Master Degree in Computer Engineering

Master Degree Thesis

Design and implementation of an Evolutionary Fuzzer for assembly program

Author:

Dimitri Masetta

Supervisors:

Prof. Giovanni Squillero, Supervisor

Prof. Alberto Tonda, Co-Supervisor

Politecnico di Torino

July 2024

Acknowledgements

I would like to express my sincere gratitude to everyone who has supported me throughout the process of completing this thesis. Their contributions, both large and small, have been invaluable.

First and foremost, I would like to thank my supervisor, Giovanni Squillero, for his unwavering support, guidance, and encouragement. Your expertise and insight have been crucial in shaping this work. I am deeply grateful for your mentorship.

On a personal note, I would like to thank my family for their unconditional love and support. To my parents, Nadia and Claudio, your belief in me has been a constant source of motivation. To my siblings, Angelica and Nicolò, thank you for your encouragement and for always being there. Finally, to my partner, Arianna, your patience, understanding, and support have been my anchor through the ups and downs of this journey. I would also like to acknowledge the support of my friends, for always being there to provide a listening ear.

This thesis would not have been possible without the support and contributions of all these individuals. To each of you, I extend my deepest gratitude.

Sincerely,

Dimitri Masetta

Abstract

This thesis describes the design and implementation of Byron, an extensible, self-adaptive, general-purpose Evolutionary Fuzzer, capable of fuzzing complex test-program in assembly language. Evolutionary Algorithm (EA) principles and techniques were first described by John H. Holland in the 1960s; in 1990 John Koza described Genetic Programming, a new approach that differs in the type of representation, using trees, and in the application area. From these roots arise various alternative representations of GP, Linear Genetic Programming (LGP), Cartesian Genetic Programming (CGP), and Graph Genetic Programming (GGP); the last one, which is used in Byron implementation. The tool represents the genotypes as Directed Acyclic Graphs (DAGs), which allow the evolution of the graph without going through an intermediate encoding, but directly manipulating the graphs. Byron, through the use of multiple genetic operators and a self-adaptation mechanism, is able to perform fuzzing of programs written in assembly language; this is due to its framework that allows the encoding of candidate solutions employing typed Directed Multi-Graphs, characterized by the possibility of having multiple edges that connect the same couple of vertices. Moreover, this graph encoding features two edge types, “Framework” or “Link”, that are bound by the type of vertices they connect. Experiments show the applicability and performance of this tool on the RISC-V Instruction Set Architecture, with results that can be extended to any general programs written in assembly language.

Contents

List of Figures

Chapter 1

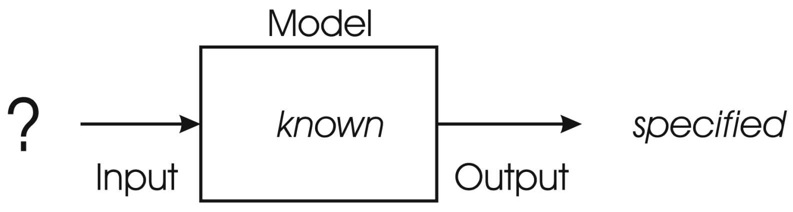
Introduction to Evolutionary Computation

Evolutionary Computing (EC) is a research area within computer science that involves a family of algorithms, called Evolutionary Algorithm (EA), for global optimization, valid in a wide range of problem settings, inspired by the natural process of biological evolution.
The basic idea of EA is that given an environment and embedded in it a population, the individuals in it will compete for reproduction and survival. To evolve the population, to better fit the given environment, mechanisms inspired by biological evolution are used, such as reproduction, mutation, recombination, and selection. The context then becomes that of a stochastic trial-and-error, where the individuals are iteratively modified through the application of evolutionary operators, with the aim of improving the performance. The evolutionary operators are inspired by the processes of natural selection and reproduction, where individuals who are better adapted to their environment have more probability to survive and reproduce. The reproduction of an individual is subjected to crossover with another one and then to mutation, to increase diversity.
EC is a rapidly growing field, that has been utilized in a broad range of applications, such as optimization, machine learning, and network design. It has proven to be an effective method for addressing complex issues that are challenging to resolve with conventional techniques.
The aim of this chapter is thus to provide an introduction to the Evolutionary Algo-

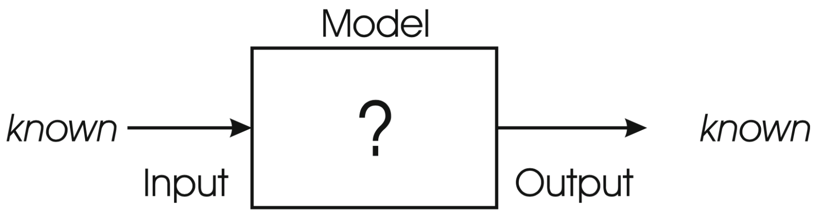
rithm and then delve into the details of Graph Genetic Programming (GGP), an EA technique that underpin the structure of Byron.

1.1 History

The foundational idea of using evolutionary principles for computing can be traced back to the work of Charles Darwin and his theory of natural selection [1]. These principles inspired early computer scientists to think about optimization and problem-solving in new ways; indeed, as early as 1948 Turing proposed “genetical or evolutionary searches” and by 1962 Bremermann performed experiments on “optimization through evolution and recombination”.
In the 1960s, John Holland at the University of Michigan pioneered the concept of genetic algorithms (GA) [2–4]. GAs use operators like selection, crossover, and mutation to evolve solutions to problems over successive generations, focusing on problems of optimization and machine learning. Around the same time, Ingo Rechenberg and Hans-Paul Schwefel in Germany developed Evolution Strategies (ES) [5, 6], initially applied to optimization problems in engineering. ES emphasizes mutation and selection but with a focus on continuous parameter optimization. Always in the same years, Lawrence J. Fogel introduced Evolutionary Programming (EP). EP focuses more on the evolution of finite-state machines and is generally used for optimization problems and machine learning.
In the early 1990s, John Koza proposed Genetic Programming (GP) [7–9]. GP differs from the other EAs in its application area, in that it can be seen as a machine learning algorithm; in fact most other EAs try to find some input to realize the maximum payoff, Figure 1.1a, while GP tries to find models with maximum fit [10], Figure 1.1b.
Storn and Price introduced Differential Evolution (DE) in the mid-1990s [11–13]. DE is a method used for optimizing complex, multi-dimensional functions without using the gradient, which means that DE does not require the problem to be differentiable or even continuous; DE has been widely applied in engineering and economic problems.
The 1990s also saw advancements in multi-objective optimization using evolutionary algorithms (MOEA). Techniques like Non-dominated Sorting Genetic Algorithm - II (NSGA-II) by Kalyanmoy Deb and Strength Pareto Evolutionary Algorithm (SPEA)



(a) Optimisation problems



(b) Modelling or system identification problems

Fig. 1.1 Types of problems to be solved

were developed to handle problems with multiple conflicting objectives.
Modern evolutionary computing often involves hybrid algorithms that combine evolutionary principles with other optimization methods and machine learning algorithms; this area of research is called memetic algorithms (MA). These hybrid techniques aim to improve efficiency and solution quality.

1.2 Evolutionary Algorithm

There are numerous variants of evolutionary algorithms available, each of which is better suited to a particular application context. However, the underlying principle across all these techniques is consistent: given an environment with limited resources and a population of individuals in it, competition leads to natural selection (survival of the fittest), enhancing the population's overall fitness. Indeed, given a quality function to maximize (fitness function), we can randomly generate a set of candidate solutions in the function's domain. For each candidate is then applied the fitness function in order to measure the quality of the solution, favoring higher values. Based on these fitness values, some superior candidates are selected to seed the next generation through recombination and/or mutation.

Recombination involves combining elements from two or more selected candidates (parents) to create new candidates (children), while mutation alters a single candidate to produce a new one. These operations generate a set of new candidates (offspring), whose fitness is then evaluated. Then the offspring compete with the old candidates for a place in the next generation, based on the quality of the individuals that is measured by their fitness. This process iterates until a terminal condition is met, such as an individual with a valid solution, a computational time limit, or a convergence criterion is reached.

At the base of every Evolutionary Algorithm two main forces determine the improvement of the fitness values across successive generations; they are variation operators (recombination and mutation), which introduce diversity within the population, and selection, which increases the mean quality of solutions in the population. The combined effect of variation and selection typically leads to an improvement in fitness values across successive generations. This process can be seen as an optimization of the fitness function by approaching optimal values over time. Consequently, it follows that the population becomes increasingly adapted to the environment.

It's important to note that the evolutionary process is not merely deterministic, but rather has some stochastic components such as selection. The parent selection doesn't deterministically choose the best individuals, even weaker individuals may become parents, although with less probability than the better ones. Recombination randomly chooses which pieces from the parents to recombine, and mutation randomly selects which pieces within a candidate solution to modify.

Different streams of EA are often characterized by how candidate solutions are represented, such as strings over a finite alphabet (generally bit-strings) in genetic algorithms (GA), real-valued vectors in evolution strategies (ES), finite state machines in classical evolutionary programming (EP), and trees in genetic programming (GP). Therefore, the various forms of EA differ only in technical details, while the general scheme of an EA remains the same across all streams, as illustrated in flowchart and pseudocode in Figure 1.3 and Figure 1.2 respectively.

