Expressive Genetic Programming: Concepts and Applications

A Tutorial at the Genetic and Evolutionary Computation Conference (GECCO-2017, Berlin)

This is an updated version of the tutorial of the same name from GECCO-2016

Lee Spector
School of Cognitive Science
Hampshire College
Amherst, MA USA
lspector@hampshire.edu

Nicholas Freitag McPhee
Division of Science & Mathematics
University of Minnesota, Morris
Morris, Minnesota USA
mcphee@morris.umn.edu

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.
Lee Spector is a Professor of Computer Science in the School of Cognitive Science at Hampshire College in Amherst, Massachusetts, and an adjunct professor in the Department of Computer Science at the University of Massachusetts, Amherst. He received a B.A. in Philosophy from Oberlin College in 1984 and a Ph.D. from the Department of Computer Science at the University of Maryland in 1992. His areas of teaching and research include genetic and evolutionary computation, quantum computation, and a variety of intersections between computer science, cognitive science, evolutionary biology, and the arts. He is the Editor-in-Chief of the journal *Genetic Programming and Evolvable Machines* (published by Springer) and a member of the editorial board of *Evolutionary Computation* (published by MIT Press). He is also a member of the SIGEVO executive committee and he was named a Fellow of the International Society for Genetic and Evolutionary Computation.

More info: [http://hampshire.edu/lspector](http://hampshire.edu/lspector)
Nicholas Freitag McPhee is a Professor of Computer Science in the Division of Science and Mathematics at the University of Minnesota, Morris, in Morris, Minnesota. He received a B.A. in Mathematics from Reed College in 1986 and a Ph.D. from the Department of Computer Sciences at the University of Texas at Austin in 1993. His areas of teaching and research include genetic and evolutionary computation, machine learning, software development, and, when circumstances allow, photography and American roots music. He is a co-author of *A field guide to genetic programming*, and a member of the editorial board of the journal *Genetic Programming and Evolvable Machines* (published by Springer).

More info: [http://facultypages.morris.umn.edu/~mcphee](http://facultypages.morris.umn.edu/~mcphee)
The language used to express evolving programs can significantly impact the dynamics and problem-solving capabilities of a genetic programming system. In GP these impacts are driven by far more than the absolute computational power of the languages used; just because a computation is theoretically possible in a language, doesn’t mean it’s readily discoverable or leveraged by evolution. Highly expressive languages can facilitate the evolution of programs for any computable function using, as appropriate, multiple data types, evolved subroutines, evolved control structures, evolved data structures, and evolved modular program and data architectures. In some cases expressive languages can even support the evolution of programs that express methods for their own reproduction and variation (and hence for the evolution of their offspring).

This tutorial will present a range of approaches that have been taken for evolving programs in expressive programming languages. We will then provide a detailed introduction to the Push programming language, which was designed specifically for expressiveness in genetic programming systems. Push programs are syntactically unconstrained but can nonetheless make use of multiple data types and express arbitrary control structures, supporting the evolution of complex, modular programs in a particularly simple and flexible way.
Scope and Content (2)

Interleaved with our description of the Push language will be demonstrations of the use of analytical tools such as graph databases to explore ways in which evolved Push programs are actually taking advantage of the language’s expressive features. We will illustrate, for example, the effective use of multiple types and type-appropriate functions, the evolution and modification of code blocks and looping/recursive constructs, and the ability of Push programs to handle multiple, potentially related tasks.

