Push-forth: a Light-weight, Strongly-typed, Stack-based Genetic Programming Language

Maarten Keijzer
Pegasystems Inc.
Claude Debussylaan 20b
Amsterdam, The Netherlands
mkeijzer@xs4all.nl

ABSTRACT

This paper defines the push-forth language, a recombination of Push [3] and Joy [7], borrowing type-safety considerations from Alp [2]. Push-forth is stack-based, strongly typed and easy to extend. The concept of an Evolutionary Development Environment is presented, and some informal experiments are described to illustrate the utility of such an environment.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Specialized application languages

Keywords

Genetic programming, Functional programming, Joy, Push, Evolutionary development environment

1. THE PUSH-FORTH ENGINE

Push-forth is a recombination of concatenative combinator languages like Joy [7], Cat [1], and genetic programming languages like Push 1,2 & 3 [3, 5, 6, 4]. Push-forth is designed to be simple to define, easy to implement, strongly typed, and extensible. The language is stack based, and instructions manipulate items on the stack. It has the concept of a program – in itself a 'stack' of instructions – that operates on a stack of data elements. Because program and stack work intimately together, it was chosen to use an implementation where the program would, by convention, be the first element on the data stack. Execution of a program would pop the program of the data stack, execute an instruction from that program using the rest of the data stack, and push the program back on the data stack.

The language thus uses a single stack. It uses predicates to ensure type-safety, and by virtue of this type-safety it can handle constrained languages such as matrix algebra, simply by coding the constraints in the definition of the instructions. In this respect it resembles a language such as ALP [2]. It is not surprising that a push-forth interpreter can be implemented in a few lines of code in Prolog.

As an inital example, below a trace of the full execution of the simple postfix program **1 1 +** is shown:

```
[ [1 1 +] ]
[ [1 +] 1]
[ [+] 1 1]
[ [] 2]
```

Execution thus halts when the first element of the list/stack is the empty list. Because the pattern of a list containing a list as the first element is ubiquitous in push-forth, a special notation is used bring code and data notationally closer together. It does this by introducing the pivot element ' · ', and reversing the order of the program, effectively showing it in prefix notation. So if c defines an instruction, and d a data element, the following convention is used:

$$[[\mathbf{c_1} \dots \mathbf{c_n}]\, \mathbf{d_1} \dots \mathbf{d_m}] \quad \equiv \quad [\mathbf{c_n} \dots \mathbf{c_1} \cdot \mathbf{d_1} \dots \mathbf{d_m}]$$

And thus

$$[[]\, \mathbf{d_1} \dots \mathbf{d_m}] \equiv [\cdot\, \mathbf{d_1} \dots \mathbf{d_m}]$$

In this altered notation, the previous computation now reads differently.

```
[+ 1 1 · ]
[+ 1 · 1]
[+ · 1 1]
[ · 2]
```

This is just notational convenience, the underlying implementation works with postfix programs pushed on the data stack. This notation is chosen as it makes some of the definitions easier to follow. In a few cases this does not hold. For these the original notation is used. Below, upper case characters such as **X** , **Y** , and **Z** are typically used to denote lists, while lower case characters **x** , **y** , and **z** denote any data type (including lists).

Using this notation, we can define an instruction **f** as an operation that takes some elements from the stack **Y** (to the left of the pivot element), and returns a continuation, **[C · D]** , where **C** is the code to run, and **D** contains the remainder of the original data stack, optionally with some results added to the stack.

GECCO'13 Companion, July 6–10, 2013, Amsterdam, The Netherlands.

$$\mathbf{f(Y) = [C \cdot D]}$$

One such instruction is **eval** . Eval takes a list, and proceeds in four different cases:

eval([x X])	=	**[· x X]**	x not list
eval([· X])	=	**[· X]**	halt
eval([X f · Y])	=	**[X C · D]**	**f(Y) = [C · D]**
eval([X x · Y])	=	**[X · x Y]**	otherwise

eval thus takes a list, and either creates a halted program out of that list, executes an instruction, or moves data from the code stack to the data stack. It is itself an instruction. As a departure from Push 1,2, and 3 that execute a list recursively, in push-forth, a list is considered to be a data element, and gets pushed on the **data** stack.