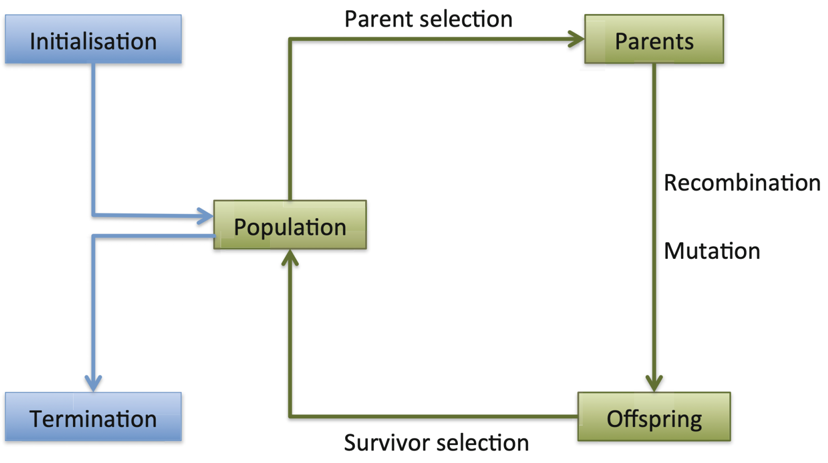


Fig. 1.2 The general scheme of an evolutionary algorithm as a flowchart

```
BEGIN
  INITIALISE population with random candidate solutions;
  EVALUATE each candidate;
  REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
    1 SELECT parents;
    2 RECOMBINE pairs of parents;
    3 MUTATE the resulting offspring;
    4 EVALUATE new candidates;
    5 SELECT individuals for the next generation;
  OD
END
```

Fig. 1.3 The general scheme of an evolutionary algorithm in pseudocode

In the next sections, we will delve into the specifics of EA, providing a definition of its key components, which include:

- **Representation**
- **Population**
- **Initialization**
- **Evaluation function**
- **Parent selection**

- **Variation operators**
- **Survivor selection**
- **Terminal conditions**

1.3 Individual representation

Representation in Evolutionary Algorithms is a critical aspect that influences the performance and efficiency of the algorithm. It determines how potential solutions to a problem are encoded and manipulated during the evolutionary process. In this context, potential solutions, which are in the solution space, are called phenotypes, while their encoded versions, which are manipulated within the EA, are referred to as genotypes. Therefore the representation involves defining the genotype structure and the mapping from genotype to phenotype (decoding), such that all possible feasible solutions can be represented within the phenotype space, and each potential phenotype has at least one genotype encoding it.
The representation of solutions in EA can take various forms, each suited to different types of problems; special care should be taken in choosing the right representation for the specific problem at hand, as it affects the efficiency of the EA. Follow a brief, non-exhaustive, introduction to the primary types of representations:

- **Binary representation**, Figure 1.4a, encodes solutions as binary strings, comprising 0s and 1s. This form is widely utilized in Genetic Algorithms and is particularly suitable for problems where solutions can be naturally expressed as a series of binary decisions, such as the knapsack problem.
- **Integer representation**, Figure 1.4b, uses strings of integers to represent solutions. This type of representation is beneficial for problems that involve selecting from a discrete set of options, including scheduling and routing problems.
- **Real-value representation**, Figure 1.4c, encodes solutions as strings of real numbers, often used in Evolution Strategies or Evolutionary Programming. This representation is ideal for continuous domains, such as function optimization, where precision and granularity are essential.

- **Permutation representation**, Figure 1.4d, represents solutions as permutations of a set of elements. It is particularly effective for ordering problems, such as the traveling salesman problem or job scheduling, in which the order of elements is crucial.
- **Tree representation**, Figure 1.4e, uses tree structures to represent solutions, a method primarily employed in Genetic Programming. This representation suits problems where solutions can be expressed hierarchically, such as symbolic regression or program synthesis.
- **Graph Representation**, Figure 1.4f, encodes solutions as graphs. This form is widely utilized in Graph Genetic Programming. It is useful for circuit design problems, neural network topology optimization, or program synthesis; in fact, it has been the representation used for Byron development.

1.4 Population

In Evolutionary Algorithms, the concept of population is fundamental; a population is a collection of potential solutions to the problem at hand. Each member of the population is typically referred to as an individual and it is characterized by its genotype; it follows that a population can be seen as a multiset (set where potentially there are multiple occurrences of the same element) of genotypes.
The population serves as the pool from which candidate solutions are selected, modified, and evaluated. The evolutionary processes operate at the level of the population, which evolves over time, while the individuals are static entities that remain unchanged; it is the population itself that undergoes adaptation and change, thanks to selection, that as the name suggests choose a set of individuals from the current population, and variation operators, that generate new individuals.
In most EA applications, the population size is constant, and it is considered a crucial parameter that needs to be balanced. A population too small may lead to premature convergence, while a population too large may be computationally expensive and slow down the convergence process.
Just like in nature, it is important to promote diversity to avoid competition and premature convergence. To measure the diversity there is not a single metric but it can be done by means of the number of distinct lineages, phenotypes, genotypes, or

0	1	0	1	1	1	0	0

(a) Binary representation

7	1	33	14	6	58	2	19

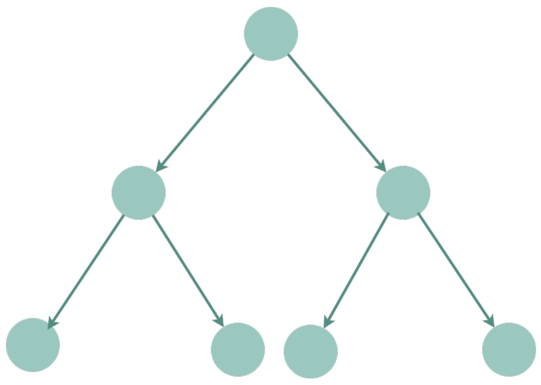
(b) Integer representation

3,2	4,3	6,1	1,4	2,6	5,8	1,9	7,8

(c) Real-value representation

7	2	3	5	6	8	1	4

(d) Permutation representation


(e) Tree representation

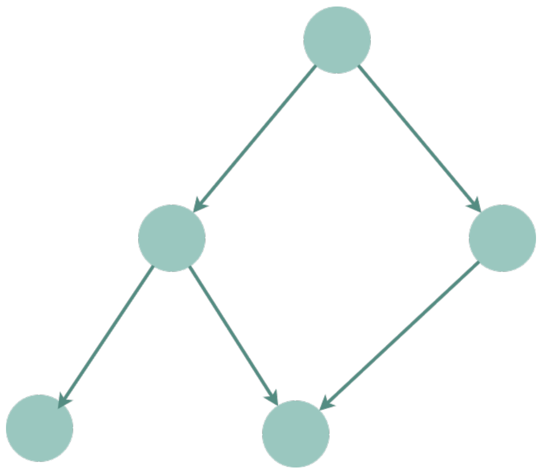

(f) Graph representation

Fig. 1.4 Types of genotype representations

fitness of the individuals or by calculating the entropy of the population. To improve diversity various population models, that manage the distance between neighbors individuals, can be used, like island or segregation models.
The design and management of the population, including its size, diversity, and selection strategies, are crucial for the success of the algorithm in finding optimal or near-optimal solutions to complex problems.

1.5 Initialization

In Evolutionary Algorithms, the initialization process sets the stage for the evolutionary process. The process begins with the creation of an initial population. This can be done in multiple ways:

- **Random initialization** randomly generates individuals within the allowable range of values (the most used solution).
- **Quasi-random sequences** uses sequences such as Sobol or Halton to generate more uniformly distributed initial populations.
- **Heuristic initialization** uses domain-specific heuristics to generate diverse and high-quality initial solutions.

Whether the additional computational effort of not using the random initialization is justified, largely hinges on the specific application, as the quality of the initial solutions is generally uninfluential in the evolutionary process.

1.6 Fitness evaluation

An evaluation function, also known as a fitness function, is a crucial component of an Evolutionary Algorithm. It is used to evaluate and assign a fitness score to each candidate solution in the population, or more technically to each genotype. Typically, this function is first constructed by decoding the genotype into the phenotype and then applying a quality measure in the phenotype space.
Thus, the score obtained reflects how well the solution solves the problem at hand. The EA uses these scores to guide the selection process for reproduction, favoring

individuals with higher fitness scores to produce the next generation, and by so directly favoring the improvements that really matter.

1.7 Parent selection

Parent selection determines which individuals from the current population will be chosen to reproduce and generate offspring for the next generation. In EA the parent selection is usually probabilistic, having as a goal the ensuring that the best-performing individuals have a higher chance of passing their genes to the next generation while maintaining genetic diversity; to maintain diversity, individuals of lower quality are still afforded a small chance to become parents. Follow a brief, not exhaustive, list of the common methods:

- **Fitness-proportionate methods:**
  - **Roulette wheel selection** individuals are selected based on their fitness proportion. The probability of selection is directly proportional to an individual's fitness.
  - **Stochastic Universal Sampling (SUS)** a variation of fitness-proportionate selection where multiple individuals are selected in one go. This ensures a more even spread of selected individuals.
  - **Boltzmann selection** the selection probability depends on both fitness and temperature parameter. As the temperature decreases, selection pressure increases, favoring fitter individuals.
- **Ordinal-based methods:**
  - **Rank-based selection** individuals are ranked based on their fitness. Selection probabilities are assigned based on these ranks, which helps to avoid issues with large fitness differences.
- **Tournament-based methods:**
  - **Tournament selection** a group of individuals is chosen randomly from the population, and after a series of pairwise competitions, the best individual from this group is selected as a parent. This process is repeated until the required number of parents is selected.

- **Deterministic methods:**
  - **Truncation selection** only the top-performing individuals, based on a predefined threshold, are selected as parents. For example, the top 20% of the population might be chosen.

1.8 Variation operators

Variation operators in Evolutionary Algorithms are crucial for generating new individuals in the population, and by so improving the solutions diversity, enabling the algorithm to explore the search space effectively.
There are two kinds of variation operators, based on the arity of the operation performed, mutation, unary, and recombination, n-ary; follow a detailed look at each.

