

Tag-Based Modules in Genetic Programming

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

Kyle Harrington
Computer Science
Brandeis University
Waltham, MA 02453 USA
kyleh@brandeis.edu

Brian Martin
Cognitive Science
Hampshire College
Amherst, MA 01002 USA
btm08@hampshire.edu

Thomas Helmuth
Computer Science
University of Massachusetts
Amherst, MA 01003 USA
thelmuth@cs.umass.edu

ABSTRACT

In this paper we present a new technique for evolving modular programs with genetic programming. The technique is based on the use of “tags” that evolving programs may use to label and later to refer to code fragments. Tags may refer inexactly, permitting the labeling and use of code fragments to co-evolve in an incremental way. The technique can be implemented as a minor modification to an existing, general purpose genetic programming system, and it does not require pre-specification of the module architecture of evolved programs. We demonstrate that tag-based modules readily evolve and that this allows problem solving effort to scale well with problem size. We also show that the tag-based module technique is effective even in complex, non-uniform problem environments for which previous techniques perform poorly. We demonstrate the technique in the context of the stack-based genetic programming system PushGP, but we also briefly discuss ways in which it may be used with other kinds of genetic programming systems.

Categories and Subject Descriptors

I.2.2 [Artificial Intelligence]: Automatic Programming—*Program synthesis*; D.3.3 [Programming Languages]: Language Constructs and Features—*Procedures, functions, and subroutines*

General Terms

Algorithms

Keywords

Push, PushGP, genetic programming, stack-based genetic programming, modularity, automatically defined functions, tags, lawnmower problem, obstacle-avoiding robot problem

Permission to make digital or hard copies of all or part 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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GECCO'11, July 12–16, 2011, Dublin, Ireland.

Copyright 2011 ACM 978-1-4503-0557-0/11/07 ...\$10.00.

1. INTRODUCTION

It has long been recognized that genetic programming (GP) systems will be more powerful when they can more readily evolve programs with modular architectures, just as human programmers can be more productive when they make proper use of the modularization facilities that are provided by modern programming languages. The factors underlying the utility of modularity may ultimately be quite deep, as Simon claims in his discussions of “nearly decomposable” hierarchical systems [23], but for the present discussion we will merely stipulate the common belief that modularity is often useful and that systems that support modularization will therefore often be more powerful.

While many previous genetic systems have employed mechanisms that allow for the evolution of modular programs, the most popular of these (described more fully below) are either complex or limiting with respect to the kinds of modules that can evolve. One exception, on which the present work builds, involves the evolution of programs expressed in the Push programming language [25, 31, 29]. Push supports the evolution of arbitrary modules using only the facilities for code self-manipulation that are built into the core of the language, even when using a GP system that is quite generic aside from its use of Push as the representation for evolving programs (as with PushGP). In particular, it is possible to evolve modular programs in Push without pre-specifying the modular architecture and without modifying genetic operators for mutation and crossover. Features of Push that support this capability are described more fully below.

That said, some of the facilities for modularization in Push are rarely used in programs produced by evolution. One Push feature in particular, the ability to name arbitrary pieces of code and later to run those pieces of code by re-using the same names, seems closest in spirit to the modularization mechanisms most frequently used by human programmers. But in practice, this feature has almost never been used in evolved Push programs. In order to improve the usefulness of named modularity we devised an alternative modularization scheme, based not on names but on a closely related concept, tags [6, 7, 18]. Whereas a name-based reference refers only to an entity with an *exactly matching name*, a tag-based reference refers to the tagged entity with the *closest matching tag*. This small difference in functionality can make a big difference in evolution. In a tag-based

system every reference refers to some entity as long as at least one entity has been tagged. Additionally, the number of tagged entities maintained by a program can grow in an incremental way over the course of evolution.

We studied the effectiveness of tag-based modules first in the context of the “lawnmower problem,” which was used by Koza to demonstrate the utility of his own modularization scheme (“automatically defined functions”—see below) for problems of increasing size [13]. We found (see below) that systems using tag-based modules also scale well on this problem, as do systems that make use of a mechanism that has long been available in Push: direct manipulation of the execution stack.

We hypothesized that the good performance of execution stack manipulation on the lawnmower problem depended in part on the exceptional regularity of the problem, and that tag-based modules might perform better in less regular problem environments. The lawnmower problem is so exceptionally regular that even the simplest forms of modularity, in which one merely repeats exactly the same code over and over again, will suffice. Facilities that would permit more flexible modular architectures are unnecessary.

To explore this hypothesis we conducted runs on a more difficult and less regular problem called the “dirt-sensing, obstacle-avoiding robot problem” [24]. Here we found that the new tag-based modularity mechanism does in fact perform dramatically better than execution stack manipulation, confirming our intuition that it provides a robust facility for the evolution of modularity in complex environments.

