grammar-Guided Genetic Programming [37] (GGGP) was created to reduce the number of operation combinations and allowing a faster synthesis of programs.

In GGGP, a structure of the solution is provided in the form of a grammar. Grammars are then used to reduce the search space, but also to sub-induce what sub-functions should be used in the target function. GGGP power on restricting the search space, make it a state of the art approach, widely used in different areas: from generating Super Mario levels [49] to mining association rules [35].

Coevolving programs with Genetic Programming [4] The work developed by Arcuri and Yao tries to use Genetic Programming and a specification to generate programs automatically. The user provides the specification which is converted into a continuous fitness function to evaluate individuals better. This type of conversion of the program specifications, allows a more expressive evaluation of the program candidates. Arcuri and Yaos' approach coevolves tests and the program at the same time, if a test is good at testing a specific program, then it should progress throughout generations. The proposed approach, however, is not capable of narrowing the number of operator combinations, since it does not take into consideration any specification restriction over the synthesis components.

3.2 Inductive Synthesis

Inductive synthesis or example-based synthesis requires the user to provide a test suite to declare its intention. The objective is to extrapolate the program behaviour from the *IO* pairs. This approach is arguably more usable than writing specification, but if the test suite is not expressive enough [46], this approach cannot synthesize generalized programs. Different techniques [31] have been introduced in this particular area, in this section we present some of the main approaches.

CEGIS [51] is an acronym for Counterexample Guided Inductive Synthesis, and it is a crucial concept in inductive synthesis. The main objective of CEGIS is to correctly choose a set of input pairs so that

any synthesized program is generalized for those inputs. CEGIS execution is based in two main components: a checking system for the candidate solutions, capable of generating counterexample inputs; and the inductive synthesizer, which generates candidate programs according to the counterexamples. Each counterexample input represents a different behaviour of the program, which no previous counterexample was able to detect. The system continuously produces new candidates according to the detected counterexamples, until it can generate a valid candidate. Figure 3.1 presents a visual description of the procedure done by CEGIS.

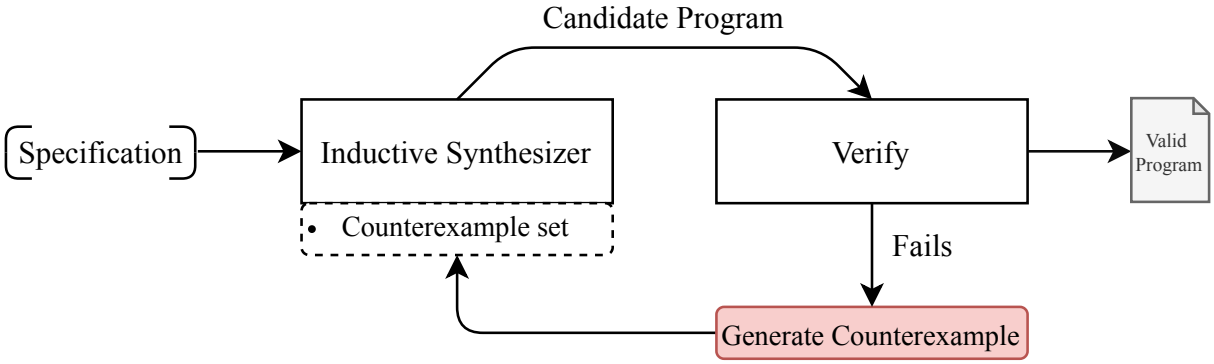


Figure 3.1: CEGIS counterexample and candidate programs synthesis flow.

MYTH [41] MYTH is a prototype tool based on OCaml for testing the Type-and-Example-directed Program Synthesis approach. Similarly to GGGP, this approach uses the type system as a refinement for the combinations of the components. The user intention is provided in the programming language from use cases. The system uses a bidirectional type checking [44] for synthesizing recursive functions.

The synthesis is divided into three main components: Type refinement, in which the outputs provided to the function are set as goals and bounded to the proper inputs parameters and respective values; Guessing, where the ill typed and non-well formed expressions and finally, it matches the example expressions as a way to guide the synthesis.

Program Synthesis Using Natural Language [15] This approach relies on neural networks and uses natural language as a description of the problem, and outputs a set of ranked programs. Internally, to build the synthesis network, the synthesis programmer is required to provide two components: a domain-specific language (DSL) with the existing operations and their combinations; and, examples of descriptions in natural language with the respective programs in the DSL. Since natural language is ambiguous, instead of presenting one final solution, it provides a ranked set with the top programs which may fit the description.

3.3 Synthesis from Sketches

SKETCH [50] allow programmers to express their insights on a problem by providing a partial program that encodes the structure of the solution. The programmer must also provide all the operations that the problem requires in order to generate the code automatically.

The SKETCH framework is composed by two parts, the core SKETCH language and the SKETCH itself constructed as syntactic sugar on top of the core. It also uses counterexample guided inductive synthesis (CEGIS) to generate the candidates, check their correctness, learn from counterexamples until it finds the correct program or fails if there are no programs that follow the specification.

The SKETCH approach main issue arises with the restriction of the synthesis components. The programmer is required to provide only the exact components and their combinations in order to generate the sketch. This type of approach does not scale for general purpose synthesis problems, where the objective is to get rid of the programmer worries on declaring the components. Not only that, but the increasing amount of variables used exponentially increases the runtime.

EPS [7] also known as Evolutionary Program Sketching, was a work later developed by Bladek and Krawiec where they take the original SKETCH approach and replace the human programmed sketches for evolved sketches with holes with Genetic Programming. This approach uses an SMT solver to synthesize the holes in the program, and test cases to evaluate the candidate programs. Unfortunately, the approach only takes into consideration constants and input variables, since adding extra information would explode in complexity for the SMT solver.

Chapter 4

The ÆON Programming Language

4.1 Main Concept

This chapter presents the essentials of the ÆON programming language, examples of its features, and the internal implementation, distinguishing the classes of refinements and their importance towards program synthesis.

We propose ÆON as a language for expressing the program specification using its rich type system. ÆON is a general-purpose functional programming language that uses refined and dependent types to synthesize complete or partial programs. ÆON syntax similarity to Python allows new developers who never engaged with refined and dependent types to program in it.

Like many other functional programming languages, such as Haskell and Scheme, ÆON contains native functions, lambda expressions, types creation, type abstractions and applications. The creation of new types and type abstractions enriches ÆON expressiveness allowing it to tackle different areas of program synthesis.

ÆON provides the programmers with the ability to create specifications in functions through the type system. These specifications not only ensure the correctness of a function but also allow the synthesis of full or partial functions. Generalized program synthesis distinguishes ÆON from the remaining state of the art approaches. The regular developer can quickly specify the wanted function by defining the input and output type refinements. The language allows the introduction of holes (■) in the program body implementation. These holes inform the compiler that it is required to synthesize an expression for each hole defined with the help of the evolutionary procedure (Chapter 7).

4.2 Examples

This section presents examples of the ÆON programming language by implementing types and functions that compose an image library. The examples are ordered by ease of comprehension and demonstrate different features of the language.

Listing 4.1: Type Aliases and Type Declarations in ÆON.

```
1 import aeon/libraries/list;
2
3 type Nat as {x:Integer | x >= 0};
```

```
4  type ColourInt {x:Nat | x <= 255};
5
6  type Colour {
7      red : ColourInt;
8      green : ColourInt;
9      blue : ColourInt;
10 }
11
12 type Image {
13     width : Nat;
14     height : Nat;
15     pixels : List[Colour];
16 }
```

Listing 4.1 presents a small introduction to the Æon programming language. Æon allows the user to import either natively implemented functions, and types or other user-defined functions.

The first feature of the language is type aliases, where the user can define new types to improve program usability. The first type alias, `Nat`, is defined as an `Integer` whose value is greater than 0. This type alias definition is then used to quickly help define a `ColourInt`, by further refining the allowed values up to 255.

It is also possible to declare new types. These type declarations may or not have ghost variables associated with them. In example 4.1, we can find two new declarations of types `Colour` and `Image`, each with their variables. In the `Colour` type, we can find three different variables, typically associated when defining a colour (RGB), which describes the behaviour of its type. We describe what composes an image, its width, height, and, in this case, following a simple representation, a list of the pixels that compose the image. In line 15, the type `List[Colour]` of the pixels variable presents the type application of the imported polymorphic type `List` to the type `Colour`.

Listing 4.2: Native function declaration in Æon.

```
1  get_red(colour:Colour) -> {c:ColourInt | c == colour.red};
2  get_green(colour:Colour) -> {c:ColourInt | c == colour.green};
3  get_blue(colour:Colour) -> {c:ColourInt | c == colour.blue};
4  build_colour(red:ColourInt, green:ColourInt, blue:ColourInt) -> {c:Colour |
5          c.red == red && c.green == green && c.blue == blue};
6
7  load_image(path:String) -> Image;
8  save_image(img:Image, path:String) -> Image;
9  build_image(l:List[Colour]) -> {img:Image -> img.pixels == l};
10 get_colours(img:Image) -> {l:List[Colour] | img.pixels == l};
```

Æon allows the programmer to define native functions without providing the function body, as seen in Listing 4.2. The functions specification is written in Æon and the implementation in a foreign language, like Python. The user can then use the specified functions in their programs. Listing 4.2 example declares three new functions to obtain the colours in the `Colour` type, and also a function that allows it to create a new function. When declaring native implemented functions, it is recommended to refine the input and output types as much as possible, as these will be essential, not only to ensure program

correctness but also to help the program synthesis procedure.

Listing 4.3: Inverting the colours of the Guernica painting in Æon.

```
guernica : {img:Image | img.width == 766 && img.height == 349} =
↪  load_image("guernica.jpg");

invert_colours(pix:List[Colour]) -> {l:List[Colour] | l.size == pix.size} {
    if len[Colour](pix) == 0 {
      pix;
    } else {
        colour : Colour = head[Colour](pix);

        red : ColourInt = 255 - get_red(colour);
        green : ColourInt = 255 - get_green(colour);
        blue : ColourInt = 255 - get_blue(colour);

        result : List[Colour] = invert_colours(tail[Colour](pix));

        add_first[Colour](build_colour(red, green, blue), result);
    }
}

cool_guernica : {i:Image | i.width == guernica.width && i.height ==
↪  guernica.height} = build_image(invert_colours(get_colours(guernica)));

save_image(cool_guernica, "cool_guernica.jpg");
```

Listing 4.3 presents a simple example of an implementation of a function in Æon. In this case, the goal is to load the famous painting from Pablo Picasso, Guernica, naively invert its colours and save the new image.

The program starts by creating a new definition of the guernica variable by loading it using the native implemented function `load_image`. The result is saved on the guernica variable, and its type is refined to ensure the proper image sizes.

The `invert_colours` function receives a list of colours as input and outputs a new list with the same size as the original one, but with its colours inverted. The first statement checks whether the list is empty or not. It is essential to notice that every method containing a refined polymorphic type must have a type application for each type abstraction. Every function's invocation on the list library functions also needs to be applied to the type `Colour`.

The else body explanation is pretty straightforward. We obtain the colour we intend to invert from the head of the list, invert its colours, recursively apply the function to the remaining list, and append the inverted colour to the head of the result list.

Then we build a new image by invoking the `invert_colours` function we just implemented. The definition type of the `cool_guernica` variable shows a dependent type, where the width and height must be the same as those of the original image.

Finally, the image is saved by providing the previously defined cool_guernica inverted image and its location path.

Listing 4.4: Gallery of pictures in Æon.

```
type Date {
    day : {d:Nat | d <= 31};
    month : {m:Nat | m <= 12};
    year : {y:Nat | y < 2020};
}

type Picture {
    image : Image;
    date : Date;
    author : String;
}

type Gallery[T] {
    files : List[T];
    creation_date : Date;
    author : String;
}

get_pictures(g:Gallery[Picture]) -> List[Picture];
exists_picture(img:Picture, g:Gallery[Picture]) -> Boolean;
remove_picture(img:Picture, g1:Gallery[Picture]) -> {g2:Gallery[Picture] |
↪   !exists_picture(img, g2)};
copy_picture(img:Picture, g1:Gallery[Picture], g2:Gallery[Picture]) ->
↪   {g3:Gallery[Picture] | g3.files.size == g2.files.size + 1 and
↪   exists_picture(img, g3)};

copy_all_pictures(g1:Gallery[Picture], g2:Gallery[Picture]) -> {g3:Gallery
        | (g3.files.size == g1.files.size + g2.files.size) and
        forall(\p:Picture -> exists_picture(p, g3), get_pictures(g2)) and
        forall(\p:Picture -> exists_picture(p, g3), get_pictures(g1))} {
    ??;
}
```

This final example intends to present what a simple program synthesis program is like in Æon.

The first two type declarations do not introduce anything new and were already previously explained; we declare two new types, the `Date` and `Picture` type, each with their proper ghost variables. The third type declaration presents a type abstraction. Since we can have a gallery of music, images, and even videos, the type of files that belong to a gallery should be generic, and the introduction of the `T` type, denoted by a single uppercase letter, allows the introduction of the polymorphic type.

The four different native methods' declaration allows the creation of constructors for the `Gallery` type.

The crucial part of this example lies in the last declaration. The `copy_all_pictures` is a function that receives two non-mutable galleries of images and copies all the pictures from the first gallery to the second one. The body of this function is denoted with the hole, `??`, operator, indicating that this function will be synthesized. The specification is provided on the type system with three conditions.

The first condition, `g3.files.size == g1.files.size + g2.files.size`, is a liquid type which ensures that the amount of files in the output gallery contains the same amount of the two input galleries combined.

The following two conditions ensure that every image which existed in the input also exists in the output. Combining these three conditions ensure that all the images on the input exist in the output.

Even though indistinguishable in terms of syntax, there are two classes of refinements in Æon, with two different purposes. The first condition is a liquid type, meaning that it can be statically verified using a Satisfiability Modulo Theories (SMT) solver. These types will be essential on the synthesis of valid expressions on the non-deterministic synthesizer (Chapter 6). The second and third conditions are non-liquid types, meaning that they cannot be statically verified, since its verification relies on the execution of the program. These types will be necessary for the evolutionary synthesis algorithm (Chapter 7) to evaluate candidate programs and obtain their performance. At the end of this procedure, we obtain the implementation of the function.

4.3 Implementation Details

This section presents the implementation structure of Æon components essential for the synthesis of programs. Each component is further explained in its respective section in the remaining of this work.

Two essential parts compose the Æon program synthesis procedure: the non-deterministic synthesizer, responsible for synthesizing valid expressions from liquid refined types (Chapter 6), and the evolutionary program synthesis, that evolves programs according to the non-liquid refined types, allowing complete or partially holed programs to be generated, (Chapter 7).

Figure 4.1 briefly describes the flow of the Æon programming language compiler. Firstly, the file is parsed and an Abstract Syntax Tree is created using the type system of ÆonCore. In ÆonCore the language types are checked, and if a hole is found during type checking, the responsibility for synthesizing the code is delegated to the evolutionary synthesis module. However, only programs that are valid and type checked, proceed to the synthesis module. For instance, if the user provides a specification that is not valid (e.g. `{x : Integer | false}`), the compiler raises an exception.

The evolutionary synthesis receives a holed program, fully or partially incomplete, and using Genetic Programming (GP) it tries to search, within the produced valid synthesized expressions, for an expression which complies with the refinements. During the synthesis' procedure, the evolutionary program synthesis uses the synthesizer to non-deterministically generate valid programs from the liquid types and then uses the non-liquid types to check the program's correctness. By the end of this procedure, the GP produces a valid and optimized program.

