The Push3 Execution Stack and the Evolution of Control

Lee Spector[*], Jon Klein[*°], and Maarten Keijzer[‡]

[*]Cognitive Science, Hampshire College, Amherst, MA USA

[°]Physical Resource Theory, Chalmers U. Tech. & Göteborg U., Sweden

[‡]Chordiant Software Inc.

lspector@hampshire.edu, http://hampshire.edu/lspector/

Outline

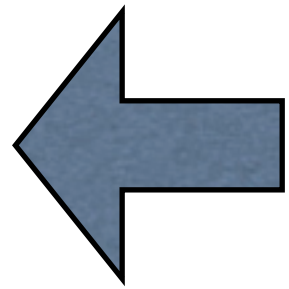
- The Push project: open-ended evolution of arbitrary computational processes.
- Push concepts and simple examples.
- The new EXEC stack and the amazing things that it can do.
- Examples: list reverse, factorial, Fibonacci, parity, exponentiation, sorting.

The Push Project

- What? Evolution of arbitrary computational processes.
- Why? Problem solving, artificial life, cognitive modeling, ...
- How? Natural selection of programs expressed in syntactically trivial, semantically rich program representation.
- How rich? Rich enough to express evolving programs of arbitrary architecture and even (in some applications but not in PushGP) the mechanisms of evolution itself.

Representations for Evolving Programs

- Lisp-like symbolic expressions
- Machine code
- Stack-based interpreter code
- Grammar indices
- Logic clauses
- ...



Push

- Stack-based postfix language with one stack per type; types include integer, float, vector, Boolean, name, code, exec, ...
- Trivial syntax.
- Evolved programs may use:
  - multiple data types (arbitrarily intermixed)
  - subroutines (evolved dynamic architecture)
  - recursion and iteration (arbitrarily structured)
- Also supports the evolution of evolutionary mechanisms (meta-evolution and *autoconstructive evolution*).

Push Syntax

program ::= instruction | literal | (program*)

Power from *That*?

Yes. Here's how:

- Instructions are written to take arguments from, and put results onto, the appropriate stacks.
- By use of the CODE stack (and now also the EXEC stack) programs can manipulate (arbitrarily) and then execute (possibly conditionally) parts of their own code, thereby implementing evolved control structures, architectures, etc.

Evolved use of Control Structures in GP

- If-then-else
- Do-until, Do-times
- Recursion (implicit/explicit)
- Evolved functions/macros/architectures
- See Koza, Kinnear, Angeline & Pollack, Racine & Schoenauer & Dague, Robert & Howard & Koza, Nordin & Banzhaf, Olsson, Whigham & McKay, Brave, Wong & Leung, Yu & Clack, Nishiguchi & Fujimoto, Schmidhuber, Kochenderfer, ...
- Claim: The Push approach is more elegant, more expressive, and more evolvable.

Sample Push Instructions

Stack manipulation instructions (all types)	POP, SWAP, YANK, DUP, STACKDEPTH, SHOVE, FLUSH, =
Math (INTEGER and FLOAT)	+, −, /, *, >, <, MIN, MAX
Logic (BOOLEAN)	AND, OR, NOT, FROMINTEGER
Code manipulation (CODE)	QUOTE, CAR, CDR, CONS, INSERT, LENGTH, LIST, MEMBER, NTH, EXTRACT
Control manipulation (CODE and EXEC)	DO*, DO*COUNT, DO*RANGE, DO*TIMES, IF

A Simple Push Program

```
( 2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE
BOOLEAN.OR )
```

Resulting stacks:

```
BOOLEAN STACK: ( TRUE )

CODE STACK: ( ( 2 3 INTEGER.* 4.1 5.2
              FLOAT.+ TRUE FALSE BOOLEAN.OR
              ) )

FLOAT STACK: ( 9.3 )

INTEGER STACK: ( 6 )
```

A Scrambled Program

```
( 5 1.23 INTEGER.+ ( 4 ) INTEGER.- 5.67
FLOAT.* )
```

Resulting stacks:

```
CODE STACK: ( ( 5 1.23 INTEGER.+ ( 4 )
              INTEGER.- 5.67 FLOAT.* ) )

FLOAT STACK: ( 6.9741 )

INTEGER STACK: ( 1 )
```

