Genetic Programming and Tag-Based Modularity

Lee Spector
based in part on work with
Brian Martin, Kyle Harrington & Thomas Helmuth

Cognitive Science, Hampshire College
Computer Science, Brandeis University
Computer Science, University of Massachusetts, Amherst
Outline

• Genetic Programming (GP)
• Push and PushGP
• Modularity in GP
• Tags and Tag-based modularity
• Results
Evolutionary Computation

Random Generation

Assessment

Selection  Variation

Solution
Traditional Genetic Algorithms

- Interesting dynamics
- Rarely solve interesting hard problems
Genetic Programming

- Evolutionary computing to produce executable computer programs.
- Programs are tested by executing them.
“Darwinian evolution is itself a designer worthy of significant respect, if not religious devotion.” *Boston Globe* OpEd, Aug 29, 2005

**And now, digital evolution**

By Lee Spector  |  August 29, 2005

RECENT developments in computer science provide new perspective on "intelligent design," the view that life's complexity could only have arisen through the hand of an intelligent designer. These developments show that complex and useful designs can indeed emerge from random Darwinian processes.
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, ...).
Mutating Lisp

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

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

\n
\[(+ (\(- (+ 2 2) Z\))
   (+ 4 (\(- Z 23\)))\)\]
Recombining Lisp

Parent 1: (+ (* X Y)
             (+ 4 (− Z 23)))

Parent 2: (− (* 17 (+ 2 X))
            (* (− (* 2 Z) 1)
                (+ 14 (/ Y X))))

Child 1: (+ (− (* 2 Z) 1)
             (+ 4 (− Z 23)))

Child 2: (− (* 17 (+ 2 X))
            (* (* X Y)
                (+ 14 (/ Y X))))
Symbolic Regression

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

Primordial ooze: $+, -, *, \%, x, 0.1$

Fitness $= \text{error} \ (\text{smaller is better})$
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

\(- \left( \% \left( * \ 0.1 \ (* \ X \ X) \right) \right) \left( - \left( \% \ 0.1 \ 0.1 \ (* \ X \ X) \right) \right) 0.1\)
Best Program, Gen 5

\(- (* (* (% X 0.1) (* 0.1 X)) (- X (% 0.1 X))) 0.1\)

- Target
- Generation 5
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))

\]

Target

Generation 12
Best Program, Gen 22

\[-(-(*X(*XX))0.1)0.1\]

Graph showing the comparison between the target and Generation 22.
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
Everybody’s Favorite Finite Algebra

Boolean algebra, \( B := \langle \{0, 1\}, \land, \lor, \neg \rangle \)

\[
\begin{array}{c|cc}
\land & 0 & 1 \\
\hline
0 & 0 & 0 \\
1 & 0 & 1 \\
\end{array}
\quad
\begin{array}{c|cc}
\lor & 0 & 1 \\
\hline
0 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\quad
\begin{array}{c|c}
\neg & \\
\hline
0 & 1 \\
1 & 0 \\
\end{array}
\]

*Primal:* every possible operation can be expressed by a term using only (and not even) \( \land, \lor, \text{and} \ \neg. \)
Bigger Finite Algebras

• Have applications in many areas of science, engineering, mathematics

• Can be *much* harder to analyze/understand

• Number of terms grows astronomically with size of underlying set
Goal

• Find terms that have certain special properties
• *Discriminator* terms, determine primality

\[ t^A(x, y, z) = \begin{cases} 
  x & \text{if } x \neq y \\
  z & \text{if } x = y 
\end{cases} \]

• *Mal'cev, majority, and Pixley* terms
• For decades there was no way to produce these terms in general, short of exhaustive search
• Current best methods produce enormous terms
## Algebras Explored

<table>
<thead>
<tr>
<th>A₁ *</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A₂ *</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A₃ *</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A₄ *</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A₅ *</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B₁ *</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Results