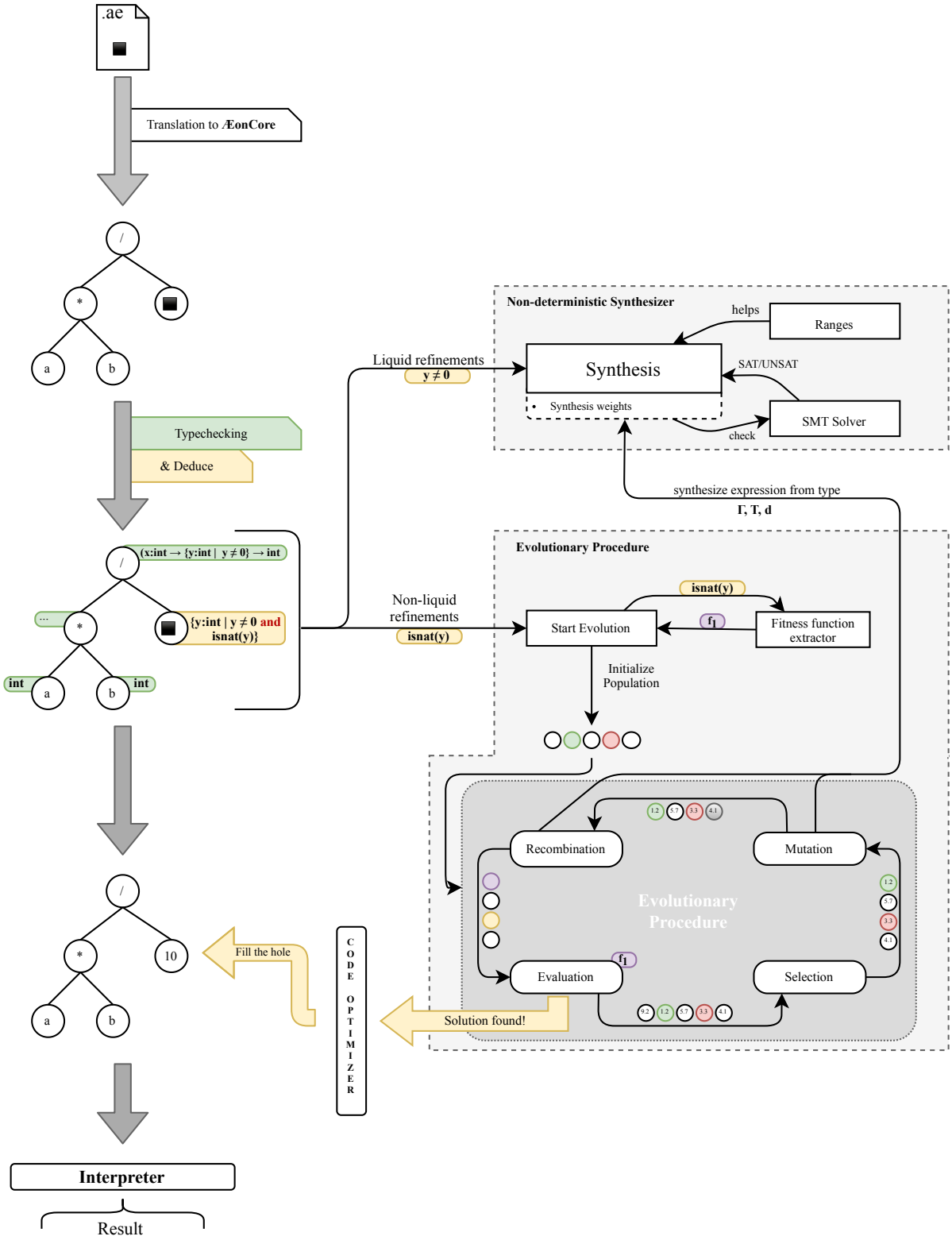


Figure 4.1: Flow of the compiler in Æon.

Chapter 5

Translation to ÆonCore

This chapter presents the type system, the essentials of ÆonCore and the motivation behind the conversion from Æon to ÆonCore, and the automatic inference of the hole type.

5.1 Æon conversion to ÆonCore

ÆonCore is a functional programming language with refinement types used internally in Æon whose tree representation allows an improved interaction between the non-deterministic and evolutionary synthesis modules. It exhibits a complex syntax that strictly follows the type system rules described in Section 5.2. The ÆonCore has an abstract syntax tree-based representation where typed nodes represent expressions of the system. The straight representation of the type system is proof of ÆonCore versatility, allowing the creation and linkage of different syntactic frontends.

In order to improve usability, Æon was created with a user-friendly syntax, allowing new programmers to engage in the language quickly. Æon acts like a syntactic frontend by being directly parsed to ÆonCore.

Listing 5.1 presents the factorial function written in a programming language that follows ÆonCore tree representation. For the sake of the reader, the type IntToInt was created and used to simplify the factorial function.

These two examples illustrate the syntax dissimilarities over ÆonCore and Æon.

Listing 5.1: Factorial in ÆonCore.

```
1 type IntToInt = (a:Integer) -> Integer
2
3 factorial : IntToInt = fix[IntToInt] (\f:IntToInt -> \n:Integer -> if (n ==
  ↪  0) then 1 else (n * f(n - 1)));
```

Listing 5.2: Factorial in Æon.

```
1 factorial(n:Integer) -> Integer {
2     if (n == 0) then 1 else n * f(n - 1);
3 }
```

As stated, ÆonCore representation is an abstract syntax tree with typed nodes that are expressions from the type system. Æon syntactic-sugar frontend is converted to this representation during the lexing/parsing of the language.

Values that represent basic types of the language, such as integers, doubles, strings and booleans, are transformed into Literal nodes.

Variables are converted to Var nodes with the variable name.

Holes in the program are directly converted to the Hole node and the type included. During the conversion, if the hole type was not provided, the deducing procedure described in Section 5.3 is triggered, and its type is automatically infered.

Import statements are temporarily converted to Import nodes with the file path. Once the conversion is over, import nodes are evaluated, and the file paths are read and loaded.

Æon allows the programmer to create a new alias for certain types. Listing 5.3 presents a type declaration of the Natural type by restricting the Integer domain. Type aliases are directly transformed into TypeAlias nodes, saving the type and alias, in ÆonCore.

Listing 5.3: Type alias in Æon.

```
type Nat as {x:Integer | x > 0};
```

Declaring a type is strictly converted to a TypeDeclaration node in ÆonCore. Let us consider the types Bus and Car declared in Listing 5.4. The type Bus is directly converted to a TypeDeclaration node. On the other hand, the Car type not only is converted to the TypeDeclaration node, but its ghost variables are used to generate uninterpreted functions. These functions are invoked when calling the variable name over the type. The type alias Ford exemplifies the syntactic sugar invocation of the uninterpreted functions, where the x.brand is converted to Car_brand(x). It is also possible to have nested uninterpreted functions (e.g. car.owner.age is converted to Person_age(Car_owner(x))).

For each ghost variable of the type Car, the proper uninterpreted functions is generated, and the ghost variables access are correctly converted in Listing 5.5.

Listing 5.4: Type declarations in Æon.

```
import person;

type Bus;

type Car {
    year : Nat;
    brand : String;
    registration : String;
    owner : {p:Person | p.age > 18};
}

type Ford as {x:Car | x.brand == "Ford"};
```

Listing 5.5: Uninterpreted functions of the Car type.

```
Car_year : (x:Car) -> year:Nat = uninterpreted;
Car_brand : (x:Car) -> brand:String = uninterpreted;
Car_registration : (x:Car) -> registration:String = uninterpreted;
Car_owner : (x:Car) -> {owner:Person | p.age > 18} = uninterpreted;

type Ford = {x:Car | Car_Brand(x) == "Ford"};
```

Æon allows the user to declare functions which are converted into a Definition node. A definition node contains the information related to the function (name, type, return type, and body). The conversion of functions from Æon to ÆonCore present a more demanding challenge. ÆonCore only allows one single expression composed by abstractions and applications. The multiple statements in Æon must be converted into multiple nested abstraction and application for each statement. For better understanding, consider the example 5.6, a function which creates and updates the ownership of a vehicle in a database. It is important to notice that these ghost variables do not exist at runtime, only at the type level.

Listing 5.6: Update of vehicle ownership in a database in Æon.

```
updateOwner(owner:String, registration:String) -> Integer {

    sql : String = "UPDATE vehicle SET owner=? where registration=?";

    print(sql);

    sql = setParameter(sql, owner);
    sql = setParameter(sql, registration);

    executeUpdate(sql);
}
```

Each statement is converted into an abstraction and the body is applied to it. For instance, the statement in line 6 would be converted to `_:Top = print(sql)`. Starting by the last statement being computed, it is nested with the application of abstraction of the previous statement.

Listing 5.7 presents the tree representation of the conversion from Æon to ÆonCore.

Listing 5.7: Vehicle ownership in a database converted to ÆonCore.

```
Application(
    Abstraction("sql", String,
        Application(
            Abstraction('_', Top,
                ...
            ),
            Application(Var("print"), Var("sql"))
        )
    ),
    Literal("Update vehicle SET owner=? where id=?")
)
```

Function invocations are traduced into multiple nested applications. For example, the sum of two values, $x + y$, is converted to ((+ x) y)

It is possible to find two different kinds of **if** in Æon. **If** statements allow multiple expressions in the body of the **then** and **else**. The body is converted into a single expression using the previous conversion procedure. **If** expressions, (e.g. **if** cond **then** expression **else** expression), only allow a single expression in the body of **then** and **else**.

Lambda functions are directly converted into Abstractions.

5.2 Syntax

Kinds	k	$::=$	$* \mid k \to k$
Types	T	$::=$	**Integer** $\mid$ **Boolean** $\mid t \mid x : T \to T$
			$\mid x : T$ **where** $e \mid \forall t : k.T \mid TT$
Expressions	e	$::=$	**true** $\mid$ **false** $\mid n \mid x \mid ??$
			$\mid$ **if** e **then** e **else** $e \mid \lambda x : T.e \mid ee$
			$\mid \Lambda t : k.e \mid e[T]$
Contexts	Γ	$::=$	$\varepsilon \mid \Gamma, x : T \mid \Gamma, t : k \mid \Gamma, e$

Figure 5.1: The syntax of ÆonCore programs.

Figure 5.1 presents the syntax of ÆonCore programs which are composed by kinds, types, expressions and contexts.

Kinds As referred in Section 2.2, a kind is associated to each type. Regular types, like **Integer**, have the kind $*$, while type applications, types applied to types, recursively have the kind $k \to k$ (e.g. `(List Integer)` with the kind $*$).

Types Æon includes native basic types, like **Integer**, **Boolean**, and also custom created types, t, through type alias or type declaration. There are also implemented two native types, **Bottom** and **Top**, which stand for the empty type, which is the subtype for all types, and the universal base type, where all the other types are its subtype, respectively.

The type $(x : T_1 \to T_2)$ is a dependent function that receives as input the type T_1, assigned to variable x, and returns a type T_2, where x occurs freely in T_2. A type $x : T \to U$ is abbreviated to $T \to U$ when x does not occur in U. For better understanding, consider Listing 5.8 written in ÆonCore, which computes the power of a given value using a nested dependent function.

Listing 5.8: Power with two dependent variables in ÆonCore.

```
1 type Nat = {x:Integer where x >= 0}
2 pow : Nat -> (Nat -> Nat) = \x:Nat -> y:Nat -> if y == 0 then 1 else x *
  ↪  ((pow x) (y - 1));
```

Listing 5.8 recursively calculates the power of a value. In order to improve readability, the first line aliases a new type `Nat` from the refined `Integer` type. The *pow* function is declared by providing its type and the assignment body. The pow function type states it receives two natural numbers and returns a natural value.

The rule $x\colon T$ **where** e expresses a refinement of the type T for which the predicate e holds; term variable x may occur in e (but not in T).

Polymorphic types are introduced with $\forall t\colon k.T$, for t a type variable and k a kind governing the "shape" of t. The body T of the type may contain free occurrences of t.

A type of the form (TU) denotes application. For better understanding, let us consider the type T of the type $\forall t\colon *.(x\colon t \to \textbf{Boolean})$ and U has kind $*$, then the application of the type T to U, TU, is convertible to $x\colon U \to \textbf{Boolean}$.

Expressions ÆonCore contains natively implemented expressions from the basic types (**Integer**, **Boolean**, **Double**, **String**). It is also possible to declare new variables, x. The ?? operator denotes a hole in the program.

Listing 5.8 presents an example of the abstractions implemented in ÆonCore. $\lambda x\colon T.e$ declares that x is of type T and may occur in the abstraction body e. In correlation to 5.8, we declare x of type `Nat`, and y of type `Nat`, then x and y are used in the abstraction body to calculate the power.

Expressions application, $e_1 e_2$, in ÆonCore is made resorting to currying, reading e_2 is applied to e_1, where e_1 type is an abstraction type. For instance, in 5.8, the expressions `y == 0` and `y - 1` are syntactic sugar to `((== y) 0)` and `((- y) 1)`, respectively.

The polymorphism over expressions is built using type abstraction expressions, $\Lambda t\colon k.e$, and type application expressions, $e[T]$. The type abstraction introduces a new type that can be used in its body. On the other hand, the type application expression takes an expression, whose type is a type abstraction, and substitutes all occurrences of the type abstraction declaration by the applied type.

Contexts ÆonCore contexts may be empty, contain variables and the respective type, declared types and its kind, and finally boolean expressions which correspond to the propagation of the refined type conditions and `if then else` conditions.

The type system and type formation have been independently and continuously developed by the research team at LASIGE and can be found in Appendix A.

5.3 Hole Type Inference

Æon allows the introduction of holes, **??**, in the program whose expressions will be synthesized and replaced in the hole, according to the type. However, writing the type of the hole can either be tedious and repetitive or too hard for the regular programmer. To prevent this, we automatically infer the type of the hole, or multiples holes according to the specification in the type system.

Figure 5.2 presents a proposal of the hole type inference algorithm to infer the type of the hole according to its context and update the holes parents types automatically. The types of the remaining nodes were annotated during typechecking and are kept unchanged, regardless of the algorithms deductions system. The notation is straightforward: by default we are only infering the hole and updating its parents, the

$e \mapsto T$ infers the type T from the expression e. Since its node is already pre-annotated with its type from the typechecker, its type is presented in the form $e : T$.

$$\frac{\vdash \Gamma \text{ context}}{\Gamma \vdash n : \textbf{Integer} \mapsto \textbf{Integer}} \qquad \frac{\vdash \Gamma \text{ context} \quad b = \textbf{true}, \textbf{false}}{\Gamma \vdash b : \textbf{Boolean} \mapsto \textbf{Boolean}} \qquad \frac{\vdash \Gamma \text{ context} \quad (t \text{ fresh})}{\Gamma \vdash ?? : \textbf{Bottom} \mapsto \forall t : \star.t}$$

(D-Int, D-Bool, D-Hole)

$$\frac{\vdash \Gamma \text{ context} \quad x : T \in \Gamma}{\Gamma \vdash x : T \mapsto T} \qquad \frac{\Gamma, x : T \vdash e \mapsto U}{\Gamma \vdash (\lambda x : T \to e) : (x : T \to U) \mapsto (x : T \to U)}$$

(D-Var, D-Abs)

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \mapsto \textbf{Boolean} \quad \Gamma, e_1 \vdash e_2 \mapsto T \quad \Gamma, e_1 \vdash e_3 \mapsto U \\ \Gamma, e_1 \vdash e_2 \mapsto \text{ glb}(V, T) \quad \Gamma, \neg e_1 \vdash e_3 \mapsto \text{ glb}(V, U)\end{array}}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : V \mapsto V}$$

(D-If)

$$\frac{\Gamma, t : k \vdash e \mapsto T}{\Gamma \vdash (\Lambda t : k.e) : (\forall t : k.T) = V \mapsto V} \qquad \frac{(t \text{ fresh}) \quad \Gamma \vdash e \mapsto \forall t : k.U \quad \Gamma \vdash T : k}{\Gamma \vdash e[T] : U[T/t] \mapsto U[T/t]}$$

(D-TAbs, D-TApp)

$$\frac{\Gamma \vdash e \mapsto T \quad (x \text{ fresh}) \quad (t \text{ fresh}) \quad \Gamma \vdash ?? \mapsto \|\forall t : \star.(x : T \to U)\|_{\blacksquare}}{\Gamma \vdash (??\ e) : U \mapsto U}$$

(D-LApp)

$$\frac{\Gamma \vdash ?? \mapsto T \quad \Gamma \vdash e \mapsto \|(x : T \to V)\|_{\blacksquare}}{\Gamma \vdash (e\ ??) : V \mapsto V}$$

(D-RApp)

$$\frac{\Gamma \vdash e_2 \mapsto T \qquad \Gamma \vdash e_1 \mapsto (x : T \to U)}{\Gamma \vdash e_1 e_2 : U \mapsto U[e_2/x]}$$

(D-App)

Figure 5.2: Hole deduction rules.