Quotation

```
( CODE.QUOTE
  ( INTEGER.DUP INTEGER.+ )
  CODE.DO
)
```

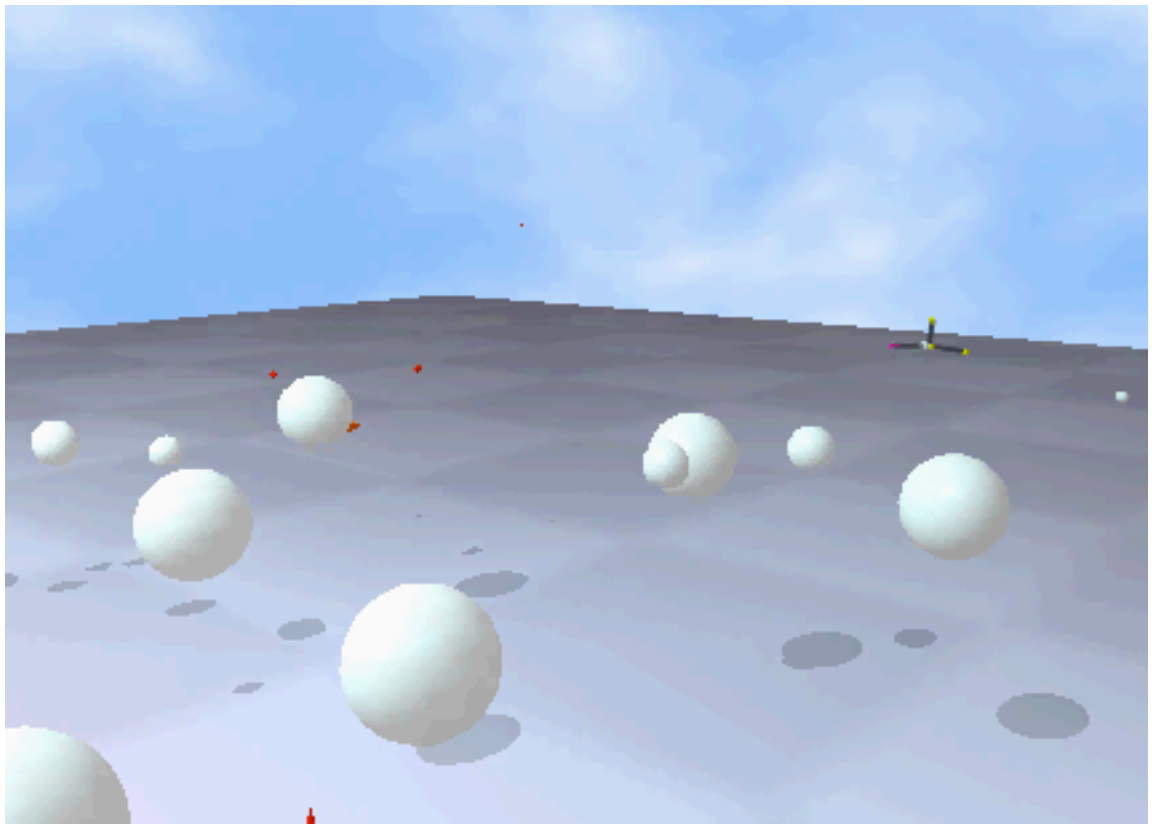
PushGP

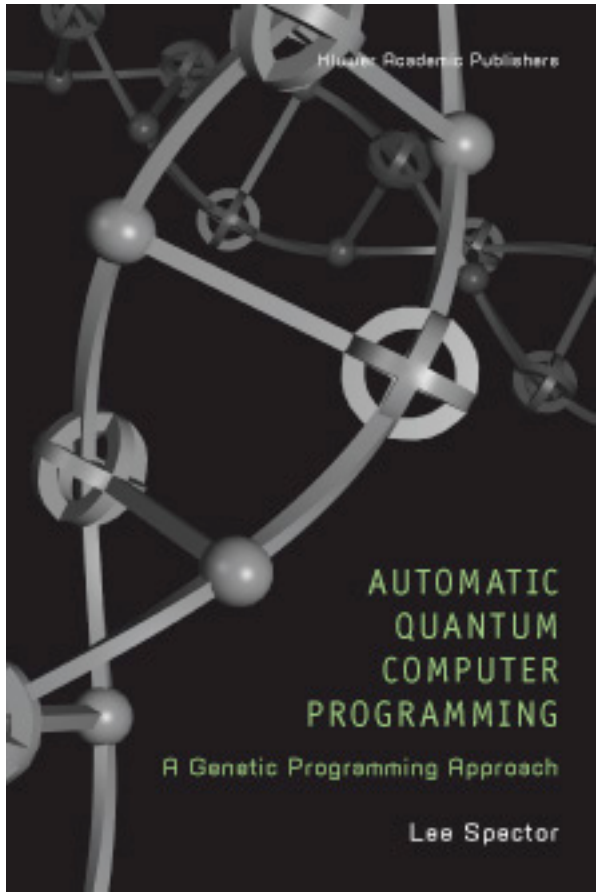
- As generic and simple a GP system you can imagine, except that it represents evolving programs in Push.
- A few minor differences from standard tree-based GP since Push programs aren't exactly S-expression trees (no function/arg place distinction).
- A few minor enhancements (e.g. size-fair genetic operators).
- Available in C++, Lisp, and as a Breve plugin (http://hampshire.edu/lspector/push.html).