1.8.1 Mutation

The mutation is a unary variation operator that alters the genotype of the parent, to generate a modified offspring. The mutation operator executes a random and unbiased change, that usually, only slightly, modifies the parent genotype; for this reason, it is defined as a purely stochastic operator. Usually, the mutation operator is characterized by the mutation rate, which defines the probability with which a mutation is applied to an individual or gene. Given that, each EA flavors have its own characteristic representation, there are different mutation operators for each of them. Follow some examples of mutation operators for the most common representations:

- **Binary representation:**
  - Bit flipping
- **Integer representation:**
  - Random resetting
  - Creep mutation
- **Float representation:**
  - Uniform mutation

    - Gaussian mutation
    - Self-adaptive mutation

- **Permutation representation:**
    - Swap mutation
    - Insert mutation
    - Scramble mutation
    - Inverse mutation

- **Tree/Graph representation:**
    - Point mutation
    - Permutation
    - Hoist mutation
    - Expansion mutation
    - Collapse mutation
    - Subtree mutation

1.8.2 Recombination

The recombination, or crossover, is a n-ary variation operator that from the genotype of usually two parents creates a new genotype for the offspring. This process is inspired by the biological mechanism of reproduction and recombination in natural evolution. Recombination leverages the genetic information of the two parents to create a new solution that has a genotype that is a mix of them. The recombination operator has some common properties with the mutation one, in fact, just like that, it is a stochastic operator, and it is representation dependent. Follow some examples of recombination operators for the most common representations:

- **Binary representation:**
    - One-point crossover
    - N-point crossover

  - Uniform crossover

- **Integer representation:**
  - One-point crossover
  - N-point crossover
  - Uniform crossover
- **Float representation:**
  - Simple arithmetic crossover
  - Single arithmetic crossover
  - Whole arithmetic crossover
  - Blend crossover
- **Permutation representation:**
  - Partially mapped crossover
  - Edge crossover
  - Order crossover
  - Cycle crossover
- **Tree/Graph representation:**
  - Subtree swap
  - Leaf swap

1.9 Survivor operators

Survivor selection, also known as replacement, works similarly to parent selection, as selecting a subset of the population based on their quality, favoring higher quality individuals. It should be noted, however, that survivor selection operates at a different phase of the evolutionary algorithms and, in doing so, acts over the population and the newly generated offspring; furthermore, in contrast to the parent selection, the decision made by the survivor selection is usually fully deterministic and based on

the individuals' fitness.
Survivor selection is crucial in guiding the evolutionary process, ensuring that, from the current population and offspring, high-quality solutions are retained for the next generation while also promoting diversity and balancing the exploration of the solution space.
Different strategies can be used depending on the specific requirements and characteristics of the optimization problem at hand. Given that the size of the population is usually maintained constant, this necessitates a decision on which individuals, from the current population and offspring, will progress to the next generation. Follow a list of the most common strategies for survivor selection, where from μ parents and λ offspring select μ individuals for the next generations.

- **Fitness-based replacement:**
  - **Elitism** ensures that the best individuals from the current generation are preserved and carried over to the next generation. This prevents the loss of high-quality solutions due to genetic operations.
  - **Round-robin tournament** pairwise tournament competitions are held between the individuals in a round-robin format (each individual competes against n other individuals). The μ individuals that have won most of the tournament are selected as survivors.
  - $(\mu + \lambda)$ **selection** the next generation individuals are selected from the combined pool of μ parents and λ offspring. The best μ individuals, ranked by their fitness, are chosen as survivors.
  - (μ, λ) **selection** the next generation individuals are selected only from the λ offspring. The best μ individuals, ranked by their fitness, are chosen as survivors. If $\mu = \lambda$ is also called a generational approach.
- **Age-based replacement:**
  - **Age selection** operates on the principle that an individual's fitness is not considered, instead, ensures that each individual remains in the population for an equal number of generations.

1.10 Terminal conditions

In Evolutionary Algorithm, termination conditions are crucial to determine when the algorithm should stop running. These conditions help ensure that the algorithm doesn't run indefinitely and also signal when a satisfactory solution has been found. Common terminal conditions include:

- **Fixed number of generations** the algorithm stops after a predetermined number of generations or iterations, regardless of the solutions' quality.
- **Convergence criteria** the algorithm terminates when the population converges, meaning that there is little or no improvement in the fitness of the best solution over a set number of generations.
- **Fitness threshold** the algorithm stops when a solution reaches or exceeds a predefined fitness level or objective value.
- **Computational time limit** the algorithm runs for a specified amount of time and then stops, even if it has not reached other termination criteria.
- **User-defined criteria** problem-specific, such as the algorithm stops when a solution reaches a specific target.
- **Resource limit** the algorithm halts if it reaches a limit on resources like memory or CPU usage.

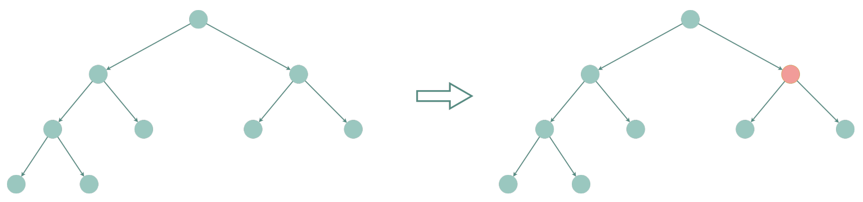
The selection of the appropriate termination condition depends on the specific problem and goals of the Evolutionary Algorithm. Often, a combination of several termination conditions is employed to ensure a balance between computational efficiency and solution quality.

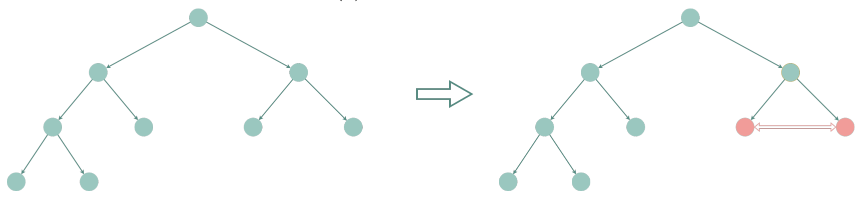
1.11 Graph Genetic Programming

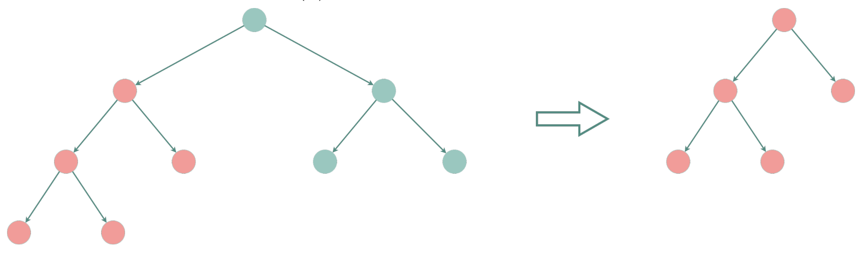
Since Byron is based on Graph Genetic Programming (GGP) a greater focus should be placed on that. Genetic Programming (GP) is a fairly new EA technique proposed by Koza in the 1990s [8] that uses a tree representation; from that derives GGP, which represents programs as directed graphs instead of tree structures. This approach,

often used in the generation of programs, leverages the flexibility and expressiveness of directed graphs data structure, where nodes represent operations, and edges represent the program flow, to evolve solutions to complex problems; given that, the definition “automatic evolution of computer programs”, provided by W. Banzhaf et al. [7], fits like a glove.
In contrast to most EAs that have as objective to find some input in order to maximize the payoff, GGP, and more in general GP, tackles a different kind of problem since tries to seek models that provide a maximum fit [10].
As introduced earlier, GGP take leverage of a directed graph representation, where the vertices are usually operations (single line of code) while edges define the program flows, thus allowing multiple edges entering a single node; this is the great flexibility that a tree representation cannot provide.
Regarding the selection operators, the parent selection is usually fitness proportional while the survival selection typically employs a generational or $(\mu + \lambda)$ approach.
The variation operators for GGP are the same that are used in GP because the tree and graph representation are fairly similar; follow a list of the most common:

- **Mutation:**
  - **Point mutation**, Figure 1.5a also known as node mutation, involves mutating the operation contained in a node. It is the most used kind of mutation and the least destructive.
  - **Permutation**, Figure 1.5binvolves the exchange of two leaf nodes.
  - **Hoist mutation** leads to the consideration of only a subtree.
  - **Expansion mutation**, Figure 1.5c involves the substitution of a node with a copy of a subtree/subgraph of the original tree/graph.
  - **Collapse mutation**, Figure 1.5d is the reverse of the expansion mutation, involves the collapsing of one subtree/subgraph.
  - **Subtree/subgraph mutation**, Figure 1.6a leads to the mutation of the structure of an entire subtree/subgraph.
- **Recombination:**, Figure 1.6b
  - **Subtree/subgraph swap**, Figure 1.6c involves the swap of to subtree/-subgraph from the two parents’ genotypes.


(a) Point mutation


(b) Permutation mutation


(c) Hoist mutation

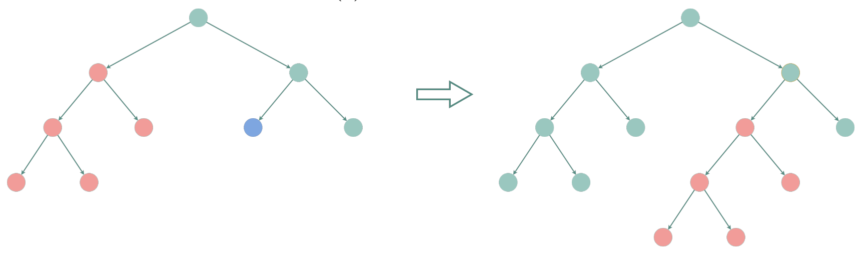
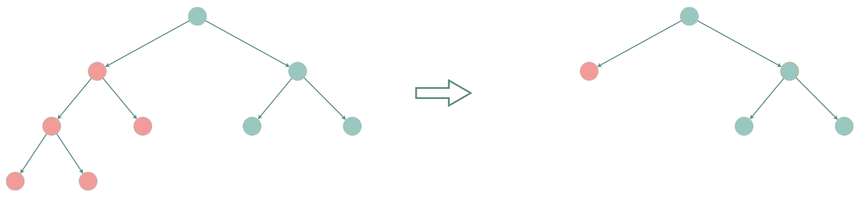
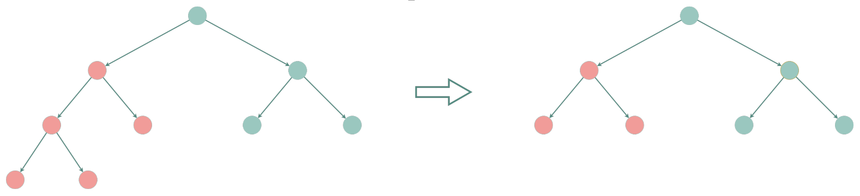

(d) Expansion mutation

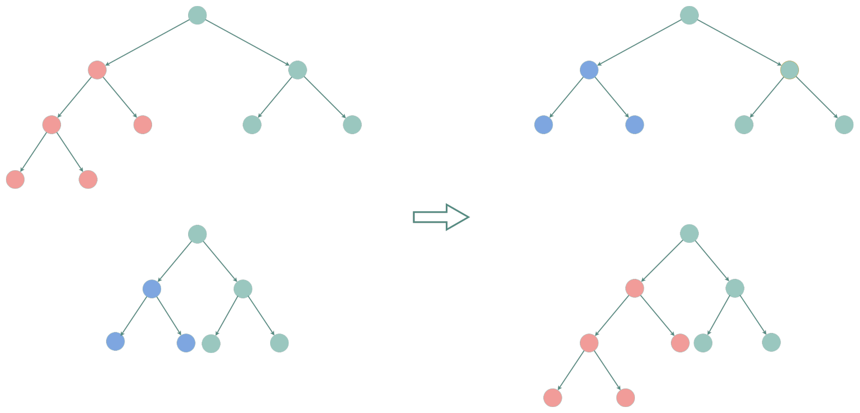
Fig. 1.5 Types of GGP variation operators

- **Leaf/node swap**, Figure 1.6d is a more specific version of the subtree/-subgraph swap, that involves the swapping of a specific node instead of an entire subtree/subgraph.