The first two rules, D-Int and D-Bool, return the types from the native expressions, such as integers and booleans. The D-Hole, returns the type of the hole, in which, if not annotated by the user it is automatically deduced as a type abstraction applied to the type itself. The introduction of the type abstractions as the type of the hole informs the evolutionary synthesis that it is required to non-deterministically synthesize the type for this hole in order to generate a proper expression. This way, we are able to introduce diversity in the synthesized expressions and create independence of the type for the same hole. The rule that expresses the type over a variable is obtained from the D-Var rule.

The D-Abs rule defines the abstractions body deduction. The return type of the abstraction is strictly linked with its body or any application it may be wrapped. The type inferer takes into consideration both types and assigns the proper return type to the abstraction. The expression e type should be the return type decided by the inferer.

The D-If deduce rule needs to deduce the type of three expressions. Firstly, the condition of the if expression is required to be a **Boolean**. Then, it tries to deduce the types in both e_2 and e_3. The type of e_1 expression is the greatest lower bound (glb) between the type of the **if** expression and e_2. In the situation where e_1 is a hole, the inferer will assign the if type. The same operation is executed over the e_3 expression.

The D-TAbs tries to deduce the type of e, and assigns it the type application of the type abstraction.

The D-TApp rule works quite similarly, it assigns to the expression a new type abstraction that is being applied to T. If e is not a hole, it keeps the previous type, otherwise, it generates the new type abstraction.

The last expression to be deduced is the Application. Deducing an application presents a stiffer challenge since there can be nested holes. The first application deduce rule, D-LApp, deduces the application target when it is a hole. Firstly, we obtain the type of the argument expression. Then we generate a new variable and a fresh type, x and t, to create the abstraction type. Since it is a hole, the abstraction type is wrapped on a type abstraction of type t and kind $*$. This rule is wrapped with the $\|T\|_{\blacksquare}$ notation. Since T can be obtained from a deduced hole, and the type abstractions inside the argument type of the abstraction type are not allowed, this notation tells us that the type abstractions over the deduced holes are propagated outside the abstraction type. For example, the following type $\forall t : \star.(x : (\forall t_2 : \star.t_2) \to U)$ is converted to $\forall t : \star.\forall t_2 : \star.(x : t_2 \to U)$.

The second application rule, D-RApp, is applied when the argument of the application is a hole. Its implementation is straightforward: first, the type of e is synthesized, and since the hole is the argument of e, its type is required to be T. If the whole expression is nested holes, the return type V, obtained from the return type of the abstraction type of e, is wrapped within the appropriated type abstractions.

The last application deduction rule is the regular application of two expressions. It synthesizes the type of e_2 and the type of e_1, ensuring that the argument is of type T. The returned type is the type U, with a substitution of expression by the expression on a type.

The following practical examples and respective derivations introduce some of the main challenges solved by this mechanism.

Listing 5.9: Deducing the hole type of a car list update in Æon.

```
import aeon/libraries/list;

updateOwner(l:List[Car], old:Person, new:Person) ->
    {l2:List[Car] | l.size == l2.size and
    length(getCars(old, l2)) == 0 and
    length(getCars(new, l2)) == length(getCars(new, l)) +
    ↪  length(getCars(old, l))} {
    ??;
}
```

Listing 5.9 shows a simple example of the deduction of the hole, where the programmer updates the ownership of its cars. In this simple case, the output of the hole corresponds to the return type of the function.

Most difficult cases arise from the partial synthesis of programs. As different holes type may depend on each other, we need to calculate and propagate types between holes that may be linked.

The following pseudocode in Æon present different examples for the deduce procedure in the partial synthesis of programs.

Listing 5.10: If expression deduction in Æon.

```
deduce1(l:List[Car], old:Person, new:Person) -> {l2:List[Car] | cond} {
    if ?? then ?? else ??;
}
```

$$\frac{\Gamma \vdash ??_1 \mapsto \textbf{Boolean} \quad \Gamma, ??_1 \vdash ??_2 \to (\forall t_2 : \star . t_2) \quad \Gamma, \neg ??_1 \vdash ??_3 \to (\forall t_3 : \star . t_3) \quad \Gamma \vdash ??_2 \mapsto \text{glb}(\forall t_2 : \star . t_2, V) = V \quad \Gamma \vdash ??_3 \mapsto \text{glb}(\forall t_3 : \star . t_3, V) = V}{\Gamma \vdash \textbf{if}\ ??_1\ \textbf{then}\ ??_2\ \textbf{else}\ ??_3 : V = \{l2 : (List\ \ Car) \mid cond\} \mapsto V}$$

Figure 5.3: Derivation of Listing 5.10 using the D-If rule.

Listing 5.10 is an example that contains three holes. The first hole, the condition of the if expression, requires it to be of **Boolean** type. The then and otherwise body types are required to be the same as the return type. The type abstraction deduced from the holes say that it accepts any type application of kind $\star$, and so, the return type of the function can be type applied, correctly deducing the type of the hole.

Listing 5.11: Deducing the hole type of a car list update in Æon.

```
1 deduce2(l:List[Car], old:Person, new:Person) -> {l2:List[Car] | cond} {
2     ??;
3     ...;
4 }
```

Listing 5.11 presents an even simpler example. All holes in the middle of statements, which are not let statements, must return deduced type from the D-Hole rule, which means that any expression of any type can be synthesized.

Listing 5.12: Target deduction of function application in Æon.

```
1 deduce3(l:List[Car], old:Person, new:Person) -> {l2:List[Car] | cond} {
2     ??(old);
3 }
```

$$\frac{\Gamma \vdash old \mapsto Person \quad (x \text{ fresh}) \quad (t \text{ fresh}) \quad \Gamma \vdash ?? \mapsto \|\forall t : \star . (x : Person \mapsto V)\|_{\blacksquare}}{\Gamma \vdash (??\ old) : V = \{l2 : (List\ \ Car) \mid cond\} \mapsto V}$$

Figure 5.4: Derivation of Listing 5.12 using the D-LApp rule.

Listing 5.12 tries to deduce the target of an application. In this case, the hole is applied to an argument; therefore its type must be an abstraction type, where its argument is the type synthesis of the *old* variable and its return type the return type of the function, `(x:Person -> {l2:List[Car] | cond})`. The return type represents one of the exception cases caught by the auxiliary notation, $\|T\|_{\blacksquare}$. Typically, the return type would be t, but since, a greater type than t is used as return type (in this case, the functions return type), then it is assigned to V.

Listing 5.13: Argument deduction of function application in Æon.

```
append_list[T](e:T, l:List[T]) -> {l2:List[T] | l2.size == l.size + 1};

deduce4(l:List[Car], old:Person, new:Person) -> {l2:List[Car] | cond} {
    append_list[Car](??, l);
    ...;
}
```

In Listing 5.13, a similar concept is applied from 5.12. The `append_list` is type applied to the `Car` type, and, all the locations of the **T** type are replaced by `Car`, being an argument of the `append_list`, it means that the hole type is deduced to be `Car`.

Listing 5.14: Full function deduction in Æon.

```
deduce5(l:List[Car], old:Person, new:Person) -> {l2:List[Car] | cond} {
    ??[Car](??, ??(l));
    ...;
}
```

$$\dfrac{\dfrac{(x_3 \text{ fresh}) \quad (t_3 \text{ fresh}) \quad \Gamma \vdash l \mapsto (List\ Car)}{\Gamma \vdash ??_3 \mapsto \|\forall t_3 : \star.(x_3 : (List\ Car) \to t_3)\|_{\blacksquare}} \; {}^{*below}}{\Gamma \vdash ((??_1\ Car)\ ??_2)\,(??_3\ l)) : V = \{l2 : (List\ \ Car) \mid cond\} \mapsto V}$$

Figure 5.5: Derivation of Listing 5.14 using multiple deduce rules.

$$\dfrac{\dfrac{\vdash \Gamma \text{ context} \quad (t_2 \text{ fresh})}{\Gamma \vdash ??_2 \mapsto \forall t_2 : \star.t_2} \quad \dfrac{(t_1 \text{ fresh}) \quad \Gamma \vdash Car : \star}{\Gamma \vdash ??_1 \mapsto \|\forall t_1 : (* \to *).(\forall t_2 : \star.\forall t_3 : \star.(x_2 : t_2 \to x_3 : t_3 \to V)\ Car)\|_{\blacksquare}}}{\Gamma \vdash ((??_1\ Car)\ ??_2) \mapsto \forall t_3 : \star.(x_3 : t_3 \to V)}$$

Figure 5.6: *Continuation:* Right side derivation of the D-App rule for Listing 5.14.

The last example, Listing 5.14, presents a more complex deduction of the holes. Starting with the first argument, from the rule D-Hole, its type is `t3:*.t3`. The second argument of the function call is a hole applied to the argument l, following the D-LApp rule, this second hole, which corresponds to the second argument, is of type `t2:*.(x2:List[Car] -> t2)` (because of the lack of space, the x_2 fresh is implicitly defined). Finally, the most external hole is applied to two arguments and type applied to `Car`. Following the rules, the last hole type is `t1:(*->*).((t2:*.t3:*.(x2:t2 -> (x3:t3 -> V))) Car)`.

By the end of this translation, we obtain a AST in ÆonCore with annotated types and infered hole types. The translation will allow the usage of the AST as part of an individual in the evolutionary procedure. The infered holes are also prepared for the further steps in RTGP. The evolutionary synthesis will provide a type for the type abstractions if necessary, and the non-deterministic synthesizer will generate all the needed expressions or types, according with the liquid refinements, for the genetic programming.

Chapter 6

Non-deterministic Synthesis From Liquid Types

This chapter presents the synthesis algorithm for the non-deterministic synthesizer for the liquid refinement types. This algorithm is capable of synthesizing kinds, types and expressions. Figure 6.1 presents the simplified procedure for the synthesis of different components.

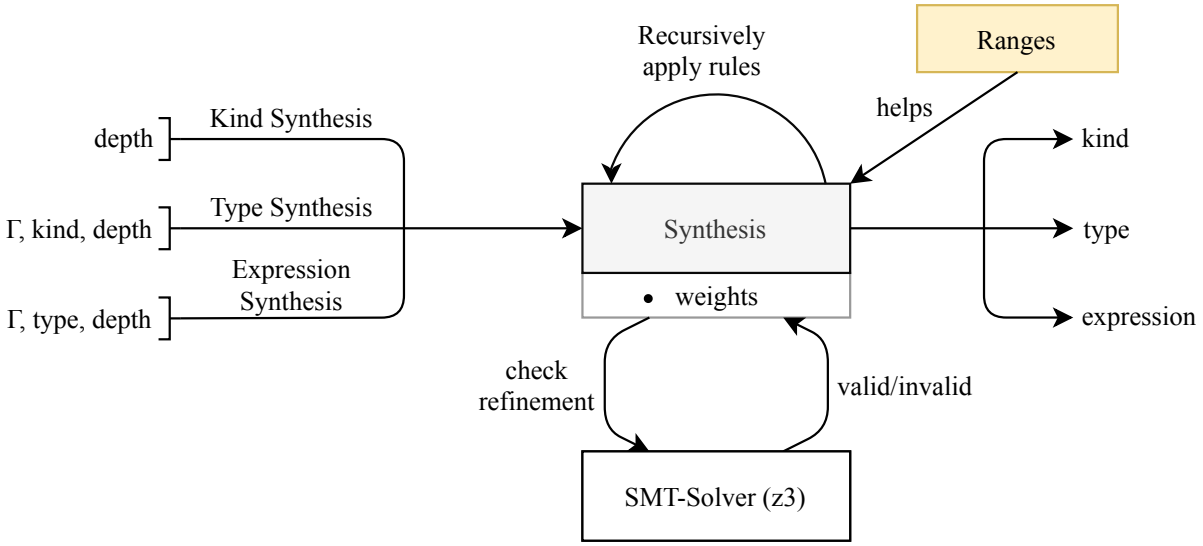


Figure 6.1: Synthesis diagram of kinds, types and expressions.

The non-deterministic synthesizer recursively applies the rules in order to generate the appropriate expression, type or kind. Each rule contains a weight which determines the likelihood of the rule to occur (the higher the value, the more probable it is to be used). Expressions are verified against liquid types with the help of an SMT solver (z3 [1] in our implementation). Since automatically generating values which comply with the refinement at the first try is unlikely with tight restrictions, we have developed a small module, described in Section 6.3, that allows us to generate discontinuous native values instantly.

Each of the following sections presents the concepts described in Figure 6.1. Section 6.1 describes the synthesis rules for the components of the language. Section 6.2 shows the influence of the weights over the synthesis rules and, finally, Section 6.3 addresses the synthesis problem of native expressions

[1] Tool available in `https://github.com/Z3Prover/z3`

over restricted refinements.

6.1 Synthesis rules

This section presents the synthesis rules for the program synthesis procedure.

$$\frac{}{\rightsquigarrow_d *} \qquad \frac{\rightsquigarrow_d k \qquad \rightsquigarrow_d k'}{\rightsquigarrow_{d+1} k \to k'} \qquad \text{(SK-Star, SK-Rec)}$$

Figure 6.2: Kind synthesis, $\boxed{\rightsquigarrow_d k}$.

Figure 6.2 presents the first set of synthesis rules for generating kinds. The first rule, SK-Star, given a non-negative depth d returns a kind. The second rule, SK-Rec, creates multiple type constructors via currying. Considering a depth greater than 0, $d+1$, the rule will generate two kinds, k and k', at maximum depth d, that together create an n-ary type constructor.

$$\frac{}{\Gamma \vdash *\rightsquigarrow_d \textbf{Integer}} \qquad \frac{}{\Gamma \vdash *\rightsquigarrow_d \textbf{Boolean}} \qquad \frac{t\colon k \in \Gamma}{\Gamma \vdash k \rightsquigarrow_d t} \qquad \text{(ST-Int, ST-Bool, ST-Var)}$$

$$\frac{\Gamma \vdash *\rightsquigarrow_d T \quad (x \text{ fresh}) \quad \Gamma, x\colon T \vdash *\rightsquigarrow_d U}{\Gamma \vdash *\rightsquigarrow_{d+1} (x\colon T \to U)} \qquad \text{(ST-Abs)}$$

$$\frac{\Gamma \vdash k\rightsquigarrow_d T \quad (x \text{ fresh}) \quad \Gamma, x\colon T \vdash \textbf{Boolean}\rightsquigarrow_d e}{\Gamma \vdash k\rightsquigarrow_{d+1} (x\colon T \textbf{ where } e)} \qquad \text{(ST-Where)}$$

$$\frac{\Gamma, t\colon k \vdash k'\rightsquigarrow_d T}{\Gamma \vdash (k \to k')\rightsquigarrow_{d+1} (\forall t\colon k.T)} \qquad \text{(ST-TAbs)}$$

$$\frac{\rightsquigarrow_d k' \quad \Gamma \vdash (k' \to k)\rightsquigarrow_d T \quad \Gamma \vdash k'\rightsquigarrow_d U}{\Gamma \vdash k\rightsquigarrow_{d+1} TU} \qquad \text{(ST-TApp)}$$

Figure 6.3: Type synthesis, $\boxed{\Gamma \vdash k\rightsquigarrow_d T}$.

Figure 6.3 describes the synthesis rules for types. Every rule requires to be given the typing context, the kind of the type to be synthesized and the synthesis depth.

The first two rules describe the synthesis of native types, such as the `Integer`, and `Boolean`. Although not represented, the `Double` and `String` types are also synthesized. Each native type is of kind $\star$ and can be synthesized at any depth.

The third rule, ST-Var, describes the synthesis of user-defined types. The only verification required is that the returned type exists in the context and has the appropriate kind.

The ST-Abs rule defines the creation of an abstraction type through the synthesis of a fresh variable and two types. The rule reads as follows: given the typing context, a kind k, and the non-negative synthesis depth, d, generate a type T. Then, create a fresh variable x. By providing the context the variable x with the type T, and depth d, generate a type U, where x can occur in U. Finally, build the AbstractionType for the generated variable x with type T and return type U.

The ST-Where is responsible for generating a refined type. Given the kind k, synthesize the type T. Then, create a fresh variable x. Given the typing context, Γ, with the variable x of type T, synthesize a `Boolean` expression e, where x can occur freely in e. Afterwards create the refined type with name x, type T and refinement expression e.