Push's Glorious Past

- Superior scale-up on certain standard GP problems.
- Discovery of novel solutions to multi-type problems.
- Demonstration of automatic modular architecture evolution.
- Use in development of general GP techniques such as size-fair operators and trivial geography.
- Application to quantum computing problems, producing several human-competitive results.
- Application to many and various problems in the evolution of multi-agent systems.
- Use in development of novel self-organizing evolutionary computation paradigms.

Push's Glorious Past





See http://hampshire.edu/lspector/push.html

Push(1&2) Semantics

- To execute program P:
  1. If P is an INSTRUCTION: execute P (accessing whatever stacks are required).
  2. If P is a LITERAL: push P onto the appropriate stack.
  3. If P is a LIST: recursively execute each subprogram in P.

Push(3) Semantics

- To execute program P:

  1. Push P onto the `EXEC` stack.
  2. While the `EXEC` stack is not empty, pop and process the top element of the `EXEC` stack, E:
     (a) If E is an instruction: execute E (accessing whatever stacks are required).
     (b) If E is a literal: push E onto the appropriate stack.
     (c) If E is a list: push each element of E onto the `EXEC` stack, in reverse order.

What's the difference?

- Re-entrance *(∴ broader applicability)*
- Expressive parsimony for iteration, naming, and other language features. *(∴ enhanced evolvability of programs using these features)*
- Combinators and novel control regimes via explicit EXEC stack manipulation. *(∴ enhanced evolvability of control)*

Naming a Subroutine

```
Push2:
( TIMES2 CODE.QUOTE ( 2 INTEGER.* )  CODE.SET )

Push3:
( TIMES2 EXEC.DEFINE ( 2 INTEGER.* ) )
```

Calling a Subroutine

```
Push2:
( TIMES2 CODE.GET CODE.DO )

Push3:
( TIMES2 )
```

Iterators

- `CODE.DO*TIMES, CODE.DO*COUNT, CODE.DO*RANGE`
- `EXEC.DO*TIMES, EXEC.DO*COUNT, EXEC.DO*RANGE`
- Additional forms of iteration are supported through code manipulation (e.g. via `CODE.DUP CODE.APPEND CODE.DO`)

CODE vs. EXEC Iterators

- The following produce the same net effects:

```
( 5 CODE.QUOTE ( INTEGER.+ ) CODE.DO*COUNT )
( 5 EXEC.DO*COUNT ( INTEGER.+ ) )
```

Combinators

- Standard *K*, *S*, and *Y* combinators:
  - `EXEC.K` removes the second item from the `EXEC` stack.
  - `EXEC.S` pops three items (call them `A`, `B`, and `C`) and then pushes `(B C)`, `C`, and then `A`.
  - `EXEC.Y` inserts `(EXEC.Y` *T*`)` under the top item (*T*).
- A *Y*-based "while" loop:

```
( EXEC.Y
  ( <BODY/CONDITION> EXEC.IF
  ( ) EXEC.POP ) )
```

Evolved List Reverse

- Input is list of integers on the CODE stack.
- PushGP produced the following general solution:

```
(CODE.DO*TIMES (CODE.DO* CODE.LIST (((INTEGER.STACKDEPTH EXEC.DO*TIMES)
(BOOLEAN.YANKDUP CODE.FROMINTEGER)) CODE.FROMINTEGER INTEGER.SWAP)
(CODE.YANKDUP INTEGER.% (BOOLEAN.AND) CODE.STACKDEPTH EXEC.DO*TIMES))
(CODE.CONS) (BOOLEAN.SHOVE (CODE.EXTRACT EXEC.S (EXEC.FLUSH CODE.IF
BOOLEAN.YANK (CODE.FROMINTEGER CODE.ATOM (CODE.SWAP BOOLEAN.SHOVE
(INTEGER.MAX) (CODE.QUOTE CODE.APPEND CODE.IF)) ((CODE.ATOM CODE.SHOVE
EXEC.POP (CODE.DO*TIMES BOOLEAN.SHOVE) INTEGER.ROT) (INTEGER.>
BOOLEAN.AND CODE.DO* INTEGER.ROT) CODE.CONS INTEGER.ROT ((CODE.NTHCDR)
INTEGER.ROT BOOLEAN.DUP) INTEGER.SHOVE (CODE.FROMNAME (CODE.CONS
CODE.FROMINTEGER)))) CODE.LENGTH INTEGER.MAX EXEC.Y)) (BOOLEAN.=
(CODE.QUOTE INTEGER.SWAP) CODE.POP) INTEGER.FLUSH))
```

Evolved List Reverse (2)

- The evolved general solution simplifies to:
  `(CODE.DO* INTEGER.STACKDEPTH EXEC.DO*TIMES CODE.FROMINTEGER CODE.STACKDEPTH EXEC.DO*TIMES CODE.CONS)`
- This works by executing the input list, then moving all of the integers individually to the CODE stack, then building the reversed list.

