Plushi: An Embeddable, Language Agnostic, Push Interpreter

Edward Pantridge
MassMutual
epantridge@massmutual.com

Lee Spector
Hampshire College
lspector@hampshire.edu

ABSTRACT

In the Push programming language, all sequences of valid symbols with balanced parentheses are syntactically valid and execute without error. Push nonetheless supports arbitrary data types and is Turing complete. This combination of features makes it useful as the target language for program induction in genetic programming systems, for which it was developed, and potentially also in other program induction systems. Prior Push implementations have generally been designed for use only within the host language in which they were implemented, often in conjunction with a specific program induction system that is written in the same host language. In this paper we present Plushi, a modular, embeddable Push interpreter that is designed to interoperate with program induction systems written in any language.

CCS CONCEPTS

• **Software and its engineering → Automatic programming**; **Genetic programming**;

KEYWORDS

genetic programming; program synthesis;

ACM Reference Format:
Edward Pantridge and Lee Spector. 2018. Plushi: An Embeddable, Language Agnostic, Push Interpreter. In *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3205651.3208296

1 INTRODUCTION

The Push programming language operates on a virtual stack machine, with a separate stack for each data type. It was developed to serve as the language in which evolving programs are expressed in genetic programming systems, but it may have use in other forms of programming induction as well.

Push programs are parenthesized sequences of instructions and literals, which may also contain nested, parenthesized sequences of instructions and literals. Push implementations typically provide stacks for common data types such as integers, floating point numbers, boolean values, and strings, but they may also provide stacks for any other data types that the developers choose to include [14].

A special stack, called the execution stack (or just the exec stack), is used to manage program execution. When we wish to execute a program, we place it on the exec stack and then repeatedly pop and process the top item on the exec stack until the exec stack is empty, or until an execution step limit has been reached. At that point, result values can be taken from the interpreter's stacks.

When a parenthesized expression is processed, its outermost parentheses are removed and all of the items that they enclosed are pushed individually back onto the exec stack, with the last item in the expression pushed first. When a literal is processed, it is moved to the top of the stack that matches its type. When an instruction is processed, all of the arguments that it requires are popped from the stacks of the appropriate types. The instruction then pushes any resulting values onto the tops of the stacks of the appropriate types. If the stacks do not contain sufficient values of the needed types, then the instruction is a "no-op" and has no effect. Because the exec stack can itself be manipulated by instructions, it is easy to provide instructions that implement arbitrary control structures, including conditionals and loops [14].

One consequence of this design is that all sequences of valid symbols with balanced parentheses are syntactically valid Programs that will execute without error. This is true even though Push supports arbitrary data types and is Turing complete, which in most languages make syntax and semantic errors possible. This combination of features makes Push useful as the target language for program induction in genetic programming systems, for which it was developed, because it simplifies the random generation and variation of programs that may use arbitrary data and control structures. The same features may also make Push useful in other program induction systems.

Prior Push implementations have generally been designed for use only within the host language in which they were implemented, often in conjunction with a specific program induction system that is written in the same host language. In this paper we present Plushi, an embeddable, host-language agnostic Push interpreter that is designed to interoperate with program induction systems written in any language.

The goal of the Plushi project is to create a Push interpreter that can be interfaced with as many machine learning and artificial intelligence systems as possible. This will allow for easier comparison of search methods, and also allow more search methods to be applied to inductive program synthesis tasks that require the use of arbitrary data types and control structures [9].

The following section outlines the motivation for creating Plushi. The subsequent section explains how Plushi is used by machine learning systems and how synthesized Push programs can be deployed, for example to serve new predictions in data prediction applications. Finally, we describe the primary implementation details of Plushi and conclude with suggestions for future research and development.

ACM ISBN 978-1-4503-5764-7/18/07…$15.00
https://doi.org/10.1145/3205651.3208296

2 MOTIVATION FOR PLUSHI

There have been many push interpreters implemented in recent years. Most Push interpreters are tightly coupled with a particular genetic programming framework. These interpreters tend to support slightly different data types and instruction sets. These interpreters are also implemented in different host programming languages and generally rely on the host languages' built-in indexed data structures to store programs. For example, the Python implementation, `pyshgp`,[1] stores programs in a Python list while the Clojure implementation, `Clojush`,[2] stores programs in Clojure vectors. [10]

The consequences of these different implementations is that programs found by any given Push Genetic Programming system can only be run by the same framework which produced it. This made the productionalization of Push programs difficult. After a program had been found by the genetic programming system, all future calls to that program would require the use of the entire genetic programming system.

