Tag-based Modularity in Tree-based Genetic Programming

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

Kyle Harrington
DEMO Lab
Brandeis University
Waltham, MA 02453 USA
kyleh@cs.brandeis.edu

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

ABSTRACT

Several techniques have been developed for allowing genetic programming systems to produce programs that make use of subroutines, macros, and other modular program structures. A recently proposed technique, based on the "tagging" and tag-based retrieval of blocks of code, has been shown to have novel and desirable features, but this was demonstrated only within the context of the PushGP genetic programming system. Following a suggestion in the GECCO-2011 publication on this technique we show here how tag-based modules can be incorporated into a more standard tree-based genetic programming system. We describe the technique in detail along with some possible extensions, outline arguments for its simplicity and potential power, and present results obtained using the technique on problems for which other modularization techniques have been shown to be useful. The results are mixed; substantial benefits are seen on the lawnmower problem but not on the Boolean even-4-parity problem. We discuss the observed results and directions for future research.

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

Genetic programming, modularity, tags, automatically defined functions, lawnmower problem, parity problem

1. INTRODUCTION

In order for genetic programming [10] to become a scalable, general-purpose automatic programming methodology, its practitioners will have to have access to methods by which they can automatically evolve programs that incorporate modular structures such as functions, macros, subroutines, and co-routines. A variety of methods have been developed for allowing genetic programming to evolve such programs; the most well known of these is Koza's "Automatically Defined Function" (ADF) framework [10, 11] but a variety of other approaches have also been developed [9, 10, 1, 8, 19, 2, 15, 17, 18, 7, 26].

The methods for genetic programming with modules that are most widely used require pre-specification of the module "architecture"—that is, of the number of modules and the numbers and types of arguments that they take—or otherwise limit the generality of the module architectures that can arise automatically during evolution. The exceptions to this generalization have mostly involved much more cumbersome and *ad hoc* mechanisms to support the evolution of architectures. For example, Koza's "architecture-altering operations" permit one to evolve populations of programs with diverse architectures, and they allow for architectures to become more complex over evolutionary time, but many of the architectural modifications in Koza's framework require elaborate and potentially destructive repair strategies to be applied after architectural changes [12]. For example, if an architecture is altered by deleting an automatically defined function, or by changing the number of arguments that an automatically defined function takes, then all calls to the deleted or modified function must be changed; it is not generally clear *how* they would best be changed, and the system designer's decisions on these matters will influence the power of the evolutionary system.

A recently-developed alternative technique for evolving programs that use modules, based on the use of "tags," has shown some promise insofar as it permits arbitrary architectures to evolve, even though the changes that must be made to the genetic programming system are relatively few and relatively simple [24, 21]. The relevant notion of "tag" comes from the work of Holland, who used the term to denote initially arbitrary identifiers that come to have meaning over time in many different kinds of complex adaptive systems [5, 6]. He described examples of tagging involving biological and cultural systems, ranging from immune systems to armies on a battlefield, and he argued that tag-based matching is a general tool for the support of emergent complexity. The concept has been applied by several researchers over the last decade or so to the study of the emergence of cooperation and altruism (e.g. [16, 3, 22]), but its application

GECCO'12, July 7–11, 2012, Philadelphia, Pennsylvania, USA.

to the emergence of modularity in genetic programming is relatively new.

Tag-based modularity in genetic programming has been described in the literature previously, but only in the context of genetic programming with the Push programming language, using the PushGP genetic programming system [20, 25, 23]. Because "tree-based" genetic programming, in which one evolves programs in the form of Lisp-like symbolic expressions, is currently among the most popular forms of genetic programming, our goal in the research described in this paper is to see if and how the ideas of tag-based modularity can be applied in the tree-based genetic programming framework. A simple scheme for doing this was described briefly in the prior work on tag-based modularity [24]; here we describe this simple scheme in detail, including nuances not presented in the prior work, present results of its performance on two problems, and discuss its limitations and possible extensions.

In the following sections we first briefly describe the tag-based modularity technique as it has been previously applied in the context of PushGP. We then describe the simplest version of the idea that can be easily applied in tree-based genetic programming. Following this, we present the results of runs of the system that we have described on the lawnmower and Boolean even-4-parity problems. The mixed results that we document—positive for tags on the lawnmower problem but negative for tags on the Boolean even-4-parity problem—lead to a discussion of extensions and open issues in the subsequent section. We conclude with a discussion of suggestions for future work in this area.