In essence, the **eval** instruction defines the push-forth engine. At its core, this engine consists of an implementation of a list/stack, a type definition of an instruction, the **eval** instruction that is aware of the type of instructions, and a set of other instructions and data elements. The set of instructions is defined to do something useful, and can vary from generic to very specific. Below a subset of instructions are described that define a number of important patterns to deal with:

- Too few arguments on the stack
- Wrongly typed arguments on the stack
- Recursion

1.1 Too few arguments on the stack

Stack manipulating functions such as **dup** , **swap** and **rot** can operate on any stack element. All they need is the presence of 1,2, or 3 arguments on the stack respectively. In push-forth, too few elements on the stack lead to the instruction to be skipped (no-op).

dup([])	=	**[·]**
dup([x X])	=	**[· x x X]**
swap([])	=	**[·]**
swap([x])	=	**[· x]**
swap([x y X])	=	**[· y x X]**

1.2 Wrongly typed arguments on the stack

Type-safety is enforced by the instructions. Typically, an instruction checks the types on the stack, and returns a continuation when the types do not match. This continuation puts the data-element that was incorrectly typed onto the **code** stack, and recursively calls itself. Assuming **n** and **m** designate integers, integer addition is fully defined using the following 5 cases.

add([])	=	**[.]**	missing args
add([x])	=	**[. x]**	missing args
add([x X])	=	**[x add · X]**	**x** not int
add([m y X])	=	**[y add · m X]**	**y** not int
add([m n X])	=	**[. m+n X]**	execute

When non-integers are encountered, they are put on the **code** stack, and will, after successful execution of the addition, be put back on the **data** stack. Example:

```
1   [add · 'b' 1 'a' 2]
2   ['b' add · 1 'a' 2]
3   ['b' 'a' add · 1 2]
4   ['b' 'a' · 3]
5   ['b' · 'a' 3]
6   [ · 'b' 'a' 3]
```

Although in the example above a simple type check was performed, in general, the language can use arbitrary predicates to check type safety. For instance, an integer division instruction could simply skip any zeros it will find as denominator, and matrix operators could check row and column dimensions, skipping over instructions that cannot be used in the computation. This gives greater flexibility in the use of typed operators than for instance Push 1,2 & 3, where a separate stack is maintained for each type. The disadvantage is a possible lack of efficiency, where instructions are continuously searching for arguments on the stack. However, by consistently using the number of instructions processed as a means of gauging the speed of a program, and by *always* having this speed as an objective in the search, programs will need to find ways to work efficiently with the available resources.

1.3 Recursion, Combinators and Turing Completeness

For simple recursion, the combinator **i** is defined.

i([])	=	**[]**
i([[$c_1 \ldots c_n$] Y])	=	**[$c_n \ldots c_1$ · Y]**

Although the pivot notation obscures it, essentially, the **i** combinator takes a list from the data stack, and appends it to the code stack. **i** is a basic combinator, but many more combinators on code can be defined. Turing completeness can be established with any set of basis combinators ([7]). A few are displayed here (omitting type safety conditions for brevity):

cons([x [X] Y])	=	**[· [x X] Y]**	
pop([x X])	=	**[· X]**	
split([[x X]])	=	**[· x X]**	
car([[x X]])	=	**[· x]**	**split swap pop**
cdr([[x X]])	=	**[· X]**	**split pop**
cat([X Y Z])	=	**[· [[X Y] Z]**	
unit([x Z])	=	**[· [x] Z]**	

1.4 A push-forth interpreter written in push-forth

Define the **while** combinator such that it halts when the second argument is empty, and recursively executes the first argument otherwise. In this section, the use of the **C · D** notation becomes confusing, therefore it uses the regular style of nested brackets (postfix).

while([X [] Y)	=	**[[] Y]**
while([X Z Y)	=	**[[X [X while] i] Y]**

A push-forth interpreter can be written in push-forth as **[while [eval dup car] [[]]]** :

[[[]] [eval dup car] while] [[1 1 +]]
[[eval dup car] while] [[]] [[1 1 +]]
[while] [eval dup car] [[]] [[1 1 +]]
[eval dup car [[eval dup car] while] i] [[1 1 +]]
...
[while] [eval dup car] [1 +] [[1 +] 1]
...
[eval dup car [[eval dup car] while] i] [[+] 1 1]
[dup car [[eval dup car] while] i] [[] 2]
...
[while] [eval dup car] [] [[] 2]
[] [[] 2]

1.5 Overloading

By virtue of operating on a single stack, the language comes with a natural way of overloading functions. Whenever there are instructions that share the same symbol, say + that is defined for integers, floats, matrices, vectors, polynomials, or symbolic expressions themselves, they can be combined into a single combined instruction that will execute for the first set of matching arguments. As an example, consider the case that the symbol '+' is used concatenation of strings and addition of integers:

+ · "Hello " 2 "World!" 3
2 + · "Hello " "World!" 3
2 · "Hello World!" 3
· 2 "Hello World!" 3

