**Project Shackleton**

**A large ship in the water

Description automatically generated**

**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.

A screenshot of a cell phone

Description automatically generated

**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.

**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.

**The File Structure**

Here is a look at the filesystem of Shackleton, in tree form:

As is apparent from the tree structure shown above, the bulk of the source code of the Shackleton Framework is contained within the src/ directory. The root directory serves as a home for several subdirectories that are further discussed in this section.

There are only 2 files in the root directory of note: the main.c file and the makefile. The makefile builds all code necessary for the Shackleton Framework to be run. The main.c file is the top-level file through which the tool is run, managing the command-line interface of the tool and making the evolutionary process more interactive and understandable for the user.

**bin – Resulting files from make**

This directory contains only files resulting from the build process of source.

**docs – Descriptive and instructional documents**

This file is contained within this folder. This directory holds any instructional documents. Much of the material found in these documents can also be found in the READMEs for subdirectories of src and other directories.

**img – Images used in top README**

This directory contains images that are used in the top-level README for the Shackleton framework. These files are not used by any of the code in the src directory.

**obj – Object (.o) files from the build process**

All object files created during the make process are stored in this directory.

**src – All source files for the Shackleton Framework**

**osaka – The Osaka Structure**

The Osaka Structure is the main object type used in the Shackleton Framework. It is a doubly linked list of node\_str struct objects. Each Osaka Structure represents what we refer to as an “individual” in the world of Shackleton. All of the node\_str objects taken in order represent a single individual’s “genotype”, their sequential makeup that can be deserialized into some piece of code or a series of choices.

Here are the datatypes included in the Osaka Structure (written in C language):

typedef enum {

EMPTY = 0,

ISOLATE = 1,

HEADNODE = 2,

TAILNODE = 3,

INTERMEDIATE = 4

} osaka\_node\_typ;

typedef enum {

NOTSET = -1,

SIMPLE = 0,

ASSEMBLER = 1,

OSAKA\_STRING = 2,

    AUTOGEN\_TEST = 3,

LLVM\_PASS = 4,

    BINARY\_UP\_TO\_512 = 5

} osaka\_object\_typ;

typedef enum {

HEAD = 1,

TAIL = 2,

} append\_typ;

typedef struct node\_str {

uint32\_t uid; // a universal identifier

osaka\_object\_typ objtype; // determined type of object\_ptr

void \*object\_ptr; // pointer to the “internal” object

struct node\_str \*last\_ptr; // pointer to prev node in Osaka structure

struct node\_str \*next\_ptr; // pointer to next node in Osaka structure

} node\_str;

The osaka\_node\_typ datatype is an enum that is a helper for internal methods of the Osaka structure. This datatype is helpful for manipulating the doubly linked list structure

The osaka\_object\_typ datatype is another enum, however this enum helps to distinguish how a single node\_str object should be interpreted. The values of osaka\_object\_typ are the valid object types that can be pointed to by a node\_str object through the object\_ptr. The different build-in types will be explained in more detail in the “Modules” section of this document. Each node\_str serves as a wrapper, contained the actual value of the node within the object\_ptr object.

The append\_typ datatype is a helper enum for method that add node\_str objects to a new or pre-existing Osaka structure. A user had the option of specifying if a node should be added with an offset from the head of the structure or from the tail.

Finally, the node\_str datatype is the heart of the doubly linked list represented here. As such, each node\_str object has a pointer to the previous(last) node in the list and a pointer to the next node in the list. Each node\_str also has a unique universal identifier in the uid. During any run of the Shackleton framework, no two node\_str objects will have the same uid value. Each node\_str also had an objtype value that indicates the type of object that is pointed to by the object\_ptr value. This object\_ptr is the main object that contains the real “value” of each node\_str object.

There is a collection of macros that allows for accessing the different values in a node\_str object:

#define NEXT(s) s->next\_ptr

#define LAST(s) s->last\_ptr

#define OBJECT(s) s->object\_ptr

#define OBJECT\_TYPE(s) s->objtype

#define UID(s) s->uid

There is a one-to-one correspondence between the number of macros and the number of values in the node\_str object.

**module – All the object types legal within the Shackleton framework**

The “modules” section of the project is the section in which the different object types and their implementations are housed. For each Osaka\_object\_typ value in Osaka.h, there is a corresponding type represented by a .c file and a .h file in the module directory. Within those files are implementations for the following methods:

* osaka\_createobject
* osaka\_randomizeobject
* osaka\_printobject
* osaka\_deleteobject
* osaka\_writeobject
* osaka\_readobject
* osaka\_copyobject

For each individual module, all of the method names will have the “osaka” replaced by the \_ separated name of that object type (such as osaka\_string, or lvvm\_pass). Module.h contains the following struct and externs:

typedef struct object\_functions\_str {

uint32\_t object\_type;

char name[25];

object\_simple\_str \*(\*osaka\_createobject)(void);

void \*(\*osaka\_randomizeobject)(void \*);

void (\*osaka\_deleteobject)(void \*);

void (\*osaka\_printobject)(void \*);

void (\*osaka\_writeobject)(FILE \*stream,void \*);

void \*(\*osaka\_readobject)(FILE \*stream);

void \*(\*osaka\_copyobject)(void \*);

//uint32\_t (\*osaka\_fitnessobject)(node\_str \*);

} object\_functions\_str;

extern object\_functions\_str object\_table\_function[];

extern uint32\_t MAXTYPE;

Within module.c, all of the object types in the Shackleton framework have an instance of the object\_functions\_str struct with its specific information. These two files are the top-level through point for accessing the correct methods for each object type.