We’ll conclude with a brief review of over a decade of Push-based research, including the production of human-competitive results, along with recent enhancements to Push that are intended to support the evolution of complex and robust software systems.
Course Agenda

- Genetic programming
- Expressiveness and evolvability
- The Push programming language and PushGP
- Demos
- Results
- Ongoing Research
Evolutionary Computation

Random Generation → Assessment

Assessment → Solution

Assessment → Selection

Selection → Variation
Genetic Programming

- Evolutionary computing to produce executable computer programs
- Programs are assessed by executing them
- Automatic programming; producing software
Program Representations

- Lisp-style symbolic expressions (Koza, ...)
- Purely functional/lambda expressions (Walsh, Yu, ...)
- Linear sequences of machine/byte code (Nordin et al., ...)
- Artificial assembly-like languages (Ray, Adami, ...)
- Stack-based languages (Perkis, Spector, Stoffel, Tchernev, ...)
- Graph-structured programs (Teller, Globus, ...)
- Object hierarchies (Bruce, Abbott, Schmutter, Lucas, ...)
- Fuzzy rule systems (Tunstel, Jamshidi, ...)
- Logic programs (Osborn, Charif, Lamas, Dubossarsky, ...)
- Strings, grammar-mapped to arbitrary languages (O’Neill, Ryan, McKay, ...)
Mutating Lisp (1)

\[
\begin{align*}
( + ( * X Y ) \\
( + 4 ( - Z 23 ) ) )
\end{align*}
\]
Mutating Lisp (2)

(+ (* X Y)
   (+ 4 (− Z 23))))
Mutating Lisp (3)

\[
\begin{align*}
&\left( + \left( - \left( + \ 2 \ 2 \right) \ z \right) \\
&\quad \left( + \ 4 \left( - \ z \ 23 \right) \right) \right)
\end{align*}
\]
Recombining Lisp

\[ (+ \ (* \ X \ Y) \\
\quad (+ \ 4 \ (- \ Z \ 23)) ) \]

\[ (- \ (* \ 17 \ (+ \ 2 \ X)) \]
\[ (* \ (- \ (* \ 2 \ Z) \ 1) \]
\[ (+ \ 14 \ (/ \ Y \ X)) ) ) \]

\[ (+ \ (- \ (* \ 2 \ Z) \ 1) \]
\[ (+ \ 4 \ (- \ Z \ 23)) ) \]
Symbolic Regression

• A simple example

• Given a set of data points, evolve a program that produces y from x.

• Primordial ooze: +, -, *, %, x, 0.1

• Fitness = error (smaller is better)
Genetic Programming Parameters

- Maximum number of Generations: 51
- Size of Population: 1000
- Maximum depth of new individuals: 6
- Maximum depth of new subtrees for mutants: 4
- Maximum depth of individuals after crossover: 17
- Fitness-proportionate reproduction fraction: 0.1
- Crossover at any point fraction: 0.3
- Crossover at function points fraction: 0.5
- Selection method: FITNESS-PROPORTIONATE
- Generation method: RAMPED-HALF-AND-HALF
- Randomizer seed: 1.2
Evolving \( y = x^3 - 0.2 \)
Best Program, Gen 0

\[-(\% (* 0.1 (* X X)) (-(\% 0.1 0.1) (* X X)))) 0.1\]
Best Program, Gen 5

$$(- (* (* (% X 0.1) (* 0.1 X)) (- X (% 0.1 X)))) 0.1)$$
Best Program, Gen 12

(+ (- (- 0.1
    (- 0.1
      (- (* X X)
        (+ 0.1
          (- 0.1
            (* 0.1
              0.1)))
        ))
      ))
(* X
  (* (% 0.1
  (% (* (* (- 0.1 0.1)
        (+ X
          (- 0.1 0.1)))
      X)
    (+ X (+ (- X 0.1)
       (* X X))))))
(+ 0.1 (+ 0.1 X))))
(* X X))
Best Program, Gen 22

\((- (- (* X (* X X))) 0.1) 0.1\)
Expressiveness

- Turing machine tables
- Lambda calculus expressions
- Partial recursive functions
- Register machine programs
- Assembly language programs
- etc.
Evolvability

• The fact that a computation can be expressed in a formalism does not imply that it will be produced in by a human programmer or by evolution.