In the sections below, we first briefly describe some of the ways that modularization has been supported previously in GP. We then describe Push and PushGP, with a focus on the ways in which Push (without tags) can support the evolution of modular programs. Next we describe the new tag-based modularization scheme and show how it can be used by a GP system to evolve modular programs. We then present our results for the lawnmower problem, demonstrating good scaling performance both of systems that use tag-based modularization and of systems that use only execution stack manipulation. Next we present our results for the dirt-sensing, obstacle-avoiding robot, demonstrating exceptional performance of the new tag-based modularization mechanism. We briefly discuss how tags might be used in other forms of GP and we conclude with some suggestions for future work.

2. MODULES IN GP

A variety of specific techniques have been developed for incorporating modules (including functions, subroutines, coroutines, macros, etc.) in the programs that are produced by GP systems [11, 12, 1, 10, 24, 2, 17, 19]. The most widely used of these techniques is probably the “Automatically Defined Function” (ADF) framework presented in detail by Koza in his first and second GP books [12, 13]. In the original ADF framework the structure of all of the programs in a population is restricted to a single pre-specified modular architecture, with some fixed number of function definitions (each of which takes some fixed number of arguments) and a “result-producing branch.” The user of an ADF-enabled system also specifies, in advance, which ADFs can call which other ADFs and which ADFs can be called by the result-producing branch. For a GP system to handle programs with ADFs, its program generation and manipulation

procedures (e.g. mutation and crossover procedures) must be modified to respect the restrictions of the pre-specified modular architecture.

Koza showed that when ADFs are available a GP system can often exploit problem regularities, which may allow the system to handle much larger problem instances. He also showed that the use of ADFs often allows solutions to be found more quickly and that the solutions found are often smaller than those found by the traditional GP technique. He subsequently showed how one can add another “architecture altering” layer to the GP process to allow ADF architecture to evolve during a run [14]. Other researchers have explored the utility of “run-transferable” libraries of code that allow modules evolved in one problem-solving episode to be used again in later episodes in the same problem domains [20, 9].

Several other approaches to the evolution of modular programs have been explored in the GP literature more recently, some of which allow modular architectures to evolve dynamically or to be transferred across runs (e.g. [15, 8, 5, 33, 22, 35]). Additional work has been done on the theoretical analysis of modularity in evolutionary algorithms more generally [34]. Space constraints prohibit detailed discussion of these or other approaches to evolving modular programs; such a survey and comparisons to the method presented here are a topic for future work. We believe that the method presented here has merit on its own terms, and that the core mechanism of tag-based modularity is novel and worthy of further study.

3. MODULES IN PUSH

Push is a programming language designed specifically for use in evolutionary computation systems, as the language in which evolving programs are expressed [25, 31, 29]. Push is a stack-based language (among others that have been used for GP, e.g. [16]) with the novel feature being that a separate stack is used for each data type. Instructions are implemented to take their arguments from—and leave their results on—stacks of the appropriate types, which allows instructions and literals to be freely intermixed regardless of type while still ensuring execution safety. Instructions that find insufficient arguments on the relevant stacks act as “no-ops” (that is, they do nothing). Push implementations now exist in C++, Java, JavaScript, Python, Common Lisp, Clojure, Scheme, Erlang, and R. Many of these are available for free download from the Push project page.¹

Many of Push’s most powerful features stem from the fact that “code” is itself a Push data type, and from the fact that Push programs can easily (and often do) manipulate their own code as they run. This capability is supported by a variety of code-manipulating instructions that act on code stored on two stacks: the “exec” stack, from which the interpreter iteratively fetches program fragments for execution, and the “code” stack, which is treated as any other data stack. The code stack can be used for “off line” manipulation of arbitrary code which may later be transferred to the exec stack for execution. These data types and their associated instructions allow an evolved program to transform code in ways that produce the effects of ADFs or automatically defined macros [24] without pre-specification of the number of modules and without new mechanisms such as

¹<http://hamphshire.edu/lspector/push.html>

architecture-altering operations [14]. They support conditionals, recursion, iteration, combinators, co-routines, and novel control structures, all through combinations of the built-in code-manipulation instructions. This allows a Push-based system to evolve modular programs in a particularly simple and flexible way, even if the system is otherwise quite simple. All of the work described here uses PushGP, which was designed to be as simple and generic as possible (e.g. it has no ADFs, strong typing mechanisms, or syntactic constraints) aside from using Push as the language for evolving programs.

A particularly simple form of modularity supported by Push involves `exec` stack manipulation. Consider the program `(3 exec.dup (1 integer.+))`. This program first pushes 3 onto the integer stack and then executes `exec.dup`. The `exec.dup` instruction duplicates the item that is on the top of the `exec` stack. At the time that `exec.dup` is executed `(1 integer.+)` will be on top of the `exec` stack, so after the execution of `exec.dup` there will be *two* instances of `(1 integer.+)` on top of the `exec` stack. They will both be executed. The first will push 1 onto the integer stack (on top of the 3) and then add 3 and 1, leaving 4 on top of the integer stack. Then the second instance of `(1 integer.+)` will then be executed: it will push a 1 on top of the 4 and then add 4 and 1, leaving 5 on top of the integer stack. This shows a simple module for adding 1 to an integer being run twice; as such it is a minimal instance of modular code reuse.