Evolved Factorial

Two simplified evolved general solutions:

```
(1 EXEC.DO*RANGE INTEGER.*)
```

Runs a loop that just multiplies all of the loop counter values.

```
(INTEGER.* INTEGER.STACKDEPTH CODE.DO*RANGE
INTEGER.MAX)
```

*Recursively executes the whole program, which is on the CODE stack; INTEGER.STACKDEPTH produces the 1 for the loop index lower bound, and INTEGER.MAX pulls each product out from under each INTEGER.STACKDEPTH; only the first CODE.DO*RANGE is executed in a context with code on the CODE stack.*

Evolved Fibonacci

Two simplified evolved general solutions:

```
(EXEC.DO*TIMES (CODE.LENGTH EXEC.S)
INTEGER.STACKDEPTH CODE.YANKDUP)
```

Builds an expression with Fibonacci(input) instances of INTEGER.STACKDEPTH on the EXEC stack, then executes them all.

```
(EXEC.DO*COUNT EXEC.S CODE.QUOTE NAME.=
CODE.DO*COUNT CODE.YANKDUP CODE.DO*COUNT
CODE.CONS CODE.STACKDEPTH)
```

Builds an expression with Fibonacci(input) instances of NAME.= on the CODE stack, then executes CODE.STACKDEPTH.

Evolved Even Parity

- Input is list of Boolean values on the `CODE` stack.
- Goal is a *general* solution that solves the problem for any number of inputs.

Evolved Even Parity (2)

Two simplified evolved general solutions:

```
(CODE.DO* EXEC.Y BOOLEAN.=)
```

Terminates only when execution limit is reached; works only for even number of inputs.

```
(((CODE.POP CODE.DO BOOLEAN.STACKDEPTH)
(EXEC.DO*TIMES) (BOOLEAN.= BOOLEAN.NOT))))
```

100% correct, general, terminating; see paper for explanation.

Evolved Expt(2,*n*)

- Normally an easy problem, but here we attempted to evolve solutions without iteration instructions.
- The following evolved solution uses novel evolved control structures (but does not generalize beyond the training cases, *n*=1-8):

```
((INTEGER.DUP EXEC.YANKDUP EXEC.FLUSH 2
CODE.LENGTH) 8 (2 8  INTEGER.* INTEGER.DUP)
(EXEC.YANK 8 INTEGER.* ((CODE.IF (EXEC.ROT))
BOOLEAN.DEFINE EXEC.YANK)))
```

Evolved Sort

- Input/output in an external data structure accessed with `INTEGER.LIST-SWAP, INTEGER.LIST-LENGTH, INTEGER.LIST-GET, INTEGER.LIST-COMPARE.`
- Simplified evolved general solution that makes n*(n-1) comparisons:

```
(INTEGER.LIST-LENGTH INTEGER.SHOVE
INTEGER.STACKDEPTH CODE.DO*RANGE
INTEGER.YANKDUP INTEGER.DUP EXEC.DO*COUNT
INTEGER.LIST-COMPARE INTEGER.LIST-SWAP)
```

Conclusions

- The Push3 EXEC stack supports powerful and parsimonious control regimes through explicit manipulation of the stack of expressions that are queued for execution.
- These control regimes include standard iteration, several forms of recursion based on code manipulation, combinators, named subroutines, and less conventional strategies.
- PushGP can routinely produce solutions that incorporate a range of these control regimes.
- Examples were provided here for reversing and sorting lists and for computing factorials, Fibonacci numbers, powers of 2, and parity.

Available Technologies

- Push 3 (in Lisp and C++), PushGP 3 (Lisp and C++), and Pushpop (Lisp): http://hampshire.edu/lspector/push.html.
- Breve (Linux, Mac OS X, Windows, includes Push3): http://www.spiderland.org/breve.
- SwarmEvolve 2.0 (uses Push 2): http://hampshire.edu/lspector/gecco2003-collective.html.
- Related technology: http://hampshire.edu/lspector/code.html.

Thanks

- **Collaborators**: Raphael Crawford-Marks, Chris Perry, Alan Robinson.
- **Grants**: National Science Foundation Grants No. 0308540 and No. 0216344, Defense Advanced Research Projects Agency and Air Force Research Laboratory, agreement number F30502-00-2-0611.