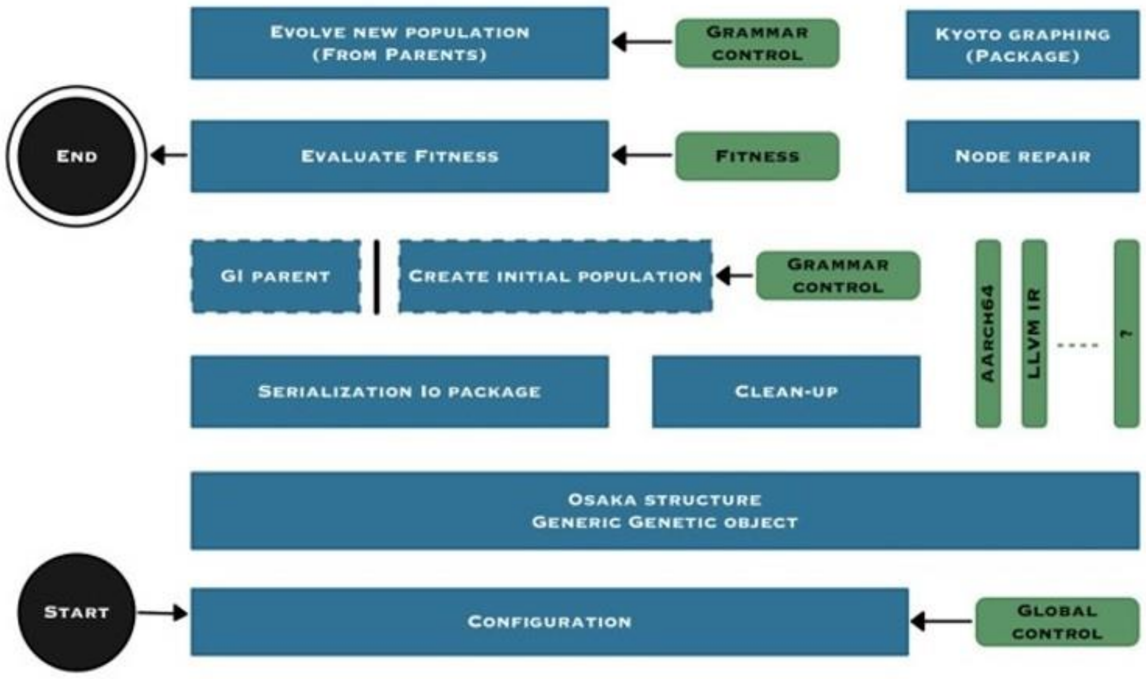
Project Shackleton

Synopsis

- **Project Shackleton** goal is to build a modular designed Linear Genetic Programming (LGP) framework to automate the discovery of optimal software solutions for hard low-level software optimization problems. Idea to is a apply LGP to a Generic Genetic Object (GGO). The GGO by default is set to AArch64 instructions but support is in place for LLVM Intermediate Code.
- Once complete the Shackleton framework can be used to explore any low-level software optimization problem if the optimal answer can be defined either as an objective or multi-objective (pareto curve) with constraints. The framework itself works with GGO's.
- Challenges revolve around interfacing, variable setting, define goal/s, testing environment and inserting constraints. Also, a complicated part is understanding the methods of outputting correct LLVM IR code or simply even outputting AArch64 code directly from the GGO.
- A dynamic method of playing with the LGP algorithm variables i.e. population size, maximum generations, recombination %, mutation %, output size, variable constraints, etc. would be advantageous since this would allow more complicated search models could be adopted.



