The calculator problem and the evolutionary synthesis of arbitrary software

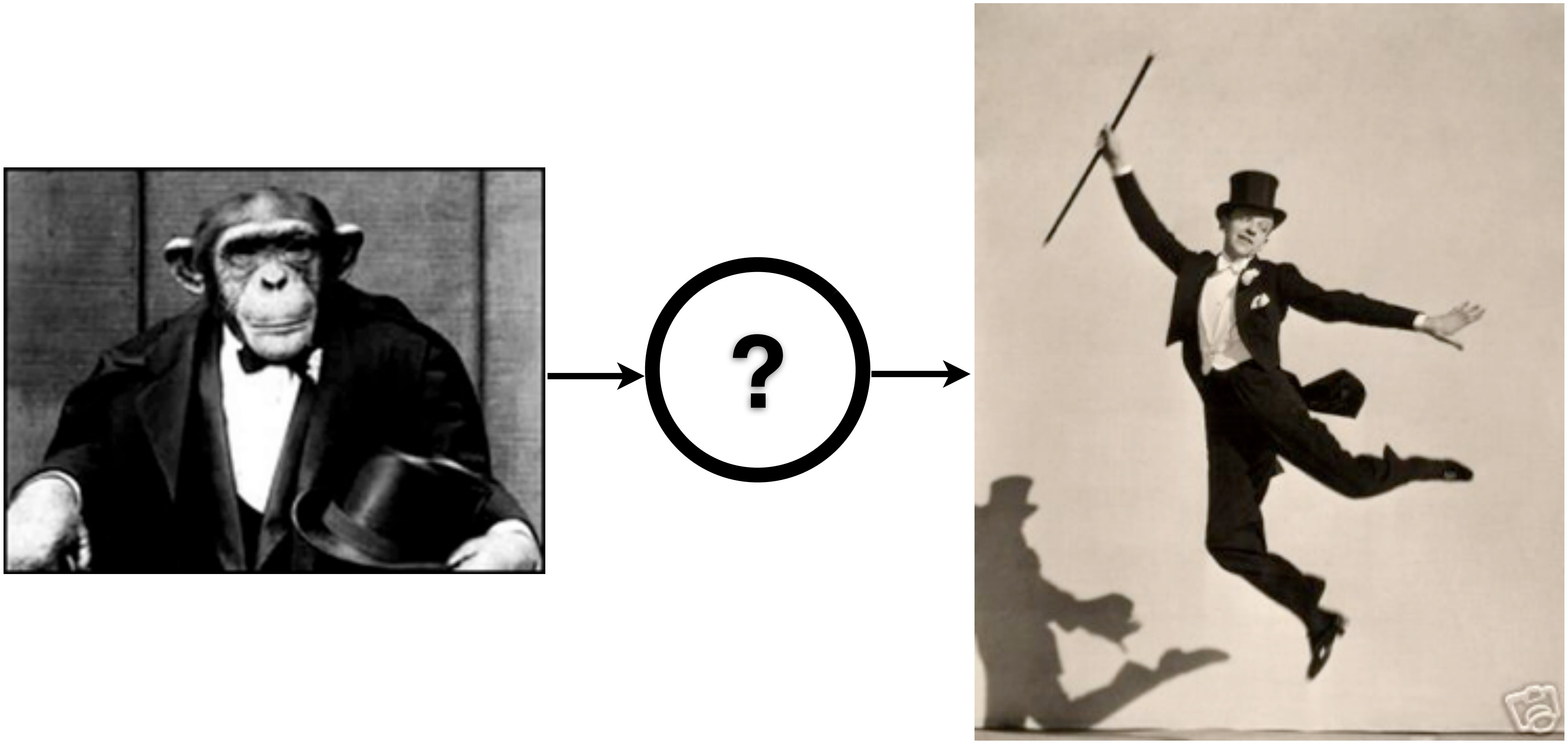
CREST Open Workshop on Genetic Programming
for Software Engineering