The variation operators for GGP, as can be seen from the description above, yield significant variations in the offspring genome compared with the parent genome. Therefore, in GGP are either used mutation or recombination, rather than recombination followed by mutation like in most EA, as can be seen in the Figure 1.7. In addition, since the recombination operators have a large effect on the genome, Koza advises not to use mutation operators [8]; by doing that, however, you don't have the opportunity to insert new elements in the genome, so the solutions are using a


(a) Collapse mutation


(b) Subtree mutation


(c) Subtree swap

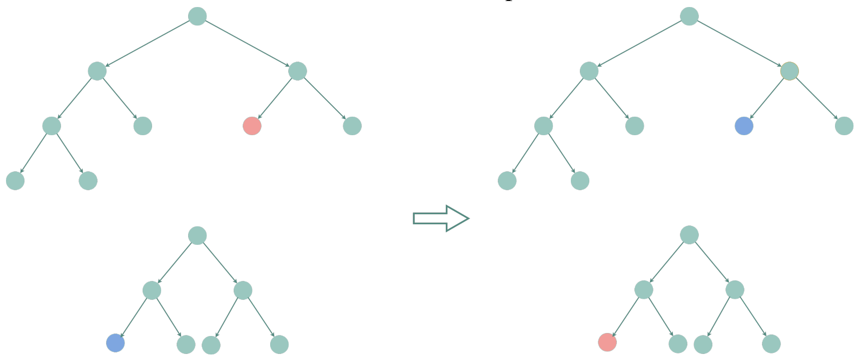

(d) Leaf swap

Fig. 1.6 Types of GGP variation operators

large populations size, to have almost all potential nodes from the start, or use a low mutation rate, like the 5% proposed by Banzhaf et al. [7].

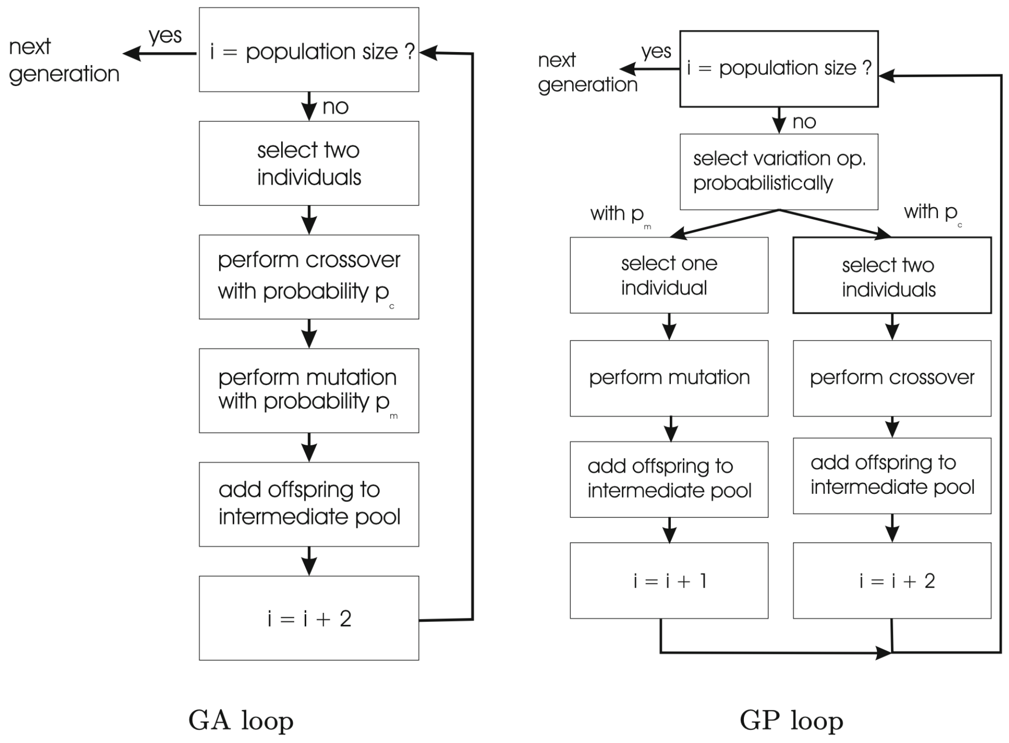


Fig. 1.7 GP flowchart versus GA flowchart.

Since the kind of problems addressed by GGP requires a variable length representation, in contrast to most EAs, can happen that over the generations the solutions suffer from the bloating problem; which means that the graph size tends to grow during the evolution process. To overcome this problem the following solutions can be applied:

- Limiting the action of the operators that induce the growth of the graph.
- Ad-hoc operators that destroy unused paths in the graph.
- Apply parsimony pressure, through the introduction of a penalty term in the fitness that increases with the dimension of the graph.
- Using multi-objective techniques.

Despite the GGP representation introduces additional complexity and computational overhead, the benefits in terms of flexibility and expressiveness make it a valuable tool for solving a wide range of problems and for effectively discovering high-quality solutions.

Chapter 2

RISC-V

The original Byron implementation was focused on the development and study of an evolutionary fuzzer for programs written in assembly language for x86 microprocessors. Given the rise in popularity of new microprocessors, developed to support the RISC-V Instruction Set Architecture (ISA) we choose to develop the Byron capability further, so as to support this new type of ISA.
In the context of computer architecture, the dominance of proprietary ISAs was the norm, however, RISC-V placed itself as an open-source ISA provided under royalty-free licenses. Introduced in the early 2010s at the University of California, Berkeley, RISC-V aims to provide a flexible, modular, and extensible ISA capable of carrying out a wide variety of use cases from low-power to high-performance implementations [14]; to greatly reduce the complexity of the systems, the RISC-V architecture is based on the load and store principles.
This chapter aims to provide a basic background on the RISC-V ISA, without going into too much detail, as it is not the purpose of this thesis, and then focus on the implementation in Byron.

2.1 RISC-V ISA

The RISC-V ISA is fundamentally structured as a base integer ISA, which is mandatory in any implementation, supplemented by optional extensions that provide further operations. The RISC-V ISA is based on settled reduced instruction set computer (RISC) principles, in fact, the base integer ISA resembles that of the early RISC

processors, but without branch delay slots and supporting variable-length instruction encodings.
The base RISC-V ISA features fixed-length 32-bit instructions, each of which must be aligned at 32-bit intervals. However, the standard allows to accommodate ISA extensions with variable-length instructions. These variable-length instructions can consist of any number of 16-bit instruction parcels, with each parcel aligned at 16-bit boundaries. The addition of optional variable-length instructions serves a dual purpose by broadening the available instruction encoding space and enabling a dense instruction encoding [14].
The base integer ISA is made by a minimal set of instructions, integer computational instructions, integer loads, integer stores, and control-flow instructions, adequate to provide a target for compilers, assemblers, linkers, and operating systems; thus, providing a framework for building customized processor ISAs around this base. For signed integers, the base instruction sets utilize two's complement representation.
Actually, there are multiple base integer instruction sets, listed below, that are distinguished by the number and width of the integer registers, that are dictating the user address space size:

- **RV32I** base instruction set integer variant that provides a 32-bit address space size, with 32 registers.
- **RV32E** base instruction set integer subset variant of RV32I, specifically developed for small and embedded microcontrollers, that provide a 32-bit address space size, with 16 registers.
- **RV64I** base instruction set integer variant that provides a 64-bit address space size, with 32 registers.
- **RV128I** base instruction set integer variant that provides a 128-bit address space size, with 32 registers.

All the base instruction sets can be implemented in Byron, as it will be defined in section 4.1, but for academic purposes, it is just analyzed the use of the base instruction set integer variant RV32I, which will be defined in section 2.2.
As seen above, the base ISA has a limited number of instructions available; to offer other functionalities that support a more general software development multiple

standard extensions are provided, which offer integer multiply and divide, single-precision arithmetic, double-precision arithmetic, and atomic operations. A complete list of all the standard extensions available is outside the scope of this thesis, therefore follow a list of the most important ones and corresponding labels:

- **M**, integer multiply and divide extension, provides instructions to multiply and divide values held in two integer registers.
- **A**, atomic instructions extension, provides instructions to atomically execute read-modify-write operations, to support synchronization across multiple threads.
- **F**, single-precision floating-point extension, provides instructions that allow single-precision floating-point arithmetic that are compliant with the IEEE 754-2008 standard.
- **D**, double-precision floating-point extension, extends the single-precision floating-point to support double-precision arithmetic.

2.2 RISC-V RV32I Base Integer Instruction Set

The RISC-V base integer ISA RV32I consists of 31 general-purpose registers x1-x31, which hold integer values of 32 bits of lengths, plus the x0 register that is hardwired to the constant value of 0. While there isn't a dedicated return address register, the x1 register is typically used for this purpose. There is another register visible to the user which is the program-counter (pc), that tracks the current instruction address.
A defining feature of the RISC-V architecture is the load and store approach, which allows to execute operations only between registers or immediates, therefore all the access to data in memory must be performed through load and store operations. It follows that the RV32I base ISA utilizes six instruction formats R/I/S/B/U/J, as can be seen in the Figure 2.1, that allow to perform computational, memory access, control flow, and system instructions. All instructions are fixed at 32 bits of length and are aligned to four-byte boundaries. To define a common interface to all instructions and streamline the decoding, RISC-V ISA consistently places the source

registers, rs1 and rs2, and the destination register, rd, at the same position across all the instruction formats. Follow a list of all the available instruction formats:

- **R-type** performs operations between registers.
- **I-type** performs operations between a register and an immediate.
- **S-type** performs store operations.
- **B-type** performs branch operations.
- **U-type** performs operations with 20-bit immediates.
- **J-type** performs jump operations.