- Discriminators for $A_1, A_2, A_3, A_4, A_5$
- Mal’cev and majority terms for $B_1$
- Example Mal’cev term for $B_1$:

$$((((((x*(y*x))*x)*z)*(z*x))*(x*(z*(x*(z*y)))))*z)))*z)*z)*(z*(((x*(((z*z)*x)*(z*x)))*x)*(y)*(((y*(z*(z*y)))*((y*y)*x)*z))*(x*(((z*z)*x)*(z*(x*(z*y))))))$$
## Significance, Time

<table>
<thead>
<tr>
<th></th>
<th>Uninformed Search Expected Time (Trials)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>3 element algebras</strong></td>
<td></td>
</tr>
<tr>
<td>Mal’cev</td>
<td>5 seconds ($3^{15} \approx 10^7$)</td>
</tr>
<tr>
<td>Pixley/majority discriminator</td>
<td>1 hour ($3^{21} \approx 10^{10}$)</td>
</tr>
<tr>
<td></td>
<td>1 month ($3^{27} \approx 10^{13}$)</td>
</tr>
<tr>
<td><strong>4 element algebras</strong></td>
<td></td>
</tr>
<tr>
<td>Mal’cev</td>
<td>$10^3$ years ($4^{28} \approx 10^{17}$)</td>
</tr>
<tr>
<td>Pixley/majority discriminator</td>
<td>$10^{10}$ years ($4^{40} \approx 10^{24}$)</td>
</tr>
<tr>
<td></td>
<td>$10^{24}$ years ($4^{64} \approx 10^{38}$)</td>
</tr>
</tbody>
</table>
## Significance, Time

<table>
<thead>
<tr>
<th></th>
<th>Uninformed Search Expected Time (Trials)</th>
<th>GP Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>3 element algebras</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mal’cev</td>
<td>5 seconds ((3^{15} \approx 10^7))</td>
<td>1 minute</td>
</tr>
<tr>
<td>Pixley/majority</td>
<td>1 hour ((3^{21} \approx 10^{10}))</td>
<td>3 minutes</td>
</tr>
<tr>
<td>discriminator</td>
<td>1 month ((3^{27} \approx 10^{13}))</td>
<td>5 minutes</td>
</tr>
<tr>
<td><strong>4 element algebras</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mal’cev</td>
<td>10^3 years ((4^{28} \approx 10^{17}))</td>
<td>30 minutes</td>
</tr>
<tr>
<td>Pixley/majority</td>
<td>10^{10} years ((4^{40} \approx 10^{24}))</td>
<td>2 hours</td>
</tr>
<tr>
<td>discriminator</td>
<td>10^{24} years ((4^{64} \approx 10^{38}))</td>
<td>?</td>
</tr>
</tbody>
</table>
## Significance, Size

<table>
<thead>
<tr>
<th>Term Type</th>
<th>Primality Theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mal’cev</td>
<td>10,060, 219</td>
</tr>
<tr>
<td>Majority</td>
<td>6,847, 499</td>
</tr>
<tr>
<td>Pixley</td>
<td>1,257, 556, 499</td>
</tr>
<tr>
<td>Discriminator</td>
<td>12,575, 109</td>
</tr>
</tbody>
</table>

(for $A_1$)
Significance, Size

<table>
<thead>
<tr>
<th>Term Type</th>
<th>Primality Theorem</th>
<th>GP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mal’cev</td>
<td>10,060,219</td>
<td>12</td>
</tr>
<tr>
<td>Majority</td>
<td>6,847,499</td>
<td>49</td>
</tr>
<tr>
<td>Pixley</td>
<td>1,257,556,499</td>
<td>59</td>
</tr>
<tr>
<td>Discriminator</td>
<td>12,575,109</td>
<td>39</td>
</tr>
</tbody>
</table>

(for $A_i$)
Human Competitive?

• Rather: human-**WHOMPING**!