• Research program:
  1. Provide expressiveness
  2. Study/enhance evolvability
Data/Control Structure

- Data abstraction and organization
  Data types, variables, data structures, name spaces, ...

- Control abstraction and organization
  Conditionals, loops, modules, threads, ...
Evolving Structure (1)

• Specialize GP techniques to directly support human programming language abstractions

• Examples:
  1. Structured data via strongly typed GP
  2. Structured control via automatically defined functions
Strongly Typed GP (Montana)

- Primitives annotated with types
- Constrained code generation
- Constrained mutation and recombination
Automatically Defined Functions (Koza)

• All programs in the population have the same, pre-specified architecture

• Genetic operators respect that architecture

• Significant implementation costs

• Significant pre-specification

• Architecture-altering operations: more power and higher costs
Evolving Structure (2)

• Specialize GP techniques to **indirectly** support human programming language abstractions

• Map from unstructured genomes to programs in languages that support abstraction (e.g. via grammars)
Evolving Structure (3)

- Evolve programs in a minimal-syntax language that nonetheless supports a full range of data and control abstractions

- For example: orchestrate data flows via stacks, not via syntax

- Minimal syntax + maximal, flexible semantics

- Push
Push (1)

- Designed for program evolution
- Data flows via stacks, not syntax
- One stack per type:
  integer, float, boolean, string, code, exec, vector, ...
- program → instruction | literal | ( program* )
- Turing complete, with rich data and control structures
Push (2)

• Missing argument? NOOP

• Argument order: Generally reflect expected order from text of canonical usage

• Time/step limits ensure termination, with results available from stacks in all cases

• PushGP is a GP system that evolves Push programs

• http://pushlanguage.org
Push (3)

- It is simple to write a minimal Push system in any language; instructions can then be added incrementally.

- Implementations in C++, C#, Clojure, Common Lisp, Java, Javascript, Python, Racket, Ruby, Scala, Scheme, Swift, ...

- Most examples in this presentation use Clojush, a Push/PushGP implementation in Clojure.

- A recent addition that might be more approachable is Pysh, a Push/PushGP implementation in Python.
Clojush

Lee Spector (lspector@hampshire.edu), started 20100227 See version history. Older version history is in old-version-history.txt.

This is the README file accompanying Clojush, an implementation of the Push programming language and the PushGP genetic programming system in the Clojure programming language. Among other features this implementation takes advantage of Clojure's facilities for multi-core concurrency.

Availability

https://github.com/lspector/Clojush/

Requirements

To use this code you must have a Clojure programming environment; see http://clojure.org/. The current version of Clojush requires Clojure 1.7.0.

Clojure is available for most OS platforms. A good starting point for obtaining and using Clojure.

Quickstart

Using Leiningen you can run an example from the OS command line (in the Clojush directory) with a call like:

```
lein run clojush.problems.demos.simple-regression
```
Pysh

Push Genetic Programming in Python. For the most complete documentation, refer to the ReadTheDocs.


Push Genetic Programming

Push is a programming language that plays nice with evolutionary computing / genetic programming. It is a stack-based language that features 1 stack per data type, including code. Programs are represented by lists of instructions, which modify the values on the stacks. Instructions are executed in order.

More information about PushGP can be found on the Push Redux and the Push Homepage.

For the most cutting edge PushGP framework, see the Clojure implementation called Clojush.

Installing Pysh

Pysh is compatible with python 2.7.x and 3.5.x

Install from pip

Coming with the first beta release of pyshgp. Check the Roadmap to get a sense of how far off this is.
Why Push?

• Expressive: data types, data structures, variables, conditionals, loops, recursion, modules, ...

• Elegant: minimal syntax and a simple, stack-based execution architecture

• Supports several forms of meta-evolution

• Evolvable? At minimum, supports investigation of relations between expressiveness and evolvability
Plush

- Linear genomes for Push programs
- Facilitates useful placement of code blocks
- Permits uniform linear genetic operators
- Allows for epigenetic hill-climbing
Push Program Execution

• Push the program onto the exec stack.