Description

- Shackleton is designed to be highly adaptive framework targeting AArch64, LLVM IR, …
- Using population-based optimization and exploration
- Frameworks is designed to speed up automated software production
- Either taking existing code (GI) or creating entirely new code (GP)
- Broken into a module design with specific methods of communication
- Each module can be developed independently providing the I/O rules are adhered-to
- Designed to be run in hosted (x86) or native, in continuous or fixed fashion
- Optimized purposely as for ARM only hardware

Genetic Programming

Genetic Programming (GP) by contrast to Genetic Algorithms (GA) manipulates structures and in-particular programs that are executable or mathematical equations i.e. symbolic regression. Traditionally GP used a tree-based representation and used the LISP programming language, in more recent times Python is extremely popular. Relies more on crossover than mutation as the method of evolution. GPs handle manipulating structures so any problems which is structurally based can be tackled by this algorithm i.e. linear, tree and direct graph. GPs can produce original source code and in fact can find new novel solutions to any structural style problems. In industry GP to mostly used to discover best fit mathematical equations.

Genetic Improvement

Genetic Improvement (GI) is a subclass of GP (section 4.3), where instead of a random initial seeded population, a working program is inserted as the starting point to spawn the children entities of the first population. This is a powerful concept since it does not only search for a better optimized solution but also has the potential to discover and correct faults in the original" working" code. Solves an interesting problem, where either the working code is potentially un-optimized, and a more optimized version is required or bringing legacy code up to current standards.

Linear Genetic Programming

Linear Genetic Programming (LGP) is a subclass of GP and as the name implies uses a linear structure representation. The linear structure has some advantages over the more complicated tree or directed-graph structures. LGP is particularly useful for problems which are more sequential. For example, optimizing low level assembly output. It also makes the problem of manipulating complex structures easier since it is a linear flow that is being evolved. Constructs like if-style control flow or loops are superimposed onto the linear structure.

LPG solves problems that are sequential. This is useful for optimizing programs and low-level assembly style output. Or any problem-domain where the problem being explored is about sequential ordering.

Implementation

The Osaka Structure

Our main vehicle for enabling more generics within the framework, the Osaka structure is a specialized doubly linked list for which all nodes are wrappers for internal node types that vary based on the application. There are some node types that are built into the base structure, such as “osaka_string” and “simple”, but users are also given the option to create their own types (discussed in the editor tool later).

Editor tool

Built into the framework itself, this folder contains a separate tool from the rest of Shackleton that is aimed at eliminating much of the headaches of adding new object types to the Shackleton framework. These new object types are automatically generated by running the tool with a json string input file and after that fact can be used as internal types within the Osaka structure.

This tool makes the following changes to the structure of Shackleton:

- Creates a new autogen_.h file where is the name of the new object type as specified by the user. All .h module files are built off of the simple.h template with added object items and macros based on user input
- Creates a new autogen_.c file where is the name of the new object type as specified by the user. All .h module files are built off of the simple.c template and contain implementations of the methods described in autogen_.h
- Edits shackleton/osaka/osaka.h by adding a new entry in the osaka_object_type enum declaration. This gives the item a name and a number which must match that which appears in module/modules.c
- Edits shackleton/modules.h to add a new item to the object_table_function array along with new versions of all osaka object methods. This tool also increased the MAXTYPE variable by 1 to account for the new type
- Edits shackleton/module/modules.c to add a new include statement for the .h file created and described above
- Edits the shackleton/makefile by adding a new object declaration for autogen_.o and adding a new build statement for the .h and .c created and described above

This tool is a work in progress and may still have bugs to work out. It is recommended that a backup of all edited files listed above be created in the case that the autogenerated documents do not compile/make.

In order to add a new object type that has internal parameters, a new object type json file must be supplied. Here is an example of such a file:

```
{
       "object": {
           "name": "test_type"
       },
       "params": [
            {
                "name": "number",
                "type": "uint32_t",
                "macro_name": "my_number",
                "has_valid_values" : "false"
            },
            {
                "name": "binary",
                "type": "uint32_t",
                "macro_name": "my_binary",
                "has_valid_values" : "true",
                "values" : [
                    {
                        "value": 1
                    },
                    {
                        "value": 2
                    },
                    {
                        "value": 4
                    },
                    {
                        "value": 8
                    },
                    {
                        "value": 16
                    },
                    {
                        "value": 32
                    },
                    {
                        "value": 64
                    },
                    {
                        "value": 128
                    },
                    {
                        "value": 256
                    },
                    {
                        "value": 512
                    }
                ]
            }
        ]
}
```

The new parameters file should always have two top-most items: object and params. The object name category is an additional name that adds a layer of redundancy to avoid duplicate naming across files in the tool.