Other forms of modularity can be implemented using other stack-manipulation instructions on the `exec` stack, for example `exec.swap` (which swaps the top two items on the `exec` stack) and `exec.rot` (which rotates the top three items on the `exec` stack), perhaps in conjunction with other calls to `exec.dup`. And much more exotic forms of modularity can be implemented by manipulating code in arbitrary ways prior to execution; the full Push instruction set includes a rich suite of code manipulation instructions inspired by Lisp’s list manipulation functions, including versions of `car`, `cdr`, `list`, `append`, `subst`, etc.

Another elegant way to implement complex, modular control structures in Push involves the use of the *K*, *S* and *Y* *combinator* instructions, which are derived from combinatory logic [21, 3]. The `exec.k` combinator instruction simply removes and discards the second element from the `exec` stack. The `exec.s` combinator instruction pops three items, *A*, *B* and *C* from the `exec` stack and then pushes back three separate items: *(B C)*, *C* and *A* (leaving the *A* on top). This produces two calls to *C*, so it has a `dup`-like effect in terms of potentially inducing a module. Note that *C* might itself be a large, complex expression. The `exec.y` combinator inspects (but does not pop) the top of the `exec` stack, *A*, and then inserts the list `(exec.y A)` as the second item on the `exec` stack. This generates a recursive call that can be terminated through further manipulation of the `exec` stack; this further manipulation may occur within *A*.

These various mechanisms have been shown to support the evolution of programs with rich, evolved control structures that are in some senses modular and that solve a wide range of problems. For example, even before the introduction of the `exec` stack manipulation instructions (which appeared with “Push 3” in 2005 [29]) Push’s code manipulation mechanisms had been shown to automatically produce modular solutions to parity problems [31] and to induce modules in response to a dynamic fitness environment [32]. More recent

work has produced solutions to a wide range of standard problems (including list reversal, factorial regression, Fibonacci regression, parity, exponentiation, and sorting [29]) along with the production of human-competitive results in quantum circuit design [26] and pure mathematics [27].

Nonetheless, one modularity-supporting facility of Push—the facility for naming values—has failed to produce significant results even though it is arguably the mechanism closest in spirit to the tools most commonly used by human programmers and even though it has been revised repeatedly with an aim toward making the use of naming more evolvable. The most recent Push specification (version 3.0, [30]) allows one to name a module with an expression such as `(plus1 exec.define (1 integer.+))`. When the Push interpreter sees `plus1` for the first time it does not recognize it as an instruction, so it pushes it onto the “name” stack. The `exec.define` instruction takes the name on top of the name stack (which will be `plus1`) and binds it to the item on top of the `exec` stack (which will be `(1 integer.+)`). Henceforth when the Push interpreter sees `plus1` it will treat it essentially as an instruction that pushes its bound value onto the `exec` stack. This implements a named module for adding 1 to the top item on the integer stack, and of course one could do the same thing with any arbitrarily complex module code.

When using names, however, the question arises of how many names to include in the instruction set, or of how many names to allow an ephemeral random constant mechanism to introduce into the population. If the number is very small then the number of modules that can be used will be quite limited, but on the other hand if it is *not* very small then it will be extremely unlikely that random programs will successfully bind and then reference the same names. A previous approach to this problem was to start with a very small number of names and to allow this number to grow only slowly, but this is ad hoc and unsatisfying, and in any case it has not been shown to be effective. It is in this context that the idea of tag-based modules was developed as a more principled and potentially more useful alternative to name-based modules.

4. TAG-BASED MODULES

The concept of a “tag” that we use here derives from Holland’s work on general principles of complex adaptive systems [6, 7], but it has also been adopted in more specific contexts such as the evolution of cooperation [18, 4, 28]. The essential idea of a tag is that it is a mechanism that serves to allow selective binding or aggregation through matching, even though specific tags may initially have no intrinsic meaning. Examples given by Holland include banners or flags used by armies and “the ‘active sites’ that enable antibodies to attach themselves to antigens” [7, p. 13]. Particularly in the work on the evolution of cooperation the aspect of *inexact* tag-matching is stressed. For example, in the model of Rilo, Cohen, and Axelrod an agent will donate to a second agent if the tags of the two agents are more similar than allowed by the “tolerance” threshold of the donor, and both tags and tolerances are allowed to change over evolutionary time [18].

Our implementation of tags in Push does not reify tags as an independent data type; instead, we add instructions for tagging and tag-based reference that incorporate the tags into the instruction names. Tags themselves are simply integers between zero and a pre-set maximum—this choice is

Table 1: Push instructions available for use in evolved programs for the lawnmower problem (see text for instruction descriptions).