The ST-TAbs creates a new type abstraction from a new type. First, we generate a fresh t, with the kind k. This new type is provided to the context in order to synthesize the type T from the kind k', where t can occur in T.

The ST-TApp rule synthesizes the type application of two types. The rule starts by describing the synthesis of a new kind, k'. Then, we generate the type T, from the binary type constructor $k' \to k$. Lastly, synthesize the type U, which is the argument of the type application.

$$\frac{}{\Gamma \vdash \textbf{Boolean} \rightsquigarrow_d \textbf{true}, \textbf{false}} \qquad \frac{}{\Gamma \vdash \textbf{Integer} \rightsquigarrow_d n} \tag{SE-Bool, SE-Int}$$

$$\frac{\Gamma, x : T \vdash U \rightsquigarrow_d e}{\Gamma \vdash (x : T \to U) \rightsquigarrow_{d+1} (\lambda x : T.e)} \qquad \frac{\Gamma, e_1 \vdash T \rightsquigarrow_d e_2 \quad \Gamma \vDash e_1[e_2/x]}{\Gamma \vdash (x : T \textbf{ where } e_1) \rightsquigarrow_{d+1} e_2} \tag{SE-Abs, SE-Where}$$

$$\frac{\Gamma, t : k \vdash T \rightsquigarrow_d e}{\Gamma \vdash (\forall t : k.T) \rightsquigarrow_{d+1} (\Lambda t : k.e)} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash T \rightsquigarrow_{d+1} x} \tag{SE-TAbs, SE-Var}$$

$$\frac{\Gamma \vdash \textbf{Boolean} \rightsquigarrow_d e_1 \quad \Gamma, e_1 \vdash T \rightsquigarrow_d e_2 \quad \Gamma, \neg e_1 \vdash T \rightsquigarrow_d e_3}{\Gamma \vdash T \rightsquigarrow_{d+1} \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3} \tag{SE-If}$$

$$\frac{\begin{array}{c} \rightsquigarrow_d k \quad \Gamma \vdash k \rightsquigarrow_d U \quad \Gamma \vdash U \rightsquigarrow_d e_2 \quad (x \text{ fresh}) \\ \Gamma, x : U \vdash_{[e_2/x]} T \rightsquigarrow_d V \quad \Gamma \vdash (x : U \to V) \rightsquigarrow_d e_1 \end{array}}{\Gamma \vdash T \rightsquigarrow_{d+1} e_1 e_2} \tag{SE-App}$$

$$\frac{\begin{array}{c} \rightsquigarrow_d k \quad \Gamma \vdash k \rightsquigarrow_d U \quad (t \text{ fresh}) \\ \Gamma, t : k \vdash_{[U/t]} T \rightsquigarrow_d V \quad \Gamma \vdash (\forall t : k.V) \rightsquigarrow_d e \end{array}}{\Gamma \vdash T \rightsquigarrow_{d+1} e[U]} \tag{SE-TApp}$$

$$\frac{\Gamma \vdash T \overset{:>}{\rightsquigarrow}_d U \quad \Gamma \vdash U \rightsquigarrow_d e}{\Gamma \vdash T \rightsquigarrow_{d+1} e} \tag{SE-Sub}$$

Figure 6.4: Expression synthesis, $\boxed{\Gamma \vdash T \rightsquigarrow_d e}$.

The expression synthesis rules correspond to the ones that can generate valid expressions for the program synthesis. These rules use the previously defined kind and type synthesis rules to aid expression synthesis.

By providing the native types to the synthesizer, it is possible to generate the native values for the native types. The first two rules, SE-Bool and SE-Int, synthesize default values for the **Boolean** and **Integer** types by providing the context and maximum depth. Not only these types are natively generated but also the **String** and **Double** types are too. The synthesis of these values respects the restrictions propagated in the typing context. In order to automatically generate valid values from the restrictions, we use the framework described in Section 6.3.

The SE-Abs rule is responsible for generating an abstraction expression from an abstraction type. The synthesis of this expression is simple as we only need to obtain the expression, e, for the abstraction body. To do so, we synthesize the expression e, from the return type U, with the depth d, and providing the typing context with the variable x and its type T, allowing x to occur in the generated expression.

It is also possible to synthesize a restricted expression with a refined type from the SE-Where rule. We propagate the refinement restriction by providing it to the typing context, Γ, on the synthesis of the expression e_2 from the type T, at depth d. We then check condition entailment when replacing x by e_2 in e_1. If so, then the synthesized expression is considered valid and can be returned.

The synthesizer also generates variables, SE-Var, from a given type, T, as long as the variable is a subtype of T, and it belongs to the typing context.

The language also contains if expressions that can be produced from the SE-If rule. The if expression is composed of three body expressions. The condition, e_1, which is generated by providing the type **Boolean**. The then body expression, e_2, created by providing the context with the holding condition e_1 and the input type T, and, finally, the else body which generates an expression e_3 by propagating the not holding condition $\neg e_2$ on the context with the type T.

The SE-App rule is responsible for creating the application of two expressions from a given type T. The procedure goes as follows: first, we generate a new kind at depth d. This kind is used to synthesize the type U, needed for the synthesis of the argument expression e_2. Then, we need to generate the target side of the final application expression. The target expression is required to be an abstraction, and so, we create a fresh variable x and include in the typing context with the type U to synthesize the type V from the inverse expression on expression substitution of e_2 by x in the type T. To generate the expression e_1 we create the abstraction type, that given the variable x of type U returns the type V. We can posteriorly build the $e_1 e_2$ application from the synthesized expressions.

The SE-TApp rule creates the application of a type from a generated type abstraction. First, we generate a new kind at synthesis depth d. With the new type constructor and the context, we generate the argument of the type application, U, in which, U belongs to the typing context. Then, by providing to the context a fresh generated type t with the kind k, we generate the type V with inverse substitution of type in type of t by U in the type T. Finally we generate the expression that is type applied to U from the type abstraction t with kind k.

The SE-Sub rule allows this synthesis procedure to generate any expression that is a subtype of the provided type.

6.2 Weights over synthesis rules

The refinements applied over the types help to reduce the search space for the valid program the user is seeking. However, the combination of components may still be too broad and, the introduction of weights will guide the synthesis towards a faster and improved solution.

Each rule is followed by a weight, which tells the synthesizer the probability of it being chosen. This kind of optimization allows the tool to generate programs that may follow user-based programming patterns.

For instance, the probability of generating an if expression for each nested if is smaller as we get

deeper in the nesting, in other words, the regular programmer typically does not write many nested if expressions (five ore more, for instance). The weights help to control these type of situations, by reducing the weight every time the SE-If rule is chosen.

Another example is when a specific variable is chosen within the condition of the SE-If synthesis rule. The synthesizer may deduce that if a specific variable x is used in the condition of the if-expression, then the likelihood of it being used within the bodies of the if-expression is higher.

Table 6.1 presents an example of initial weights provided for the expression synthesis rules.

Table 6.1: Weights on the expression synthesis rules.

Rule	Weight	*(continued)*	
SE-Int	20	SE-If	15
SE-Bool	20	SE-App	30
SE-Var	40	SE-TAbs	5
SE-Where	10	SE-TApp	5
SE-Abs	10	SE-Sub	10

From the provided rules and weights, we can reason for some information. Firstly, the rule with the most weight is the SE-Var, thus being the one with the most probability of being chosen. Then, application rule has the second-highest weight, followed by the native value synthesis rules, SE-Int and SE-Bool. In this particular case, it is possible to tell that the non-deterministic synthesizer tries to generate with high probability function invocations. This weights can be customized for the particular synthesis problem the programmer may have.

For better understanding, let us create an example: consider the type `Integer` and a subset of chosen rules that can generate an expression from this type: SE-Int, SE-Var, SE-App. The probability for each rule to be chosen is,

$$\frac{w_i}{\sum_{j=0}^{n-1} w_j} \tag{6.1}$$

Since SE-Int has weight 20, the likelihood of being chosen is 22.22% from the calculation of $20/(20+40+30)$. Similarly, the SE-Var since has weight 40, its probability is 44.44%, and finally the probability of chosing SE-App is 33.33%.

6.3 Ranges over refinements

The synthesis rules allow the generation of native values from their proper types. When synthesizing a value from a given refined type, this must automatically comply with the predicate of the liquid refinement. The unoptimized version of the synthesizer would generate any value, independent from the refinement, according to a distribution, and have it verified by the SMT-Solver. This process, however, introduces a challenge with the `Integer` and `Double` types: with tight restrictions, there is a high probability for the synthesizer to fail when generating native values.

The solution for this challenge was the introduction of the synthesis of some discontinuous ranges over the tight refinements with the help of a mathematical framework in Python called SymPy [2]. The

[2]Tool available in `https://www.sympy.org/`

objective is to have a different interpretation from refinements to ranges and use SymPy to solve the rational inequalities.

For instance, consider the following refinement, `x >= 0 && x <= 10`, which reads, generate any integer value between 0 and 10. By providing this restriction to the SymPy Solver, it will output the lower bounds and upper bounds (or ranges) of the refinement, 0 and 10 in this case. Using Pythons random generator and these bounds, we can non-determinstically generate valid integer values.

The previous example does not evidence the true potential of the framework on the new interpretation of the refinements. When new operations appear and discontinuous intervals arise, then we can verify the tool usefulness. For example, the refinement `x - 1 >= -1 && x <= 10` is equivalent to the previous one, but this time we have an extra arithmetic operation in between, and also if the condition is further refined by stating that `x != 5`, then we can conclude that, as we provide more and more information, the bounds calculation is not trivial.

For this reason, the non-deterministic synthesizer uses the SymPy to do a best effort on calculating values that comply with the tight restrictions. The steps below show the refinement translation from ÆonCore to SymPy and some instrumentation for generating the native values.

1. **Translation to SymPy** The first step is to translate the liquid refinements from ÆonCore to SymPy. The ÆonCore representation allows a pretty straightforward translation since all that is required is to create the bindings from the specific native operations (e.g. sum, minus, div, and, or, and so on), and the SymPy representation of these expressions.

2. **Bounded intervals** Then, the translated refinements are converted into intervals, resulting in a list of intervals or lists. Each interval contains a lower bound and an upper bound. A nested list is created when an Or operation is created from the intervals conversion (e.g. when there are discontinuations), and so, each list has its lower and upper bounds.

   Each interval is translated to proper values to be used within Pythons randoms generator. Infinity and negative infinity are respectively translated to maximum integer and a minimum integer. The lower bounds and upper bounds are converted from SymPy values to native values. Moreover, if the left bound is open or the right bound is open, the bounds are adjusted with an offset.

3. **Generating the values** By the end of this procedure, the synthesis has obtained a list of minimum and maximum ranges, in which, one is non-determinstically chosen. Since typically the values closer to 0 are more likely to be used in regular programming standards, the synthesizer will choose and generate a value between the minimum and maximum that follows a normal distribution of $\mu = 0$ and $\sigma = 75$. In the cases where it has to choose between intervals and a specific value obtained from the refinement (e.g. `x == 5`), it will give all the conditions in the disjunction the same probability, and if an interval is chosen, then the final value is selected according to the normal distribution.

For better understanding, consider Figure 6.5 which presents an example of this system where the user wants to generate a value between 0 and 30, and it is not 15, or any value greater than 100.

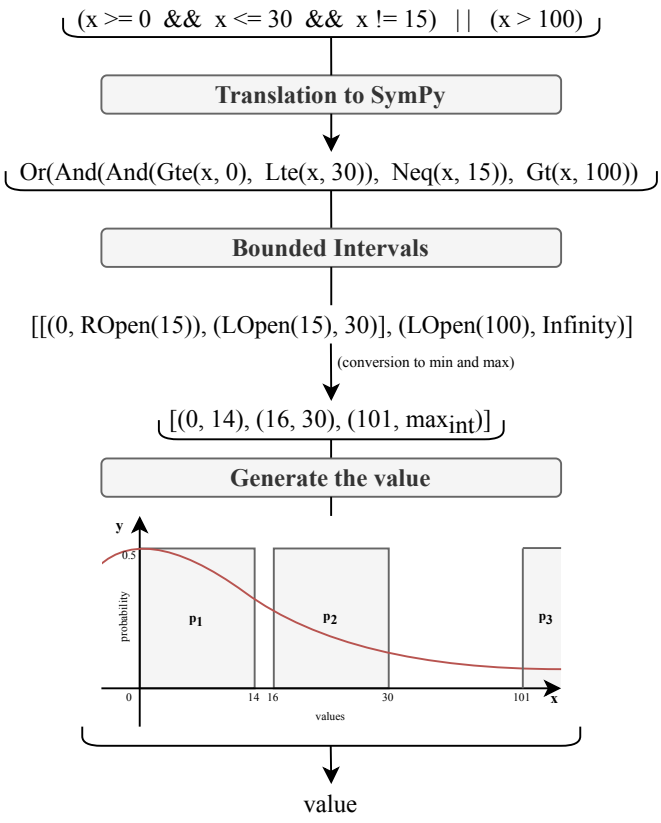


Figure 6.5: Example of bounds from a liquid refinement predicate.

The first step is the translation from ÆonCore to the SymPy notation. The translated refinement is then delegated to SymPy inequalities solver, from where we obtain the bounded intervals for each disjunctions' conditions. In the first condition, the refinement between 0 and 30 was split into two intervals, in order to take into consideration the $x \neq 15$ restriction. Each interval is then converted into pairs of minimum and maximum, the offsets are set according to the open intervals, and the infinity values are converted into Pythons *maximum* integer size.

The following step is to calculate the probability of choosing one of the intervals. Instead of randomly choosing one of them, or non-deterministically choosing them according to the amount of elements the interval has, we have decided to calculate the area of the interval within the normal curve with the help of the z-score table and obtain the probability of that certain interval to be chosen. The main motivation behind this choice is the assumption that programmers typically use values within small ranges [30]. For instance, the value 0 should have a higher probability than a random value such as 149 since it is typically more used.

Example 6.3.1 performs the calculations and the final likelihood for choosing the intervals.

Example 6.3.1. Intervals probability calculation

Consider the following equations essential for the calculation of the probability of each interval:

$$z_{min} = \frac{min - \mu}{\sigma} \qquad z_{max} = \frac{max - \mu}{\sigma} \qquad p_i = |zscore_{min} - zscore_{max}| \tag{6.2}$$

Assuming $\mu = 0$ and $\sigma = 75$, and taking into consideration the previous formulas, we can calculate the z minimum and maximum values for each interval:

$$v_0 = 0.0 \qquad v_{14} = \frac{14}{75} = 0.19 \qquad v_{16} = \frac{16}{75} \approx 0.21 \qquad v_{30} = \frac{30}{75} = 0.40 \qquad v_{101} = \frac{101}{75} \approx 1.35$$

Each value is then provided to the z-score table library and we obtain the adequate z value:

$$z_0 = 0.5 \qquad z_{14} = 0.5753 \qquad z_{16} = 0.5832 \qquad z_{30} = 0.6554 \qquad z_{101} = 0.9115$$

The probability for each area $p_1, ..., p_n$, is then calculated using the p_i formula. If the restriction does not have an upperbound or lowerbound, or if it is too high, then it is replaced by 1. The probability of each area can be found below:

$$p_1 = |0.5 - 0.5753| = 0.0753 \qquad p_2 = |0.5832 - 0.6554| = 0.0722 \qquad p_3 = |1 - 0.9115| = 0.0885$$

Each probability is then multiplied by 10000 and used as a weight to calculate the likelihood of choosing a certain interval, where, in this particular case: $w_1 = 753$, $w_2 = 722$ and $w_3 = 885$. The final probability of selecting each interval is: $p_1 = 31.9\%$, $p_2 = 30.6\%$, and $p_3 = 37.5\%$.

To conclude, although the set of values of the first disjunctions restriction is smaller than the second one, the likelihood of generating a value inbetween that range is actually greater than generating any value greater than 100.

The usage of this framework still has some limitations. Firstly, dependent types have not been taken into consideration for this particular case, as they present a higher challenge for the synthesizer.

Secondly, SymPy cannot create intervals for all the discontinuous functions, for example, `x % 2 == 0`, where the condition tries to obtain even values. As we progress towards second-degree inequalities, the tool also has its limitations. So, when it is not possible to synthesize an expression from these specific liquid types, the type is delegated to the SMT-Solver, and it will output a possible solution. Using z3 to generate the values is the last resort since it sacrifices the diversity of the solutions for the computation of the solution itself.