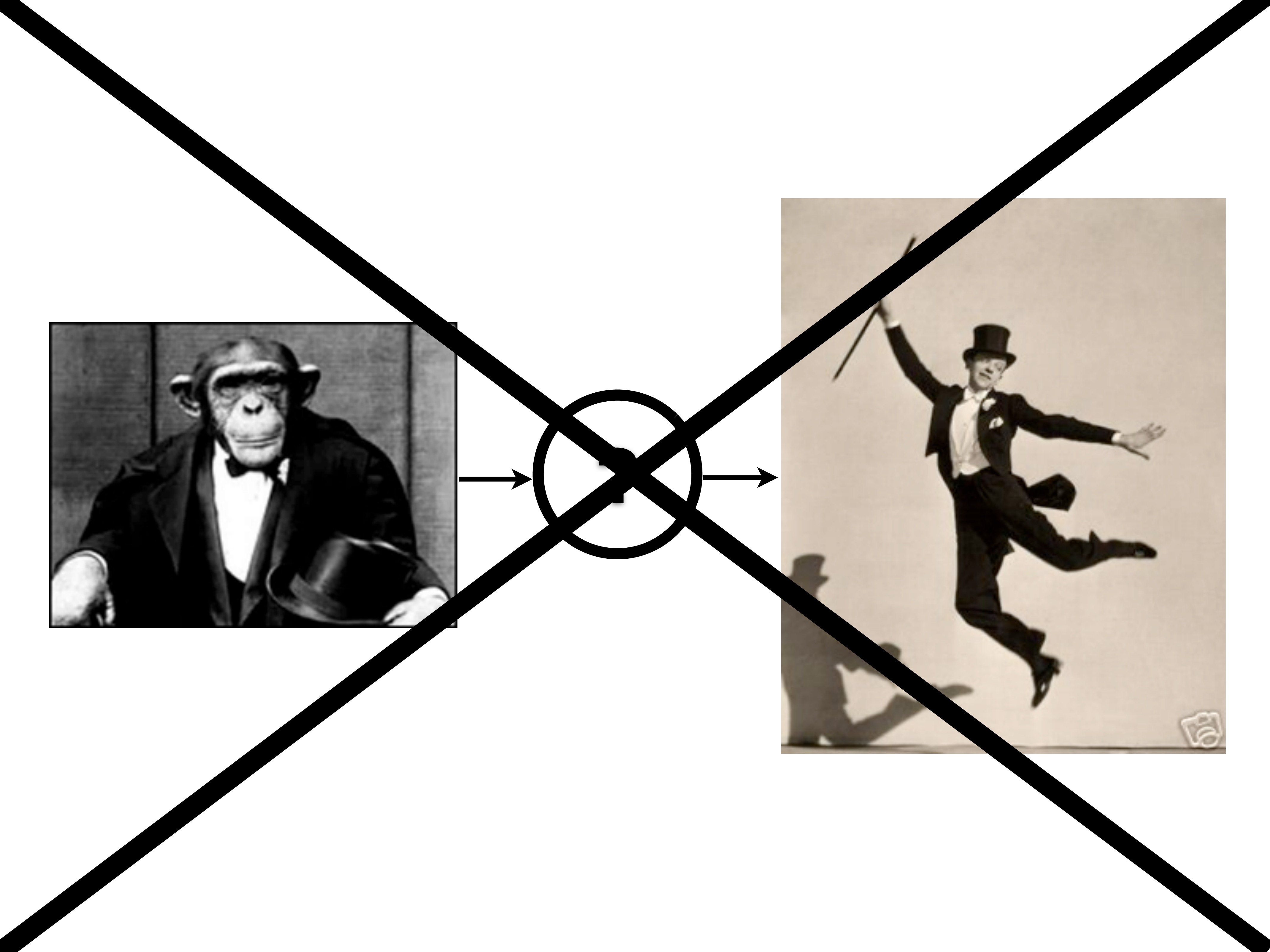
October 14, 2013

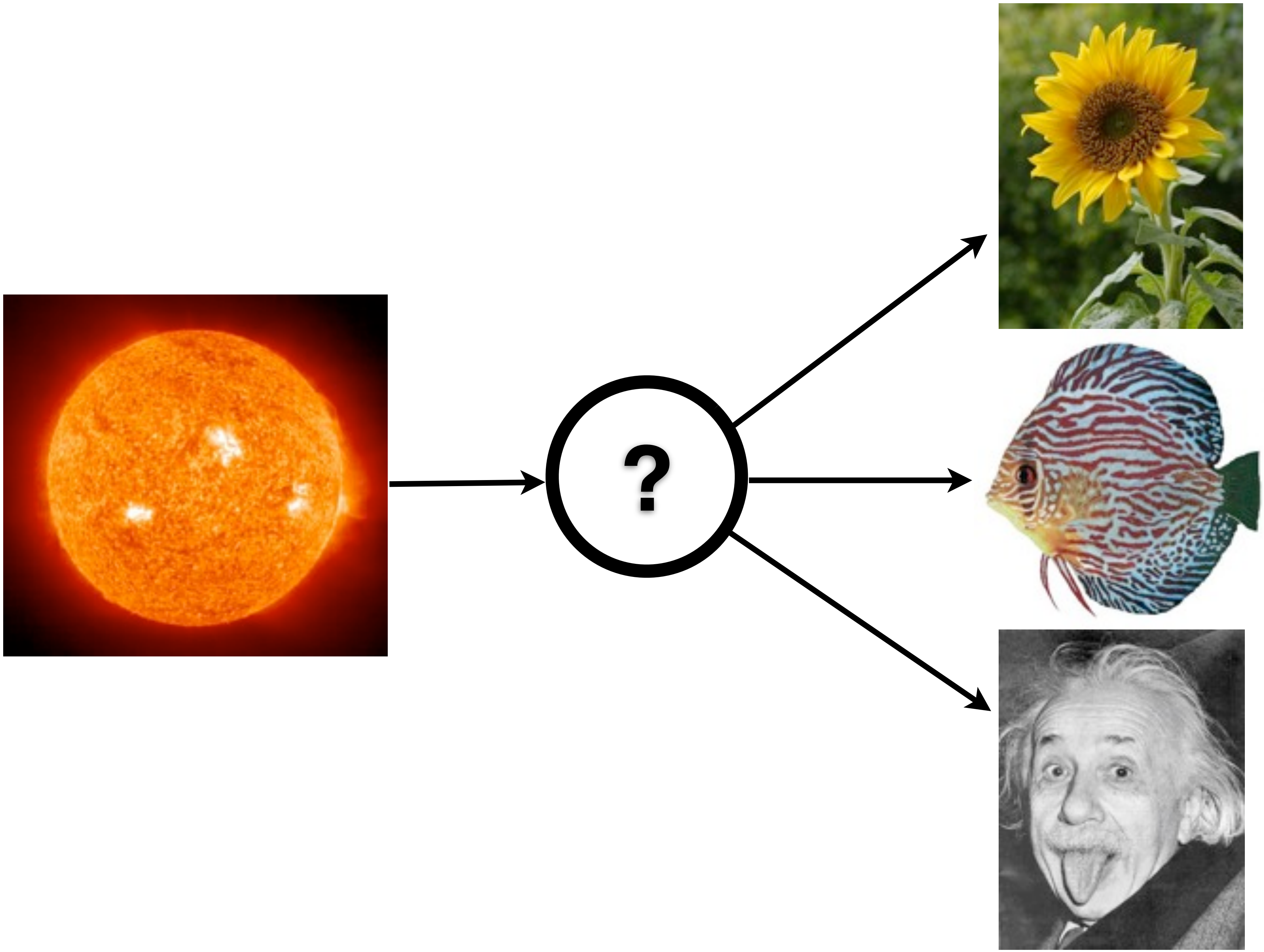
Lee Spector
Hampshire College
Amherst, MA USA