This called for the design of a standardized, serialized program representation. If Push programs could be represented in a standard text format that most programming languages are capable of producing, that would help arbitrary machine learning systems produce Push programs capable of being run by the same Push interpreter. Plushi chose to represent programs as JSON lists because of the ubiquity of the format [1].

Aside from a standardized program encoding, it was also beneficial to create a fully featured Push interpreter that is decoupled from any machine learning framework. The interpreter should be usable from as many environments as possible and be able to accept programs encoded in the standard way described above.

Plushi achieves these desired properties by running an HTTP server in a dedicated process that receives JSON encoded programs via POST requests. The server responds with the result of running the program. This allows many machine learning systems to be used in tandem with Plushi to attempt program synthesis tasks.

3 PLUSHI USAGE

There are many commonly used machine learning frameworks that implement a variety of search and optimization methods. These machine learning frameworks are implemented across a variety of programming languages. In the Python programming language alone there are a wide range of machine learning systems including: DEAP, TensorFlow, Keras, Sci-kit Learn, Spark ML and more.

These machine learning systems implement algorithms from various fields such as evolutionary computation and artificial neural networks. Given the success of Push in genetic programming systems, it may be that simply adding Plushi to some of the more mature evolutionary computation frameworks, such as HeuristicLab [3] and DEAP [4], could yield improved results over existing results achieved by genetic programming frameworks that synthesize Push programs [6].

[1]https://github.com/erp12/pyshgp
[2]https://github.com/lspector/Clojush

The Plushi system aims to provide an interpreter for the Push language that will enable users to leverage the diverse community of machine learning systems for the purposes of automatic programming.

In order to achieve an interface that can be used from as many environments as possible Plushi is implemented as an HTTP server. The Plushi server can be run locally, or on dedicated hardware. The user, or users, interact with the Plushi server through POST requests. The body of each POST request is assumed to be JSON which the Plushi server parses, and interprets in various ways described in section 4.1. The response Plushi gives to each POST request is also serialized in JSON format.

It is very common for programming languages to have built-in or library support for making POST requests. It is also very common for programming languages to support the reading and writing of JSON data. This implies that many systems which are commonly used for machine learning can now utilize Plushi in addition to existing machine learning frameworks in order to perform automatic programming tasks.

The Plushi server is written in Clojure, and is built as a standalone `.jar` artifact. This makes distribution and deployment on a wide range of computer systems trivial. Any device that has access to a JVM can use the Plushi server to execute Push programs [8].

3.1 Plushi Request Types

For every request sent to the Plushi server, the request body is parsed as JSON data. Each request requires that the `action` key be specified with a string for its value. There are two values for action that are run while using the Plushi interpreter to generate and execute Push programs. The first action returns the set of supported instructions which can appear in a program. The second action returns the output of running a given program on a given dataset.

3.1.1 The "instructions" Action.

When the value of the "action" field in the request body JSON is "instructions" the Plushi server will return the set of supported instructions which can appear in programs that will be run by the Plushi interpreter. The instruction set is encoded in JSON as a list of one key-value object per instruction. The schema of each instruction object contains a key-value pair for the instruction name, a list of types the instruction requires as input, and a list of types the instruction produces as output. Each of these instruction objects is referred to as an instruction signature, because it holds name and type information related to the instruction, but not a full specification of its behavior. Below is an example of a single instruction JSON object for the `integer_add` instruction which produces the sum of two integers.

```
{
  "name": "plushi:integer_add",
  "input-types": ["integer", "integer"],
  "output-types": ["integer"],
}
```

When making a request with the "instructions" action, it is required that the "arity" key be present in the request body JSON.

This field specifies the number of inputs (or features) the program being searched for, or learned, will take. Plushi needs this information when creating the instruction set because each input value to a Plushi program receives a dedicated instruction that produces the input value each time it is evaluated.