31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

Fig. 2.1 RISC-V base instruction formats

Immediates are sign-extended and packed towards the leftmost bits for hardware simplification, with the sign bit consistently in bit 31 for all immediates. Figure 2.2 provides a visual representation of how immediate values are derived from different RISC-V instruction formats.

31	30 20	19 12	11	10 5	4 1	0	
— inst[31] —				inst[30:25]	inst[24:21]	inst[20]	I-immediate
— inst[31] —				inst[30:25]	inst[11:8]	inst[7]	S-immediate
— inst[31] —			inst[7]	inst[30:25]	inst[11:8]	0	B-immediate
inst[31]	inst[30:20]	inst[19:12]	— 0 —				U-immediate
— inst[31] —		inst[19:12]	inst[20]	inst[30:25]	inst[24:21]	0	J-immediate

Fig. 2.2 RISC-V base instruction formats showing immediate variants

The RISC-V RV32I base ISA consists of 47 instructions that can be divided into the following groups:

- **Computational operations** they include arithmetic, logic, shift, and comparison operations, that can be further subdivided according to the operations format (the [U] variants are the unsigned ones):
  - **R-type** with the arithmetic instructions ADD, SUB, the shift instructions SLL, SRL, SRA, the logic instructions AND, OR, XOR, and the comparison instruction SLT[U].
  - **I-type** with the counterpart of the R-type instructions for register-immediate operations; the arithmetic instruction ADDI, the shift instructions SLLI, SRLI, SRAI, the logic instructions ANDI, ORI, XORI, and the comparison instruction SLTI[U].
  - **U-type** with the LUI(Load Upper Immediate) and AUIPC(Add Upper Immediate to PC) instructions.
- **Control flow operations** they include unconditional and conditional jumps, that can be further subdivided according to the operations format:
  - **J-type** with the plain unconditional jump instruction JAL (Jump And Link).
  - **I-type** with the indirect jump instruction JALR (Jump And Link Register).
  - **B-type** with the conditional jump instructions BEQ, BNE, BLT[U], BGE[U].
- **Load and store instructions** can be further subdivided according to the operations format. Should be noted that, being RISC-V a load and store architecture, they are the only instructions that can perform memory access.
  - **I-type** with the various load instructions LB, LH, LW, LBU, LHU
  - **S-type** with the store instructions SB, SH, SW
- **System instructions** they include operation in the I-type format that allows ordered device I/O and memory access with the FENCE instruction, synchronize

the operation and data stream with the FENCE.I instruction, perform environment calls with the ECALL instruction, place breakpoints with the EBREAK instruction, and manage the control and status registers with the following instructions CSRRW, CSRRS, CSRRC, CSRRWI, CSRRSI, CSRRCI.

2.3 RISC-V Extensions

The modularity of RISC-V allows developers to tailor the ISA to specific requirements. The base ISA provides essential integer operations, while standard extensions can be added for additional functionality. This modularity facilitates customization and optimization for various applications. RISC-V's extensibility enables the addition of custom instructions and extensions, fostering innovation and specialization. This feature is particularly valuable for emerging fields like artificial intelligence and machine learning, where specialized hardware acceleration is often required. To enhance its capabilities beyond general software development, a suite of standard extensions is provided. These extensions offer functionalities such as integer multiplication and division(M), single-precision(F), and double-precision(D) arithmetic, as well as atomic operations(A).
Focusing on the RISC-V M extension, it consists of 8 instructions (with 5 additional if we extend the base RV64I) that can be divided into the following groups:

- **Multiplication operations** with all R-type operations; MUL performs a 32-bit x 32-bit multiplication of the input registers by placing the lower 32-bit of the result in the destination register; MULH, MULHU and MULHSU perform the same multiplication but returning the upper 32 bit of the result for signed x signed, unsigned x unsigned, and signed x unsigned multiplication respectively.

- **Division operations** with all R-type operations; DIV and DIVU perform signed and unsigned integer division respectively; REM and REMU return signed and unsigned integer division remainder respectively.

Chapter 3

Byron

Byron is an extensible, self-adaptive, general-purpose Evolutionary Fuzzer, that can be categorized as part of the family of Graph Genetic Programming (GGP). Its primary goal is to generate Turing-complete assembly programs.
A fuzzer is a program that performs testing by generating pseudo-random sequences of input data to uncover issues or provoke unintended behaviors in software. Over the past few decades, fuzzing has enabled the detection of thousands of bugs and vulnerabilities across a wide range of applications, making it an essential tool in all security-related contexts [15–19]. In that context, following the work of [19, 20], Byron exploits the capability of GGP to create original code, through an evolutionary process, to perform fuzzing of assembly-language test programs.
In Byron, candidate solutions are encoded as typed, directed multigraphs. Typed because, the type system specifies the data that can be stored in vertices and the constraints over the edges; while the multigraph data structure encodes the program's control and data flow.
Byron is the successor of the C++ tool MicroGP (μGP) [21], sharing similar objectives and key concepts. However, Byron has been redesigned to enhance usability, flexibility, and expandability, and it has been re-implemented in Python from scratch. The tool is Free and Open Software (FOSS), and it is distributed under the permissive Apache 2.0 license; it is available as a package through PyPi, with ongoing active development hosted on GitHub[22].
The following sections describe the structure and the functioning of the current version of Byron, the 0.8.

3.1 Individual representations

The byron.classes.Individual class represents a candidate solution that competes in the evolutionary process. The set of all individuals is managed by the Byron.classes.Population class. Every individual is characterized by the following main attributes, which will be analyzed below:

- **ID**
- **Genome**
- **Fitness**
- **Lineage**
- **Age**

The ID is a unique identifier that allows to distingue and identify individuals.
As the name suggests, the genotype of each individual is stored inside the genome attribute; the genome is encoded as a typed Directed Multi-Graph, which is stored as a NetworkX MultiDiGraph, which is a directed graph class that can have multi-edges; that is, it can have multiple edges between the same pair of vertices. Therefore, the genome data structure of the individual can be seen as an ordered tuple G = (V, T, E, es, et, t) where:

- **V** is a set of vertices
- **T** is a set of types
- **E** is a set of edges
- **es: E -> V** is a function that maps each edge to its source vertex
- **et: E -> V** is a function that maps each edge to its target vertex
- **t: V -> T** is a function that associates each vertex with its type

As introduced before, the kind of genome graph is typed; a typed graph is a graph in which each vertex and edge is assigned a type from a predefined set. These types can impose constraints on how the vertices and edges can be connected, as can be seen

in the following section. Going down into more detail, each vertex, in an individual genome, can be seen as an instance of a specific class, that defines its type and by so specifies the possible attributes of the vertex, while each edge is defined by its type and a unique label. With regard to the vertices, the potential set of classes available, that can be instantiated, are either macro or frame, further details are available in the next section; while regarding the edge there are two kinds, structural or FRAMEWORK, that store the program structure, and reference or LINK, that store parameters or references. If only structural edges are considered, the definition of the representation of an individual genome as a Directed Multi-Graph can be further refined as a forest, that is a disjoint union of trees; each tree is a connected acyclic graph with a node, defined as root, and all the edges of structural type.
Must be noted that, given the forest-like representation of the individual genome, a concept of relationship between the nodes can be established; each node can have one predecessor and one or more successors, and by so each node can have a set of parents and a set of descendants. Additionally, the successors of a given node are ordered.
For implementation purposes, all the disjoint trees are actually connected together to a common root node, called "node zero"; therefore transforming the representation definition from a forest to a single tree. As introduced before, edges of type reference are also possible; they are used to connect leaf nodes among them, whether or not they are part of the same tree. In contrast to structural edges, in this case no parental binding or ordering between nodes is defined. Thus, considering jointly the two types of edge, the representation falls under the initially proposed definition of typed Directed Multi-Graph, which provides Byron with a great deal of flexibility regarding the graph representation, meeting every need.
The fitness attribute, as the name suggests, stores the fitness of the individual; after the evaluation of the candidate solution, the fitness and the statistics on the performance of the variation operator, used for the individual generation, are updated; the update in the statistics allows to employ the self-adaptation mechanism, for more details see section 3.5. To calculate the fitness of an individual, its genotype must be decoded into the phenotype representation; this is done by the dump method, which deals with the typed Directed Multi-Graph representation and through a deep-first convert it in a text file, more details on section 3.3.
The lineage attribute stores the parent or parents and the variational operator that has generated the individual.

The age attribute is updated each generation, and it is used in case an aging approach is used, instead of a $(\mu + \lambda)$ ones, for survivor selection.
Should be finally noted that individuals are created by passing a reference to the top frame, which will be analyzed in section 4.1.

3.2 The Type System

The type system is in charge of defining the type of the vertices and the constraints over the edges, found in the typed Directed Multi-Graph representation of each individual; this type system allows Byron to offer a huge flexibility and expressiveness.
The type system defines two main types for the vertices, either macro or frame, from which the user can define his own different macros or frames. Follow two sections that explain in detail the two vertex types.