Condition	Instructions
Basic	left, mow, v8a, frog, \mathcal{R}_{v8}
Tag	left, mow, v8a, frog, \mathcal{R}_{v8} , tag.exec.[1000], tagged.[1000]
Exec	left, mow, v8a, frog, \mathcal{R}_{v8} , exec.dup, exec.pop, exec.rot, exec.swap, exec.k, exec.s, exec.y

arbitrary but allows for a simple implementation—and the tagging and tag-based reference instructions are produced during code generation by an ephemeral random constant mechanism. The associations between tags and tagged values are stored in an auxiliary data structure (not on a stack), much as name bindings have long been stored in Push. For example, an instruction such as `tag.exec.123` pops the exec stack and creates a binding, stored in the auxiliary data structure, between the tag 123 and the popped value. A later call to the instruction `tagged.123` will push the associated value back onto the exec stack, from whence it will next be executed. Inexact matching is implemented in a particularly simple way: if there is no exact match then we count upwards until a stored tag is reached, wrapping back to zero if the maximum is reached without finding a stored tag. So in the previous example `tagged.100` will have the same effect as `tagged.123` as long as nothing has been tagged with a tag that is greater than or equal to 100 and less than 123. If no value has been tagged at all then a call to a `tagged` instruction will act as a no-op; otherwise every `tagged` instruction will refer to some stored value and retrieve it for execution.

Our full implementation of the tags includes not only `tag.exec` and `tagged` instructions but also `untag` instructions for removing tag/value associations and variants that operate on stacks other than the exec stack. For the experiments presented here, however, only the `tag.exec` and `tagged` instructions were used.

5. LAWNMOWER PROBLEM RESULTS

In the lawnmower problem the goal is to completely “mow” a virtual lawn with a programmable virtual lawnmower [13]. In the original version of the problem, the grid contains 64 lawn squares arranged in an 8 x 8 grid with location (0,0), the lawnmower’s initial location, in the upper left. To formulate this problem for traditional GP, Koza specified a terminal set containing (`left`), an operator that takes no arguments and rotates the lawnmower 90° counterclockwise, (`mow`), an operator that takes no arguments and moves the lawnmower one space forward, mowing the grass that is in the destination square (and wrapping around toroidally if necessary), and the ephemeral random constant generator \mathcal{R}_{v8} that can produce constant vectors of the form (i, j) where i and j range from 0 to 7. He specified a function set containing `V8A`, a two-argument vector addition function modulo 8, `FROG`, a one-argument operator that jumps the lawnmower ahead and sideways an amount indicated by its vector argument (mowing the destination square), and `PROGN`, a two argument sequencing function. He allowed a program to execute a total of 100 turns (calls to (`left`)) or

Table 2: Parameters for PushGP runs on the lawnmower problem and the dirt-sensing, obstacle-avoiding robot problems. Additional details of the runs may be found in the source code at <http://hampshire.edu/l spectator/tags-gecco-2011>.

fitness	squares unmowed/unmopped (lower is better)
runs per condition	100
population size	1000
max generations	1001
tournament size	7
mutation percent	45
crossover percent	45
reproduction percent	10
node selection	90% internal nodes, 10% leaves
limits on moves and turns	problem size 8x4: 50 problem size 8x6: 75 problem size 8x8: 100 problem size 8x10: 125 problem size 8x12: 150
limits on program size (in points) and execution steps	problem size 8x4: 500 problem size 8x6: 750 problem size 8x8: 1000 problem size 8x10: 1250 problem size 8x12: 1500

Table 3: Number of runs that succeeded, out of 100, in each combination of problem size (columns) and instruction set (rows), on the lawnmower problem.

	8x4	8x6	8x8	8x10	8x12
Basic	100	100	99	85	66
Tag	100	100	100	100	99
Exec	100	100	100	100	99

100 movement operators (calls to (`mow`) or `FROG`) before it would be aborted.² Although neither the toroidal structure of the lawn nor the `FROG` function relate particularly well to real lawnmower environments, this function set nonetheless defines a clear problem that involves regularity that can be exploited by modular systems.

In Push there is no distinction between functions and terminals. Instead the relevant distinction is between instructions and literals, with these being distinguished only by the fact that instructions are callable. There is also no need for a dedicated sequencing function because sequencing is implicit. To implement the lawnmower problem for PushGP we added a stack for 2D vectors (of the form (i, j)), used instructions `left`, `mow`, `v8a`, `frog`, each of which was identical to Koza’s operators aside from taking arguments from stacks and pushing results onto stacks, and used an ephemeral random constant generator \mathcal{R}_{v8} .

We ran tests in this “basic” condition and also in two other conditions: “tag” in which we also allowed instructions of the form `tag.exec.i` and `tagged.i` where i could range from 0 to 999, and “exec” in which we also allowed standard exec stack manipulation and combinator instructions. Table 1 shows the instruction sets for all three conditions.

²A program would be aborted when it reached 100 moves or 100 turns, whichever came first.

Table 4: Mean best fitnesses (lower is better) achieved in each combination of problem size (columns) and instruction set (rows), on the lawn-mower problem.