Below is a snippet of python code that uses the `requests` and `json` libraries to make an "instructions" request. It assumes a Plushi interpreter is running locally on port 8075.

```
import json as j
import requests as r
body = {
  'action': 'instructions',
  'arity': 4
}
instr_set = r.post("http://localhost:8075/",
                   json=j.dumps(body)).json()
```

After making a request to retrieve the instruction set from Plushi server, the user can filter the list of instruction signatures down to only the instructions that deal data types relevant to the problem. For example, if a machine learning system is using Plushi to produce a real-valued regression model there does not need to be instructions that manipulate strings in the programs produced by the machine learning system. Performing this intuitive filtering of the instruction set helps limit the search space, but it is not required.

3.1.2 The "run" action.

When the value of the "action" field in the request body JSON is "run" the Plushi server will return a list of output values produced when running a Plushi program on each case in a dataset. The required field of a Plushi request with the "run" action are given in figure 1.

Below is a snippet of python code that uses the `requests` and `json` libraries to run a program that adds the numbers 1 and 2 on a Plushi interpreter running locally on port 8075. It assumes that there is a JSON file containing the dataset to run the program on in a file named `data.json`.

```
import json as j
import requests as r
X = j.load("data.json")
b = {
  'action': 'run',
  'code': [1, 2, "plushi:integer_add"],
  'arity': 1,
  'output-types': ['integer'],
  'dataset': X
}
outputs = r.post("http://localhost:8075/",
                 json=json.dumps(b)).json()
```

3.2 During Training or Search

The core function of a machine learning system is to search for some kind of optimal structure that solves a problem. Some examples of this include: optimizing a set of coefficients for a linear model, selecting splits to use in a decision tree, and finding parameter values for the weights of an artificial neural network that minimizes the loss function. In order to guide these search/optimization procedures, a training dataset is generally used to evaluate intermediate models until a fit model is produced.

Plushi is designed with this training phase in mind. In order to minimize the time spent on communication between the machine learning framework and the Plushi server, a program can be executed on an entire dataset using one request. In other words, Plushi expects that most use cases would prefer higher throughput without much regard to maintaining a low latency.

3.3 Deploying Plushi Models

One of the failings of previous Push systems is the ability to easily save and deploy the synthesized programs for later use. A benefit of Plushi's JSON program representation is that programs found by machine learning frameworks can easily be persisted. Programs can also be run in any environment that has access to a Plushi interpreter, regardless of if the environment has access to the machine learning system used to produce the program. This also implies that if multiple different machine learning methods produce a collection of Plushi programs, all programs can be deployed into a system that uses a single Plushi interpreter.

When models are deployed, or productionalized, they are used to make predictions on new data. Predictions are often made on single data records in real-time. Predictions can also be made on larger batches data on a less frequent schedule. Both use cases are possible with the Plushi server, however the use case of many, frequent, single prediction requests has not yet been extensively tested. The use case of larger, infrequent batches bears resemblance to the training stage discussed in Section 4.2.

The Plushi server can run a different program on each request, thus it is possible to run a single Plushi server that can make predictions using all productionalized programs produced by the machine learning systems within an organization. This use case could benefit from multiple Plushi servers running on different machines behind a load balancer.

3.4 Example Usage

To evaluate the ease of Plushi usage, a simple Python implementation of simulated annealing was created that produces Plushi programs. The full application (simulated annealing and Plushi calls) was written in under 150 lines of Python code.[3]

To start a separate process which runs the Plushi server from within Python, the `Popen` class from the `subprocess` library was used. In other applications if the ability to spawn a new process is not available, the Plushi server can be started beforehand. Assuming we have built the Plushi software into a `.jar` file named "plushi-standalone.jar" the system call which is needed to start the server is as follows:

```
java -jar plushi-standalone.jar --start
```

To generate random programs, the simulated annealing scripts first get the set of supported instructions through an "instructions"

[3]The code for this example can be found in the python files here: https://github.com/erp12/plushi-annealing

Key	Description
`code`	A serialized Plushi program. Must be a JSON list with no nested lists.
`arity`	The number of inputs (or features) the program will takes. Plushi needs this to create the instruction set because each input value to a Plushi program receives a dedicated instruction that produces the input value each time it is evaluated.
`output-types`	A list of type names which will be returned in a list as the output value of running the program.
`dataset`	The dataset to run the program on serialized into JSON. The dataset should be a list or records. Each record would be a JSON object where the keys are the feature names and the values are the feature values.