Chapter 7

Evolutionary Program Synthesis

This chapter presents the evolutionary procedure from non-liquid refined types using genetic programming. ÆonCore generates random valid expressions from the liquid refinement. Valid expressions correctness is checked using the non-liquid refined types in the genetic programming process.

7.1 Concept

Automatically generating programs from a given specification is called program synthesis. In Æon, introducing the hole, `??`, operator and the function specification initiates the program synthesis procedure. During type checking, it gathers all the holes and their local contexts and provides this information to the genetic algorithm approach in order to synthesize, with the help of the liquid and non-liquid refined types, a valid and correct solution.

Let us consider the synthesis example 7.1 that tries to generate a simple Caesar cipher.

Listing 7.1: Synthesis of the cipher function in Æon.

```
type Key {
    {key : Integer | key >= 0 && key <= 1024};
}

decipher(i:Integer, k:Key) -> {j:Integer | j > 0} {
    i - getKey(k);
}

cipher(i:Integer, k:Key) -> {j:Integer | j > 0 and
        i == decipher(j, getKey(k))} {
    ??;
}
```

The `??` denotes the hole in Æon. The type of the hole to be synthesized is automatically deduced using the algorithm described in 5.3. Detecting and gathering the holes and their local contexts trigger the program synthesis.

The cipher definition contains refinements over the input and output types of the function. Each predicate is separated by an **and**. These predicates are essential for program synthesis to work correctly.

As stated in previous chapters, even though ÆON does not distinguish them, there are two classes of refinements over the types:

Liquid Refined Types: used to express the problem and restrict the search space of correct programs in order to find a valid solution faster. The cipher definition contains one refined type, `j > 0`, which indicates that the output value must be greater than 0. The evolutionary procedure will use the non-deterministic synthesizer, described in Chapter 6, to generate random expressions for each hole.

Non-liquid Refined Types: are essential to evaluate the quality of generated valid individuals. The cipher function presents one non-liquid refined type, `i == decipher(j, getKey(k))`, which states that doing the reverse process of the cipher function, we should obtain the initial value. Each predicate is used to generate a fitness function that evaluates individuals. Each non-liquid refinement is an objective that should be accomplished in order to synthesize a correct program.

Although the examples so far have presented the synthesis of full programs only, the whole ÆON infrastructure is also prepared to compute partial programs. In order to help the synthesis procedure, the previous example could also be presented as in Listing 7.2. In this case, the evolutionary synthesizer would have to discover what expressions should be generated in order to respect the return type specification.

Listing 7.2: Partial synthesis of the cipher function in ÆON.

```
1 cipher(i:Integer, k:Key) -> {j:Integer | j > 0 and
2         i == decipher(j, getKey(k))} {
3     ?? + ??;
4 }
```

7.2 The Evolutionary Synthesis System

Representation

The first step is to choose the representation of the individuals for the genetic programming procedure. Three attributes compose each individual: **Types & Contexts**, a list which contains the types and contexts of the holes that are being synthesized, **Expressions**: each expression represents a synthesized expression from the non-deterministic synthesizer, an AST, for each hole in the program, and **Fitness**: a list of fitness values for each objective obtained from the non-liquid refined types.

Initialization Procedure

The evolutionary process starts with the initialization of a population of program candidates. By providing the context and type of each hole, and the maximum depth to the non-deterministic synthesizer, it generates random expressions to be filled in the holes. The initialization procedure uses an attempt to ramped-half-half initialization algorithm. The attempt raises from using the non-deterministic synthesizer, and it is not possible to guarantee that the generated expression complies with the maximum depth (not possible to enforce it as no expression may exist at required depth).

Selection

Selection takes a vital role in the genetic programming algorithm. It is used to select individuals from genetic operations (recombination and mutation) and for elitism. Although different kind of approaches can be used for selection in multi-objective optimization problems, recent work extols the dominance of lexicase-based variants [25, 28]. In this work, we use a combination of two techniques: ϵ-lexicase selection and fitness proportionate selection.

Helmuth, Spector, and Matheson introduced lexicase selection for parent selection in multi-objective program synthesis problems. The main objective is to improve population diversity and improve convergence by selecting individuals that are the best fit for randomly chosen objectives at each generation. Listing 7.3 presents the algorithm described by Helmuth.

Listing 7.3: Lexicase selection algorithm pseudocode.

```
1. Start with the initial population and test cases set.

2. While there remain test cases or population size greater than 1:
    2.1. Randomly choose a test case.
    2.2. Obtain the best score from the individuals for the test case.
    2.3. Filter the individuals without the best score.
    2.4. Remove the test case from the test case set.

3. Randomly choose an individual from the remaining population.
```

ϵ-lexicase selection [9] is a variant of the regular lexicase selection algorithm and the one used in the multi-objective optimization on the evolutionary synthesis. This selection algorithm is used to choose the individuals to recombine in the crossover. The difference with the regular lexicase selections is that this instead of filtering the individuals without the best score, it filters the ones that do not fall within the range of, best score + error (ϵ), dynamically calculated using the median absolute deviation [9] (Equation (7.1)), reading, obtain the median from the list of absolute errors between the fitness value for a test case and the median errors of that test case. This strategy was chosen over the regular lexicase selection since it allows a more diverse selection of the individuals for the population.

$$\epsilon = median(|e_{t_j} - median(e_{t_k})|) \tag{7.1}$$

If only one objective is provided to the evolutionary synthesis, and since the ϵ-lexicase selection always chooses the same individual if it is better than the remaining individuals, leading to a less diverse population, the selection strategy used is the fitness proportionated. The population is sorted according to their fitness, and an individual is randomly chosen according to a gaussian curve.

Fitness Evaluation

The evaluation plays an essential role in the evolutionary procedure. Each individual must be evaluated according to the non-liquid refinements provided by the programmer. The evaluation for each individual occurs as follows:

1. Fill each hole of the function with the synthesized expressions;

2. Randomly generate a set of inputs;

3. Obtain the output value from evaluating each filled function with the test set;

4. Evaluate the output against the non-liquid refined types.

In order to evaluate the output against the specification, the first naive approach to consider is a step conversor. If when checking the output value, the condition holds, then it returns 0, otherwise returns 1. The objective is to minimize the error and ensure that every condition in the refinement is fulfilled.

This naive approach, however, is not capable of distinguishing between bad from not-so-bad solutions. For instance, a given individual A may be closer to the final solution than individual B, even though both of them do not comply with the specification. In this matter, each logical condition is converted into a continuous fitness function [38], according to the rules in Table 7.1.

Boolean	Continuous
true, **false**	0.0, 1.0
$x = y$	norm($\|x - y\|$)
$x \neq y$	$1 - f(x == y)$
$a \wedge b$	$(f(a) + f(b))/2$
$a \vee b$	min$(f(a), f(b))$
$a \rightarrow b$	$f(\neg a \vee b)$
$\neg a$	$1 - f(a)$
$x \leq y$	norm$((x - y))$
$x < y$	norm$((x - y + \delta))$

Table 7.1: Conversion function f between boolean expressions and continuous values.

The rules stated in Table 7.1 are used to convert each non-liquid refinement into a a continuous fitness function, an objective that candidate programs try to comply with. Each clause is first reduced to the conjunctive normal form (CNF) and then converted from the predicate into the continuous function. For example, the previous example in Listing 7.1 is converted into the function:

$$norm(|i - decipher(j, getKey(k))|) \tag{7.2}$$

Since the output of f is an error, the value **true** is converted to 0.0, stating that condition holds, otherwise 1.0, this being the maximum value of not complying with the condition. Variables and function calls are also converted to 0.0 and 1.0 on whether the condition holds or not. Equalities of numeric values are converted into the normalized absolute difference between the arguments. The normalization is required as it allows different clauses to have the same importance on the given specification. Inequalities are converted to equalities and its difference with 1, negating the fitness result from equality; conjunctions are converted to the average of the sum of the fitness extraction of both operands; and, finally, disjunctions value is obtained by extracting the minimum fitness value of both clauses. The minimum value indicates what clause is the closest to no error. Conditional statements fitness is recursively extracted by using the material implication rule. Similarly to inequalities, the negation of conditions denies the value returned by the truth of the condition. Numeric value comparisons represented a laborious challenge as there are intervals where the condition holds. We use the difference of values to represent the error. In the $<$ and $>$

rules, the δ constant depends on the type of the numerical value, 1.0 for integers and 0.00001 for doubles, and is essential for the extra step required for the condition to hold its truth value. A rectifier linear unit was used to ensure that if the condition holds, it is set to the maximum between the negative number and 0, otherwise, if the value is greater than 0, the positive fitness value is normalized.

The fitness function is the result of applying each f_i for each non-liquid refinement to a set of randomly generated (using the liquid synthesis algorithm in Section 2.3) input values. The individuals fitness is composed by a set of values, each obtained with the sum of running the f_i fitness function in all the random input values.

Recombination

Recombination is used to combine individuals and produce offspring with characteristics of both parents. The recombination is similar to the Strongly Typed Genetic Programming (STGP) approach but differs on the details. Figure 7.1 presents the recombination process between two selected individuals. While in STGP the recombination exchanges nodes of the AST that have the exact type, in RTGP, the source parent node can receive any node that is a subtype of the previous source son node. This approach increases the likelihood of successful crossovers. On the other hand, if on the second parent, no type complies with the replacement node, the recombination algorithm works similar to the mutation. It uses the non-deterministic synthesizer with all the genetic information gathered in the second parent and generates a new replacement that complies with the type and maximum depth required to the AST.

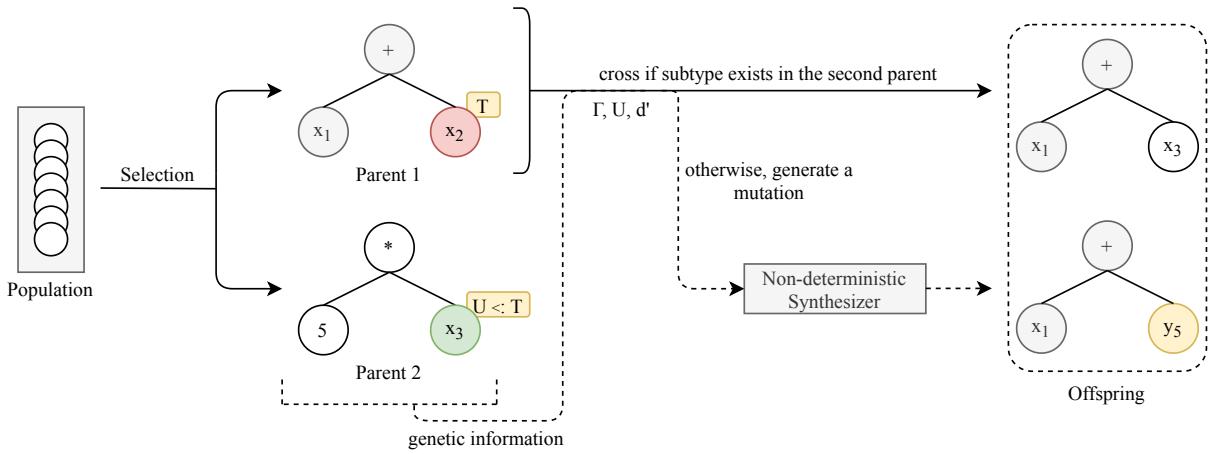


Figure 7.1: Recombination diagram in the evolutionary computation.

In RTGP, it is possible to define multiple holes for partial synthesis. The recombination not only non-deterministically selects individuals but also randomly chooses the synthesized expressions for the holes, allowing a N-target-multi-objective synthesis, where distinct synthesized holes exchange genetic material between each other.

Mutation

The mutation works similarly to the recombination. Each individual in the population has a probability for a mutation in one of the synthesized holes. Figure 7.2 presents the mutation flow of a selected in-

dividual. A random node from the AST is selected for the mutation and, by providing its local context, its type, the calculated synthesis depth (maximum allowed depth minus node depth), and the node itself as genetic material, the non-deterministic synthesizer generates a replacement which complies with the type and maximum depth of the program.

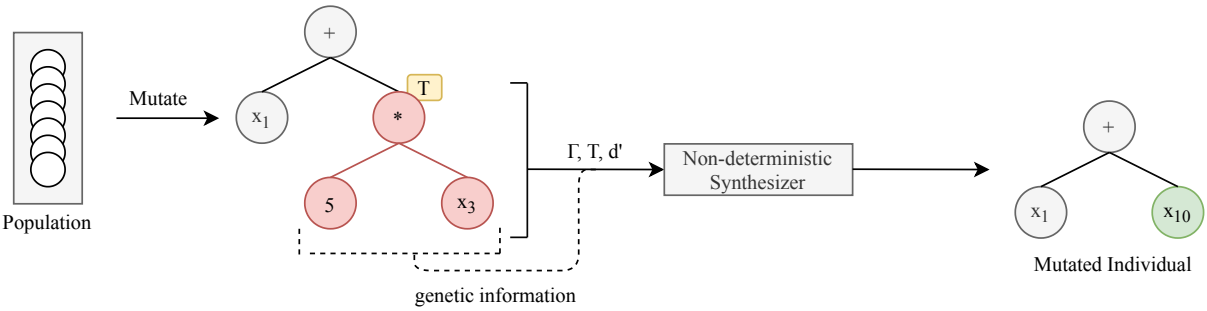


Figure 7.2: Mutation diagram in the evolutionary computation.

7.3 Code optimizer

By the end of the evolutionary process, a final solution is found with expressions for each hole. Each expression, however, may not be the most optimized one, as it can contain dead branches, unused variables, simples calculations, so on. Therefore, in order to improve readability, each synthesized expression runs over a code optimizer algorithm, which improves the overall code quality.

Rule 1. Algebraic and Boolean expressions simplification The first code optimization is related to expressions computation. Basic arithmetic expressions, such as sums, divisions, and multiplication, are computed and replaced by the final value.

The simplification only occurs if the arguments of the application do not contain any variable nor function invocation. There are two exceptions though: when applying the Zero Product/Sum law and the Multiplicative Identity Property.

Similarly, the boolean expressions are optimized based on the computation of the values, if possible, and variables are kept unchanged. In this situation, one exception applies with short circuits.

Listing 7.4 presents the different algebraic and boolean optimizations made according to the current rule and its exceptions. For each expression e, the respective optimization, `~>`, is introduced.

Listing 7.4: Algebraic and Boolean expression optimizations of the synthesized code.

```
1 -- Algebraic optimizations
2 1 + 1 ~> 2
3 1 + x ~> 1 + x
4 0 * f(x) ~> 0
5 0 + f(x) ~> f(x)
6 1 * f(x) ~> f(x)
7
8 -- Boolean optimizations: short circuits example
9 true && x ~> x
```

```
false && x ~> false
true || x ~> true
```

Rule 2. Variables value propagation The optimizer also tries to improve the usage of variables. If the synthesizer decided to create a variable, x, with a constant value, and x is not redefined in the rest of the program but used, the code optimizer replaces all instances of x with the assigned constant value.

Rule 3. Abstraction types removal Variable declarations are defined in ÆonCore with abstractions, where a variable x, of type T, can occur in the abstraction body. If the synthesizer generates a particular abstraction which is not used in its body, then the abstraction is replaced by its body.

Rule 4. Type abstractions reduction Type abstractions allow the introduction of polymorphism in the language. The synthesis procedure may have decided to generate superfluous type abstractions, which are not used in its body, or, a type application is immediately before applied to its abstraction, making the type abstraction redundant. So, two mechanisms are deployed to solve this issue: unused type abstractions are removed, and immediate type applications on type abstraction are propagated through the program.

Listing 7.5 contains two examples of removal and propagation over the type abstraction. In the first example, the type T is removed as it does not occur in its body. In the second example, the `Integer` is propagated in the body of the type abstraction, replacing the type T.