• While exec isn't empty, pop and do the top:
  • If it's an instruction, execute it.
  • If it's a literal, push it onto the appropriate stack.
  • If it's a list, push its elements back onto the exec stack one at a time.
Instructions Implemented for Most Types

- `<type>_dup`
- `<type>_empty`
- `<type>_eq`
- `<type>_flush`
- `<type>_pop`
- `<type>_rot`
- `<type>_shove`
- `<type>_stackdepth`
- `<type>_swap`
- `<type>_yank`
- `<type>_yankdup`
Selected Integer Instructions

integer_add integer_dec integer_div
integer_gt integer_fromstring integer_min
integer_mult integer_rand

Selected Boolean Instructions

boolean_and boolean_xor boolean_frominteger

Selected String Instructions

string_concat string_contains string_length
string_removechar string_replacechar
Selected Exec Instructions

Conditionals:
  exec_if exec_when

General loops:
  exec_do*while

“For” loops:
  exec_do*range exec_do*times

Looping over structures:
  exec_do*vector_integer exec_string_iterate

Combinators:
  exec_k exec_y exec_s
Many More Types and Instructions
;; https://github.com/lsp Hector/Clojush/

=> (run-push '(1 2 integer_add) (make-push-state))

:exec ((1 2 integer_add))
:integer ()

:exec (1 2 integer_add)
:integer ()

:exec (2 integer_add)
:integer (1)

:exec (integer_add)
:integer (2 1)

:exec ()
:integer (3)
=> (run-push '(2 3 integer_mult 4.1 5.2 float_add
        true false boolean_or)
    (make-push-state))

:exec ()
:integer (6)
:float (9.3)
:boolean (true)

In other words
  • Put $2 \times 3$ on the integer stack
  • Put $4.1 + 5.2$ on the float stack
  • Put $true \lor false$ on the boolean stack
Same as before, but
  • Several operations (e.g., boolean_and) become NOOPs
  • Interleaved operations
=> (run-push
   '(4.0 exec_dup (3.13 float_mult) 10.0 float_div)
   (make-push-state))

:exec ((4.0 exec_dup (3.13 float_mult) 10.0 float_div))
:float ()

:exec (4.0 exec_dup (3.13 float_mult) 10.0 float_div)
:float ()

:exec (exec_dup (3.13 float_mult) 10.0 float_div)
:float (4.0)

:exec((3.13 float_mult) (3.13 float_mult) 10.0 float_div)
:float (4.0)

...

Computes 4.0 × 3.13 × 3.13 / 10.0

:exec ()
:float (3.91876)
=> (run-push '(1 8 exec_do*range integer_mult)
   (make-push-state))

:integer (40320)

Computes 8! in a fairly “human” way
A less “obvious” evolved calculation of 8!

```lisp
=> (run-push '(code_quote
    (code_quote (integer_pop 1)
        (code_quote (code_dup integer_dup
            1 integer_sub code_do
            integer_mult)
        integer_dup 2 integer_lt code_if)
    code_dup
8
code_do)
(make-push-state))

:code ((code_quote (integer_pop 1) code_quote (code_dup
    integer_dup 1 integer_sub code_do integer_mult)
    integer_dup 2 integer_lt code_if))

:integer (40320)
```
(run-push '((0 true exec_while (1 integer_add true)) (make-push-state)))

:exec (1 integer_add true exec_while (1 integer_add true))
:integer (199)
:termination :abnormal

- An infinite loop
- Terminated by eval limit
- Result taken from appropriate stack(s) upon termination
=> (run-push '(in1 in1 float_mult 3.141592 float_mult)
  (push-item 2.5 :input (make-push-state)))

:float (19.63495)
:input (2.5)