**evolution – Implementation of evolutionary operators**

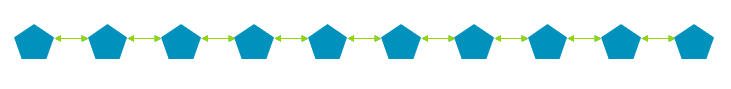
This directory contains the core of the implementation of the Shackleton Framework as a genetic programming/evolutionary algorithms framework. All evolutionary operators (mutation, crossover, selection, etc.) are contained within this directory. These files are referenced by the top-level evolution files: evolution.c and evolution.h. This section will not only discuss the organization of the evolution files, but also the implementation of the evolutionary operators contained therein and how the Osaka structure is manipulated using them.

**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.

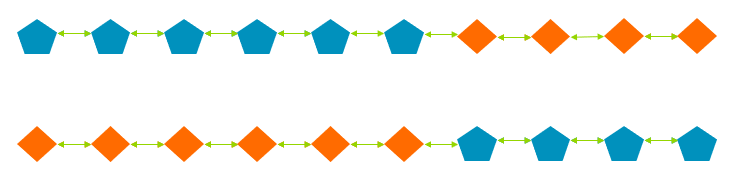
**Basic One-Point Crossover**

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:

A picture containing object

Description automatically generated

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

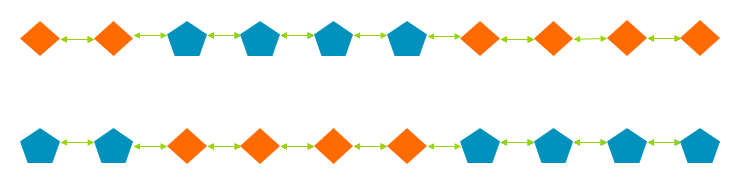


**Basic Two-Point Crossover**

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.

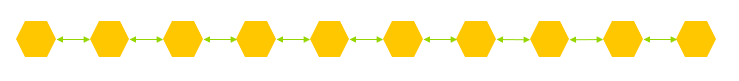
**Two-Point Crossover Diff**

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:

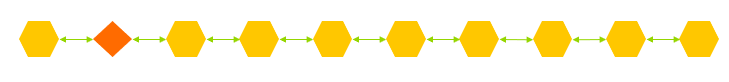


**---- Mutation ----**

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.

**files – Other files used by the tools**

Any files that will be needed by the top-level Shackleton tool must be housed in this directory or some subdirectory of this directory. Currently, the only established subdirectory in this folder is the llvm/ subdirectory that houses any code files used with the LLVM integration portion of the Shackleton framework. Any files that are automatically created due to the terminal commands created and run by the llvm.c and llvm.h files will be loaded here, so this subdirectory also acts as storage for these resulting files.

**support – Files that contain methods used by multiple other files**

This directory contains files that are used by many other files across the src directory. This is where the main testing methods can be found in test.c and test.h. These tests have coverage over the evolutionary operators detailed in the evolution section, as well as specific utility methods for llvm integration.

The utility.c and utility,h files contain commonly needed general methods, such as methods for converting int values to strings or processing a json string from a .json file into a cJSON object. These files are used by both the top-level Shackleton tool as well as the editor tool for adding new object types to the framework.

The cJSON.c and cJSON.h files are from the popular open-source c library for handling .json files in c code. These files are only used by the editor tool to handle the inputted json files for adding new object types. Json is not used in the top-level Shackleton tool itself.

The visualization .c and visualization.h files contain methods for visualizing individuals and generations during the evolution process. Each object type in the module section has their own printobject method that prints the internal values of note for that object. These files use those methods for entire Osaka structures to print a more human readable version of the object. Visualization is turned off by default for the Shackleton tool. Visualization can be enabled by setting the -visualization flag on start up of the tool.

Finally, there are the llvm.c and llvm.h files. These files are representative of the motivation to integrate the Shackleton tool with the open-source LLVM compiler framework. These files contain methods for both creating and running commands from the LLVM framework when given a filename from the user of the tool. The files must always be located in the files/llvm/ directory in order to be found by the llvm-related files.

**editor\_tool – Tool for adding new object types to Shackleton**

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/src/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/src/module/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/src/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.

**Future Work/ Additional Functionality in Progress**

This section outlines some of the ideas for implementing support for loop and conditional constructs in Shackleton. One major goal was to attempt to change the Osaka structure implementation as little as possible. There will be some changes required in order to implement these more complex constructs, but the aim is to make those changes as non-intrusive to the current implementation as possible.

My inspiration and intuition came from the way that DNA bunches up on and folds against itself naturally when representing certain proteins and other structures. What if we can that concept and represent these constructs in a similar way? The whole point is to allow these constructs to be represented in an intuitive way to a normal user, while still be unmistakably linear in its construction.

We could represent an if…then statement as something like this:

Which, once unwrapped would be represented like so:

In a similar vein, an if…then…else statement would look something like this:

And when unwrapped would be represented like so:

In the above visualization, nodes with a light green color represent regular nodes that will be executed as they are and are not explicitly linked to other nodes in any way outside of the overall doubly linked list structure. The nodes with the darker green color represent the nodes that are included in the structural elements of the if…then (…else) construct. These nodes would need some form of link to the other nodes they share the construct with.

For this to be implemented in the current Shackleton framework, each node would have an optional field that indicates a choice to be “linked” to another node. This linkage indicates that they are a part of the same if…then (…else) construct. It would almost be a phantom doubly linked list, one that only links things in the same construct together and does not include any other nodes. During evolution, linked nodes cannot be separated from one another and are not good candidates for mutation. However, segments of code within two nodes of any construct can freely be mutated and crossed over with any other segment. I think this had promise for limiting some of the “bad” changes that may arise from the evolutionary process as well.

Implementing the loop conditions in this way would require a relatively non-intrusive way to include these in the Osaka structure and allow for the structures to still be treated as a fully linear representation.