2. GENETIC PROGRAMMING IN AN EVOLUTIONARY DEVELOPMENT ENVIRONMENT

For representing code to a genetic programming system, push-forth remains faithful to its *FORTH* ancestry. In push-forth, a program is simply a string of elements. The elements are either an instruction (which will get executed when encountered), or a list containing a single instruction (which will get put on the data stack). Using combinators such as **cons** on lists containing instructions, the genetic programming system has sufficient expressive power to be able to create arbitrary pieces of code, and by virtue of combinators such as *i*, it has the ability to execute them. It is not needed for the evolving programs to itself contain complex lists. By executing, a complex list can be built:

[cat [3] cons cat [1] cat [2] [*] [+] ·]
[cat [3] cons cat [1] cat [2] [*] · [+]]
[cat [3] cons cat [1] cat [2] · [*] [+]]
[cat [3] cons cat [1] cat · [2] [*] [+]]
[cat [3] cons cat [1] · [2 *] [+]]
[cat [3] cons cat · [1] [2 *] [+]]
[cat [3] cons · [1 2 *] [+]]
[cat [3] · [[1 2 *] +]]
[cat · [3] [[1 2 *] +]]
[· [3 [1 2 *] +]]

The goal of push-forth is to define a light-weight language to be used in a light-weight genetic programming system. Basic genetic operators such as one-point crossover and point-mutation can readily be defined. Having the representation and the genetic operators defined in this way, a genetic programming system is easily built using these.

2.1 Evolving instructions

So far a set of primitive instructions have been defined, but an extension mechanism (words or names) is lacking. In *FORTH* a 'word' is a user-defined instructions, an association between a name and a piece of code. In *Push*, 'words' are called 'names', and have been part of the genetic apparatus from the onset. Push programs are allowed to dynamically create instructions and use them. Unlike their counterpart in tree-based genetic programming, ADFs, where at least some experiments have been conducted that clearly show advantage, names have not lived up the promise: no evidence has been found where names have an evolutionary advantage over just using code directly [1]. The modularity achieved by being able to manipulate and use code is apparently sufficiently powerful to not need an additional method of indirection.

There are several reasons for discontinuing the use of names in push-forth as part of the language itself. The most important of it lies in breaking the expectation of lexical closure. Given a set of instructions, a push-forth program is fully defined by its data stack containing the (partly executed) program. Adding names to the representation would mean that additional information needs to be stored next to the evolving program to determine the current name bindings. In push-forth, such additional information should reside on the data-stack, possibly as a dictionary with corresponding **put** and **get** instructions. Using names in this way is not explored further here.

In this section the use of push-forth as a means to build an Evolutionary Development Environment (EDE) is explored. In such a development environment, the user is assisted by a genetic programming system. The user's task is to do some of the task decomposition – the user does this by defining names for instructions, and test-cases defining the semantics of those instructions. The EDE will take the definition of such an instruction and will try to find a push-forth program that solves all of the test-cases. Once such a program is found, it is added to the global library and can henceforth be used by all evolving programs. This evolved instruction will only be replaced by an instruction that is more efficient, i.e., that solves the problem faster.

Correctness in the EDE thus defines a particular level of competence that a program must achieve before it is promoted to inclusion in the library. Once an instruction is included in the library it becomes available for other genetic programming runs to solve different problems. This development environment works using the following loop:

1. Flip a coin to see if we're trying to solve an unsolved problem or improve upon an existing solution
2. Pick an instruction from the category selected above
3. Create a multi-objective search finding a solution
4. If a solution has solved all test cases, compare it with the best so far (if existing), and add it to the library if it is faster overall
5. Start again

[1] Lee Spector, personal communication

The human developer (in this case the author), lets the system run, and adds new instructions or new testcases if the evolved instructions have issues because of ommissions in the test cases. It's an interactive cycle.

An example input file used in the current system defines the combinator **cake** [7] as:

```
cake

test:
b a cake
[[b] a] [a [b]]

test:
[dup b] [dup a] cake
[[[dup b]] [dup a]] [[dup a] [[dup b]]]
```

These test cases are evolved on four objectives:

- Hit – a tree based distance measure captures the distance between the output stack and the desired output. Distance between numbers is measured by difference in magnitude
- Speed – number of instructions processed
- Flat Hits – number of hits between elements ignoring elaborate tree structure (array comparison between flattened lists)
- Flat Type hits – number of correct types on flattened arrays

The two latter objectives provide additional guidance to the search by giving points to focussing on producing the right types and the right elements. Each test case will be judged on those four objectives. In the case of **cake** there are thus 8 objectives in the search. An evolutionary multi-objective algorithm is used that keeps an archive of the front of non-dominated individuals and a population of 2000 individuals, picking parents either from the archive or from the population. Due to the large number of objectives the EDE uses, the Pareto-front can grow very large. However, the constant pressure on size in the objectives and the light-weight nature of the individuals themselves keep this manageable.