Tests

Software







STAPLES
SPL-110
12345678
%
√
OFF
ON/C
MRC
M-
M+
CE
7
8
9
÷
4
5
6
×
1
2
3
−
0
•
=
+

Outline

- Arbitrary software
- Requirements and ways to meet them
- Tags, uniform variation, and lexicase selection
- The calculator problem
- Other problems and prospects

Arbitrary Software

- OS utilities
- Word processors
- Web browsers
- Accounting systems
- Image processing systems
- Everything

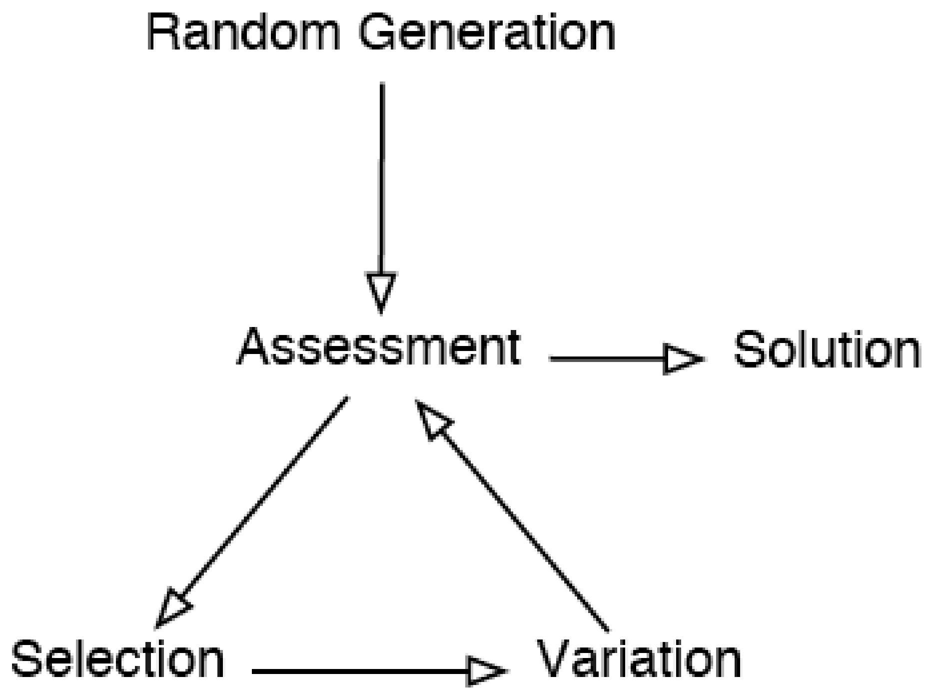
Arbitrary Software

- May be stateful, with multiple entry points
- May have a variety of interfaces involving a variety of types
- May require arbitrary Turing-computable functions
- Can be specified with behavioral tests

Requirements

- Represent and evolve arbitrary computable functions on arbitrary types (Push, uniform variation)
- Represent and evolve arbitrary computational architectures (modules; tags, tagged entry points)
- Drive evolution with performance tests (lexicase selection)

Evolutionary Computation



Genetic Programming

- Evolutionary computing to produce executable computer programs
- Programs are assessed by executing them
- Automatic programming; producing software
- Potential (?): evolve software at all scales, including and surpassing the most ambitious and successful products of human software engineering

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, ...).

Evolvability

The fact that a computation *can* be expressed in a formalism does not imply that a correct program can be produced in that formalism by a human programmer or by an evolutionary process.

Data/Control Structure

- Data abstraction and organization

  Data types, variables, name spaces, data structures, ...

- Control abstraction and organization

  Conditionals, loops, modules, threads, ...

Structure via GP (1)

- Specialize GP techniques to support human programming language abstractions
- Strongly typed genetic programming
- Automatically defined functions/macros
- Architecture altering operations
- Map from unstructured genomes to programs in languages that support abstraction (e.g. via grammars)

Structure via GP (2)

- Forget about human programming abstractions (mostly)
- 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
- **Push**

Push

- A programming language developed specifically for evolutionary computation, as the language in which evolving programs are expressed
- Intended to maximize the evolvability of arbitrary computational processes

Push

- Stack-based postfix language with one stack per type
- Types include: integer, float, boolean, code, exec, vector, matrix, quantum gate, [add more as needed]
- Missing argument? NO-OP
- Minimal syntax:
  program $\rightarrow$ instruction | literal | (program*)

Why Push?

- Highly expressive: data types, data structures, variables, conditionals, loops, recursion, modules, ...
- Elegant: minimal syntax and a simple, stack-based execution architecture
- Elegance simplifies a variety of things ranging from uniform variation to meta-evolution
- Evolvable
- Extensible