	8x4	8x6	8x8	8x10	8x12
Basic	0	0	0.01	0.42	1.47
Tag	0	0	0	0	0.01
Exec	0	0	0	0	0.10

In order to explore how performance scaled with problem size we followed Koza in considering lawns of five sizes: 8x4, 8x6, 8x8, 8x10, and 8x12. It is not clear from Koza’s text if or exactly how other parameters of the problem—such as program size limits, execution step limits, turn and move limits, ranges of constants, and the modulus used on the results of vector addition—were scaled along with the lawn size, but we thought it was reasonable to scale all limits in proportion to the lawn size; see Table 2 for the specific values that we used, along with the other parameters for our PushGP runs. We also scaled the ranges of constants and the modulus use on the results of vector addition, ensuring that all and only valid coordinates could arise (but we retained the names `v8a` and \mathcal{R}_{vs} for historical consistency).

In our implementation the initial placement of the mower does not cause the starting square to be mowed; the mower must move to that square again in order to mow it.

We examined the results by looking at the number of successful runs, the mean best fitness achieved, and the “computational effort” of finding a solution in each condition. Computational effort was computed as specified by Koza [12, pp. 99–103], by first calculating $P(M, i)$, the cumulative probability of success by generation i with population size M ; this is the number of runs that succeeded on or before the i th generation, divided by the number of runs conducted. $I(M, i, z)$, the number of individuals that must be processed to produce a solution by generation i with probability greater than z (here $z = 99\%$), is then calculated as:

$$I(M, i, z) = M * (i + 1) * \left\lceil \frac{\log(1 - z)}{\log(1 - P(M, i))} \right\rceil$$

The minimum of $I(M, i, z)$ over all values of i is defined to be the “computational effort” required to solve the problem.

Table 3 shows the percentage of runs that succeeded and Table 4 shows the mean best fitnesses achieved in each condition. Figure 1 shows a graph of computational effort for each instruction set as the size of the problem increases. We can see that the problem is generally quite easy but that performance of the basic instruction set drops off substantially as the problems get bigger. This is qualitatively similar to the results obtained by Koza [13, pp. 266–267], although we would not expect the numbers to match exactly since the Push representation and the standard tree representation are not identical and several other parameters also differ. Nonetheless it is clear that performance degrades badly without access to modularity mechanisms, while both of the modularity-facilitating instruction sets that we provided scaled much better. We hypothesized that the good performance in the “exec” condition, but not in the “tag” condition, relied on the extreme uniformity of the lawn-mower problem. To test this hypothesis we next experimented with a somewhat more difficult and less uniform problem.

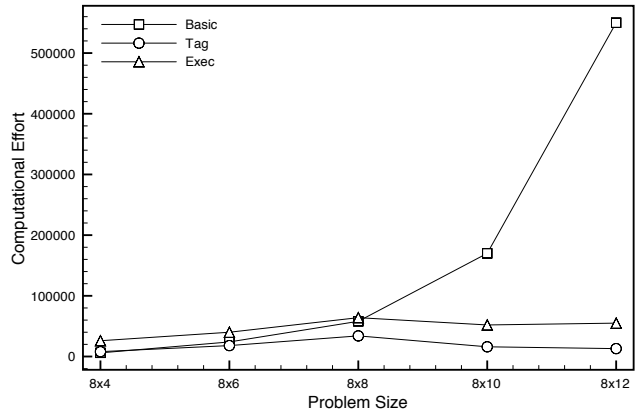


Figure 1: Computational effort required to solve the lawn-mower problem at various sizes and with various instruction sets.

Table 5: Push instructions available for use in evolved programs for the dirt-sensing, obstacle-avoiding problem (see text for instruction descriptions).

Condition	Instructions
Basic	if-dirty, if-obstacle, left, mop, v8a, frog, \mathcal{R}_{vs}
Tag	if-dirty, if-obstacle, left, mop, v8a, frog, \mathcal{R}_{vs} , tag.exec.[1000], tagged.[1000]
Exec	if-dirty, if-obstacle, left, mop, v8a, frog, \mathcal{R}_{vs} , exec.dup, exec.pop, exec.rot, exec.swap, exec.k, exec.s, exec.y

6. DIRT-SENSING, OBSTACLE-AVOIDING ROBOT RESULTS

The “dirt-sensing, obstacle-avoiding robot problem” [24] is a variant of the “obstacle-avoiding robot problem,” which is itself a variant of the lawn-mower problem. The obstacle-avoiding robot problem [12] is the same as the lawn-mower problem except that the lawn-mower metaphor is dropped in favor of a floor-mopping robot metaphor (which has no practical impact, although the `mop` function is replaced with `mop`), there are obstacles through which the robot cannot pass (and there is no dirt to mop in the squares with obstacles³), and a new `if-obstacle` conditional branching operator is added to the function set; `if-obstacle` executes its first argument if the robot is currently facing an obstacle, or its second argument otherwise. In the dirt-sensing, obstacle-avoiding robot problem we also add a new `if-dirty` conditional branching operator that executes its first argument if the robot is currently facing a dirty (un-mopped) square, or its second argument otherwise. These new functions are translated into Push instructions by taking the arguments from the exec stack.