2. TAGS IN PUSH

The previous work on tag-based modularity in genetic programming has been conducted in the context of the PushGP genetic programming system, in which the evolved programs are expressed in the Push programming language [20, 25, 23]. Push is a stack-based programming language with an independent stack for each data type, including a stack for code and a separate "exec" stack for code that is queued for execution. Full descriptions of the Push programming language and of PushGP are beyond the scope of the present paper, but sufficient detail will be presented here to describe how programs with tag-based modules can be evolved in PushGP.

The essential idea of tag-based modularity is that a block of code—a module—is *tagged* with an initially arbitrary identifier that permits retrieval based on possibly-inexact matching. For simplicity the tags used in prior work have been positive integers, and the closest match has been determined by numerical difference in one direction. If we refer to a tag t_{ref} then the closest match, among the tags that have been used to tag values, will be the smallest tag t_{match} for which $t_{ref} \leq t_{match}$. If no used tag is greater than or equal to t_{ref} then we consider the numerically lowest tag to be the closest—that is, we wrap around. For example, if the tags that have been used to tag values are $\{123, 456, 789\}$ and we retrieve a value by referring to tag 200, then we will retrieve whatever value had been tagged with 456. A reference to tag 800 will retrieve the value tagged with 123.

The PushGP implementation of tag-based modularity includes instructions that can tag any data item and retrieve items by closest match to a tag. Because code in Push is also data, tag-based modularity can be implemented by allowing the tagging of code and the tag-based retrieval of code to the exec stack, from where it will then be executed. For example, the following Push program[1] first defines and then twice uses a tag-based module for squaring an integer:

```
(tag.exec.123 (integer.dup integer.*)
  3 tagged.100 tagged.107)
```

The first instruction here, `tag.exec.123`, is produced by an "ephemeral random instruction" token in the instruction set that is much like the ephemeral random constants employed in traditional genetic programming systems [10] except that here we generate instructions that incorporate randomly-chosen tag values. This particular tag instruction pops and tags the item on top of the exec stack (which will be the expression `(integer.dup integer.*)` when this instruction executes) with the tag 123. Then `3` will be executed, and since this is an integer literal it will be pushed onto the integer stack. The call to `tagged.100` will retrieve the item that has been tagged with the tag that most closely matches 100 and push that item onto the exec stack, from where it will then be executed. The retrieved item will be `(integer.dup integer.*)` in this case; when executed this will duplicate the top item on the integer stack and then multiply the top two items, effectively squaring it and leaving 9 on top of the integer stack. Next, the call to `tagged.107` will be processed, with the final result on the integer stack being 81.

Several potentially problematic special cases that might arise in tag-based modularity are handled in natural ways in the Push implementation but may require more elaborate solutions in tree-based genetic programming, as will be discussed below. For example, if one attempts to retrieve a value based on a tag when no data has yet been tagged then the natural approach in Push is simply to treat the retrieval instruction as a "no-op," leaving the stacks unchanged. In tree-based genetic programming every sub-expression must return a value, so *some* value must be returned from a retrieval expression in every circumstance. Similarly, it is quite possible for programs that use tags to loop/recurse infinitely, but because many of Push's other code-manipulation instructions also permit the expression of non-terminating algorithms, all Push implementations already enforce execution step limits and allow for the delivery of results from the stacks even when the step limits have been reached.

It is also relevant that in Push, arguments are passed to instructions via stacks and values are returned from instructions via stacks. This means that a tagged chunk of code may access any number of inputs (which could be considered to be function arguments) from the stacks even though there is no syntactic restriction on the code that corresponds to the number of arguments that it may take. Indeed, a mutation to code that gets tagged may cause it to "take" more or fewer arguments, but this need not be reflected anywhere else in the program's syntax. A tagged chunk of code may also "return" multiple values to its caller via the stacks, without any requirement that this be explicitly indicated or maintained in the program's syntax.

As a result of the Push features described above, the incorporation of tag-based modularity into PushGP is partic-

[1]Different implementations of Push use slightly different notational conventions; for example some use "_" in place of "." and have somewhat different instruction names. Here we use the notational conventions that have been used most frequently in prior Push publications.

ularly simple and the resulting system is particularly flexible. Any number of tagged modules, each of which takes any number of arguments and returns any number of values, can arise during evolution without any pre-specification of the numbers of modules or arguments. The "closest match" semantics ensures that any reference to a tag will retrieve *something* as long as *something* has been tagged, and the number and targets of these references may grow gradually over evolutionary time. The results of previous studies have shown that this leads to problem-solving efforts for the lawnmower and "dirt-sensing, obstacle-avoiding robot" problems that scale well with problem size, indicating that tag-based modularity may be a powerful while nonetheless simple method for evolving modular programs more generally [24, 21]. But because tree-based representations are currently more popular than Push among genetic programming researchers and practitioners, the extent to which such results can be obtained in tree-based genetic programming is clearly of interest.