Figure 1: A table of keys which must appear in the body of the POST request sent to Plushi when the action field is set to "run". The response to these requests will be a JSON blob containing a list of output values produced by running the program from the "`code`" field on each of the records in the dataset from the "`dataset`" field.

Processor	Intel Core i7
Processor Speed	2.8 GHz
Total Number of Cores	4
Memory	16 GB

Figure 2: A table of hardware (MacBook Pro) specs used to benchmark the program executions per second achieved by the simulated annealing example.

request as described in section 3.1. A random linear sequence of the name fields from any of these instructions is a valid program. Programs can also include integers, floats, booleans, and strings that are not equivalent to instruction names.

The simulated annealing algorithm was selected because it is fairly lightweight and most of the computation is done during evaluation of programs. On the hardware described in figure 2 the Plushi server was able to perform 1998 program executions per second, where each program was 100 elements long. The number of program executions per second for various program lengths is given by Figure 3. At this time, the Plushi request handling does not utilize any parallel or asynchronous computation during program execution. This is a valuable area or future research and development.

4 IMPLEMENTATION

Plushi was written in the Clojure programming language, and is built as a standalone `.jar` artifact. Clojure is a modern Lisp language that runs on the Java Virtual Machine (JVM) thus it is possible for other programming languages that run on the JVM to use Plushi via direct interop [8]. This is not the primary use case Plushi is designed for, because producing a standalone service allows Plushi to be more language agnostic.

The two main implementation differences between Plushi and other Push interpreters is the variant of the Push language specification that was used to maintain linear programs, and the use of an HTTP server to allow for the Plushi interpreter to be embedded in arbitrary systems.

Lastly, it is common to implement custom, domain specific instructions to allow for Push programs to operate in particular domains. For example, previous applications of the Push language inside of genetic programming systems have been augmented with instructions that allow Push programs to express quantum circuits or artificial neural network architectures [11, 13]. To add these instructions to Plushi a change to the source code is required. A new instruction can be registered with a single function call, as described later in Section 4.3.

4.1 Linear Push Variant

The contemporary specification of the Push language is simply nested sequences of Push atoms, where an atom is either an instruction supported by the interpreter or a literal value of some primitive data type. The nested structure was generally considered to be important because a nested sequence is commonly used as the body of control structures, such as conditionals and loops. The problem introduced by the nested structure of Push programs is that many machine learning methods cannot search over arbitrarily nested structures. The original uses of the Push language in genetic programming systems did not have this obstacle because it is common to use genetic operators that manipulate nested structures, such as trees.

Plushi implements a new specification of the Push language where all programs are represented in a linear structure. Despite the linear structure, this variant of the push language can still express synonymous programs to the nested specification, as described in the following paragraphs. This is achieved by through two implementation details in the Plushi instruction set.

First, each instruction includes a field which denotes the number of code blocks which should follow it in the program. For example, the `exec_if` instruction (which implements a simple if-else logic) should be followed by two code blocks: one for the if clause, and one for the else clause. The `exec_while` instruction (which implements a `while` loop) should be followed by one code block which is considered to be the body of the while loop.

Second, programs can contain placeholder atoms called the "close" atoms which denote the end of a code block. The Plushi interpreter use these "close" atoms along with number of code blocks opened by the definitions of each instruction to translate the linear program representations proposed in this paper into the traditional, nested Push programs. This preserves the expressiveness of Push because control structures, such as loops, can have arbitrarily long code blocks acting as the body of the loop.