The second test-case for this problem seems supperfluous, but the use of the **dup** instruction is aimed to prevent solutions that will execute (i.e., put on the program stack) the arguments to the **cake** function.

For the **cake** problem, a solution was induced in the library. It scored perfect on the three 'error' based objectives and did that in 34 instruction executions. The resulting program reads: [**swap unit unit swap cons dup unpair unit cat unit eval**] . The solution makes use of an instruction **unpair** . This instruction was defined elsewhere in the system as:

```
unpair

test:
[[a] [b]] unpair
[a] [b]

test:
[a b] unpair
a b
```

It was defined for a different reason, but evolution saw fit to use it for solving the **cake** problem. A wide range of instructions have been induced in this way: combinators such as *cake*, *fold* and *zap*, polynomials, numbers (π), conditionals, etc. Not all

2.2 Limitations & Future Work

The EDE in its current state is rudimentary. Only instructions in a simple input-output relationship can be defined, and no auto-constructive evolution [3] takes place in the system. Crossovers and mutations are hard-coded, and do not evolve. The representation is limited to a single array, and no mechanism is in place to exploit further structural changes of code. Also the method of measuring fitness needs work. Finally, the number of instructions is growing, making it harder for the evolutionary search to find good solutions from scratch.

The concept of an EDE is compelling: being able to develop computer programs that are by definition testable and which can therefore be continuously improved. The goal of an EDE is to be able to create computer programs faster and in a more reliable fashion. Letting a computer create the code makes this possible.

3. CONCLUSION

The push-forth language is a concatenative combinator language that is largely compatible with Joy and Cat, although push-forth lacks the concept of words/names. Legal Joy or Cat programs are typically legal push-forth programs. Due to the type-safety inherent in push-forth, the reverse does not hold. It is a very simple language but has a number of features that can be important for being a general purpose genetic programming language:

- Re-entrant and lexical closure – a program is the first element on the stack. Each evaluation is primitive and returns control immediately. A program thus executes in a self-contained manner with no additional setup. This allows easy storage of partly executed programs. It is trivial to 'freeze' a program and pick it up for further execution later.
- Type safety – skipping data elements on the stack as a generic mechanism to find correct types
- Predicates for defining instructions – the use of predicates and skipping illegal data elements allow for a straightforward mechanism to define instructions and add them to the language
- Computationally complete – loops, recursions are all part
- Ability to sandbox – computer programs can evaluate other computer programs without the underlying program taking control

It is hypothesized that these features allow pursuing the vision for auto-constructive evolution, as set forth by Lee Spector[3], in particular in the form of an Evolutionary Development Environment: a place where the human operator no longer creates code, but let a computer continuously create and improve programs that solve test-cases.

4. REFERENCES

[1] Christopher Diggins. Simple type inference for higher-order stack oriented languages. Technical Report Cat-TR-2008-001, http://www.cdiggins.com, USA, 4 September 2008.

[2] M. Keijzer, V. Babovic, C. Ryan, M. O'Neill, and M. Cattolico. Adaptive logic programming. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 42–49, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[3] Lee Spector. Autoconstructive evolution: Push, pushGP, and pushpop. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 137–146, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[4] Lee Spector, Jon Klein, and Maarten Keijzer. The push3 execution stack and the evolution of control. In Hans-Georg Beyer, Una-May O'Reilly, Dirk V. Arnold, Wolfgang Banzhaf, Christian Blum, Eric W. Bonabeau, Erick Cantu-Paz, Dipankar Dasgupta, Kalyanmoy Deb, James A. Foster, Edwin D. de Jong, Hod Lipson, Xavier Llora, Spiros Mancoridis, Martin Pelikan, Guenther R. Raidl, Terence Soule, Andy M. Tyrrell, Jean-Paul Watson, and Eckart Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1689–1696, Washington DC, USA, 25-29 June 2005. ACM Press.

[5] Lee Spector, Chris Perry, and Jon Klein. Push 2.0 programming language description. Technical report, School of Cognitive Science, Hampshire College, April 2004.

[6] Lee Spector, Chris Perry, Jon Klein, and Maarten Keijzer. Push 3.0 programming language description. Technical Report HC-CSTR-2004-02, School of Cognitive Science, Hampshire College, USA, 10 September 2004.

[7] Manfred von Thun. Joy: Forth's functional cousin. In *Proceedings from the 17th EuroForth Conference*, 2001.