Sample Push Instructions

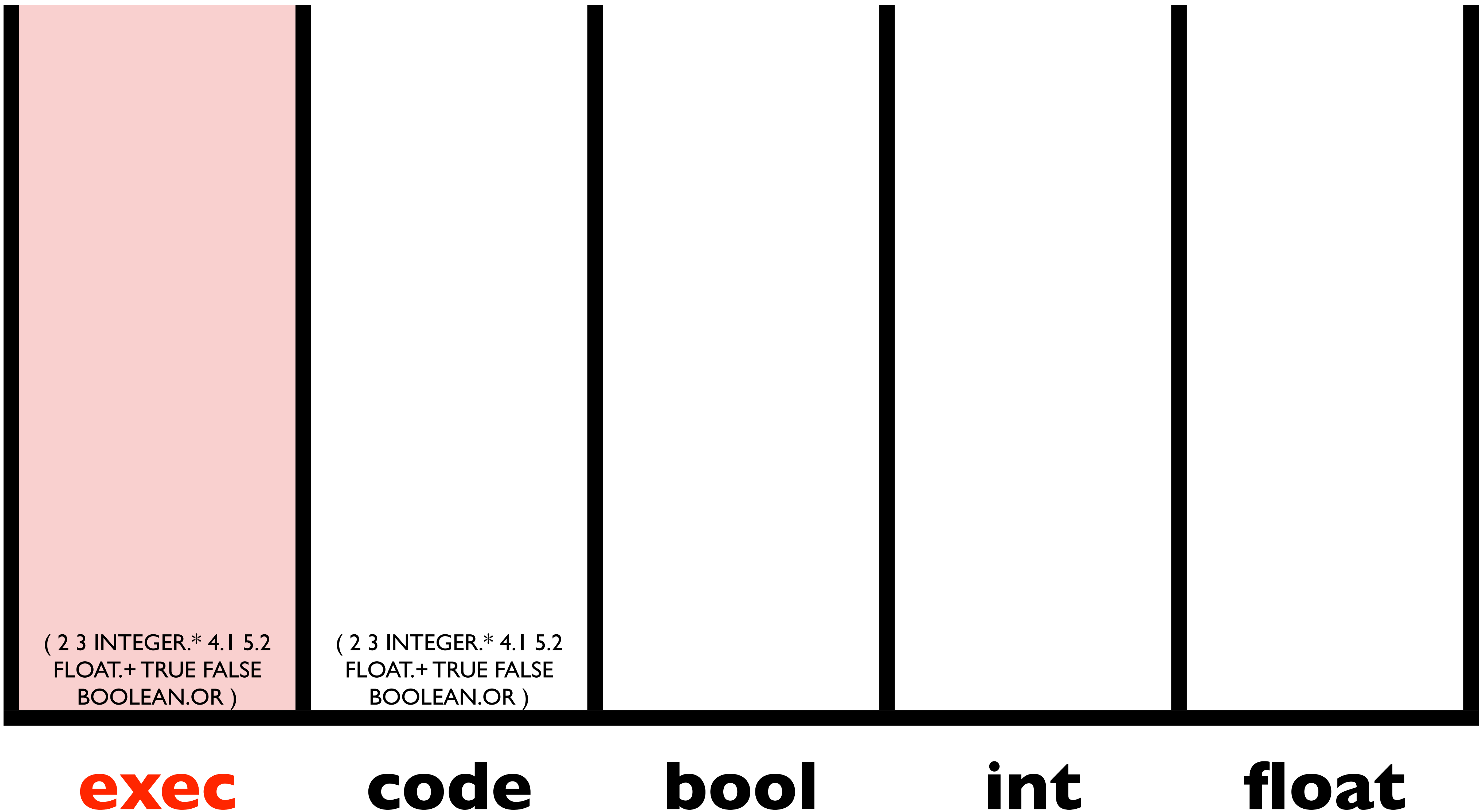
Stack manipulation instructions (all types)	POP, SWAP, YANK, DUP, STACKDEPTH, SHOVE, FLUSH, $=$
Math (INTEGER and FLOAT)	$+$, $-$, $/$, $*$, $>$, $<$, MIN, MAX
Logic (BOOLEAN)	AND, OR, NOT, FROMINTEGER
Code manipulation (CODE)	QUOTE, CAR, CDR, CONS, INSERT, LENGTH, LIST, MEMBER, NTH, EXTRACT
Control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