3.2.1 Macro

The macro type defines a parametric fragment of text, with as the main attribute the text of the macro and the parameters used in the macro.
The text of the macro is defined in the f-string format, that is a particular format defined by [23], where a string is defined by affixing a pair of curly brackets around each parameter, which are specified as keyword arguments. When a macro is decoded, its text string is parsed by replacing the parameters keyword with the actual parameters' values, thus returning the complete string.
The parameters, employed inside the macros, are the smallest building block of the type system; it is thanks to them that the macros provide so much expressiveness and dynamism. Each parameter provides a value along with the specification of the type of parameter. In this way, a parameter can be mutated, through a mutation operator, and can provide a new value that complies with the parameter's specifics. To cover as many cases as possible, a series of parameter classes is provided:

- **Integer parameter** provides an integer number in the half-open range [min, max)
- **Float parameter** provides a float number in the half-open range [min, max)

- **Choice parameter** provides an element from a fixed list of alternatives
- **Array parameter** provides a fixed-length array of symbols
- **Counter parameter** provides a value that automatically updates at each mutation
- **Local reference parameter** provides a reference to a leaf node in the same tree of the macro in which is used
- **Global reference parameter** provides a reference to a leaf node in another tree, to respect the ones in which the macro is used

Considering that the main use case, of Byron, is that of a fuzzer for assembly programs, the definitions provided above can be reevaluated in this context. Thus, each macro can be seen as a parametric fragment of application code, which can be one or more lines of code, composed of a fixed part of text and one or more parameters; thus, each parameter could be the name of a variable, the name of a function, a number, an array, or everything else that can be depicted with one of the parameter classes presented above.
For instance, an addition instruction between a register and an immediate can be encoded in the following macro:

```
addi {r_out}, {r_in}, {imm}
```

In the f-string representation of the macro shown above, there is a fixed fragment of text, that comprises the name of the instruction "addi" and the commas, plus three parameters; the first two could be two choice parameters, that have as potential set of value the registers' names, while the third could be an integer parameter between 0 and the maximum integer representable on the specific architecture. Given hypothetical values of r_out = "r1", r_in = "r2", imm = 10 the decoded version of the macro above became:

```
addi r1, r2, 10
```

As seen previously, in the typed Directed Multi-Graph representation of the individual there may be edges of type references, that connect a pair of leaf nodes together; to define these, relationship parameters of type local reference or global reference are employed, based on whether the target node is on the same structural tree. For instance, a call to a subroutine, that is in a different tree, can be represented as follows:

```
call {target}
```

Where “target” is a global reference parameter that denotes the target node, with which the subroutine will begin its execution.

3.2.2 Frame

The second type of vertex available is the frame, in contrast to the type macro the frames are only structural nodes and are not decoded into any code; frames can be seen as empty containers, whose main purpose is to organize the hierarchical relationships between the nodes and enforce constraints. There are different kinds of frames, already predefined, each of which organizes its successors in a different way. The following types of frames are provided:

- **Macro bunch** defines a vertex that can have a variable number of successor vertices of type macro. The macro bunch can be constrained by a minimum and maximum number of successors, that is specified during initialization; always during the initialization a set of macros is specified, from which the successor nodes are selected. In the context of code generation, the macro bunch could, for example, be used to represent a sequence of instructions.
- **Frame sequence** defines a vertex that has a fixed number of successors that can be of type macro or frame. The successor nodes are orderly selected from a list that is defined during the initialization. In the context of code generation, the frame sequence could, for example, be used to represent a subroutine; where a frame sequence has as successors a macro node, that contains the header, followed by a macro bunch node, that indexes the body, in turn, followed by a macro node, that contains the return statement.
- **Frame alternative** defines a vertex that has a single successor which can be of type macro or frame, and is randomly selected from a set of valid types, provided during the initialization process.

The internal representation of the individual, as a typed Directed Multi-Graph, is therefore composed of two kinds of nodes, the macro nodes which are in the leaf

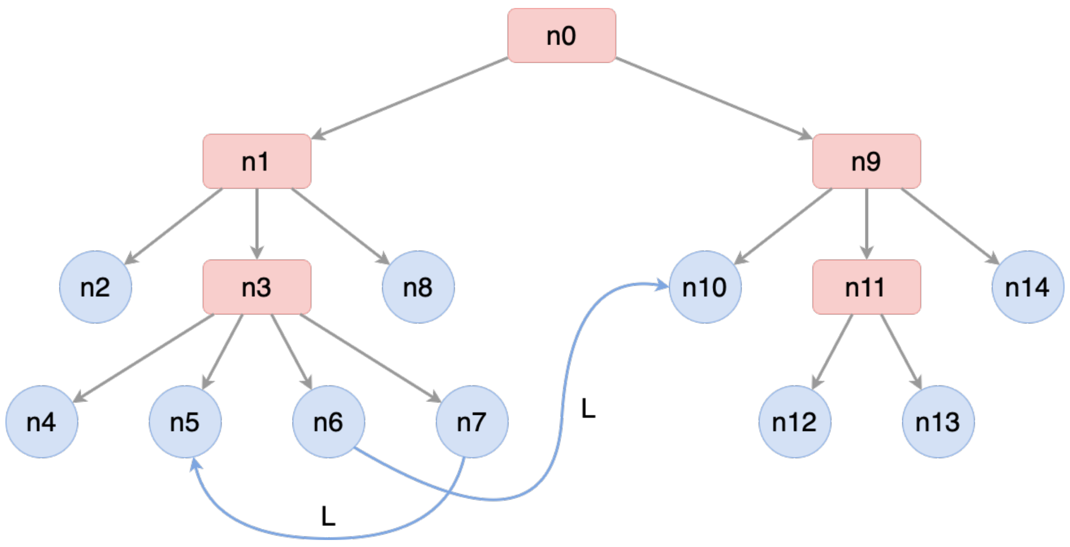


Fig. 3.1 Internal representation of a candidate solution as a multigraph. Square vertexes are frames, round vertexes are macros.

nodes, of the trees composing the forest, and the frame nodes which are all the intermediate ones. They are all connected by structural edges, plus eventually some reference edges, that connect pairs of leaf nodes. An individual genome example can be seen in Figure 3.1, where the round nodes are of type macro while the square ones are of type frame.

3.3 Evaluator

During the evaluation process, to calculate the fitness of each individual, its genotype, represented as a graph, as shown in Figure 3.1, must be firstly decoded in an evaluable phenotype representation; this is done by linearizing the genotype of the individual into a text. The linearization process consists of visiting the typed Directed Multi-Graph, starting from the root node of each tree in the forest, with a depth-first approach, and for each leaf node encountered, the macro contained in it is decoded and converted into its string representations; by that, a text with the program code, that was decoded inside the genome, is obtained. The linearization process transforms the genome in a representation that resembles that of an individual generated with Linear Genetic Programming [24], as shown in Figure 3.2; despite the importance in the evaluation of fitness, the linear representations lose most of the information regarding the internal structure of the genotype and its constraints.

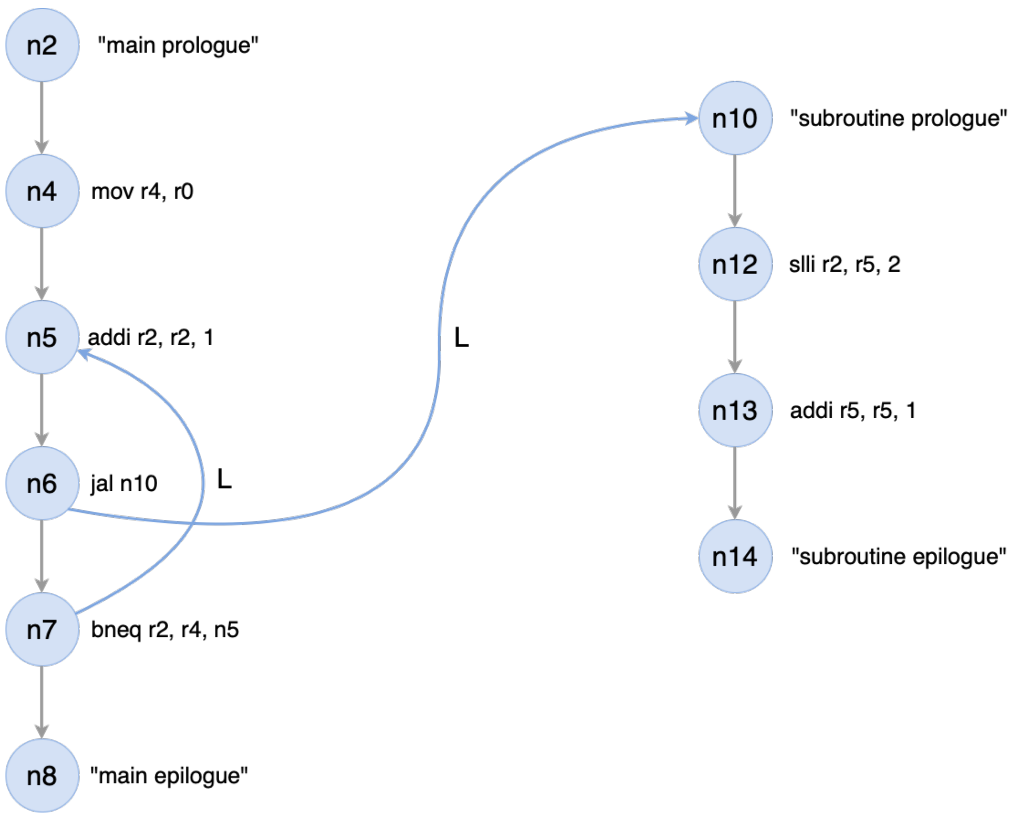


Fig. 3.2 Candidate solution, of the individual represented in Figure 3.1, seen as a linear GP representation.

Multiple evaluation solutions are then provided to allow the needed flexibility to support various types of languages and industrial requirements; thus, to obtain the texts of the programs, that were encoded in the individual genome, through the linearization process, the following evaluation solutions are provided:

- **Python evaluator** it uses a Python function, with the phenotype text as parameter, to calculate the fitness values of individuals. Support three execution methods:
  - **Sequential** execute the evaluation of all the individuals sequentially.
  - **Thread-based parallelization** executes the concurrent evaluation of multiple functions in different threads; it is useful if the evaluation process accesses to external resources given the multithreading limitations of Python.

  - **Process-based parallelization** executes the concurrent evaluation of multiple functions in different processes; it is useful if the evaluation process is computationally intensive because despite not having the multithreading limitations of Python, the generation of multiple processes introduces additional overhead.

- **Makefile evaluator** uses an external Makefile to perform the evaluation; it initially creates a temporary directory for each individual, where it dumps the phenotype of the individual, the Makefile, and every necessary file, then runs the "make" command and obtains the fitness result as standard output. Provide a thread-based parallelization approach, without incurring the previous limitations.

- **Script evaluator** uses an external shell script to perform the evaluation of each individual sequentially.

- **Parallel script evaluator** uses an external shell script to perform the evaluation of each individual, but implements a thread-based approach to the parallelism similar to the one used by the Makefile evaluator.