• *Outperforms* humans and *all other known methods* on significant problems, providing benefits of *several orders of magnitude* with respect to search speed and result size

• Because there were no prior methods for generating practical terms in practical amounts of time, GP has provided the first solution to a previously open problem in the field
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.
Expressive Languages

- Strongly typed genetic programming
- Automatically defined functions
- Automatically defined macros
- Architecture-altering operations
- Developmental genetic programming
Expressive Languages

• Strongly typed genetic programming
• Automatically defined functions
• Automatically defined macros
• Architecture-altering operations
• Developmental genetic programming

• Push provides all of the above and more, all without any mechanisms beyond the stack-based execution architecture
Why Push?

- Multiple data types
- User-defined procedures & functions
- User-defined macros & control structures
- User-defined representations
- Dynamic definition & redefinition
- All of the above provided without any mechanisms beyond the stack-based execution architecture
And I won’t even mention

- Automatic simplification
- Autoconstructive evolution
- Iterators and combinators
- Code self reference
- Ontogenetic programming
- etc. See http://hampshire.edu/lspector/push.html
Push

- Stack-based postfix language with one stack per type
- Types include: integer, float, Boolean, name, code, exec, vector, matrix, quantum gate, [add more as needed]
- Missing argument? NOOP
- Trivial syntax:
  program \(\rightarrow\) instruction \(\mid\) literal \(\mid\) ( program\(^*\) )
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.
( 2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR )

exec  code  bool  int  float

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

( 2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR )
<table>
<thead>
<tr>
<th>exec</th>
<th>code</th>
<th>bool</th>
<th>int</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>INTEGER.*</td>
<td>4.1</td>
<td>5.2</td>
</tr>
<tr>
<td>FLOAT.+</td>
<td>TRUE</td>
<td>FALSE</td>
<td>BOOLEAN.OR</td>
<td></td>
</tr>
</tbody>
</table>

```sql
( 2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR )
```
<table>
<thead>
<tr>
<th>exec</th>
<th>code</th>
<th>bool</th>
<th>int</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER.*</td>
<td>4.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOAT.+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRUE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FALSE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOOLEAN.OR</td>
<td>( 2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR )</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER.*</td>
<td>4.1</td>
<td>5.2</td>
<td>FLOAT.+</td>
<td>TRUE</td>
</tr>
<tr>
<td>-----------</td>
<td>-----</td>
<td>-----</td>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>exec</th>
<th>code</th>
<th>bool</th>
<th>int</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>exec</td>
<td>code</td>
<td>bool</td>
<td>int</td>
<td>float</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>4.1</td>
<td>5.2</td>
<td>FLOAT.+</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>FLOAT.+</td>
<td>TRUE</td>
<td>FALSE</td>
<td>BOOLEAN.OR</td>
<td>6</td>
</tr>
</tbody>
</table>

(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
<table>
<thead>
<tr>
<th>exec</th>
<th>code</th>
<th>bool</th>
<th>int</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOAT.+</td>
<td>TRUE</td>
<td>FALSE</td>
<td>BOOLEAN.OR</td>
<td>(2 3 INTEGER. 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR )</td>
</tr>
<tr>
<td>TRUE</td>
<td>False</td>
<td>BOOLEAN.OR</td>
<td>(2 3 INTEGER.* 4.1 5.2</td>
<td>FLOAT.+ TRUE FALSE BOOLEAN.OR</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>------------</td>
<td>--------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>exec</td>
<td>code</td>
<td>bool int</td>
<td>float</td>
<td></td>
</tr>
</tbody>
</table>
exec code bool int float

FALSE

BOOLEAN.OR

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

TRUE

6

9.3
<table>
<thead>
<tr>
<th>exec</th>
<th>code</th>
<th>bool</th>
<th>int</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>6</td>
<td></td>
<td></td>
<td>9.3</td>
</tr>
<tr>
<td>FALSE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>exec</td>
<td>code</td>
<td>bool</td>
<td>int</td>
<td>float</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TRUE</td>
<td>6</td>
<td>9.3</td>
</tr>
</tbody>
</table>