Figure 4 shows a JSON list which could be executed by Plushi and contains a control structure which utilizes close atoms. The definition of `exec_if` specifies that there should be two code blocks following the instruction. Given the two instances of the `close` atom in the program, we know that the first code block contains the integer literal 1 and the second code block contains the integer

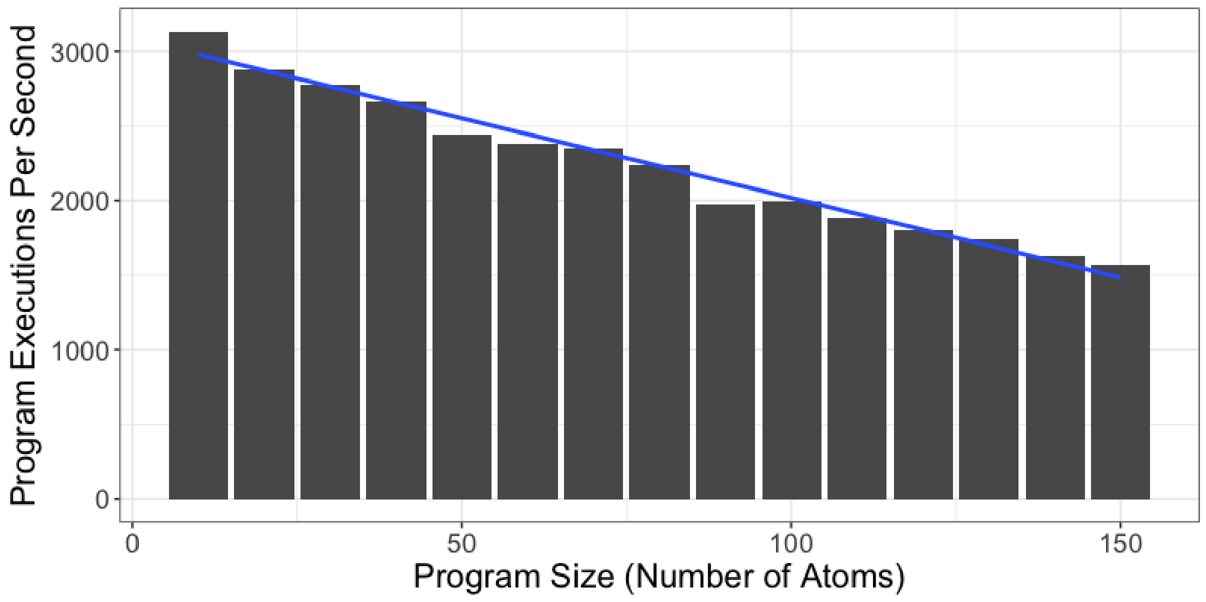

Figure 3: A bar chart with a linear trend line showing the number of program executions the Plushi server was able to run per second as it relates to the size of the programs being sent to it. All programs were produced by the simulated annealing example attempting to synthesize the ReLU function [5]. These timings were seen on the hardware described in figure 2.

```
["plushi:input_0", "plushi:exec_if", 1, "plushi:close", 0, "plushi:close"]
```

Figure 4: An example program demonstrating the use of the close atoms. The `input_0` instruction pushes the value of the first feature in the feature vector. If we assume that the first first feature is a boolean value the program will output 0 if the input feature value is true, and zero if it is false. The close atoms mark the ending points of the if and else

literal 0. It is possible for an arbitrary number of instructions and literals to appear in a code block.

When translating from the linear program representation to the nested representation, close atoms can be ignored if: 1) no instructions with open code blocks have been present in the program or 2) all instructions which denote the opening of code blocks have been closed by previous close atoms. If the entire contents of the linear program representation have been translated and there are code blocks which have not yet been closed, additional close atoms are added to the end of the program until all code blocks have been closed.

The motivation for keeping the program structure linear is that it removes all syntax requirements from the language specification. Any sequence of Plushi atoms is an executable program. It is not possible for Plushi programs to produce runtime errors. This helps machine learning systems search the space of programs unrestricted.

It should be noted that the concept of creating a linear representation of Push programs is not a novel contribution put forth by this work. Some genetic programming systems have developed linear genomes that are translated into nested Push programs. These genomes are generally referred to as Plush genomes [7]. These genomes do not utilize close atoms, but rather mark certain genes with epigenetic markers to denote the insertion of a close parenthesis.