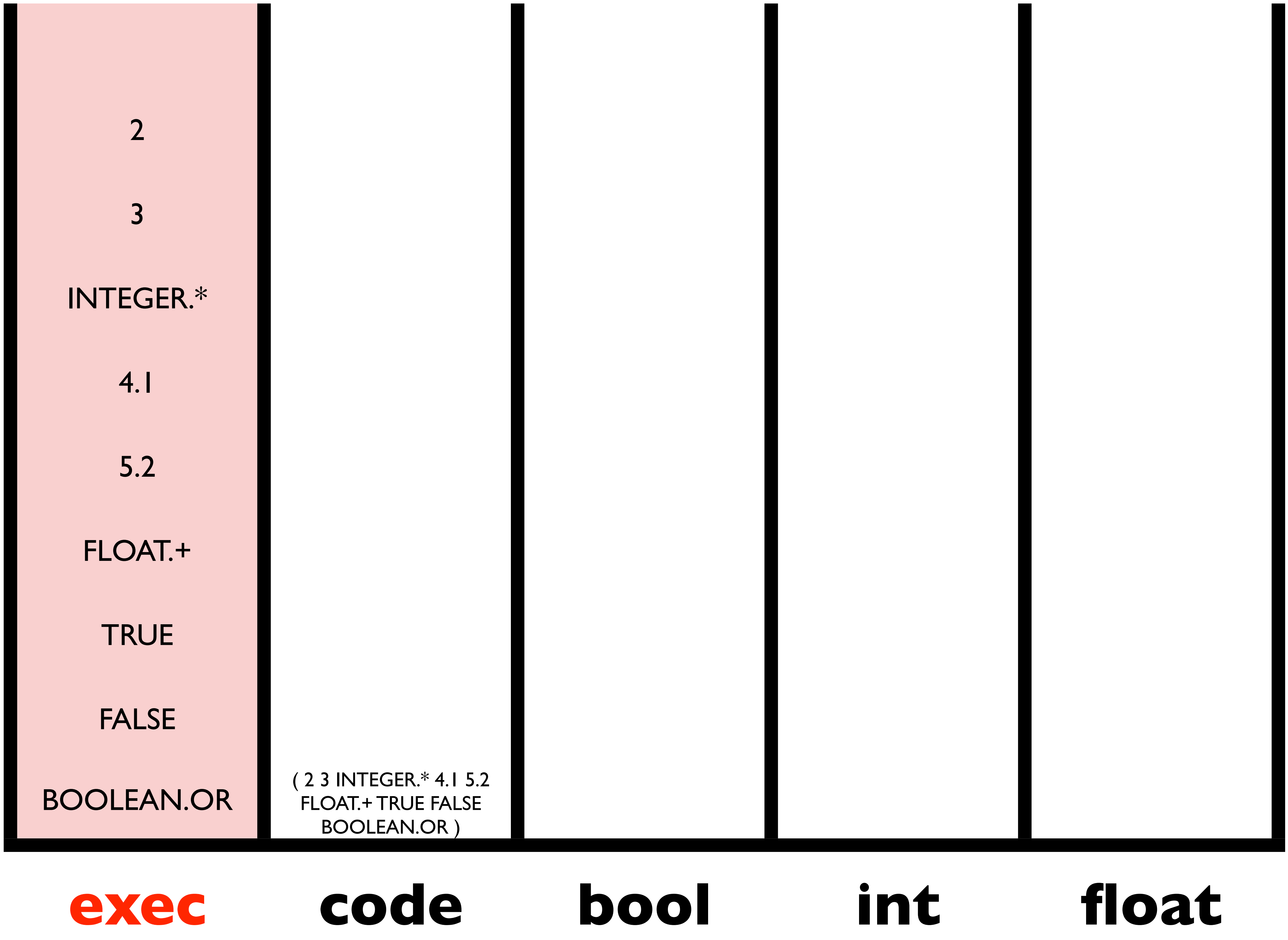
Push(3) Semantics

- 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 )
```




2
3
INTEGER.*
4.1
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
exec
code
bool
int
float

exec	code	bool	int	float
3				
INTEGER.*				
4.1				
5.2				
FLOAT.+				
TRUE				
FALSE				
BOOLEAN.OR	(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)		2	

exec	code	bool	int	float
INTEGER.*				
4.1				
5.2				
FLOAT.+				
TRUE				
FALSE			3	
BOOLEAN.OR	(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)		2	

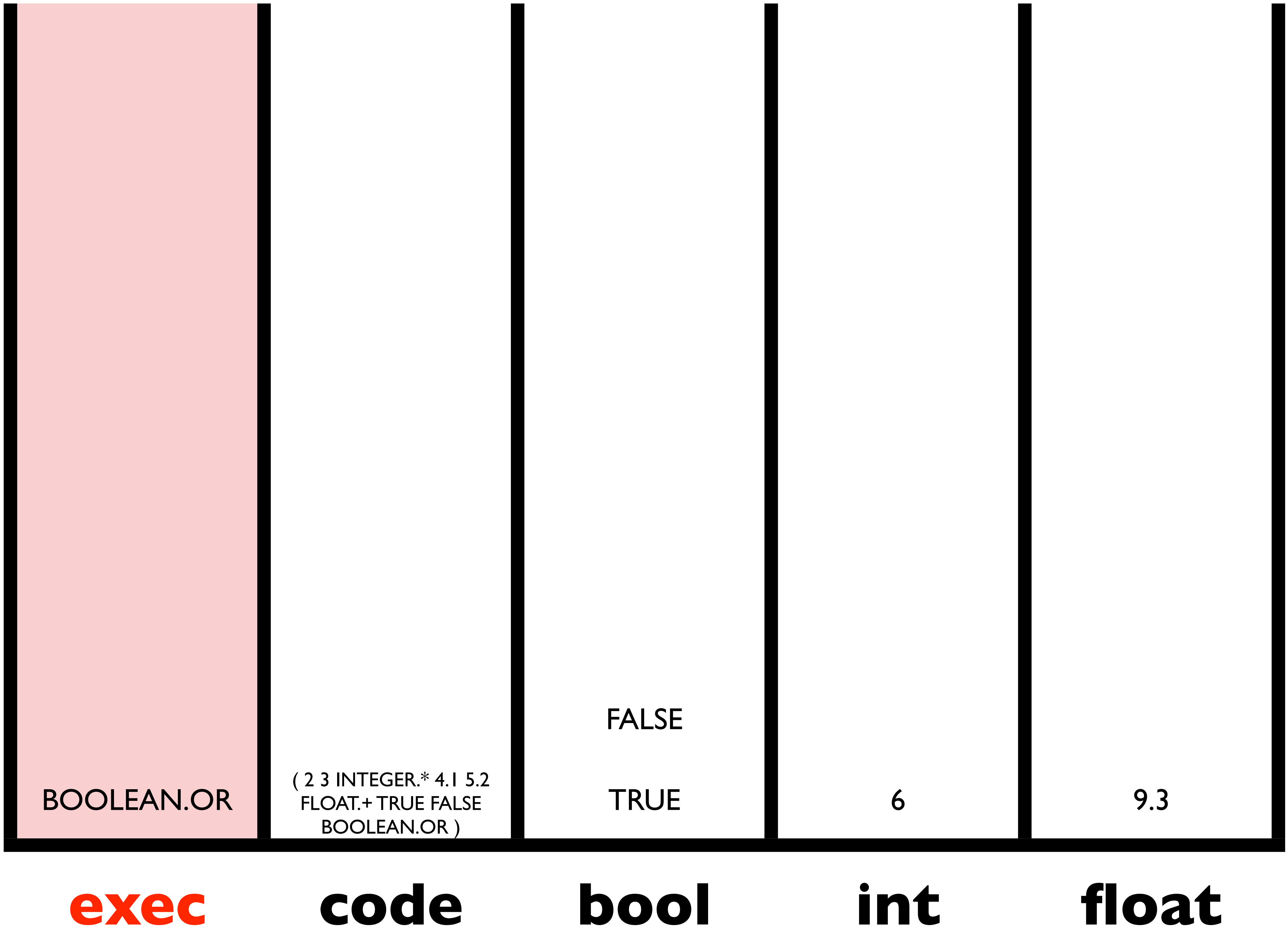
exec	code	bool	int	float
4.1				
5.2				
FLOAT.+				
TRUE				
FALSE				
BOOLEAN.OR	(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)		6	

exec	code	bool	int	float
5.2				
FLOAT.+				
TRUE				
FALSE				
BOOLEAN.OR	(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)		6	4.1

exec	code	bool	int	float
FLOAT.+				
TRUE				
FALSE				5.2
BOOLEAN.OR	(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)		6	4.1

exec	code	bool	int	float
TRUE				
FALSE				
BOOLEAN.OR	(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)		6	9.3

exec	code	bool	int	float
FALSE				
BOOLEAN.OR	(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)	TRUE	6	9.3


BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
FALSE
TRUE
6
9.3
exec
code
bool
int
float

exec	code	bool	int	float
	(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)	TRUE	6	9.3

No Time to Show How

- Push enables a trivial form of auto-simplification
- Push programs are often robust to reordering and other changes, producing a search space with high neutrality
- Push programs that modify their own code and/or the execution stack dynamically can thereby implement arbitrary control structures and several forms of modularity

STAPLES
SPL-110
12345678
%
OFF
ON/C
MRC
M-
M+
CE
7
8
9
÷
4
5
6
×
1
2
3
−
0
•
=
+

Calculator Test Cases

Keys pressed => number, error flag

- Digit entry tests
- Digit entry pair tests
- Double digit float entry tests
- Single digit math tests
- Single digit incomplete math tests
- Single digit chained math tests
- Division by zero tests

Digit Entry Tests

- :zero => 0.0, false
- :one => 1.0, false
- :two => 2.0, false
- :three => 3.0, false
- ...

Digit Entry Pair Tests

- :zero :zero => 0.0, false
- :zero :one => 1.0, false
- :two :three => 23.0, false
- :nine :nine => 99.0, false
- ...

Float Entry Tests

- :zero :point :nine => 0.9, false
- :zero :point :two => 0.2 false
- :seven :point :nine => 7.9, false
- :three :point :two => 3.2, false
- ...

Single Digit Math Tests

- :zero :plus :nine :equals => 9.0, false
- :three :times :four :equals => 12.0, false
- :three :minus :nine :equals => -6.0, false
- :three :divided-by :four :equals => 0.75, false
- ...

Incomplete Math Tests

- :three :plus :four => 4.0, false