Listing 7.5: Type abstractions removal and reduction of the synthesized code.

```
-- Type abstraction removal
T:* => f[Integer](x) ~> f[Integer](x)

-- Type abstraction propagation
(T:* => f[T](x) Integer) ~> f[Integer](x)
```

Rule 5. Superfluous branches optimization The final optimization takes into consideration the if expressions. The objective is to optimize dead branches and if expressions that can be considered redundant. Firstly the condition of the if expression is evaluated, if it can be evaluated to a constant value of *true* or *false*, then the if statement is replaced with the **then** or **else** body, respectively. A second optimization is done if the condition cannot be optimized: if both bodies of the if statement are equivalent, the optimization of both body statements is equal or the computation value is the same, then the most optimized body expression replaces the if statement. Listing 7.6 presents two different examples of the if statements optimization.

Listing 7.6: If expressions optimization of the synthesized code.

```
-- Dead branch removal
if true then x else y   ~>  x

-- If expression simplification
if f(x) then 0 * x else 0   ~>  0
```

In the first example, since the condition always evaluates to true, the if expression is replaced with the **then** body. On the second example, since both the **then** and **else** bodies are equivalent, the expression is replaced with the most minimal one, 0.

By the end of the evolutionary procedure, RTGP has created an individual with the optimized synthesized expressions which should comply as close as possible with the user specification. Each hole in the original function is then replaced with the respective generated expression and the full program with no holes is provided to the interpreter to obtain the final result.

Chapter 8

Evaluation

This chapter presents the evaluation of the synthesis framework with two main challenges in GP and the usability argument of RTGP over GGGP. We also prove the versatility of the concepts described in this work, with the creation of a property-based prototype testing tool.

8.1 RTGP vs GGGP: An Usability Perspective

This section presents the usability perspective of RTGP over GGGP. Since RTGP and GGGP similarly restrict the search space, we argue that RTGP has improved usability over GGGP. Fonseca et al. have already presented the usability arguments of RTGP [20]. In this section, we revisit and strengthen those arguments with direct comparisons between RTGP and GGGP.

Listing 8.1 presents the code used for the synthesis of the Mona Lisa painting. In Æon, the programmer is required to do three steps for the Mona Lisa generation: first, import the components required for the synthesis (which we luckily defined in Chapter 4); then, load the image from the file; and, finally, create the function that will generate the improved Mona Lisa. The user leaves a hole, ??, for the program being generated.

Listing 8.1: Mona Lisa in Æon.

```
1 import aeon/libraries/image;
2
3 monalisa : {img:Image | img.width == 732 && img.height == 1024} =
  ↪  load_image("examples/aeon/mona.jpg");
4
5 generate_mona() -> {img : Image | img.width == mona.width and
6                                   img.height == mona.height and
7                                   @minimize(image_diff(monalisa, img))} {
8     ??;
9 }
```

The goal is written as a specification on the `generate_mona` function, and is composed by two liquid refinements and one non-liquid refinement: the first two liquid conditions tell the synthesizer to

maintain the output image size, the last non-liquid condition is used as a fitness function in the Genetic Programming. The final synthesized code will then be replaced and executed in the same infrastructure where the problem was described.

One of the main objectives of ÆON is to have a robust and expressive standard library. As seen in this particular case, a robust library allows the users to quickly describe their problems without the need of implementing the synthesis components.

However, in GGGP, the user needs to provide the grammar, the implementation of the problem in a foreign language and develop the evolutionary procedure to synthesize the programs. Not only that but creating the bindings between the context and the implementation may present a challenge. Listing 8.2 describes the grammar in GGGP of the mona lisa challenge.

Listing 8.2: Mona Lisa in GGGP.

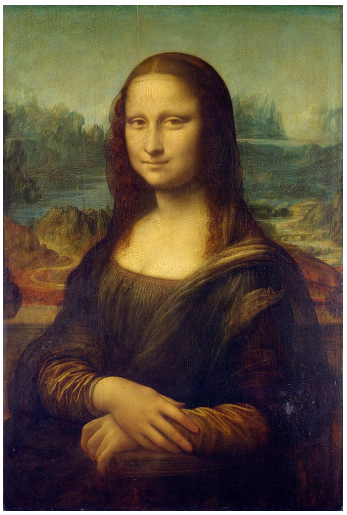
```
<expr> ::= <draw>
         | <expr><expr>

<draw> ::= triangle(<coord>, <coord>, <coord>, <color>)
         | rectangle(<coord>, <coord>, <coord>, <coord>, <color>)

<coord> ::= coordinate(<x>, <y>)
<color> ::= create_color(<r>, <g>, <b>)

<x> ::= [0..732]    <y> ::= [0..1024]
<r> ::= [0..255]    <g> ::= [0..255]    <b> ::= [0..255]
```

By the end of the synthesis, both approaches should have conceptually generated a new and improved Mona Lisa:



(a) Mona Lisa.

(b) Conceptual synthesized Mona Lisa.

Figure 8.1: Original and conceptual synthesis of Mona Lisa.

The second example, presented in Listing 8.3, is the Santa Fe Trail problem. The Santa Fe Trail is a

typical genetic programming problem in which food pellets are displayed in a path, and artificial ants try to consume all the available pellets. For better understanding, consider the Figure 8.2 [53], in which the black squares represent the food pellets and the grey spaces the gaps in the path.

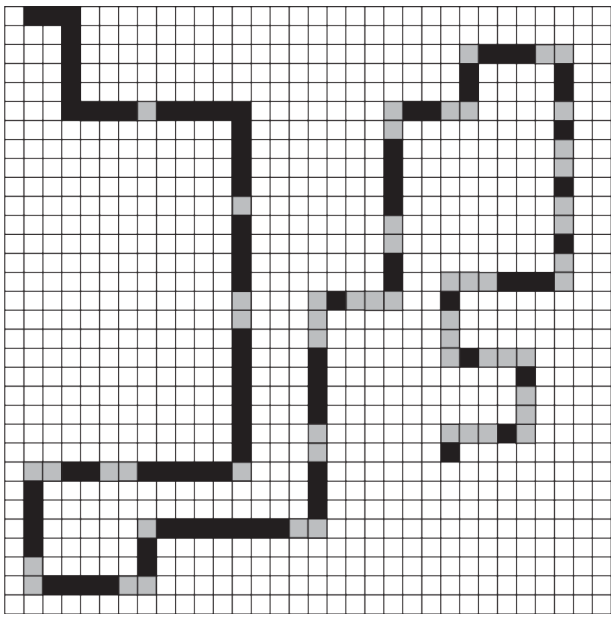


Figure 8.2: Santa Fe Trail path with gaps.

Differently from the previous example, where a robust standard library with all the components already was implemented, in this case, the user is required to implement all the required synthesis components. For the sake of the reader, some implementation details were natively implemented and imported. In terms of readability, even for the writer of this example, it may present a challenge. These limitations will be described in Section 8.3, and some solutions presented.

In this example, we plan to generate a program capable of following the food pellets incomplete path and minimize the number of pellets in the grid. Some main synthesis components are required: `food_ahead` a function to check whether a food pellet is ahead or not (if there is, the ant will hopefully move forward to consume it), `turn_left` and `turn_right`, to allow the ant reach every cell of the grid, and finally, `move`, which will move the ant and consume the food pellet if present.

Listing 8.3: Santa Fe Trail in ÆON.

```
import aeon/libraries/list;
import aeon/libraries/pair;
import aeon/libraries/santafe;

size : {x:Integer | x > 0} = 10;

type BoolInt as {x : Integer | x == 0 || x == 1};
type SmallInt as {x : Integer | x >= -1 && x <= 1};
type BoundedInt as {x : Integer | x >= 0 && x <= size};

type Grid {
    grid : List[List[BoolInt]];
    food : {food:Integer | food >= 0};
```

```
    pos : Pair[BoundedInt, BoundedInt];
    dir : Pair[SmallInt, SmallInt];
}

food_ahead({grid:Grid | 0 <= grid.pos.e1 + grid.dir.e1 < grid.size &&
                        0 <= grid.pos.e2 + grid.dir.e1 < grid.size}) ->
                         ↪  Boolean {

    g : List[List[BoolInt]] = get_grid(grid);

    x : BoundedInt = pair_first[BoundedInt, BoundedInt](position(grid));
    y : BoundedInt = pair_second[BoundedInt, BoundedInt](position(grid));

    d_x : SmallInt = pair_first[SmallInt, SmallInt](direction(grid));
    d_y : SmallInt = pair_second[SmallInt, SmallInt](direction(grid));

    get_elem[List[List[BoolInt]]](x + d_x, y + d_y, g) == 1;
}

turn_left(grid:Grid) -> {g2:Grid | g2.dir.e1 == -grid.dir.e2 &&
                                   g2.dir.e2 == grid.dir.e1} {

    d_x:SmallInt = pair_first[SmallInt, SmallInt](get_direction(grid));
    d_y:SmallInt = pair_second[SmallInt, SmallInt](get_direction(grid));

    d_x2:SmallInt = d_x;
    d_x = -d_y;
    d_y = d_x2;

    grid = set_direction(create_pair[SmallInt, SmallInt](d_x, d_y), grid);
}

move({grid:Grid | 0 <= grid.pos.e1 + grid.dir.e1 < grid.size && 0 <=
↪   grid.pos.e2 + grid.dir.e1 < grid.size}) -> {g2:Grid | g2.pos.e1 ==
↪   grid.pos.e1 + grid.dir.e1 && g2.pos.e2 == grid.pos.e2 + grid.dir.e2} {

    x : BoundedInt = pair_first[BoundedInt, BoundedInt](position(grid));
    y : BoundedInt = pair_second[BoundedInt, BoundedInt](position(grid));

    d_x:SmallInt = pair_first[SmallInt, SmallInt](get_direction(grid));
    d_y:SmallInt = pair_second[SmallInt, SmallInt](get_direction(grid));

    grid = set_position(create_pair[BoundedInt, BoundedInt](x + d_x, y +
     ↪  d_y), grid);

    if has_pos_food(grid) then eat_food(grid) else grid;
}

santafe(trail:Grid) -> {out:Grid | @minimize(food_present(trail))} { ??; }
```

In RTGP, we can use liquid refinements to restrict the search space of available programs. One in-

stance of an edge case arises in the `food_ahead` argument restriction, were we ensure that the `food_ahead` function is never called with grid out of bounds values, thus excluding all these states from being synthesized. However, in GGGP, with the approach presented by Urbano and Georgiou, this kind of restriction is not possible, and so, it would still generate these states but would require some code instrumentation in order not to allow out of bounds exceptions.

The same happens in the `move` function, which restricts the input type values. This time, RTGP makes use of dependent refined types to track information on the position of the ant and ensure the next synthesized grids keep the ant within bounds. In this sense, GGGP tries to do the same thing by calling `ifalse` function over the food_ahead, ensuring that the ant is kept in the grid, but still synthesizing the ill-state if no instrumentation is done.

Listing 8.4: Santa Fe Trail grammar [55] in GGGP.

```
<code> ::= <line> | <code> <line>
<line> ::= <condition> | <op>
<condition> ::= ifalse food_ahead
                [ <line> ] [ <line> ]
<op> ::= turn_left | turn_right | move
```

The following listings condensates not only all the previous arguments, but also described in Fonseca et al. [20], and argues the usability of RTGP against GGGP.

Advantages of RTGP

1. Ability to declaratively restrict the search space A type system is used instead of a grammar to express the restriction.

2. Problem Structure Problem domains that already follow a grammar structure can be easily encoded in RTGP. RTGP can more directly encode several problems than a grammar. For instance, the Mona Lisa challenge (in Section 8.1).

3. Flexible Extension Extensions to GP can be encoded both in grammars and dependent types. Both approaches can be used as engines to test other GP concepts.

Disadvantages of GGGP

1. Feasibility Constraints Both GGGP and RTGP make the design of new operators a more significant challenge than in STGP, since operators should follow the system constraints. All RTGP operators are shared among any problem and rely solely on the type checker and expression synthesis.

2. Limited Flexibility GGGP is flexible when the program can be directly encoded in a context-free grammar. Some GGGP approaches use context-sensitive grammars [40], but specifying the constraints of the grammar increases its complexity, making it less desirable by practicioners and revising.

3. Language Robustness Æon provides all the characteristics of a regular programming language (e.g. recursion), and already a list of synthesis components which can be imported. On the other hand, in GGGP all components must be explicitly defined by the user.

8.2 Application of RTGP for Propert-Based Testing in Python

This section presents pyCheck, a property-based testing prototype tool that automatically tests annotated Python code. This tool proves the concept versatility by reusing the refinements types, non-deterministic synthesis and fitness extraction concept to evaluate Python code correctness.

Property-based testing [19] (PBT) uses properties provided by the user to evaluate the system correctness. Hypothesis [1] is a modern framework based on property-based testing to evaluate mainstream languages, like Python and in the future, Java. This tool has been widely used in open-source projects, like PyPy and Pyrsistent, to check the correctness of the projects. Listing 8.5 presents an example where we try to make a special sum of two values.

Listing 8.5: Function verification of the special_sum with Hypothesis.

```
from hypothesis import given
from hypothesis.strategies import lists, integers

def special_sum(x, y):
    # Bug here, this entire if should be removed
    if x > 7 or y > 7:
        x = x * y
    return x + y

@given(integers(0, 10), integers(-5, 10))
def test_is_good_value(x, y):
    assert special_sum(x, y) == (x + y)
```

Listing 8.5 the example presents the special_sum and a unit test. The **given** decorator provides to the test randomly generated values for each argument. In this example, the simple usage of the Hypothesis tool is used to verify the special_sum function. The test is run once, and a random integer between 0 and 10 is given to x, and another between -5 and 10 is provided to y. When running the tool, it outputs a *Falsifying example* for the failing test.

The pyCheck tool allows the user to express more meaningful properties by using refinement types, which improves the code behavior specification. Lets consider the Listing 8.6 example, similar to the previous example, decorated with the pyCheck framework.

Listing 8.6: Function verification of the special_sum with pyCheck.

```
from pyCheck.pyCheck import provide, runall

# Decorated function
@provide('{x:Integer | (x >= 0) && (x <= 10) && (x != 5)}',
         '{y:Integer | (y >= -5) && (y <= x)}',
         expected='{z:Integer | z == x + y}',
         repeat=100)
def special_sum(x, y):
    # Bug here, this entire if should be removed
    if x > 7 or y > 7:
```

[1] Tool available in `https://hypothesis.works/`

```
11          x = x * y
12      return x + y
13
14  # Test all the annotated functions in the demo file
15  runall('pyCheck.demo')
```

Listing 8.6 presents the special_sum function responsible for calculating the sum of two restricted values. The function is annotated with a **provide** decorator, which will tell the programmer to test this function when running the runall command. The **provide** decorator receives two mandatory parameters: the variables of the function, and the expected return, and one optional parameter, the number of tests. In this case, the programmer has to provide two input types for the arguments x and y, and the expected output. For this function, we wanted to demonstrate two key elements, the refinement types on the input and output parameters and the dependent type, on the second parameter.

How does this tool work? Firstly, the arguments and expected types are parsed and type-checked. Then, for each run test, the following procedure is done:

1. Generate the values for each argument of the function using the non-deterministic synthesizer (see Chapter 6);
2. Run the Python function with the synthesized values and retrieve the returned value;
3. Use the evolutionary synthesis fitness evaluation to compare the expected and retrieved values;

After all the test runs are completed, the framework provides a small report to the user. This small report, presented in Listing 8.7, provides the user information on the non-deterministic tests made, displays the ones that failed and determines the accuracy of the function for each objective.

Listing 8.7: Final report of the pyCheck.

```
1  ...
2  ERROR: Refined test failed for input values: [9, -3]
3  SUCCESS: Refined test passed for input values: [6, 2]
4  SUCCESS: Refined test passed for input values: [4, -4]
5
6  ----------------------------------------------------------------------
7  Report:
8  Tests passed: 80 / 100
9  Function Accuracy: 94.88%
10
11 Function failed for the following random generated input tests
12 (x = 9, y = -3)
13 (x = 9, y = 4)
14 (x = 10, y = 6)
15 ...
```

On Listing 8.7, it is possible to verify that, although the number of tests passed was 80 out of 100, the function accuracy is 94.88%. The difference occurs since the evaluation of the objectives does not only check whether the test passed but also how close the program is to be correct.