(2, 3, INTEGER.* 4, 1, 5, 2, FLOAT.+ TRUE, FALSE, BOOLEAN.OR)
Same Results

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

( 2 BOOLEAN.AND 4.1 TRUE INTEGER./ FALSE 3 5.2 BOOLEAN.OR INTEGER.* FLOAT.+ )
( 3.14 CODE.REVERSE CODE.CDR IN IN 5.0 FLOAT.> (CODE.QUOTE FLOAT.*) CODE.IF )

IN=4.0
3.14
CODE.REVERSE
CODE.CDR
IN
IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT*)
CODE.IF
(C.14 CODE.REVERSE
CODE.CDR IN IN
5.0 FLOAT.>

exec code bool int float
CODE.REVERSE
CODE.CDR
IN
IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT*)
CODE.IF
( 3.14 CODE.REVERSE
CODE.CDR IN IN
5.0 FLOAT>)
3.14
exec code bool int float
(CODE.IF (CODE.QUOTE FLOAT*) FLOAT.> 5.0 IN IN CODE.CDR)

3.14
(CODE.QUOTE FLOAT.*)

CODE.IF (FLOAT.> 5.0 IN IN CODE.CDR)

exec code bool int float
<table>
<thead>
<tr>
<th>exec</th>
<th>code</th>
<th>bool</th>
<th>int</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.14</td>
</tr>
</tbody>
</table>

```lisp
(Code_quote (float > 5.0) (in (in (code cdr) (code reverse 3.14)))
```
<table>
<thead>
<tr>
<th>exec</th>
<th>code</th>
<th>bool</th>
<th>int</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>FLOAT.&gt;</td>
<td>(CODE.QUOTE FLOAT.*)</td>
<td>FLOAT.&gt; 5.0 IN IN CODE.CDR</td>
<td>4.0</td>
</tr>
<tr>
<td>3.14</td>
<td></td>
<td></td>
<td></td>
<td>4.0</td>
</tr>
</tbody>
</table>

The code snippet translates to: `CODE.IF ((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR) exec code bool int float`
FLOAT.> (CODE.QUOTE FLOAT*)

CODE.IF

( (CODE.QUOTE FLOAT*)
  FLOAT.> 5.0 IN IN
  CODE.CDR

3.14

exec   code   bool   int   float
<table>
<thead>
<tr>
<th>exec</th>
<th>code</th>
<th>bool</th>
<th>int</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE.IF</td>
<td>(((CODE.QUOTE FLOAT.*) FLOAT.&gt; 5.0) IN IN CODE.CDR)</td>
<td>FALSE</td>
<td></td>
<td>3.14</td>
</tr>
<tr>
<td>exec</td>
<td>code</td>
<td>bool</td>
<td>int</td>
<td>float</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>CODE.QUOTE</td>
<td>FLOAT.*</td>
<td>((CODE.QUOTE FLOAT.*) FLOAT.&gt; 5.0 IN IN CODE.CDR</td>
<td>FALSE</td>
<td>3.14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>false</td>
<td>4.0</td>
</tr>
<tr>
<td>exec</td>
<td>code</td>
<td>bool</td>
<td>int</td>
<td>float</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>CODE.IF</td>
<td>((CODE.QUOTE FLOAT.*) FLOAT.&gt; 5.0 IN IN CODE.CDR)</td>
<td>FALSE</td>
<td>4.0</td>
<td>3.14</td>
</tr>
<tr>
<td>exec</td>
<td>code</td>
<td>bool</td>
<td>int</td>
<td>float</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>FLOAT.*</td>
<td></td>
<td></td>
<td></td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.14</td>
</tr>
<tr>
<td>exec</td>
<td>code</td>
<td>bool</td>
<td>int</td>
<td>float</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12.56</td>
</tr>
</tbody>
</table>
\[
\text{IN EXEC.DUP (3.13 \text{ FLOAT}.* \)} \\
10.0 \text{ FLOAT}./
\]