Computes the area of a circle with the given radius: 3.141592 × in1 × in1
Set up run for target $x^3 - 2x^2 - x$

\[(\text{pushgp})\]

\[
{:\text{error-function} \ (fn \ \text{[program]} \ (\text{vec} \ (\text{for} \ [\text{input} \ (\text{range} \ 10)] \ (\text{let} \ [\text{output} \ (-\to>) \ (\text{make-push-state}) \ (\text{push-item} \ \text{input} :\text{input}) \ (\text{run-push} \ \text{program}) \ (\text{top-item} :\text{integer})))]] \ (\text{if} \ \text{(number? output)} \ (\text{Math/abs} \ (\text{float} \ (- \ \text{output} \ (- \ (* \ \text{input} \ \text{input} \ \text{input}) \ (* \ 2 \ \text{input} \ \text{input})))) \ 1000)))))\]

{:atom-generators (list (fn [] (lrand-int 10)) 'in1 'integer_div 'integer_mult 'integer_add 'integer_sub) :population-size 100} \]
Auto-Simplification

- Loop:
  - Make it randomly simpler
  - Keep simpler if as good or better; otherwise revert

- GECCO-2014 poster: efficiently and reliably reduces the size of the evolved programs

- GECCO-2014 student paper: used as genetic operator

- GECCO-2017 *GP best paper nominee*: improves generalization
SUCCESS at generation 29
Successful program: (5 4 in1 integer_sub in1 integer_mult
integer_sub integer_div integer_mult 6 integer_sub integer_add
in1 5 integer_sub integer_add in1 5 integer_add integer_add
integer_mult in1 integer_mult)
Errors: [0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0]
Total error: 0.0
History: null
Size: 24

Auto-simplifying with starting size: 24
...

step: 1000
program: (5 4 in1 integer_sub in1 integer_mult integer_sub 6
integer_sub in1 5 integer_sub integer_add in1 5 integer_add
integer_add in1 integer_mult)
errors: [0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0]
total: 0.0
size: 20
SUCCESS at generation 20

Successful program: (boolean_and boolean_shove exec_do*count (exec_swap (integer_empty char_yank boolean_or integer_fromboolean \space \newline) (exec_dup (char_yank char_iswhitespace string_butlast in1) string_empty boolean_fromboolean string_substring) exec_do*times (integer_empty string_dup) string_replacechar print_string string_rot print_char integer_fromboolean string_length integer_eq string_last integer_yankdup) string_swap string_containschar "Wx{ " exec_stackdepth char_empty integer_swap integer_rot string_last boolean_swap integer_yankdup string_swap string_containschar "Wx{ " exec_stackdepth char_empty integer_swap integer_rot integer_fromstring string_pop string_shove char_eq char_empty integer_swap integer_rot integer_fromstring string_pop string_shove char_rot integer_stackdepth integer_min char_yankdup char_eq char_empty tagged_349 exec_yank string_rot exec_dup (boolean_eq string_removechar exec_s (exec_dup (boolean_eq exec_rot (exec_s (string_eq string_fromboolean exec_noop char_eq) () (string_butlast) integer_pop) (char_eq char_empty) (integer_swap integer_yankdup char_isletter integer_gt integer_yankdup) exec_when (string_emptystring string_nth exec_do*range \space integer_yankdup string_dup exec_shove) (integer_swap string_removechar exec_yank string_dup exec_empty) char_eq exec_do*times (tagged_349 boolean_pop exec_when (string_removechar integer_swap integer_inc in1 boolean_shove boolean_swap char_isletter integer_gt string_last) integer_mult string_last string_parse_to_chars boolean_frominteger boolean_yank exec_when (string_nth exec_do*range (\space integer_yankdup string_dup exec_shove) (integer_swap string_removechar exec_yank exec_while (boolean_or)) char_isdigit boolean_swap char_isletter) integer_gt integer_yankdup integer_mult string_last string_parse_to_chars boolean_frominteger char_isletter exec_when (string_nth exec_do*range (\space integer_yankdup string_dup exec_shove) (integer_swap string_removechar exec_yank integer_yank integer_mult integer_inc in1 boolean_shove boolean_swap char_isletter integer_gt string_last) boolean_invert_second_then_and exec_empty string_rot) boolean_rot char_iswhitespace integer_yank string_conjchar boolean_dup) integer_add char_dup string_length integer_fromchar string_split char_isdigit boolean_swap boolean_eq char_isdigit exec_shove (boolean_invert_second_then_and_string_empty string_conjchar string_shove) string_fromchar boolean_not string_stackdepth exec_y (integer_empty exec_do*range (in1 string_replace)))))))) () ()