The "params" array is where the parameters themselves need to be described. Every parameter is required to have 4 objects inside of it:

- "name" - the name appended to the struct name that will be created

- "type" - the C compliant datatype. Right now, the approved datatypes are uint32_t, int, and char*
- "macro_name" - the name of a macro that will be used to access the value of the struct created
- "has_valid_values" - a string field that must either be "true" or "false". If "false", then the struct created will not enforce what values can be put into that variable If "true", then you are required to create an addition member of the param called "values" which is an array of valid values.

The example shown above is an example of a new object type that will create two new internal parameters and thus two new structs in the autogenerated files. One of the params shown does not have valid value constraints, while the other does, and this is further reflected in the absence and presence of the valid fields.

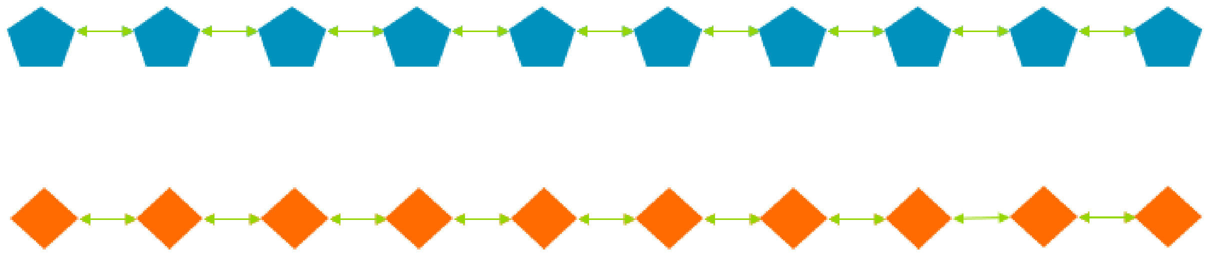
Evolution Operators

---- Crossover ----

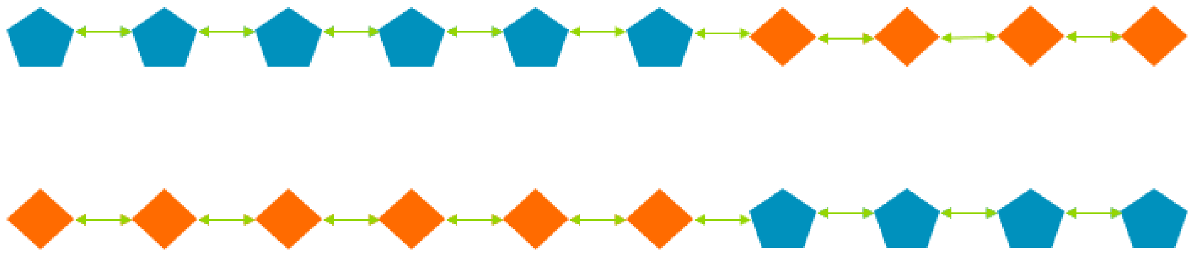
This application implements multiple versions of the crossover operator. Regardless of type, crossover takes in two individuals and combines their genetic material in order to create two new individuals from them.

Basic One-Point Crossover

Basic one-point crossover chooses only one point at which the two individuals will share their "genetic material". If the two individuals have the same length, then any point can be chosen. If one of them is longer than the other, then a point is chosen within the shorter of the two. Each individual is split into two sections at the point taken. Then, the first section of one individual is joined with the last section of the other to form two new individuals. For example, given these two individuals:



if we were to perform basic one-point crossover on them at point 7, the result would be these two individuals:

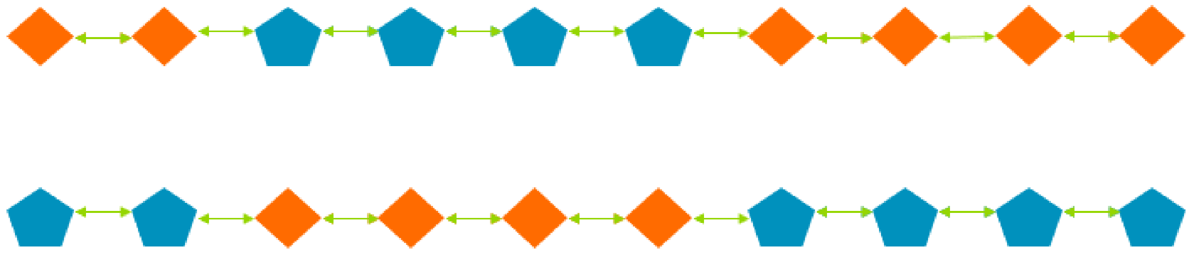


Basic Two-Point Crossover

Basic two-point crossover builds off of basic one-point crossover and simply performs basic one-point crossover twice on the same individuals. Basic two-point crossover does not guarantee that the two points chosen will be different values, so it is possible that basic one-point crossover will be applied at the same point twice, effectively leaving the individuals unchanged.

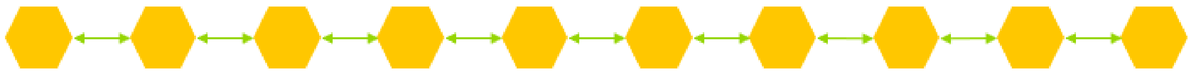
Two-Point Crossover Diff

This version of two-point crossover guarantees that the two points chose for crossover will be different points. This means that one each individual resulting from this operator will consist of the head and tail nodes of one individual, and some body nodes of the other. For instance, given the same individuals shown about for one-point crossover, if they were to have this two-point crossover performed at point 3 and at point 7, the result would be these new individuals:

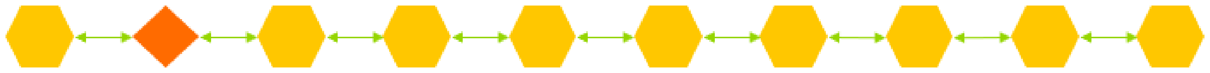


---- Mutation ----

Currently only one mutation operator in the Shackleton Framework. This mutation operator picks a single node from an Osaka structure and changes all parameters within that node with a uniform probability. If given an individual with this representation:



the application of the mutation operator once may result in an individual with this representation:



The actual nature of the mutation will depend on the internal unit being mutated. Some modules have attributes for which there are a finite number of valid values. If that is the case, then when going through mutation, that attribute will only take on a new value that exists in a known list of valid values. If the attribute does not have a finite list of valid values, then a completely random value of the appropriate type will be loaded to it. All attributes of the mutated node will change, but it is not guaranteed that the value will be different before and after the mutation.

---- Selection ----

Selection is the operator by which individuals from one generation are selected to be the parents of the next generation, to be mutated and crossed with one another and create new individuals. The main selection operator used in the Shackleton tool is tournament selection. For a k-way tournament, k individuals are chosen from the previous generation and the individual in that k size group with the highest fitness becomes a parent for the next generation. During each iteration of evolution, tournament selection is repeated until the final population size is at the target size.

---- Top-level Evolution ----

At the top-level of Shackleton, evolution utilizes all the tools that are described above. The Shackleton tool takes in the following parameters:

- generations : The maximum number of generations that will be created during the course of the evolutionary process.
- population_size : The target size of each population.
- crossover : The percent chance that crossover will occur for any given individual in the population. The final percent chance is treated as /100%
- mutation : The percent chance that mutation will occur for any given individual in the population The final percent chance is treated as /100%
- visualization: A flag that indicates if visualization of the process will be used. If the flag is present in the command then visualization will be used, it will not be used otherwise.

These parameters are set by the user and are passed to the respective operators that use them. There are defaults for each of them, so if the user does not specify an input then the defaults are used throughout the program.

Main Contributors

From within Arm, Andrew Sloss and Hannah Peeler. Progress completed in collaboration with Wolfgang Banzhaf and Yuan Yuan working out of Michigan State University, MSU.