Table 5 shows the instructions used in each condition; the other parameters were identical with those for the lawn-

³It is possible for there to be an obstacle in the starting square; this causes no conflicts, as the existence of an obstacle only prevents *entry* to a square. Attempts to `mop` or `frog` to a square with an obstacle are no-ops (they do nothing).

Table 6: Obstacle locations for the dirt-sensing, obstacle-avoiding robot problem. Two obstacle configurations were used for each problem size.

Size	Obstacle locations for each of two rooms.
8x4	$\{(2,2)(1,1)(3,1)(5,2)(6,2)\}$ $\{(3,2)(3,1)(3,0)(4,0)(0,3)\}$
8x6	$\{(0,0)(4,3)(7,4)(3,0)(5,0)(0,2)\}$ $\{(5,5)(4,3)(2,0)(7,4)(7,1)(0,4)\}$
8x8	$\{(4,3)(6,5)(7,6)(6,3)(4,1)(0,7)(3,6)(3,5)\}$ $\{(3,3)(6,5)(4,2)(3,0)(6,2)(4,0)(0,7)(2,7)\}$
8x10	$\{(0,0)(3,0)(5,0)(7,1)(0,7)(2,9)(2,8)(4,9)\}$ $\{(5,6)(0,1)(2,2)(2,0)(6,1)(3,6)(5,8)(6,8)\}$
8x12	$\{(6,5)(2,0)(5,0)(0,9)(0,8)(2,10)(5,9)(6,10)(4,7)\}$ $\{(6,6)(4,1)(4,0)(1,9)(0,7)(3,8)(6,11)(1,5)(4,6)\}$

Table 7: Number of runs that succeeded, out of 100, in each combination of problem size (columns) and instruction set (rows), on the dirt-sensing, obstacle-avoiding robot problem.

	8x4	8x6	8x8	8x10	8x12
Basic	61	35	1	0	0
Tag	86	74	17	4	2
Exec	55	44	4	0	1

Table 8: Mean best fitnesses (lower is better) achieved in each combination of problem size (columns) and instruction set (rows), on the dirt-sensing, obstacle-avoiding robot problem.

	8x4	8x6	8x8	8x10	8x12
Basic	1.04	3.97	9.83	11.39	25.15
Tag	0.15	0.42	2.4	3.71	5.16
Exec	2.04	2.97	9.78	17.77	33.35

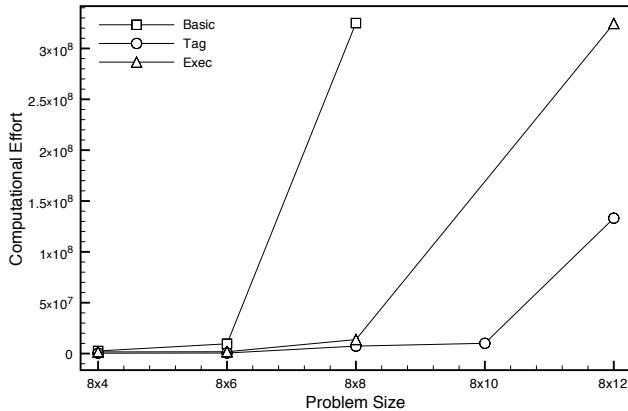


Figure 2: Computational effort required to solve the dirt-sensing, obstacle-avoiding robot problem at various sizes and with various instruction sets. Missing data points reflect conditions in which no successes were achieved, in which case the computational effort is undefined.

mower problem runs (see Table 2). We tested each individual on two rooms with different configurations of obstacles, summing the fitnesses of the two rooms. The number of obstacles was calculated as the floor of the square root of the room area: 8x4: 5, 8x6: 6, 8x8: 8, 8x10: 8, 8x12: 9. The specific obstacle locations were initially generated randomly but were then held constant through all experiments; they are listed in Table 6.

Table 7 shows the percentage of runs that succeeded and Table 8 shows the mean best fitnesses for each condition. Figure 2 shows a graph of computational effort for each instruction set as the size of the problem increases. We can see that this problem is quite a bit more difficult than the lawnmower problem, and that the basic instruction set fails completely for the larger problem instances. In fact, this problem is difficult in all conditions, but the tag condition is the clear winner overall, according to all three measures.

This data appears to lend credence to our hypothesis that tags provide a more flexible modularization mechanism than exec stack manipulation, and that tag-based modularity is likely to prove more useful than exec-stack-based modularity for complex, non-uniform problems.

7. TAGS IN OTHER FORMS OF GP

The core ideas of tag-based modularity are independent of Push, and there are a variety of ways in which tags might be incorporated into GP systems that use different representations.