\[
\text{IN=4.0}
\]
IN
EXEC.DUP
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
exec
code
bool
int
float
<table>
<thead>
<tr>
<th>exec</th>
<th>code</th>
<th>bool</th>
<th>int</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3.13 FLOAT.*)</td>
<td>10.0</td>
<td>FLOAT./</td>
<td>(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)</td>
<td>4.0</td>
</tr>
</tbody>
</table>
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
<table>
<thead>
<tr>
<th>exec</th>
<th>code</th>
<th>bool</th>
<th>int</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(3.13 FLOAT.*)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10.0 FLOAT./</td>
</tr>
<tr>
<td>exec</td>
<td>code</td>
<td>bool</td>
<td>int</td>
<td>float</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>3.13</td>
<td>FLOAT.*</td>
<td>10.0</td>
<td>FLOAT./</td>
<td>(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)</td>
</tr>
<tr>
<td>exec</td>
<td>code</td>
<td>bool</td>
<td>int</td>
<td>float</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>FLOAT.*</td>
<td>10.0</td>
<td>(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)</td>
<td></td>
<td>3.13</td>
</tr>
<tr>
<td>exec</td>
<td>code</td>
<td>bool</td>
<td>int</td>
<td>float</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>10.0</td>
<td>FLOAT./</td>
<td>(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./) 39.1876
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./) 3.91876
Modularity is Everywhere
Modules in GP

• Automatically-defined functions (Koza), macros (Spector)
• Architecture-altering operations (Koza)
• Module acquisition/encapsulation systems (Kinnear, Roberts, many others)
• Modules in GE (Swafford et al., others)
• In Push: code-manipulation instructions that build/execute modules as programs run

*We will return to this later!*
ADFs

- All programs in the population have the same, pre-specified architecture
- Genetic operators respect that architecture
- (progn (defn adf0 (arg0 arg1) ...) 
  (defn adf1 (arg0 arg1 arg2) ...) 
  (.... (adf1 ...) (adf0 ...) ...))
- Complicated, brittle, limited...
- Architecture-altering operations: more so
Modules in Push

• Transform/execute code as data: Works, emerges, but stack-based module reference won’t scale well

• Execution stack manipulation:
  (3 exec.dup (1 integer.+))
  More parsimonious, but same scaling issue

• Named modules:
  (plus1 exec.define (1 integer.+)) ... plus1
  Coordinating definitions/references is tricky and this never arises in evolution!
Modularity
Ackley and Van Belle

Figure 2: Average fitness values at the start ($F_s$) and end ($F_e$) of each epoch when regressing to $y = A \sin(Ax)$. $A$ is selected at the start of each epoch uniformly from the range $[0,6)$. 
Code-as-data
Modularity in Push
Tags

- Roots in John Holland’s work on principles of complex adaptive systems
- Applied in models of the evolution of altruism, with agents having tags and tag-difference thresholds for donation
- A tag is *an initially meaningless identifier that can come to have meaning through the matches in which it participates*
- Matches may be inexact
Tag-based Modules in GP

- Add mechanisms for tagging code
- Add mechanisms for retrieving/branching to code with closest matching tag
- As long as any code has been tagged, all branches go somewhere
- Number of tagged modules can grow incrementally over evolutionary time
Tags in Push

- Tags are integers embedded in instruction names
- Instructions like `tag.exec.123` tag values
- Instructions like `tagged.456` recall values by closest matching tag
- If a single value has been tagged then all tag references will recall (and execute) values
- The number of tagged values can grow incrementally over evolutionary time
Lawnmower Problem