- :seven :plus => 7.0, false
- ...

Chained Math Tests

- :three :plus :nine :minus :five :equals
  => 7.0, false
- :three :times :two :divided-by :eight :equals
  => 0.75, false
- :three :divided-by :nine :minus :five :equals
  => -4.6666665, false
- ...

Division by Zero Tests

- :zero :divided-by :zero :equals => 0.0, true
- :seven :divided-by :zero :equals => 0.0, true
- :three :divided-by :zero :equals => 0.0, true
- ...

Architectural Requirements

- Every key press is an entry point
- Answers (a floating point number and a boolean value) should provided after every key press
- State must be maintained between key presses
- Stacks + tags provide an elegant way to meet these requirements

Holland’s Tags

- Initially arbitrary identifiers that come to have meaning over time
- Matches may be inexact
- Appear to be present in some form in many different kinds of complex adaptive systems
- Examples range from immune systems to armies on a battlefield
- A general tool for the support of emergent complexity

Tag-based Modules

- Include instructions that tag code (modules)
- Include instructions that recall and execute modules by *closest matching* tag
- If a single module has been tagged then all tag references will recall modules
- The number of tagged modules can grow incrementally over evolutionary time
- Expressive and evolvable

Calculator Architecture

- Run program once to tag modules
- Clear stacks
- For each pressed key, execute the module that best matches the corresponding tag, maintaining stacks across key presses
- The top of the float stack is the number output; the top of the boolean stack is the error flag output

And?

With Push and the tagged-entry-point architecture we can run GP on the calculator problem.

And it fails miserably:

- Large programs are required
- Must allow growth without bloating
- Must allow arbitrary recombination

Uniform Variation

- All genetic material that a child inherits should be $\approx$ likely to be mutated
- Parts of both parents should be $\approx$ likely to appear in children (at least if they are $\approx$ in size), and to appear in a range of combinations

Why Uniformity?

- No hiding from mutation
- All parts of parents subject to variation and recombination
- Biological genetic variation, while not fully uniform, has uniformity properties that prevent some of the problems we see in GP; e.g. just having more genes doesn't generally "protect" any of them

Prior Work

- Point mutations or “uniform crossovers” that replace/swap nodes but only in restricted ways; cannot change structure, has depth biases (McKay et al, 1995; Page et al, 1998; Poli and Langdon, 1998; Poli and Page, 2000; Semenkin and Semenkina, 2012)
- Uniform mutation via size-based numbers of tree replacements; depth biases, little demonstrated benefit (McKay et al, 1995; Van Belle and Ackley, 2002)