3. TAGS IN TREES

The original publication on tag-based modularity in genetic programming [24] included the following suggestion for incorporating tags into tree-based genetic programming systems:

> 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 some constant value). 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.

In the work described in the present paper we have essentially implemented and tested this idea, discovering some of its weaknesses in the process.

In the remainder of this section we describe the issues that we faced in implementing these ideas and the ways that we resolved them for the sake of the runs described in the following sections. Other possible approaches are described after the presentation of those results.

3.1 Return values of tagging operations

In Push a tagging operation, implemented with an instruction like `tag.exec.123`, tags an item taken from a stack but it need not return any value to any stacks as a result of this operation. In contrast, in tree-based genetic programming it is generally required that all subtrees return values, so a call to a function of the form `tag.`i will have to return some value to its caller.

As suggested in the quotation above one option is to return some constant value, chosen to be a reasonable default value for the problem being solved (e.g. possibly 0 or `false`). Another reasonable option is to evaluate the code that is being tagged once during the tagging operation and to return the value produced by that evaluation.

In the runs presented below we have tried both of these options. We call the first strategy, of returning a constant (default) value without executing the tagged code, *silent* tagging. In the non-silent condition we tag the code and then evaluate it to produce the value that will be returned to the caller of the tagging function.

3.2 Return values of tag references prior to any tagging

If a tag reference, which in the technique described here will be implemented as a zero-argument function call such as `(tagged.123)`, occurs before *any* value has been tagged, then it is not clear what value should be returned from the tag reference. In Push it is trivial to treat such a tag reference as a no-op, both in the sense that no lookup occurs and in the sense that no values are pushed onto the stacks. In tree-based genetic programming, however, some value must be returned from the evaluation of each sub-tree.

The provisional solution that we have adopted in this case for the runs presented below is to return a constant value, chosen to be a reasonable default value for the problem being solved.

3.3 Unbounded recursion

The use of tags can easily lead to unbounded looping or recursion, for example when code containing a tag reference is itself tagged. Push interpreters are always designed to obey limits on the number of steps that can be executed, and the mechanisms that enforce these limits terminate any unbounded recursion. The obvious approach in tree-based genetic programming—which we have taken for the results presented below—is do do something similar, writing the code tree evaluation function to obey a step limit.

In Push, however, one can still easily interpret the results of a program that is terminated abnormally for hitting the limit; the computations that preceded termination will still be reflected in the values on the data stacks in the interpreter, and those values can be considered to be the results of the evaluation if it is desirable to do so. No such obvious option is available when terminating the evaluation of a tree-based program prior to its natural completion, unless the tree-based program also does its work by side-effect (as in the case of the lawnmower problem below).

One approach to this issue, which we used for some of the runs presented below, is to consider any program that hits the limit, and therefore fails to produce an interpretable result, to be invalid. These invalid programs are given penalty fitness values that make them unlikely to be selected for participation in the production of the subsequent generation.

3.4 Passing arguments to tagged modules

As described above, a tagged module in Push may take any number of arguments easily and without the need to implement any additional mechanisms. This is not possible in tree-based genetic programming, in which syntax restrictions require function calls to include one subtree for each argument that a function takes. Although we describe mechanisms that allow for the evolution of tag-based modules that take arguments later in this paper, the results described be-

low used only 0-argument functions. This would appear to be a significant limitation, since many uses of modules in ordinary (human) programming practice rely on the passing of arguments to those modules.

We note, however, that something akin to argument passing is actually possible in the system that we use here, even though the tagged modules do not explicitly take arguments. This is because one can use tagging itself to implement something that we might term "pseudo-arguments," tagging a value before calling a tag-based module, and then referring to the tagged value from within the module.