Errors: 

Total error: 0.0

Size: 231

Auto-simplifying with starting size: 231

... 

step: 5000 

program: (\space \newline in1 string_replacechar print_string "Wx{ " string_last in1 string_removechar string_length)

errors: 

Total error: 0.0

Size: 11
Problems Solved by PushGP in the GECCO-2005 Paper on Push3

- Reversing a list
- Factorial (many algorithms)
- Fibonacci (many algorithms)
- Parity (any size input)
- Exponentiation
- Sorting
Figure 8.7. A gate array diagram for an evolved version of Grover’s database search algorithm for a 4-item database. The full gate array is shown at the top, with $M_1$ and $M_2$ standing for the smaller gate arrays shown at the bottom. A diagonal line through a gate symbol indicates that the matrix for the gate is transposed. The “f” gate is the oracle.
Genetic Programming for Finite Algebras

Lee Spector
Cognitive Science
Hampshire College
Amherst, MA 01002
lspector@hampshire.edu

David M. Clark
Mathematics
SUNY New Paltz
New Paltz, NY 12561
clarkd@newpaltz.edu

Ian Lindsay
Hampshire College
Amherst, MA 01002
iml04@hampshire.edu

Bradford Barr
Hampshire College
Amherst, MA 01002
bradford.barr@gmail.com

Jon Klein
Hampshire College
Amherst, MA 01002
jk@artificial.com

Humies 2008
GOLD MEDAL
29 Software Synthesis Benchmarks

- Number IO, Small or Large, For Loop Index, Compare String Lengths, Double Letters, **Collatz Numbers**, Replace Space with Newline, **String Differences**, Even Squares, **Wallis Pi**, String Lengths Backwards, Last Index of Zero, Vector Average, Count Odds, Mirror Image, **Super Anagrams**, Sum of Squares, Vectors Summed, X-Word Lines, **Pig Latin**, Negative to Zero, Scrabble Score, **Word Stats**, Checksum, Digits, Grade, Median, Smallest, Syllables

- PushGP has solved all of these except for the ones in blue

- Presented in a GECCO-2015 GP track paper
Example: Checksum

Multiple types, looping, and code blocks

Simplified solution:

("Check sum is " print_string
   inl 64 exec_string_iterate (integer_fromchar integer_add)
   64 integer_mod
   \space integer_fromchar integer_add char_frominteger
   print_char)
Example: Checksum

Multiple types, looping, and code blocks

Simplified solution:

("Check sum is " print_string
  inl 64 exec_string_iterate (integer_fromchar integer_add)
  64 integer_mod
  \space integer_fromchar integer_add char_frominteger
  print_char)

First: Print out the header
Example: Checksum

Multiple types, looping, and code blocks

Simplified solution:

("Check sum is " print_string

\[\text{inl 64 exec_string_iterate (integer_fromchar integer_add)}\]

64 integer_mod

\[\text{space integer_fromchar integer_add char_frominteger}\]

\text{print_char})

Second: Convert each character to an integer, sum, and add to 64.
Example: Checksum

Multiple types, looping, and code blocks

Simplified solution:

"Check sum is " print_string

inl 64 exec_string_iterate (integer_fromchar integer_add)