It is easiest to see how this could be done in systems that already allow some form of dynamic function definition, including systems in which new functions arise at run time (e.g. through function definition calls) and also systems in which new functions arise during the reproductive phase (as with architecture altering operations [13]). In these cases one could encode tags into function names just as we have done here, and one could dispatch function calls on the basis of tag similarity. Depending on the program representation one might have to take steps to ensure that the function signatures (numbers and types of function parameters) match between definitions and calls; this might be done, for example, by using only a single signature for all functions or by dispatching to the closest matching function with the correct signature. It is reasonable to expect that these techniques would produce effects that are similar to those that we have documented here, but clearly the dynamics of the underlying function definition system will matter.

The concepts of tag-based modularity may have worthwhile applications even in tree-based systems without dynamic function definition. One simple idea is to support calls to one-argument functions of the form `tag-i` which act to tag the code in their arguments with the tags embedded in their names (and presumably return either some constant value or the results of evaluating their arguments). One would have to alter the system's code generation routines to produce these function calls, and also to produce calls to zero-argument functions of the form `tagged-i`. Calls to the `tagged` functions would branch to the code of the tagged code with the closest matching tag (or presumably return some constant value if no code had yet been tagged). This would provide a form of dynamic function definition, in which function reference occurs through inexact tag matching, but it would only produce zero-argument functions. It

could also produce unbounded recursion, so some form of execution step limit would have to be imposed.

By use of similar substitutions, substituting tag-based definition and reference for name-based definition and reference, one should be able to evolve tag-based modularity in a wide range of GP systems.

8. CONCLUSIONS AND FUTURE WORK

We have introduced a new technique for evolving modular programs, based on tags which may be matched inexactly. We have shown that this technique, implemented in the PushGP system, allows problem-solving effort to scale well with problem size using the lawnmower problem, which was introduced by Koza to demonstrate similar scaling properties for his ADF technique (which we argue is less general or, when used in conjunction with architecture-altering operations, more complex than the new technique that we present here). Our data also showed that execution stack manipulation, which has long been available in PushGP, also scales well on the lawnmower problem. We hypothesized that the strong performance of exec stack manipulation on this problem, but not the strong performance of the new tag-based technique, relied upon the extreme uniformity of the lawnmower problem. Additional runs on a more difficult, less uniform problem (the dirt-sensing, obstacle-avoiding robot problem) produced data consistent with this hypothesis.

One avenue for further investigation concerns the reasons for the good performance of the tag-based modularity technique. Our reasons for expecting it to perform well were related to the use of inexact matching, and our expectation that this would allow uses of single modules to arise readily because any reference will “find” a referent as long as a single value has been tagged. We also expected this to allow additional modules to be added incrementally over evolutionary time. But we have not yet done the analysis to discover whether or not this is really the case. One interesting study would examine programs and their uses of modules over evolutionary time, to see if the expected incremental increases are really happening. Another would compare the tag-based mechanism to names that must match exactly.

Another area for future work concerns the application of tag-based modularity to other forms of GP, as discussed briefly above.

In the work we presented here we chose to build tags into definition and reference instruction names, but another option in Push would be to reify tags as objects of their own type, with their own stack, and use definition and reference instructions that get tags from the tag stack. This would make tag use less parsimonious but it might permit a wider range of tag use strategies.

Finally, future work should compare the effectiveness of tag-based modularity mechanisms to that of the many other modularity mechanisms that have recently been presented in the GP literature. This is a big job because there are many such mechanisms and because many of them are tied more or less directly to particular GP paradigms (such as cartesian genetic programming or grammatical evolution). Still, it is important that this be done, because the complex programs that we would all like to evolve in the future must surely take advantage of modularity, and it is important for the

field that we understand the best ways in which to evolve modular programs.⁴

Acknowledgements

Thanks to Daniel Gerow, Nathan Whitehouse, and Rebecca Neimark for feedback that helped us to improve this work in several ways, to Josiah Erikson for systems support, and to Hampshire College for support for the Hampshire College Institute for Computational Intelligence. Thanks also to Jordan Pollack and the Brandeis DEMO lab for computational resources. This material is based upon work supported by the National Science Foundation under Grant No. 1017817. 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.

9. REFERENCES

- [1] P. J. Angeline and J. B. Pollack. Evolutionary module acquisition. In D. Fogel and W. Atmar, editors, *Proceedings of the Second Annual Conference on Evolutionary Programming*, pages 154–163, La Jolla, CA, USA, 1993.
- [2] W. S. Bruce. The lawnmower problem revisited: Stack-based genetic programming and automatically defined functions. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 52–57. Morgan Kaufmann, 1997.
- [3] H. Curry and R. Feys. *Combinatory Logic*, 1, 1958.
- [4] D. Hales. Evolving specialisation, altruism, and group-level optimisation using tags. In *Proceedings of the 3rd international conference on Multi-agent-based simulation II*, MABS’02, pages 26–35, Berlin, Heidelberg, 2003. Springer-Verlag.
- [5] E. Hemberg, C. Gilligan, M. O’Neill, and A. Brabazon. A grammatical genetic programming approach to modularity in genetic algorithms. In *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2007.
- [6] J. Holland. The effect of labels (tags) on social interactions. Technical Report Working Paper 93-10-064, Santa Fe Institute, Santa Fe, NM, 1993.
- [7] J. H. Holland. *Hidden Order: How Adaptation Builds Complexity*. Perseus Books, 1995.
- [8] I. Jonyer and A. Himes. Improving modularity in genetic programming using graph-based data mining. In G. C. J. Sutcliffe and R. G. Goebel, editors, *Proceedings of the Nineteenth International Florida Artificial Intelligence Research Society Conference*, pages 556–561. American Association for Artificial Intelligence, 2006.
- [9] M. Keijzer, C. Ryan, G. Murphy, and M. Cattolico. Undirected training of run transferable libraries. In *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 361–370. Springer, 2005.