4.2 The Plushi Server

As described in section 4, Plushi is implemented as an HTTP server capable of responding to POST requests that ask the server to run a particular program on a given dataset. Plushi uses the Ring library written in the Clojure language to implement the HTTP server abstraction [2].

The Plushi server assumes the body of each request is valid JSON. The responses given by the Plushi server are also have a body encoded in JSON. One potential area for future development is to include support for more expressive file formats, such as EDN.

4.3 Extending the Instruction Set

When human programmers write software for particular domains, it is common for a library of additional functions related to the domain to be used. When synthesizing Push programs using machine learning, it is similarly beneficial to augment the standard instruction set with specialized instructions relating to the problem domain.

Some notable examples of this include a set of quantum instructions which were used in Push programs to express quantum circuits. Using these specialized instructions, a genetic programming system was able to search the space of quantum circuits and discover novel quantum computations [13]. Another example of augmenting the instruction set is the use of instructions to manipulate the architecture of a neural network. In other words, the result of executing a Push program containing these instructions is a untrained network. The number of layers, and each of their sizes, are all determined by the push program which may have utilized arbitrary control structures to find interesting architectures [11].

To define custom instruction in Plushi, the source code must be modified to include additional calls the the `register` function. This function takes the following arguments:

- An instruction name that is unique throughout the instruction set.
- A Clojure function which implements the desired behavior of the instruction. The function should return either a single value, or a vector of values.
- A vector of Clojure keywords denoting the type of each argument to the function. This is used to determine which stacks to pop arguments off of.
- A vector of Clojure keywords denoting the type of each value returned by the function. This is used to determine which stacks to route return values to.
- A integer denoting the number of code blocks following the instruction. This generally zero except when implementing control structures as described in section 4.1.
- Optionally, a doc-string describing the behavior of the instruction.

Figure 5 shows a code snippet which calls the `register` function to define a new instruction named `exec_if`. The behavior of this instruction is simply the if-else control structure, but the call to register also specifies that the instructions arguments should come from the boolean and exec stacks. Also, the result of the `exec_if` instruction should be pushed back to the exec stack.

In addition to defining new instructions, it is also common to utilize auxiliary stacks to hold values other than the standard data types. For example, an custom instruction to manipulate a network architecture may pop a network encoding off of a dedicated network stack and push a modified version of the network back onto the same stack. As shown in Figure 5, the stacks which are used as the source of arguments and the destination of returned values must be specified when registering each instruction. When a program is executed by the Plushi interpreter, the initial stacks are created based on the set of types which exist in the list of input and output types across the entire instruction set. Thus, the required stacks are implicit and no addition work by the user is required to create more stacks.

4.4 Other Features

Although the Push language, and the linear variant proposed in this paper, are designed for machine learning systems to produce programs with, it is still common that human users will want to investigate the synthesized programs to understand how they work. For this reason, it is important for the Plushi instruction set to be properly documented. When defining new instructions, users can specify a "docstring" that describes the behavior of the instruction. When users send an "instructions" request, they can specify an additional field that will include the docstrings of each instruction. The Plushi standalone `.jar` artifact can also generate an HTML page describing the instruction set by running with the `--docs` flag instead of the `--start` flag.

Plushi is an open source project, hosted on GitHub[4]. This allows for easy distribution of releases, coordination of contributions, and the use of continuous integration tools.

The Plushi repository contains a comprehensive unit test suite, which includes unit tests of each instruction supported by the Plushi interpreter. In addition to unit tests, a handful of validation tests that run sample Plushi programs on small datasets are included in the repository. Both of these sets of tests are run on every contribution to ensure the language specification and the interpreter are behaving as expected.

Documentation on the Plushi software is also generated from files within the repository. These include markdown documents explaining the use of the software and docstrings embedded in the source code. Plushi utilizes a tool called `lein-codox` to convert both these sources of documentation into a web page which is then hosted publicly on GitHub Pages[5] [12].

5 CONCLUSION

With this proposal of the Plushi system, the Push language can now be used outside of genetic programming systems. A much broader set of machine learning algorithms can now perform program synthesis tasks via the use of the push language. Programs found by these machine learning systems can be deployed and executed in real-time or on large datasets without the use of the machine learning system which produced the programs.