3.4 Variation Operators

Byron, being based on GGP, provides some of the standard variation of operators used when the representation of the genome is a graph. Since that, there is an added level of customization over the graph representation, that is due to the type system, the provided operators are adapted to handle it. Follow a list of the current available variation operators in Byron, divided based on their arity:

- **Mutation operators (have a variable strength, based on the self-adaptation results):**

  - **Single parameter mutation** generates a new individual by randomly modifying a single, randomly chosen, parameter of a random vertex of the parent.

  - **Single-element array parameter mutation** generates a new individual by randomly modifying a variable number of values, randomly chosen, inside an array parameter of a random vertex of the parent. The number of values of the array modified is proportional to the mutation strength.

  - **Add macro to bunch mutation** generates a new individual by adding a random macro as successors of a randomly selected macro bunch of the parent individual. The new random macro is selected from the pool, defined by the constraints of the macro bunch, if some macros are not yet successors of the macro bunch; else the new macro is selected by the list, ordered by frequency, of the successors.
  - **Remove macro from bunch mutation** generates a new individual by removing a random macro from the successors of a randomly selected macro bunch of the parent.

- **Crossover operators:**
  - **Array parameters uniform crossover** generates a new individual by performing a uniform crossover between two randomly selected compatible arrays inside the two different parents.
  - **Node crossover choosy** generates a new individual by swapping randomly selected compatible nodes between the parents. The nodes to be considered compatible must have exactly the same path type, meaning they must have as ancestors the same types of nodes.
  - **Node crossover unfussy** generates a new individual by swapping randomly selected compatible nodes between the parents. The nodes to be considered compatible must be of the same type.
  - **Leaf crossover unfussy** generates a new individual by swapping compatible leaves. The nodes to be considered compatible must be of the same type.

3.5 Self Adaptation

Self-adaptation refers to the process where the algorithm dynamically tweaks its own parameters during the optimization process. This capability is important because it can significantly enhance the performance and efficiency of the algorithm, across a wide range of problems. Byron exploits this concept in two different ways by dynamically modifying the operator strength and preferring the best-performing operators; A more detailed explanation of the two contexts is provided in the following sections.

3.5.1 Operators' Strength

Byron employed a self-adaptation mechanism to manage the strength of the mutation operators, as it was introduced in section 3.4. The strength of the mutation operators affects how intrusive a mutation is and, in this way, affects the balance between exploration and exploitation; high strength pushes toward exploration, while lower ones toward exploitation.
To determine the strength of the operator the temperature parameter is used. The temperature parameter determines, in case of a fitness target, the proximity of the actual population from that target; by doing so the temperature is used to tweak the strength of the operators.

3.5.2 Operators Selection

Given the huge flexibility offered by Byron, according to the problem, different genetic operators will have different performances. To improve the efficiency of the system, a mechanism has been implemented to keep track of best-performing operators, and thus use the worst-performing ones more sparingly than the best-performing ones.
Firstly, a mechanism to keep track of the performance of each operator should be devised; Byron implements this through the individual class, which after updating the fitness of the individual, assigns to the operator, that has generated the individual, some statistic. Therefore, each operator keeps track of:

- Number of times it is employed.
- Number of times it failed, that is when no valid offspring is generated.
- Number of times a valid offspring is created.
- Number of times the newly created individual has a better fitness than at least one parent.

- Number of times the newly created individual has a worse fitness than both parents.

To assign a metric to the operators' performance a reward approach is employed, where each operator receives two types of rewards; the first one is assigned each time a newly created individual has a better fitness than at least one parent while the second one is assigned each time a valid offspring is created. In this way, the best-performing operators for the given task, are rewarded. Should be noted that no penalization is used when an operator doesn't generate valid offspring, or the newly created individual has a worse fitness than their parent. Since the rewards are directly correlated to a potential penalty, they can be seen as a form of penalization per se. Having a metric for the operators' performance, the concept of regret is introduced, following the work of [25]. Therefore, can be calculated the best mean reward $\mu*$ as the max between the mean rewards $\mu(o_p)$ of all the available operators; now, assuming to randomly select T times an operator, among all the available ones, the regret at T time can be calculated as follow:

$$R(T) = \bar{\mu} * T - \sum_{t=1}^{T} o_p(t)$$

This allows to make a comparison between the cumulative reward earned by the algorithm and the reward that would be achieved using an ideal optimal strategy. By default, Byron selects the operators randomly, considering all operators independent and identically distributed; by employing the self-adaptation approach, through the Successive Elimination Algorithm [26], Byron can deactivate the underperforming operators. This algorithm minimizes the selection of operators that could be not suited to the structure of the genome in question, thereby reducing the computational time wasted on creating invalid individuals or individuals that underperform their parents. Should be noted that during the evolutionary process, an operator that was initially discarded could become more successful; for this reason, at every quarter of execution, all the operators are tested to evaluate their performances.

3.6 Survivor Selection

The default survivor selection approach employed by Byron is that of a $(\mu + \lambda)$ selection strategy, also called steady state, which is the default approach used with

GGP; where the next generation of individuals is selected from the combined pool of μ parents and λ offspring. The best μ individuals, ranked by their fitness, are chosen as survivors.
Nevertheless, also the aging selection strategy can be applied. Aging refers to the process of introducing a form of lifespan or age to the individuals in the population. This method aims to enhance the diversity of the individuals, by preventing stagnation and maintaining a dynamic population. In Byron, when aging is active, all individuals die at age AT, except the best ones AB that do not age. Specifically, by tuning AT and AB various selection strategies can be employed:

- $AT = \infty$ or $AB = \infty$, pure steady state approach $(\mu + \lambda)$
- $AT = 1$ and $AB = 0$, pure generational approach (μ, λ)
- $AT = 1$ and $AB = 1$, elitist strategy.

3.7 Parent Selection

The default parent selection approach employed by Byron is that of tournament selection, which allows the control of the selective pressure by adjusting the tournament size and it is more efficient in a parallelized context because it does not need to make access to the entire population. Tournament selection involves running "tournaments" among a predefined number of individuals, chosen randomly from the population, and selecting the best individual from each tournament to be part of the group of parents who will be in charge of restocking.

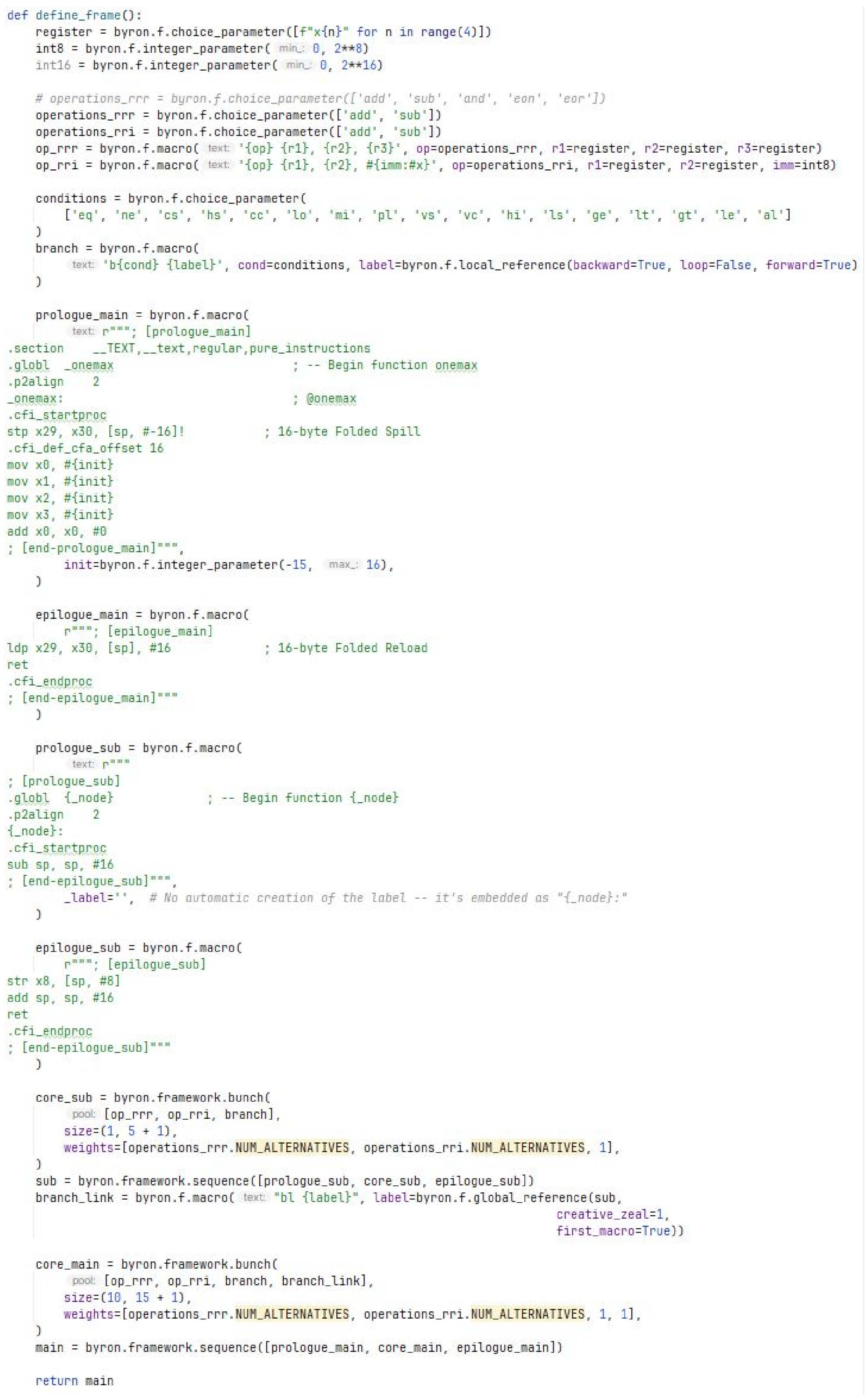
Chapter 4

Experimental Evaluation

Since Byron is still in the alpha stage and in active development, it has not been possible to use it in any commercial projects yet. Nevertheless, it already presents a vanilla and an adaptive Evolutionary Algorithm, that allows the execution of some benchmarks, based on the classic One Max problem, details of which are available on the GitHub repository used for its development.