64 integer_mod

\space integer_fromchar integer_add char_frominteger

print_char)

Third: Mod result by 64
Example: Checksum

Multiple types, looping, and code blocks

Simplified solution:

("Check sum is " print_string
    inl 64 exec_string_iterate (integer_fromchar integer_add)
    64 integer_mod
    \space integer_fromchar integer_add char_frominteger
    print_char)

Third: Add modulus result to 32 and convert to char
Example: Checksum

Multiple types, looping, and code blocks

Simplified solution:

"Check sum is " print_string
  inl 64 exec_string_iterate (integer_fromchar integer_add)
  64 integer_mod
  \space integer_fromchar integer_add char_frominteger
  print_char)

Fourth: Print resulting char
Example: Replace Space With Newline

Multiple types, looping, multiple tasks

Simplified solution:

```
\space char_dup exec_dup in1
    \newline string_replacechar print_string
    string_removechar string_length)
```
Example: Replace Space With Newline

Multiple types, looping, multiple tasks

Simplified solution:

```
(space char dup exec dup in1
newline string_replacechar print_string
string_removechar string_length)
```

First: Duplicate space character and input string for use in both tasks
Example: Replace Space With Newline

Multiple types, looping, multiple tasks

Simplified solution:

```
(spaced char_dup exec_dup in1
    newline string_replacechar print_string
    string_removechar string_length)
```

Second: Replace spaces with newlines and print
Example: Replace Space With Newline

Multiple types, looping, multiple tasks

Simplified solution:

(
        \space char_dup exec_dup in1
        \newline string_replacechar print_string
        string_removechar string_length
)

Third: Remove all spaces from second copy of input, and push length of result on integer stack for return
DEMO
Autoconstructive Evolution

- Individual programs make their own children
- Hence they control their own mutation and recombination methods and rates
- The machinery of reproduction and diversification (i.e., the machinery of evolution) evolves
- In Push, experimentation with autoconconstructive evolution is easy and natural
Variation in Genetic Programming

Program → Mutation → Program

Program → Crossover → Program

Program → Program
Autoconstruction

Program

Execute!

Program
Autoconstruction

- It works!
- Evolved solutions to non-trivial problems (e.g., replace-space-with-newlines)
- Only known evolved solution to string-differences
- Slower and less consistent than human engineered operators, but much more work to do
- More to explore
Expressiveness and Assessment

- Expressive languages ease representation of programs that over-fit training sets
- Expressive languages ease representation of programs that work only on subsets of training sets
- Lexicase selection may help: Select parents by starting with a pool of candidates and then filtering by performance on individual fitness cases, considered one at a time in random order
Evolving Modular Programs with Push

- Via code manipulation on the code or exec stacks
- Via naming and a name stack
- Via tags, inspired by Holland's work on arbitrary identifiers with inexact matching
- Evolvability challenges abound! The relationship between evolvability, robustness, and modularity is complex
- Push facilitates experimentation in this space
Future Work

• Expression of variable scope and environments (implemented, but not yet studied systematically)

• Expression of concurrency and parallelism

• Applications for which expressiveness is likely to be essential, e.g. complete software applications, agents in complex, dynamic environments
Conclusions

• GP in expressive languages may allow for the evolution of complex software

• Minimal-syntax languages can be expressive, and GP systems that evolve programs in such languages can be unusually simple and powerful

• Push has produced significant successes and provides a rich framework for experimentation

• http://pushlanguage.org
Thanks

This material is based upon work supported by the National Science Foundation under Grant Nos. 1617087, 1129139 and 1331283. 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. Thanks also to Thomas Helmuth and the other members of the Hampshire College Computational Intelligence Lab, to Josiah Erikson for systems support, and to Hampshire College for support for the Hampshire College Institute for Computational Intelligence.
References

- The Push language website: http://pushlanguage.org


General References on Genetic Programming