⁴Note: A minor error was found in this paper as it was going to press; see explanatory note at <http://hampshire.edu/lsp/lector/tags-gecco-2011>.

- [10] K. E. Kinnear, Jr. Alternatives in automatic function definition: A comparison of performance. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 6, pages 119–141. MIT Press, 1994.
- [11] J. Koza. Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical Report STAN-CS-90-1314, Dept. of Computer Science, Stanford University, June 1990.
- [12] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
- [13] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.
- [14] J. R. Koza, David Andre, F. H. Bennett III, and M. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufman, Apr. 1999.
- [15] X. Li, C. Zhou, W. Xiao, and P. C. Nelson. Direct evolution of hierarchical solutions with self-emergent substructures. In *The Fourth International Conference on Machine Learning and Applications (ICMLA '05)*, pages 337–342. IEEE press, 2005.
- [16] T. Perkis. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 148–153. IEEE Press, 1994.
- [17] A. Racine, M. Schoenauer, and P. Dague. A dynamic lattice to evolve hierarchically shared subroutines: DL'GP. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 220–232. Springer-Verlag, 1998.
- [18] R. L. Riolo, M. D. Cohen, and R. Axelrod. Evolution of cooperation without reciprocity. *Nature*, 414:441–443, 2001.
- [19] S. C. Roberts, D. Howard, and J. R. Koza. Evolving modules in genetic programming by subtree encapsulation. In *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 160–175. Springer-Verlag, 2001.
- [20] C. Ryan, M. Keijzer, and M. Cattolico. Favorable biasing of function sets using run transferable libraries. In U.-M. O'Reilly, T. Yu, R. L. Riolo, and B. Worzel, editors, *Genetic Programming Theory and Practice II*, chapter 7, pages 103–120. Springer, 2004.
- [21] M. Schönfinkel. Über die bausteine der mathematischen logik. *Mathematische Annalen*, 92:307–316, 1924.
- [22] S. Shirakawa and T. Nagao. Graph structured program evolution with automatically defined nodes. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1107–1114. ACM, 2009.
- [23] H. Simon. The architecture of complexity. In H. A. Simon, editor, *The Sciences of the Artificial*, pages 84–118. The MIT Press, 1969.
- [24] L. Spector. Simultaneous evolution of programs and their control structures. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 7, pages 137–154. MIT Press, Cambridge, MA, USA, 1996.
- [25] L. Spector. Autoconstructive evolution: Push, pushGP, and pushpop. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 137–146. Morgan Kaufmann, 2001.
- [26] L. Spector. *Automatic Quantum Computer Programming: A Genetic Programming Approach*, volume 7 of *Genetic Programming*. Kluwer Academic Publishers, Boston/Dordrecht/New York/London, 2004.
- [27] L. Spector, D. M. Clark, I. Lindsay, B. Barr, and J. Klein. Genetic programming for finite algebras. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1291–1298. ACM, 2008.
- [28] L. Spector and J. Klein. Genetic stability and territorial structure facilitate the evolution of tag-mediated altruism. *Artificial Life*, 12(4):1–8, 2006.
- [29] L. Spector, J. Klein, and M. Keijzer. The Push3 execution stack and the evolution of control. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1689–1696. ACM Press, 2005.
- [30] L. Spector, C. Perry, J. Klein, and M. Keijzer. Push 3.0 programming language description. Technical Report HC-CSTR-2004-02, School of Cognitive Science, Hampshire College, USA, 10 Sept. 2004.
- [31] L. Spector and A. Robinson. Genetic programming and autoconstructive evolution with the Push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, Mar. 2002.
- [32] L. Spector and A. Robinson. Multi-type, self-adaptive genetic programming as an agent creation tool. In A. M. Barry, editor, *GECCO 2002: Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference*, pages 73–80. AAAI, 2002.
- [33] J. A. Walker and J. F. Miller. The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 12(4):397–417, Aug. 2008.
- [34] R. P. Wiegand, G. Anil, I. I. Garibay, O. O. Garibay, and A. S. Wu. On the performance effects of unbiased module encapsulation. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1729–1736. ACM, 2009.
- [35] G. Wijesinghe and V. Ciesielski. Evolving programs with parameters and loops. In *IEEE Congress on Evolutionary Computation (CEC 2010)*. IEEE Press, 2010.