4.1 Top Frame

Based on the programming language used to run the benchmarks the related top frame should be defined as introduced in section 3.1. The top frame is a collection of all the rules and constraints of a programming language that is mapped into the type system provided by Byron. Since the top frame is employed in each individual initialization, Byron can perform its optimization process with maximum flexibility; by just changing the top frame definition the programming language, with which Byron optimizes the problem, or the constraints over that programming language can be changed. In Figure 4.1 is shown an example of the definition of the top frame used with 64-bit ARM assembly.



```python
def define_frame():
    register = byron.f.choice_parameter([f"x{n}" for n in range(4)])
    int8 = byron.f.integer_parameter( min_: 0, 2**8)
    int16 = byron.f.integer_parameter( min_: 0, 2**16)

    # operations_rrr = byron.f.choice_parameter(['add', 'sub', 'and', 'eon', 'eor'])
    operations_rrr = byron.f.choice_parameter(['add', 'sub'])
    operations_rri = byron.f.choice_parameter(['add', 'sub'])
    op_rrr = byron.f.macro( text: '{op} {r1}, {r2}, {r3}', op=operations_rrr, r1=register, r2=register, r3=register)
    op_rri = byron.f.macro( text: '{op} {r1}, {r2}, #{imm:#x}', op=operations_rri, r1=register, r2=register, imm=int8)

    conditions = byron.f.choice_parameter(
        ['eq', 'ne', 'cs', 'hs', 'cc', 'lo', 'mi', 'pl', 'vs', 'vc', 'hi', 'ls', 'ge', 'lt', 'gt', 'le', 'al']
    )
    branch = byron.f.macro(
         text: 'b{cond} {label}', cond=conditions, label=byron.f.local_reference(backward=True, loop=False, forward=True)
    )

    prologue_main = byron.f.macro(
         text: r"""; [prologue_main]
.section    __TEXT,__text,regular,pure_instructions
.globl  _onemax                         ; -- Begin function onemax
.p2align    2
_onemax:                                ; @onemax
.cfi_startproc
stp x29, x30, [sp, #-16]!           ; 16-byte Folded Spill
.cfi_def_cfa_offset 16
mov x0, #{init}
mov x1, #{init}
mov x2, #{init}
mov x3, #{init}
add x0, x0, #0
; [end-prologue_main]""",
        init=byron.f.integer_parameter(-15,  max_: 16),
    )

    epilogue_main = byron.f.macro(
        r"""; [epilogue_main]
ldp x29, x30, [sp], #16             ; 16-byte Folded Reload
ret
.cfi_endproc
; [end-epilogue_main]"""
    )

    prologue_sub = byron.f.macro(
         text: r"""
; [prologue_sub]
.globl  {_node}             ; -- Begin function {_node}
.p2align    2
{_node}:
.cfi_startproc
sub sp, sp, #16
; [end-epilogue_sub]""",
        _label='',  # No automatic creation of the label -- it's embedded as "{_node}:"
    )

    epilogue_sub = byron.f.macro(
        r"""; [epilogue_sub]
str x8, [sp, #8]
add sp, sp, #16
ret
.cfi_endproc
; [end-epilogue_sub]"""
    )

    core_sub = byron.framework.bunch(
         pool: [op_rrr, op_rri, branch],
        size=(1, 5 + 1),
        weights=[operations_rrr.NUM_ALTERNATIVES, operations_rri.NUM_ALTERNATIVES, 1],
    )
    sub = byron.framework.sequence([prologue_sub, core_sub, epilogue_sub])
    branch_link = byron.f.macro( text: "bl {label}", label=byron.f.global_reference(sub,
                                                                                   creative_zeal=1,
                                                                                   first_macro=True))

    core_main = byron.framework.bunch(
         pool: [op_rrr, op_rri, branch, branch_link],
        size=(10, 15 + 1),
        weights=[operations_rrr.NUM_ALTERNATIVES, operations_rri.NUM_ALTERNATIVES, 1, 1],
    )
    main = byron.framework.sequence([prologue_main, core_main, epilogue_main])

    return main
```

Fig. 4.1 The definition of the top frame for ARM64

Since one of the main objectives of this thesis is the application of the RISC-V ISA to Byron, a custom library of classes was implemented; such library allows an easy exploitation of the RISC-V ISA, thanks to a builder class that allows adding the following main options:

- Choosing what register to utilize and how to initialize them.
- Choosing between the main RISC-V ISA bases.
- Choosing between the main RISC-V ISA extensions.
- Defining custom RISC-V ISA bases, that are usually a subset of a standard base.
- Defining custom RISC-V ISA extensions.

Thanks to that library, various configurations of the RISC-V ISA can be rapidly and easily tested; moreover, the library automatically deals with the creation of the top frame, by dynamically defining the framework rules, based on the selected configuration.

4.2 Experiments

The One Max benchmark has been used with both the vanilla and the adaptive EA to test the performance of Byron with various programming languages. The One Max problem is a classic optimization problem in the field of EA. The objective is to maximize the number of 1s in a binary string of a fixed length.
Follow a list of the various tests that have been carried out:

- **Classical One Max:** Two variants of the classical One Max problem, with the individual defined as a frame of type macro bunch with exactly N macros, each one with a single parameter encoding 1 bit; alternatively, with exactly N/m macros, each one with a single parameter encoding an array of m bits. Individuals are evaluated by a pure Python function, with the best possible individual being 1, 1, ..., 1.

- **Assembly One Max (desktop):** Byron is being asked to generate an assembly program that sets all the bits in a given register to 1; the individuals are then evaluated on the host computer by assembling the code, linking it, and finally running it obtaining as standard output the value contained in the evaluated register. Two top frames are provided to run on the 64-bit version of the x86 assembly and the newer 64-bit ARM assembly.

- **Assembly One Max (simulator):** As in the previous benchmark, Byron is asked to generate an assembly program that sets all the bits in a given register to 1. Individuals are evaluated by exploiting an architectural simulator of the RISC-V ISA.

- **One Max Go:** Byron is asked to generate a high-level Go function, employing sub-functions, global and local variables, that returns precisely an unsigned 64-bit integer with all bits set at 1. Individuals are evaluated by compiling and executing the code on the host computer.

The tool was successful in all these simple benchmarks, being able to find the optimal solution in all of them. In that case, since the simple nature of the task, performance was not important; the main purpose of these tests was to demonstrate that Byron can handle all the following:

- Handle different structures, from a flat string of bits to a complex high-level program.

- Handle different types of assembly languages, IA64, ARM64, and RISC-V

- Handle different types of evaluators, from a Python function to an external simulator, on different operating systems.

Chapter 5

Conclusions

This thesis discusses the process behind the development of Byron, a GP-based tool designed to be an efficient assembly-language fuzzer; that can be used for test-programs generation in industrial settings, mostly by electronic engineers during the design, verification, and test phase of devices such as microprocessors and microcontrollers. At the current development stage, Byron can optimally perform on sector benchmarks, thanks to the joint work of all its main building blocks, that are already implemented.

Furthermore, this thesis has shown that Byron can perform fuzzing of complex test-program written in assembly language for the RISC-V ISA, through the use of an external simulator of the aforementioned architecture.

To conclude, has been demonstrated the effectiveness and flexibility that Byron, an extensible, self-adaptive, general-purpose Evolutionary Fuzzer, can provide, regardless of the language and the evaluators that are employed, thereby allowing to extend the results obtained to any general program.

References

[1] C. Darwin. *The Origin of Species*. John Murray, 1859.

[2] K.A. De Jong. *An Analysis of the Behaviour of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan, 1975.

[3] J.H. Holland. *Genetic algorithms and the optimal allocation of trials*. SIAM J. of Computing, 1973.

[4] J.H. Holland. *Adaption in Natural and Artificial Systems*. MIT Press, Cambridge, MA, 1992.

[5] I. Rechenberg. *Evolutionstrategie: Optimierung technisher Systeme nach Prinzipien des biologischen Evolution*. Frommann-Hollboog Verlag, Stuttgart, 1973.

[6] H.-P. Schwefel. *Evolution and Optimum Seeking*. Wiley, New York, 1995.

[7] R.E. Keller W. Banzhaf, P. Nordin and F.D. Francone. *Genetic Programming: An Introduction*. Morgan Kaufmann, San Francisco, 1998.

[8] J.R. Koza. *Genetic Programming*. MIT Press, Cambridge, MA, 1992.

[9] J.R. Koza. *Genetic Programming II*. MIT Press, Cambridge, MA, 1994.

[10] J.E. Smith A.E. Eiben. *Introduction to Evolutionary Computing*. Springer, 2015.

[11] Rainer Storn. *Differential evolution—a simple and efficient scheme for global optimization over continuous spaces*. International Computer Science Institute, 1995.

[12] R. Storn. *Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces*. Journal of Global Optimization, 1997.

[13] R. Storn. *On the usage of differential evolution for function optimization*. Biennial Conference of the North American Fuzzy Information Processing Society, 1996.

[14] Andrew Waterman. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V Foundation, 2019.

[15] Abhik Marcel Boehme, Cristian Cadar. *Fuzzing: Challenges and Reflections*. IEEE Software 38, 2021.

[16] Jinxin Ma Chen Chen, Baojiang Cui. *A Systematic Review of Fuzzing Techniques*. Computers Security 75, 2018.

[17] Chao Zhang Jun Li, Bodong Zhao. *Fuzzing: A Survey*. Cybersecurity 1, 2018.

[18] Xiaodong Jia Hongliang Liang, Xiaoxiao Pei. *Fuzzing: State of the Art*. IEEE Transactions on Reliability 67, 2018.

[19] Seyit Camtepe Xiaogang Zhu, Sheng Wen. *Fuzzing: A Survey for Roadmap*. Comput. Surveys 54, 2022.

[20] Thomas Vogel Martin Eberlein, Yannic Noller and Lars Grunske. *Evolutionary Grammar-Based Fuzzing*. Springer International Publishing, 2020.

[21] Massimiliano Schillaci Ernesto Sanchez and Giovanni Squillero. *Evolutionary Optimization: The uGP Toolkit*. Springer Science Business Media, 2011.

[22] https://github.com/cad-polito-it/byron.

[23] https://docs.python.org/3/library/string.htmlformatspec.

[24] *Linear Genetic Programming*. Springer US, Boston, 2007.

[25] Aleksandrs Slivkins et al. *Foundations and Trends in Machine Learning 12*. 2019.

[26] Shie Mannor Eyal Even-Dar and Yishay Mansour. *PAC bounds for multi-armed bandit and Markov decision processes*. Springer, 2002.