This example evidences the differences between Hypothesis and pyCheck. The first difference is the random generator. Hypothesis can more easily provide random values generated between two values, using, for instance, the `integers()` function. However, it cannot easily randomly generate discontinuous values within a range. By using the non-deterministic synthesizer, pyCheck is capable of non-deterministically generating these values on discontinuous ranges, like the one generated for the x argument. On the second difference, Hypothesis does not allow dependent types or dependent values based on previously generated inputs. It would require instrumentation on the test code in order to allow this kind of features. On the other hand, pyCheck can easily use previous synthesized arguments as a restriction to the current type. Finally, the last difference is the accuracy of each objective. The pyCheck tool can, similarly to Hypothesis, provide the user the tests failed but also extra information on the test accuracy and how correct the program is.

8.3 Limitations and Challenges

This section presents the limitations found in the ÆON programming language and the RTGP concept.

Readability with Polymorphism An issue identified during the evaluation was the readability when we have type applications. Currently, in order for the program to typecheck, the programmer is required to apply the proper type to the type abstraction, which may lead to readability issues like the ones seen in Listing 8.4. Recently, programming languages with polymorphism, like Java, have been working towards minimizing the type applications with generics, in order to improve the overall code quality.

Error handling An important feature of programming languages is to provide useful and accessible error messages to the user. The translation from ÆON to ÆONCORE loses some information related to original source code, such as line and column. Adding such information to ÆONCORE would make it dependent on the syntactic sugar frontend (which is not ideal). In order to more easily understand error messages, we have developed a translator from ÆONCORE to ÆON, but still, the information lost in the first translation creates a challenge users in the programming language.

Inconsistent refinements The crucial component for RTGP to work is tied with the quality of the refinement types. Writing specifications is a tedious work and sometimes requires some creativity from the programmer (e.g., the direction change in the turn_left function in Listing 8.3). Errors in the specification of the components or the synthesis target function will induce RTGP in error and not produce the intended program.

Search complexity in the non-deterministic synthesis The non-deterministic synthesizer works on a trial and error during the expression synthesis. There are currently two main problems with the synthesizer, which prevents further work with the evolutionary approach.

The first issue is related to the synthesis depth. At each point, a synthesis rule is non-deterministically chosen to generate an expression. When choosing a rule at depth d, the synthesizer in unacknowledged if it is possible to synthesize that expression at specified depth. Figure 8.3 presents an example where the synthesizer, at a point, chooses a rule for the synthesis. In this particular case, the context only contains

a function which requires the depth $d = 2$ to return the type Image, but the initial chosen rule already consumed part of the required depth for the synthesis, requiring the synthesizer to try again until it can synthesize an expression or a maximum amount of tries is reached.

$$\frac{\Gamma \vdash \textsf{Boolean} \rightsquigarrow_1 \textsf{true} \quad \dfrac{fail : \text{Not enough depth}}{\Gamma, \textsf{true} \vdash \text{Image} \rightsquigarrow_1 fail} \quad \dfrac{fail : \text{Not enough depth}}{\Gamma, \neg\textsf{true} \vdash \text{Image} \rightsquigarrow_1 fail}}{\Gamma, f : (f : \textsf{Integer} \to \text{Image}) \vdash \text{Image} \rightsquigarrow_2 \textsf{if true then } fail \textsf{ else } fail} \quad \text{(SE-If)}$$

Figure 8.3: Maximum depth issue in expression synthesis.

The second issue addressed is related to the application rule. The way the rule is built is based on trials and errors. Let us revisit the application expression synthesis rule:

$$\frac{\rightsquigarrow_d k \quad \Gamma \vdash k \rightsquigarrow_d U \quad \Gamma \vdash U \rightsquigarrow_d e_2 \quad (x \text{ fresh}) \quad \Gamma, x : U \vdash_{[e_2/x]} T \rightsquigarrow_d V \quad \Gamma \vdash (x : U \to V) \rightsquigarrow_d e_1}{\Gamma \vdash T \rightsquigarrow_{d+1} e_1 e_2} \quad \text{(SE-App)}$$

Figure 8.4: Application expression synthesis rule.

The synthesizer starts by generating the e_2 expression. Firstly, the synthesis of the type U does not ensure there is an expression in the context or one which can be synthesized, that can inhabit the type U. Without any instrumentation, the synthesizer is forced to try different types until it is able to find one able to synthesize e_2. The similar happens when generating a new type V from the type T.

One of the final and main limitation is the interaction between the SE-Where rule and the remaining synthesis rules. The SE-Where rule propagates the refinement through the context, and asks the remaining rules to synthesize the type T. With the exception of the SE-Bool and SE-Int because of the intrumentation detailed in Section 6.3, the other rules do not consider the refinement propagated in the context, and only try to synthesize an expression which complies with T. After the synthesis, when checking the synthesized expression against the refinement the likelihood of failing is high with tight refinements.

These issues currently present a bottleneck in RTGP, as it has a huge impact in the algorithms performance.

Genetic information overload The evolutionary procedure uses the extra non-used genetic information from an individual or mate to generate mutations or recombinations. However, using all the remaining information from an individual may introduce unecessary genetics in the genetic pool. For instance, there could be duplicated subtrees or subtrees with the same semantics within the set of genetic information. In the end, the unnecessary amount of genetic information worsens the non-deterministic synthesizer performance.

Chapter 9

Future Work

This chapter presents the future work to bridge the synthesis framework and evolutionary procedure issues.

Adaptive inductive biased hyperparameter optimization The ÆON programming language relies on the non-deterministic synthesizer (Chapter 6) and its weighted rules to generate randomly biased expressions from types. These rule weights are currently hardcoded for every synthesis problem. The objective would be to adapt the weights for each synthesis rule according, not only to the problem itself, but also the context and current depth during the synthesis.

As a simple example, let us look at the **If** synthesis rule. Typically, an ordinary programmer does not compute extremely nested **if** expressions (e.g. 5+ nested if conditions), thus these kinds of expressions should not be generated. A naive solution for this rule passes on decreasing its weight every time the synthesis enters a new if-body context.

The optimization, however, is not as this trivial as it may appear. A first question is: how much should we decrease? Not only we need to ensure this weight never reaches 0 (ensuring that all programs can still be synthesized) but also, introducing this control parameter may influence the synthesis not to follow a what we can consider standard program outline. Furthermore, other non-noticeable programming patterns may arise in this simple example and not taken into consideration if the adaptive rules are not expressive enough (e.g. if I check a variable on the if condition, the likelihood of being used in its body is higher, but the probability of rechecking the same condition is lower).

Not only the weights of the synthesis rules are needed to be adaptively improved, but also the weights inside the rules themselves need to be enhanced. Currently, the native values (booleans, integers, doubles and strings) already have some improvement (Section 6.2). The same would be required to happen on the variables and when combining expressions, for example, on the expression (`x <= ??`), it may not make sense to synthesize x again.

The improvement of the synthesis rules needs to be made by studying the patterns on an expressive set of programs. By building a model using a genetic algorithm would allow us to quickly obtain synthesis weights based on the context, depth, and type.

Type-safe genetic operators The evolutionary synthesis depends on the genetic operators in order to generate the correct individuals. Different strategies for each component for the genetic operators,

which the evolutionary procedure relies on, are continuously being developed. When creating the current evolutionary approach, standard crossover and mutation strategies were used. The objective would be to study and experiment with the combination of different strategies of genetic operators in order to allow a faster convergence.

Non-functional requirements optimization Currently, the ÆON programming language allows the synthesizes of programs that comply with the formal specification, the functional requirements. The objective is to allow the user to either annotate or provide on the specification the intention on optimizing non-functional requirements, such as execution time, memory or energy consumption. The non-functional optimization of programs is not novel and has been recently studied in the area of genetic improvement and program synthesis [52, 58]; thus, its addition would only improve the language features.

Listing 9.1 presents a proposal on optimizing the energy consumption of the previous cipher function. By providing the `@minimize(energy)` condition, natively implemented in the foreign language, the user indicates to the framework that after the functional requirements are met, it should start optimizing this non-functional requirement.

Listing 9.1: Energy consumption optimization in ÆON.

```
1 cipher(i:Integer, k:Key) -> {j:Integer | j > 0 and @minimize(energy) ...} {
2     ??;
3 }
```

Extend the standard library The current ÆON standard library is currently quite limited. One of the main objectives is to provide the user all the functions he may need for the program synthesis, reducing the necessity of programming its components. Currently, some basic libraries have been implemented, with lists, strings, map, and image. The main objective is to improve the library set by introducing new libraries and extending the class of problems ÆON can tackle.

Non-liquid types verification at runtime Non-liquid refinement types are currently just used in ÆON for extracting the fitness functions for the program synthesis. However, the real purpose of these refinements is to verify at runtime whether the conditions hold or not. Such verification, however, has an impact on the execution time of the program. We aim to provide the user of the language the option to enable the runtime verification of its program, allowing the non-liquid refinements to fulfil their purpose.

Automatic repair tool By the time the non-liquid refinement types verification is implemented it is possible to introduce a new strategy for the framework, an automatic repair tool. Figure 9.1 shows a simplified scheme on the behaviour of this automatic repair tool.

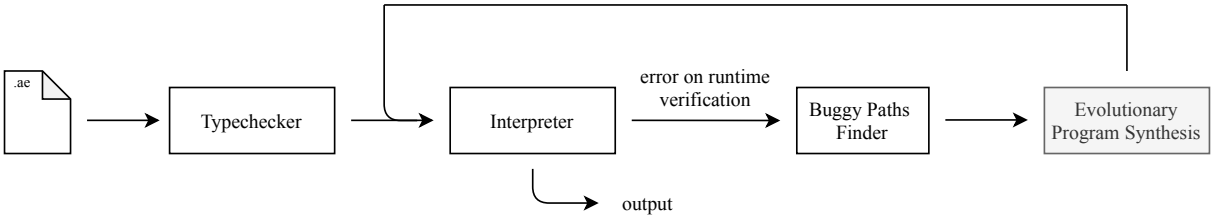


Figure 9.1: Automatic repair tool from non-liquid refinements.

The first part of the repair tool works like a typical programming language. It uses the verifier of the non-liquid types that we can check at runtime whether the condition on the refinement holds or not. If the condition does not hold, then the repair process triggers, and similarly to other automatic repair approaches [22], the program will try to find the buggy paths on the function. It will then create a random population with mutations on the defected target functions paths, and provide it to the evolutionary program synthesis, to generate a program that complies with the entire specification. With barely any changes required on the synthesis framework, the creation of this automatic repair tool would be proof of the versatility on the concepts presented by this work.

Evaluation on an extensive benchmark suite The evolutionary program synthesis approach was tested on a small subset of the general program synthesis benchmark suite [26] and was meaningful enough to evaluate different kind of synthesis problems, like recursive-based problems. The evaluation, however, is not expressive to consider more polymorphic types, other than `List`. The objective is to create a more complex benchmark suite that considers more complex types and evaluate Æon on it.

Chapter 10

Conclusion

Program Synthesis (PS) is the task of generating programs from a specification. This specification is typically provided through examples or a formal specification. Synthesis with examples occurs by giving pairs of input/output to the system, and expect the generated program to fit the test suite. This kind of approaches, however, frequently overfit to the test suite and are not able to generalize to the right solution. Synthesis from a formal specification, on the other hand, is capable of synthesizing general solutions according to the program behaviour from the specification. In the genetic programming area, nevertheless, two main problems arise when trying to apply this kind of approach: the usability and the extensive combination of the synthesis components. STGP presented a naive approach by using the type system of the language to limit the amount of operations combination and ensure program validity, thus reducing the search space for the correct program. However, the limitation of the search space with only the standard types is still not enough. GGGP bridges this problem by allowing the programmer to define the grammar and the operator combinations for the synthesis. However, not only this approach requires the programmer to define a new grammar for each problem, as some computational paradigms such as recursion are not supported.

This work introduces a new framework capable of restricting the search space of valid programs with the help of the liquid refined types and generate generalized programs from random tests evaluated with the non-liquid refined types. Refined Typed Genetic Programming (RTGP) is the new concept presented by this work for the area of Genetic Programming, and namely the area of program synthesis. The ÆON programming language was created on the concept of RTGP to allow the user to specify and synthesize their programs with ease by introducing holes in the program.

The ÆON language works as a syntactic frontend from the ÆONCORE language, used on the RTGP for the program synthesis. This frontend improves its core language usability by hiding and deducing underlined concepts. The overall synthesis framework contains two main components: the non-deterministic synthesizer and the evolutionary procedure.

The non-deterministic synthesizer is responsible for synthesizing the expressions from the liquid types. By providing the typing context, maximum depth and the liquid type, it is possible to generate any valid expression. This is accomplished with the creation and optimization of synthesis rules for the ÆONCORE type system. Since even with the refined types, the search space can be too vast, each synthesis rule is given a weight to help the synthesizer. This weight introduced the probability of a particular

synthesis expression to be used, thus reducing the space of programs we are searching. Furthermore, we provided an auxiliary module capable of automatically generating values within tight restrictions and discontinuous values.

Finally, the evolutionary procedure uses the non-liquid types to evaluate the correctness of the synthesized programs. The non-liquid types are essential in this part, as they are converted to continuous fitness function to provide information on the program correctness. This procedure uses a non-deterministic synthesizer to generate valid expressions. Then, with the help of genetic programming, it continuously evaluates and evolves populations of programs until we have reached a correct and valid program.

This work then proved its potential with a strict comparison with GGGP. Not only that but each component was proven its versatility when applied to a different kind of system: a property-based testing prototype tool. pyCheck was created using the non-deterministic synthesizer and the fitness extraction for continuous evaluation, allowing more meaningful testing information to the user other than pass or fail.

To conclude, although this work has its limitations and much future work is yet to be done, we believe that these components and the system overall have contributed towards the current state of the art approaches based on genetic programming, and have improved the synthesis of programs with the unlikely combination of refined type theory and evolutionary computation.

Appendix A

Type System

$$\frac{}{\vdash \varepsilon \text{ context}} \qquad \frac{\vdash \Gamma \text{ context} \quad \Gamma \vdash T : k}{\vdash \Gamma, x\colon T \text{ context}}$$

$$\frac{\vdash \Gamma \text{ context}}{\vdash \Gamma, t\colon k \text{ context}} \qquad \frac{\vdash \Gamma \text{ context} \quad \Gamma \vdash e : \textbf{Boolean}}{\vdash \Gamma, e \text{ context}}$$

Figure A.1: Context formation, $\boxed{\vdash \Gamma \text{ context}}$.

$$\frac{}{\Gamma \vdash \textbf{Integer} \Rightarrow *} \qquad \frac{}{\Gamma \vdash \textbf{Boolean} \Rightarrow *} \qquad \text{(K-Int, K-Bool)}$$

$$\frac{t\colon k \in \Gamma}{\Gamma \vdash t \Rightarrow k} \qquad \text{(K-Var)}$$

$$\frac{\Gamma \vdash T \Rightarrow k \quad \Gamma, x\colon T \vdash e \Leftarrow \textbf{Boolean}}{\Gamma \vdash (x\colon T \textbf{ where } e) \Rightarrow k} \qquad \text{(K-Where)}$$

$$\frac{\Gamma \vdash T \Leftarrow * \quad \Gamma, x\colon T \vdash U \Leftarrow *}{\Gamma \vdash (x\colon T \to U) \Rightarrow *} \qquad \text{(K-Abs)}$$

$$\frac{\Gamma, t\colon k \vdash T \Rightarrow k'}{\Gamma \vdash (\forall t\colon k.T) \Rightarrow k \to k'} \qquad \text{(K-TAbs)}$$

$$\frac{\Gamma \vdash T \Rightarrow k \to k' \quad \Gamma \vdash U \Leftarrow k}{\Gamma \vdash TU \Rightarrow k'} \qquad \text{(K-TApp)}$$

Figure A.2: Type formation, $\boxed{\Gamma \vdash T \Rightarrow k}$.

$$\frac{\vdash \Gamma \text{ context} \quad T = \textbf{Integer}, \textbf{Boolean}}{\Gamma \vdash T{<:}T : *} \qquad \frac{\Gamma \vdash t : k}{\Gamma \vdash t{<:}t : k} \qquad \text{(K-S-Int, S-Bool, S-Var)}$$

$$\frac{\Gamma \vdash T'{<:}T : * \quad \Gamma, x\colon T' \vdash U{<:}U' : * \quad \Gamma, x\colon T \vdash U : *}{\Gamma \vdash (x\colon T \to U){<:}(x\colon T' \to U') : *} \qquad \text{(S-Abs)}$$

$$\frac{\Gamma \vdash T{<:}U : k \quad \Gamma, x\colon T \vdash e \ \mathsf{true}}{\Gamma \vdash T{<:}(x\colon U \ \textbf{where}\ e) : k} \qquad \text{(S-WhereR)}$$

$$\frac{\Gamma \vdash T{<:}U : k \quad \Gamma, x\colon T \vdash e : \textbf{Boolean}}{\Gamma \vdash (x\colon T \ \textbf{where}\ e){<:}U : k} \qquad \text{(S-WhereL)}$$

$$\frac{\Gamma, t\colon k \vdash T{<:}U : k'}{\Gamma \vdash (\forall t\colon k.T){<:}(\forall t\colon k.U) : k \to k'} \qquad \text{(S-TAbs)}$$

$$\frac{\Gamma \vdash t : (k \to k') \quad \Gamma \vdash U{<:}U' : k}{\Gamma \vdash tU{<:}tU' : k'} \qquad \text{(S-TApp)}$$

$$\frac{\Gamma, t\colon k \vdash T : k' \quad \Gamma \vdash U : k \quad \Gamma \vdash T[U/t]{<:}V : k'}{\Gamma \vdash (\forall t\colon k.T)U{<:}V : k'} \qquad \text{(S-TAppL)}$$

$$\frac{\Gamma, t\colon k \vdash U : k' \quad \Gamma \vdash V : k \quad \Gamma \vdash T{<:}U[V/t] : k'}{\Gamma \vdash T{<:}(\forall t\colon k.U)V : k'} \qquad \text{(S-TAppR)}$$

Figure A.3: Subtyping, $\boxed{\Gamma \vdash T{<:}U : k}$.