For example, suppose that we are performing symbolic regression with three inputs and looking for the solution $x^3 + y^3 + z^3$. A one-argument function that cubes a number would be helpful here. If we are using silent tagging with a default value of 0 then the following program is a solution that employs pseudo-tagging:

```
(+ (tag.10 (* (* tagged.20 tagged.20) tagged.20))
   (+ (tag.20 x)
      (+ tagged.10
         (+ (tag.20 y)
            (+ tagged.10
               (+ (tag.20 z)
                  tagged.10))))))
```

This program essentially uses the code tagged with tag 10 as a function of one argument, and tag 20 as the argument of the function. The expression in the first line tags the function definition and the remainder of the expression sets the argument and then evaluates the function for each of the desired arguments.

Whether or not such pseudo-arguments would arise naturally during evolution is an open question, but it is a logical possibility.

3.5 Returning multiple values

As was also described above, a tagged module in Push may easily return multiple values by modifying multiple data stacks. Because tree-based genetic programming is based on function trees that return single values, we see no obvious way of incorporating this feature into tree-based genetic programming systems.

4. RESULTS

We have conducted a large number of runs of tree-based genetic programming with the simple system for tag-based modularity described above, with mixed results. We present here the results for two problems, the 8×8 lawnmower problem and the Boolean even-4-parity problem, that were shown by Koza to benefit significantly from the kind of modularity provided by ADFs [11]. The lawnmower problem (at various lawn sizes) was also shown to benefit significantly from tag-based modules in PushGP [24].

4.1 Lawnmower

The lawnmower problem involves a simulated lawnmower which must "mow" a grid-based lawn [11]. We use the 8×8 version of the problem here, in which the grid contains 64 lawn squares arranged in an 8×8 grid and in which the mower starts at location $(0, 0)$. The terminal set contains an ephemeral random constant generator $\mathcal{R}_{v8}$ which produces constant vectors of the form (i, j) where i and j range from 0 to 7. The function set contains: `left`, a zero-argument function that rotates the lawnmower 90° counterclockwise; `mow`, a zero-argument function that moves the lawnmower one space forward (mowing the grass in the destination square and wrapping around toroidally if necessary); `v8a`, a two-argument vector addition (modulo 8) function; `frog`, a one-argument function that jumps the lawnmower ahead and to the side by the amount indicated by its vector argument (mowing the destination square); and `progn`, a two function that allows two subexpressions (in its argument positions) to be executed in sequence and returns the value of its second argument. A program is allowed to execute a total of 100 turns (calls to `left`) or 100 movement operators (calls to `mow` or `frog`) before being aborted; it will be aborted when it reaches 100 moves or 100 turns, or completes execution of its code, whichever comes first.

Note that several of the lawnmower functions work by producing side effects on the state of the lawn—for example the `mow` and `frog` functions have side effects of mowing grid squares—and that a program that is terminated early for hitting the execution step limit can nonetheless be evaluated by examining the state of the lawn. For this reason we impose no penalty for hitting the execution step limit on this problem.

Table 1: Parameters for genetic programming runs on the 8×8 lawnmower problem.

runs per condition	100
fitness	squares unmowed (out of 64, lower is better)
limits on moves	100
population size	1000
max generations	51
max program depth	17
tournament size	7
mutation percent	5
crossover percent	90
reproduction percent	5
node selection	90% internal nodes, 10% leaves
tree generation	ramped half-and-half
ramp depth range	2–6
execution step limit	1000
penalty for exceeding limit	0
terminals	$\mathcal{R}_{v8}$
basic functions	`left`, `mow`, `v8a`, `frog`, `progn`
tagging functions	`tag.`n, `tagged.`n
tag range	0–999
no-op functions	`noop0`, `noop1`
default value	`(0, 0)`

Table 2: Results of genetic programming in several conditions related to tagging on the lawnmower problem.

Tags	Silent	No-op	Successes	MBF	Effort
No	—	No	63	0.45	282,000
No	—	Yes	53	0.62	357,000
Yes	No	—	97	0.13	30,000
Yes	Yes	—	65	0.57	144,000

The parameters for our runs on the lawnmower problem are shown in Table 1 and the results are shown in Table 2. The default value, a vector (0,0), is returned by silent tagging and by tagged functions when nothing has yet been tagged. In order to control for changes to the size of the function set, and effects that this may have on the combinatorics of random code generation, we conducted not only a set of runs that lacked tagging functions but also another set of runs that replaced the tagging functions with two "no-op" functions: `noop0`, which takes no arguments and returns the default value, and `noop1`, which takes one argument and returns it unchanged. We conducted runs with tagging both in the silent condition, in which tagging operations return the default value without evaluating the code that they tag, and the non-silent condition, in which the code that will be tagged is first evaluated and its value is returned. For each condition we show the number of runs that were successful (out of 100), the mean best fitness achieved in each run, and the computational effort of finding a solution. Computational effort was computed using Koza's method [10, pp. 99–103], by first calculating $P(M, i)$, the cumulative probability of success by generation i with population size M, and then $I(M, i, z)$, the number of individuals that must be processed to produce a solution by generation i with probability greater than z (here z =99%):

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

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

The results here show that compared to the basic condition (no tags, no no-ops): no-ops make things worse by all measures, (non-silent) tags make things dramatically better by all measures, and silent tags are a mixed bag, with a few more successes and improved computational effort but worse mean best fitness.