ULTRA

- Achieve uniformity by treating genomes as linear sequences, even if they are hierarchically structured
- Repair after transform to ensure structural validity

The ULTRA Operator

- **U**niform **L**inear **T**ransformation with **R**epair and **A**lternation
- Linearize 2 parents, treating "(" and ")" as ordinary tokens
- Start at the beginning of one parent and copy tokens to the child, switching parents stochastically (according to the *alternation rate*, and subject to an *alignment deviation*)
- Post-process with uniform mutation (according to a *mutation rate*) and repair

Parents:

`( a b ( c ( d ) ) e ( f g ) )`

`( 1 ( 2 ( 3 4 ) 5 ) 6 )`

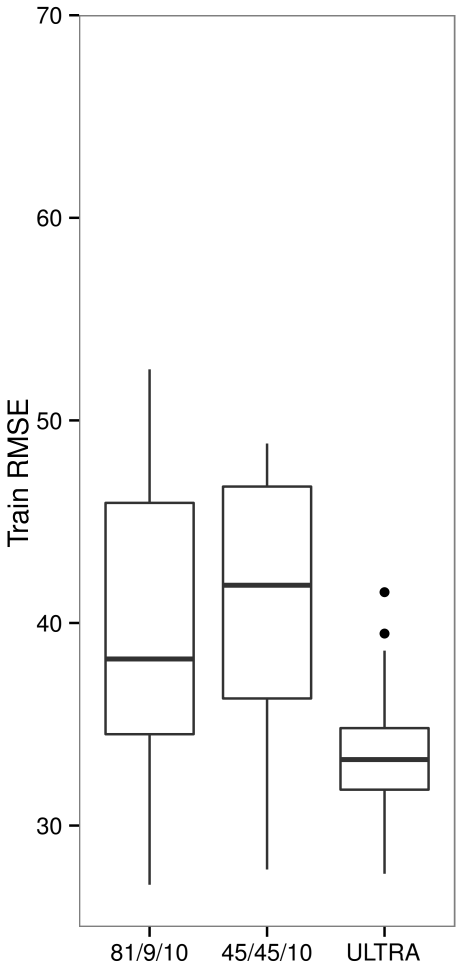
Result of alternation:

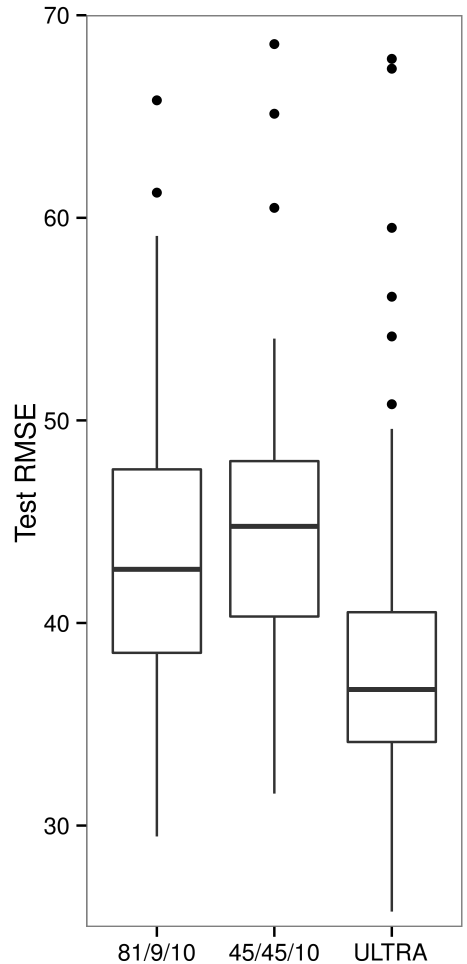
`( a b 2 ( 3 4 d ) ) 6 )`

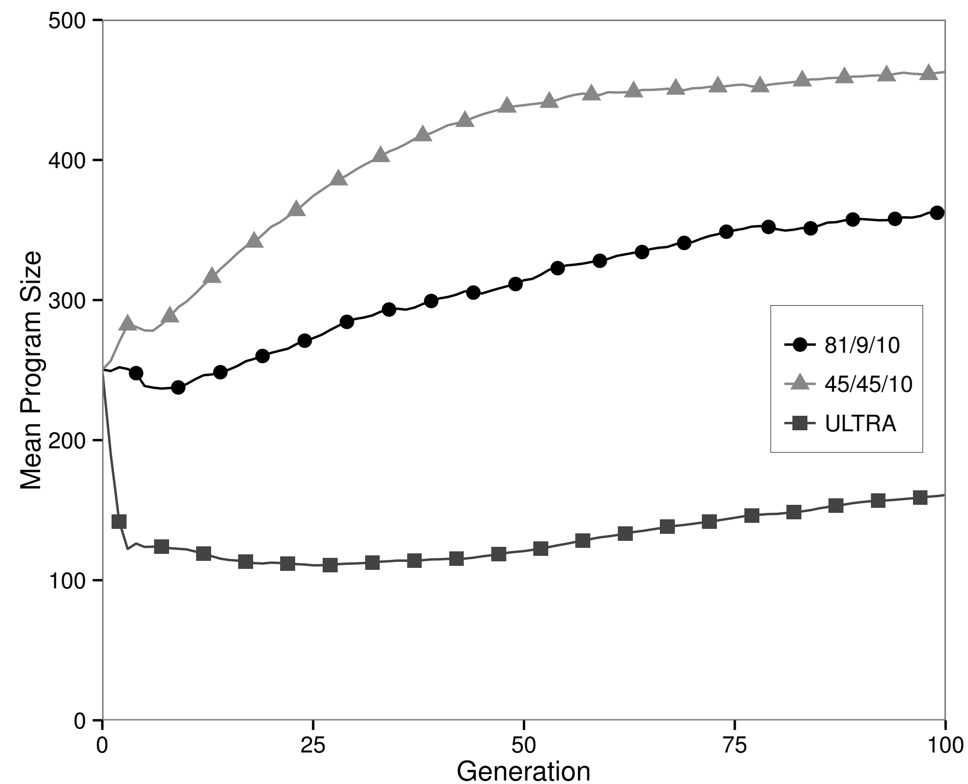
Result of repair:

`( a ( b 2 ( 3 4 d ) ) 6 )`

ULTRA on the bioavailability problem







And?

With Push, the tagged-entry-point architecture, and ULTRA... we still fail. But not quite as miserably.

Issues:

- Different test cases require qualitatively different modes of response
- Numbers of cases of different types have an undue influence
- Average performance across cases does not guide search appropriately

Lexicase Selection

- Each parent is selected by filtering the entire population, one one case at a time (in random order), keeping only the elite at each stage
- Useful for "modal" problems, which require qualitatively different responses to different inputs
- Useful for "uncompromising" problems, in which solutions must be optimal on each case
- All comparisons are "within case," so may be useful whenever cases are non-comparable

Lexicase Selection

Initialize:

Candidates = the entire population

Cases = a list of all of the test cases in random order

Loop:

Candidates = the subset of **Candidates** with exactly the best performance of any current candidate for the first case in **Cases**

If **Candidates** or **Cases** contains just a single element then return a randomly selected individual from **Candidates**

Otherwise remove the first case from **Cases** and go to **Loop**

Finite Algebras

$\mathbf{A}_1$ $*$	0	1	2
0	2	1	2
1	1	0	0
2	0	0	1

A1 Mal'cev Term

Selection	Successes	CE	MBF
Tournament Size 2	35	532,000	0.75
Tournament Size 3	43	420,000	0.70
Tournament Size 4	31	440,000	0.75
Tournament Size 5	22	616,000	0.77
Tournament Size 6	25	750,000	0.90
Tournament Size 7	23	403,000	0.92
Tournament Size 8	26	464,000	0.94
Tournament Size 9	21	550,000	1.06
Lexicase	94	90,000	0.05

Digital Multiplier

- Evolve a digital circuit to multiply two binary numbers
- *n*-bit digital multiplier: 2 x *n* bits → 2*n* bits
- Multiple outputs
- Scalable