Now that there is an embeddable, language agnostic interpreter for the Push language, we hope to see Push introduced into many existing machine learning frameworks, both within and outside the evolutionary computation community. It would be valuable to closely monitor any project that attempts to add Plushi as a dependency in order to start using the Push language to perform program synthesis.

We hope to treat any such integration as a case study to evaluate the effectiveness of the Plushi server as an interface between the machine learning system and the Push interpreter. Possible flaws surrounding the feasibility of the current Plushi system could, and will, be identified as Plushi sees more application.

More testing needs to be done to determine how well the Plushi server scales to larger problems, and larger datasets. This may require research into an appropriate application of parallel computation that Plushi could leverage to see increased throughput.

Another area of future development that may improve the Plushi system is support for additional formats in which to encode serialized programs. One very promising format is EDN, which supports a wider range of data types than JSON.

[4]https://github.com/erp12/plushi
[5]https://erp12.github.io/plushi/index.html

```
(register "exec_if"
          (fn [b then else] (if b then else))
          [:boolean :exec :exec] [:exec] 2
          "If the top boolean is true, execute the top element of the exec
           stack and skip the second. Otherwise, skip the top element of the
           exec stack and execute the second.")
```

Figure 5: A Clojure snippet showing an example call to the `register` function which creates the `exec_if` instruction. This instruction implements the if-else control structure.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1617087. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] [n. d.]. JSON. ([n. d.]). https://www.json.org/
[2] [n. d.]. ring-clojure. ([n. d.]). https://github.com/ring-clojure/ring
[3] Achiya Elyasaf and Moshe Sipper. 2014. Software review: the HeuristicLab framework. *Genetic Programming and Evolvable Machines* 15, 2 (01 Jun 2014), 215–218. https://doi.org/10.1007/s10710-014-9214-4
[4] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research* 13, Jul (2012), 2171–2175.
[5] Richard H. R. Hahnloser, Rahul Sarpeshkar, Misha A. Mahowald, Rodney J. Douglas, and H. Sebastian Seung. 2000. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature* 405 (22 06 2000), 947 EP –. http://dx.doi.org/10.1038/35016072
[6] Thomas Helmuth and Lee Spector. 2015. General Program Synthesis Benchmark Suite. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO '15)*. ACM, New York, NY, USA, 1039–1046. https://doi.org/10.1145/2739480.2754769
[7] Thomas Helmuth, Lee Spector, Nicholas Freitag McPhee, and Saul Shanabrook. 2016. Linear Genomes for Structured Programs. *Genetic Programming Theory and Practice XIV (Genetic and Evolutionary Computation)* (2016).
[8] Rich Hickey. [n. d.]. Clojure. ([n. d.]). https://clojure.org/
[9] Edward Pantridge, Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2017. On the Difficulty of Benchmarking Inductive Program Synthesis Methods. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '17)*. ACM, New York, NY, USA, 1589–1596. https://doi.org/10.1145/3067695.3082533
[10] Edward Pantridge and Lee Spector. 2017. PyshGP: PushGP in Python. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '17)*. ACM, New York, NY, USA, 1255–1262. https://doi.org/10.1145/3067695.3082468
[11] Edward R. Pantridge and Lee Spector. 2016. Evolution of Layer Based Neural Networks: Preliminary Report. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion (GECCO '16 Companion)*. ACM, New York, NY, USA, 1015–1022. https://doi.org/10.1145/2908961.2931664
[12] James Reeves. [n. d.]. codox. ([n. d.]). https://github.com/weavejester/codox
[13] Lee Spector and Jon Klein. [n. d.]. Machine Invention of Quantum Computing Circuits by Means of Genetic Programming. *AI-EDAM: Artificial Intelligence for Engineering Design, Analysis and Manufacturing, year=2008, volume=22, number=3 pages=275–283,* ([n. d.]).
[14] Lee Spector, Jon Klein, and Maarten Keijzer. 2005. The Push3 Execution Stack and the Evolution of Control. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation (GECCO '05)*. ACM, New York, NY, USA, 1689–1696. https://doi.org/10.1145/1068009.1068292