Bibliography

[1] Chang Wook Ahn and Rudrapatna S. Ramakrishna. Elitism-based compact genetic algorithms. *IEEE Trans. Evol. Comput.*, 7(4):367–385, 2003. doi: 10.1109/TEVC.2003.814633. URL https://doi.org/10.1109/TEVC.2003.814633.

[2] Bernhard K. Aichernig. *Contract-Based Testing*, volume 2757 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2002. doi: 10.1007/978-3-540-40007-3_3. URL https://doi.org/10.1007/978-3-540-40007-3_3.

[3] 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. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8. IEEE, 2013. URL http://ieeexplore.ieee.org/document/6679385/.

[4] Andrea Arcuri and Xin Yao. Coevolving programs and unit tests from their specification. In R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer, editors, *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pages 397–400. ACM, 2007. doi: 10.1145/1321631.1321693. URL https://doi.org/10.1145/1321631.1321693.

[5] João E. Batista and Sara Silva. Improving the detection of burnt areas in remote sensing using hyper-features evolved by M3GP. In *IEEE Congress on Evolutionary Computation, CEC 2020, Glasgow, United Kingdom, July 19-24, 2020*, pages 1–8. IEEE, 2020. doi: 10.1109/CEC48606.2020.9185630. URL https://doi.org/10.1109/CEC48606.2020.9185630.

[6] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.*, 33(2):8:1–8:45, 2011. doi: 10.1145/1890028.1890031. URL https://doi.org/10.1145/1890028.1890031.

[7] Iwo Bladek and Krzysztof Krawiec. Evolutionary program sketching. In James McDermott, Mauro Castelli, Lukás Sekanina, Evert Haasdijk, and Pablo García-Sánchez, editors, *Genetic Programming - 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings*, volume 10196 of *Lecture Notes in Computer Science*, pages 3–18, 2017. doi: 10.1007/978-3-319-55696-3_1. URL https://doi.org/10.1007/978-3-319-55696-3_1.

[8] Nuno Miguel Pereira Burnay. Types to the rescue: verification of rest apis consumer code. Master's thesis, Universidade de Lisboa, Faculdade de Ciências, 2019. URL http://hdl.handle.net/10451/39881.

[9] William G. La Cava, Lee Spector, and Kourosh Danai. Epsilon-lexicase selection for regression. In Tobias Friedrich, Frank Neumann, and Andrew M. Sutton, editors, *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference, Denver, CO, USA, July 20 - 24, 2016*, pages 741–748. ACM, 2016. doi: 10.1145/2908812.2908898. URL https://doi.org/10.1145/2908812.2908898.

[10] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. ISBN 978-0-262-02665-9. URL http://mitpress.mit.edu/books/certified-programming-dependent-types.

[11] Iain D. Craig. *Genetic Algorithms and Simulated Annealing* edited by lawrence davis pitman, london, 1987 (£19.95). *Robotica*, 6(2):170–171, 1988. doi: 10.1017/S0263574700004215. URL https://doi.org/10.1017/S0263574700004215.

[12] Elisa Boari de Lima, Gisele L. Pappa, Jussara Marques de Almeida, Marcos André Gonçalves, and Wagner Meira Jr. Tuning genetic programming parameters with factorial designs. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2010, Barcelona, Spain, 18-23 July 2010*, pages 1–8. IEEE, 2010. doi: 10.1109/CEC.2010.5586084. URL https://doi.org/10.1109/CEC.2010.5586084.

[13] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24. URL https://doi.org/10.1007/978-3-540-78800-3_24.

[14] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015. doi: 10.1007/978-3-319-21401-6_26. URL https://doi.org/10.1007/978-3-319-21401-6_26.

[15] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. Program synthesis using natural language. pages 345–356, 2016. doi: 10.1145/2884781.2884786. URL https://doi.org/10.1145/2884781.2884786.

[16] Susana C. Esquivel, A. J. Leiva, and Raúl H. Gallard. Multiple crossover per couple in genetic algorithms. In *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC '97)*, pages 103–106, 1997.

[17] Robert Feldt and Peter Nordin. Using factorial experiments to evaluate the effect of genetic programming parameters. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming, European Conference, Edinburgh, Scotland, UK, April 15-16, 2000, Proceedings*, volume 1802 of *Lecture Notes in Computer Science*, pages 271–282. Springer, 2000. doi: 10.1007/978-3-540-46239-2_20. URL https://doi.org/10.1007/978-3-540-46239-2_20.

[18] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 420–435. ACM, 2018. doi: 10.1145/3192366.3192382. URL https://doi.org/10.1145/3192366.3192382.

[19] George Fink and Matt Bishop. Property-based testing: A new approach to testing for assurance. *SIGSOFT Softw. Eng. Notes*, 22(4):74–80, July 1997. ISSN 0163-5948. doi: 10.1145/263244.263267. URL https://doi.org/10.1145/263244.263267.

[20] Alcides Fonseca, Paulo Santos, and Sara Silva. The usability argument for refinement typed genetic programming. In Thomas Bäck, Mike Preuss, André H. Deutz, Hao Wang, Carola Doerr, Michael T. M. Emmerich, and Heike Trautmann, editors, *Parallel Problem Solving from Nature - PPSN XVI - 16th International Conference, PPSN 2020, Leiden, The Netherlands, September 5-9, 2020, Proceedings, Part II*, volume 12270 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2020. doi: 10.1007/978-3-030-58115-2_2. URL https://doi.org/10.1007/978-3-030-58115-2_2.

[21] R. Forsyth. Beagle: A darwinian approach to pattern recognition. *Kybernetes*, 10:159–166, 03 1981. doi: 10.1108/eb005587.

[22] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012. doi: 10.1109/TSE.2011.104. URL https://doi.org/10.1109/TSE.2011.104.

[23] Nabil Hassein. Notes on idris. URL https://nabilhassein.github.io/blog/notes-on-idris/.

[24] Thomas D. Haynes, Dale A. Schoenefeld, and Roger L. Wainwright. Type inheritance in strongly typed genetic programming. *Advances in genetic programming*, 2(2):359–376, 1996.

[25] Thomas Helmuth and Amr M. Abdelhady. Benchmarking parent selection for program synthesis by genetic programming. In Carlos Artemio Coello Coello, editor, *GECCO '20: Genetic and Evolutionary Computation Conference, Companion Volume, Cancún, Mexico, July 8-12, 2020*, pages 237–238. ACM, 2020. doi: 10.1145/3377929.3389987. URL https://doi.org/10.1145/3377929.3389987.

[26] Thomas Helmuth and Lee Spector. General program synthesis benchmark suite. In Sara Silva and Anna Isabel Esparcia-Alcázar, editors, *Proceedings of the Genetic and Evolutionary Computation*

Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015, pages 1039–1046. ACM, 2015. doi: 10.1145/2739480.2754769. URL https://doi.org/10.1145/2739480.2754769.

[27] Thomas Helmuth, Lee Spector, and James Matheson. Solving uncompromising problems with lexicase selection. *IEEE Trans. Evol. Comput.*, 19(5):630–643, 2015. doi: 10.1109/TEVC.2014.2362729. URL https://doi.org/10.1109/TEVC.2014.2362729.

[28] Thomas Helmuth, Nicholas Mcphee, and Lee Spector. *Lexicase Selection for Program Synthesis: A Diversity Analysis*, pages 151–167. Springer International Publishing, 12 2016. ISBN 978-3-319-34221-4. doi: 10.1007/978-3-319-34223-8_9.

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

[30] Joe. Java integer cache, Mar 2020. URL https://javapapers.com/java/java-integer-cache/.

[31] Emanuel Kitzelmann. Inductive programming: A survey of program synthesis techniques. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *Approaches and Applications of Inductive Programming, Third International Workshop, AAIP 2009, Edinburgh, UK, September 4, 2009. Revised Papers*, volume 5812 of *Lecture Notes in Computer Science*, pages 50–73. Springer, 2009. doi: 10.1007/978-3-642-11931-6_3. URL https://doi.org/10.1007/978-3-642-11931-6_3.

[32] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. Resource-guided program synthesis. pages 253–268, 2019. doi: 10.1145/3314221.3314602. URL https://doi.org/10.1145/3314221.3314602.

[33] John R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112, Jun 1994. ISSN 1573-1375. doi: 10.1007/BF00175355. URL https://doi.org/10.1007/BF00175355.

[34] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. doi: 10.1007/978-3-642-17511-4_20. URL https://doi.org/10.1007/978-3-642-17511-4_20.

[35] José María Luna, José Raúl Romero, and Sebastián Ventura. G3PARM: A grammar guided genetic programming algorithm for mining association rules. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2010, Barcelona, Spain, 18-23 July 2010*, pages 1–8. IEEE, 2010. doi: 10.1109/CEC.2010.5586504. URL https://doi.org/10.1109/CEC.2010.5586504.

[36] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. Trinity: An extensible synthesis framework for data science. *Proc. VLDB Endow.*, 12(12):1914–1917, 2019. doi: 10.14778/3352063.3352098. URL http://www.vldb.org/pvldb/vol12/p1914-martins.pdf.

[37] Robert I. McKay, Nguyen Xuan Hoai, Peter Alexander Whigham, Yin Shan, and Michael O'Neill. Grammar-based genetic programming: a survey. *Genet. Program. Evolvable Mach.*, 11(3-4):365–396, 2010. doi: 10.1007/s10710-010-9109-y. URL https://doi.org/10.1007/s10710-010-9109-y.

[38] Phil McMinn. Search-based software test data generation: a survey. *Softw. Test. Verification Reliab.*, 14(2):105–156, 2004. doi: 10.1002/stvr.294. URL https://doi.org/10.1002/stvr.294.

[39] David J. Montana. Strongly typed genetic programming. *Evol. Comput.*, 3(2):199–230, 1995. doi: 10.1162/evco.1995.3.2.199. URL https://doi.org/10.1162/evco.1995.3.2.199.

[40] Alfonso Ortega, Marina de la Cruz, and Manuel Alfonseca. Christiansen grammar evolution: Grammatical evolution with semantics. *IEEE Trans. Evol. Comput.*, 11(1):77–90, 2007. doi: 10.1109/TEVC.2006.880327. URL https://doi.org/10.1109/TEVC.2006.880327.

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

[42] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. *IEEE Transactions on Software Engineering*, 40(5):427–449, May 2014. ISSN 2326-3881. doi: 10.1109/TSE.2014.2312918.

[43] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. volume 40, pages 427–449, 2014. doi: 10.1109/TSE.2014.2312918. URL https://doi.org/10.1109/TSE.2014.2312918.

[44] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000. doi: 10.1145/345099.345100. URL https://doi.org/10.1145/345099.345100.

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

[46] Yewen Pu, Zachery Miranda, Armando Solar-Lezama, and Leslie Pack Kaelbling. Learning to select examples for program synthesis. *CoRR*, abs/1711.03243, 2017. URL http://arxiv.org/abs/1711.03243.

[47] Baishakhi Ray, Daryl Posnett, Premkumar T. Devanbu, and Vladimir Filkov. A large-scale study of programming languages and code quality in github. *Commun. ACM*, 60(10):91–100, 2017. doi: 10.1145/3126905. URL https://doi.org/10.1145/3126905.

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

[49] Noor Shaker, Miguel Nicolau, Georgios N. Yannakakis, Julian Togelius, and Michael O'Neill. Evolving levels for super mario bros using grammatical evolution. In *2012 IEEE Conference on Computational Intelligence and Games, CIG 2012, Granada, Spain, September 11-14, 2012*, pages 304–311. IEEE, 2012. doi: 10.1109/CIG.2012.6374170. URL https://doi.org/10.1109/CIG.2012.6374170.

[50] Armando Solar-Lezama. The sketching approach to program synthesis. In Zhenjiang Hu, editor, *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, volume 5904 of *Lecture Notes in Computer Science*, pages 4–13. Springer, 2009. doi: 10.1007/978-3-642-10672-9_3. URL https://doi.org/10.1007/978-3-642-10672-9_3.

[51] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. pages 404–415, 2006. doi: 10.1145/1168857.1168907. URL https://doi.org/10.1145/1168857.1168907.

[52] Ashish Tiwari, Adrià Gascón, and Bruno Dutertre. Program synthesis using dual interpretation. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 482–497. Springer, 2015. doi: 10.1007/978-3-319-21401-6_33. URL https://doi.org/10.1007/978-3-319-21401-6_33.

[53] Leonardo Trujillo, Luis Muñoz, Edgar Galván López, and Sara Silva. neat genetic programming: Controlling bloat naturally. *Inf. Sci.*, 333:21–43, 2016. doi: 10.1016/j.ins.2015.11.010. URL https://doi.org/10.1016/j.ins.2015.11.010.

[54] A. M. Turing. Computers & thought. In Edward A. Feigenbaum and Julian Feldman, editors, *Mind*, chapter Computing Machinery and Intelligence, pages 11–35. MIT Press, Cambridge, MA, USA, 1995. ISBN 0-262-56092-5. URL http://dl.acm.org/citation.cfm?id=216408.216410.

[55] Paulo Urbano and Loukas Georgiou. Improving grammatical evolution in santa fe trail using novelty search. In Pietro Liò, Orazio Miglino, Giuseppe Nicosia, Stefano Nolfi, and Mario Pavone, editors, *Proceedings of the Twelfth European Conference on the Synthesis and Simulation of Living*

Systems: Advances in Artificial Life, ECAL 2013, Sicily, Italy, September 2-6, 2013, pages 917–924. MIT Press, 2013. doi: 10.7551/978-0-262-31709-2-ch137. URL https://doi.org/10.7551/978-0-262-31709-2-ch137.

[56] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. Liquidhaskell: experience with refinement types in the real world. pages 39–51, 2014. doi: 10.1145/2633357.2633366. URL https://doi.org/10.1145/2633357.2633366.

[57] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement types for typescript. pages 310–325, 2016. doi: 10.1145/2908080.2908110. URL https://doi.org/10.1145/2908080.2908110.

[58] David Robert White, Andrea Arcuri, and John A. Clark. Evolutionary improvement of programs. *IEEE Trans. Evol. Comput.*, 15(4):515–538, 2011. doi: 10.1109/TEVC.2010.2083669. URL https://doi.org/10.1109/TEVC.2010.2083669.

[59] Tina Yu. Polymorphism and genetic programming. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea Tettamanzi, and William B. Langdon, editors, *Genetic Programming, 4th European Conference, EuroGP 2001, Lake Como, Italy, April 18-20, 2001, Proceedings*, volume 2038 of *Lecture Notes in Computer Science*, pages 218–233. Springer, 2001. doi: 10.1007/3-540-45355-5_17. URL https://doi.org/10.1007/3-540-45355-5_17.