- Recommended as a GP benchmark problem (McDermott, et al 2012, White et al 2013)

3-bit Digital Multiplier

Boolean Stack	and, or, xor, invert_first_then_and, dup, swap, rot
Input / Output	in0, ..., in2n, out0, ..., out2n

Selection	**Successes**	**MBF**
Tournament Size 7	0	0.24
Lexicase	100	0

Factorial

Boolean Stack	and, dup, eq, frominteger, not, or, pop, rot, swap
Integer Stack	add, div, dup, eq, fromBoolean, greaterThan, lessThan, mod, mult, pop, rot, sub, swap
Exec Stack	dup, eq, if , noop, pop, rot, swap, when, k, s, y
Input	in
Constants	0, 1

Selection	Successes	MBF
Tournament Size 7	0	74,545
Lexicase	61	28,980

And?

With Push, the tagged-entry-point architecture, ULTRA, and lexicase selection... we succeed!*

*On some reasonably large sets of tests (not all shown above, yet).

*But without generalizing.

Continuing Work

- Generative tests for selection and validation
- Refinements to tagging mechanisms, ULTRA, and lexicase selection
- Work on other program synthesis problems:
  - Kata bowling
  - The UNIX wc program
  - CS101 problems
- Insights from non-evolutionary program synthesis work

Conclusions

- Evolutionary synthesis of arbitrary software is hard!
- But we can learn a lot from trying to do it, both for software synthesis and for other GP applications (including others in software engineering, I suspect)
- Push, tags, tagged-entry points, uniform variation methods, and lexicase selection have all demonstrated promise

Thanks

- Thomas Helmuth, Emma Tosch, Kyle Harrington, Kwaku Yeboah Antwi, Jamie Matheson, Daniel Homer, Omri Bernstein, Jake Wisdom, Josiah Erikson
- USA National Science Foundation grants Grants No. 1017817 and 1129139.
  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.

Genetic Programming and Evolvable Machines – incl. option to publish open access

www.springer.com/computer/ai/journal/10710

Reader

GPEM-Springer

USA » Change

Springer

» New User

LOGIN

HOME | MY SPRINGER | SUBJECTS | SERVICES | PUBLISHERS | ABOUT US

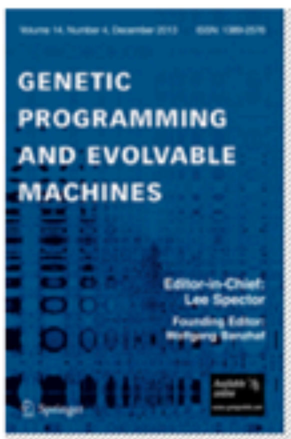
GO

Advanced Search

» *Artificial Intelligence*

Home > Computer Science > Artificial Intelligence

SUBDISCIPLINES | JOURNALS | BOOKS | SERIES | TEXTBOOKS | REFERENCE WORKS



Genetic Programming and Evolvable Machines

Main editor: L. Spector

ISSN: 1389-2576 (print version)
ISSN: 1573-7632 (electronic version)
Journal no. 10710

Free Preview

RECOMMEND TO LIBRARIAN

Like 19 Tweet 3 +1 6

ABOUT THIS JOURNAL | EDITORIAL BOARD

OPEN ACCESS

Interested in publishing your article in this journal?

READ THIS JOURNAL ON SPRINGERLINK

- Online First Articles
- All volumes & issues
- Free: Sample Articles

FOR AUTHORS AND EDITORS

2012 Impact Factor 1.333

- Aims and Scope
- Submit Online
- Open Choice - Your Way to Open Access
- Instructions for Authors
- GPEM blog

SERVICES FOR THE JOURNAL

- Contacts