- Used by Koza to demonstrate utility of ADFs for scaling GP up to larger problems
<table>
<thead>
<tr>
<th>Condition</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>left, mow, v8a, frog, $R_{v8}$</td>
</tr>
<tr>
<td>Tag</td>
<td>left, mow, v8a, frog, $R_{v8}$, tag.exec.[1000], tagged.[1000]</td>
</tr>
<tr>
<td>Exec</td>
<td>left, mow, v8a, frog, $R_{v8}$, exec.dup, exec.pop, exec.rot, exec.swap, exec.k, exec.s, exec.y</td>
</tr>
</tbody>
</table>
Lawnmower Effort*

* with frog=noop bug

Problem Size

Computational Effort

- Basic
- Tag
- Exec
Lawnmower Effort

![Graph showing computational effort vs. problem size for basic, tag, and exec methods. The graph demonstrates an increasing computational effort as the problem size increases.]
## Lawnmower Effort

<table>
<thead>
<tr>
<th>problem size</th>
<th>8x4</th>
<th>8x6</th>
<th>8x8</th>
<th>8x10</th>
<th>8x12</th>
</tr>
</thead>
<tbody>
<tr>
<td>instr set</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>basic</td>
<td>10000</td>
<td>30000</td>
<td>114000</td>
<td>320000</td>
<td>630000</td>
</tr>
<tr>
<td>tag</td>
<td>7000</td>
<td>2000</td>
<td>29000</td>
<td>&lt;1000</td>
<td>5000</td>
</tr>
<tr>
<td>exec</td>
<td>12000</td>
<td>5000</td>
<td>28000</td>
<td>5000</td>
<td>17000</td>
</tr>
</tbody>
</table>
Dirt-Sensing, Obstacle-Avoiding Robot Problem

Like the lawnmower problem but harder and less uniform
<table>
<thead>
<tr>
<th>Condition</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>if-dirty, if-obstacle, left, mop, v8a, frog, $R_{v8}$</td>
</tr>
<tr>
<td>Tag</td>
<td>if-dirty, if-obstacle, left, mop, v8a, frog, $R_{v8}$, tag.exec.[1000], tagged.[1000]</td>
</tr>
<tr>
<td>Exec</td>
<td>if-dirty, if-obstacle, left, mop, v8a, frog, $R_{v8}$, exec.dup, exec.pop, exec.rot, exec.swap, exec.k, exec.s, exec.y</td>
</tr>
</tbody>
</table>
DSOAR Effort*

* with frog=noop bug
DSOAR Effort

Computational Effort vs. Problem Size

- Tag
- Exec
## DSOAR Effort

<table>
<thead>
<tr>
<th>instr set</th>
<th>8x4</th>
<th>8x6</th>
<th>8x8</th>
<th>8x10</th>
<th>8x12</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic</td>
<td>1584000</td>
<td>430083000</td>
<td>inf</td>
<td>inf</td>
<td>inf</td>
</tr>
<tr>
<td>tag</td>
<td>216000</td>
<td>864000</td>
<td>3420000</td>
<td>2599000</td>
<td>3051000</td>
</tr>
<tr>
<td>exec</td>
<td>450000</td>
<td>2125000</td>
<td>4332000</td>
<td>16644000</td>
<td>7524000</td>
</tr>
</tbody>
</table>
More data, source code, etc, at:

http://hampshire.edu/lspector/tags-gecco-2011
Evolved DSOAR Architecture (in one environment)
Evolved DSOAR Architecture (in another environment)
Tags in S-Expressions

• A simple form:
  (progn (tag-123 (+ a b)) tagged-034)

• Must do something about endless recursion

• Must do something about return values

• Must do something fancy to support modules with arguments, particularly arguments of multiple types.
Future Work

• Tags in s-expression-based GP
• Tag usage over evolutionary time
• No-pop tagging in PushGP
• Tags in autoconstructive evolution
• Applications, application, applications
Conclusions

• Execution stack manipulation supports the evolution of modular programs in many situations

• Tag-based modules are more effective in complex, non-uniform problem environments

• Tag-based modules may help to evolve complex software and solutions to unsolved problems in the future