4.2 Even-4-Parity

In the even-4-parity problem [10] the goal is to determine whether or not an even number of the four Boolean inputs have the value `true`. The inputs are designated by the terminals `d0`, `d1`, `d2`, and `d3`. The function set includes the four Boolean functions `and`, `or`, `nand`, and `nor`. In the context of tagging operators (which have side-effects with respect to tag-based retrieval) it is important to note that we do not "short-circuit" the evaluation of these Boolean functions; for example, if the first argument of a call to `and` returns `false` we nonetheless evaluate the second argument as well (and return `false`).

The parameters for our runs on the even-4-parity problem are shown in Table 3 and the results are shown in Table 4. As with the runs for the lawnmower problem presented above, we conducted a set of runs that lacked tagging functions but replaced them with two "no-op" functions: `noop0` which takes no arguments and returns the default value and `noop1` which takes one argument and returns it unchanged. We again conducted runs with tagging both in the silent and non-silent conditions, and we present the same measures that were presented for the lawnmower problem above.

Here the results are unequivocally bad for tagging. The best condition, on all measures, is the simplest one with no tagging and no no-ops. No-ops make things worse and tagging makes them much worse indeed, with silent tagging being the worst of all.

Table 3: Parameters for genetic programming runs on the even-4-parity problem.

runs per condition	100
fitness	assignments incorrect (out of 16, lower is better)
population size	1000
max generations	51
max program depth	17
tournament size	7
mutation percent	5
crossover percent	90
reproduction percent	5
node selection	90% internal nodes, 10% leaves
tree generation	ramped half-and-half
ramp depth range	2–6
execution step limit	1000
penalty for exceeding limit	10^{13}
terminals	`d0`, `d1`, `d2`, `d3`
basic functions	`and`, `or`, `nand`, `nor`
tagging functions	`tag.`n, `tagged.`n
tag range	0–999
no-op functions	`noop0`, `noop1`
default value	`false`

Table 4: Results of genetic programming in several conditions related to tagging on the even-4-parity problem.

Tags	Silent	No-op	Succ	MBF	Effort
No	—	No	58	0.56	234,000
No	—	Yes	36	1.02	495,000
Yes	No	—	22	1.44	950,000
Yes	Yes	—	19	1.61	1,044,000

5. DISCUSSION

It appears clear from the results above that tag-based modularity of the simplest form, which was suggested in previous work [24], can indeed help to solve certain problems (such as the lawnmower problem) but it fails badly on other problems (such as even-4-parity). There may be many reasons for these failures, and we do not yet have definitive explanations or modifications to the technique that will allow for better performance. But we can see several factors that probably hinder performance and we can envision several modifications to the technique that may help to ameliorate the problems. In this section we present some of these factors and proposed modifications.

5.1 Passing arguments

It is perhaps not surprising that the technique we tested here, which allows only for zero-argument tagged modules, fails badly on the even-4-parity problem because one would expect that useful functions for this problem would take Boolean inputs. Indeed, in the configuration within which Koza showed this problem to benefit from ADFs, each ADF took three Boolean arguments [11, p. 178]. As we noted above, it is possible for programs to use "pseudo-arguments"

even with the very limited system that we have described here, but a fair amount of code is required to implement pseudo-arguments and solutions using this strategy may be difficult to reach through evolutionary search.

It is therefore interesting to consider alternative mechanisms that would allow for the passing of arguments to tagged modules in tree-based genetic programming. One mechanism that we have developed (but not yet tested sufficiently to present results) uses "substitution-based" arguments as follows: We add some number of argument symbols, such as `arg0`, `arg1`, etc., and additional tag retrieval functions, such as `tagged-1-arg`, `tagged-2-arg`, etc., to the instruction set. The `tagged-1-arg` function takes one argument, and the code that appears in the argument position is substituted for the symbol `arg0` in the retrieved code before it is executed.[2] The `tagged-2-arg` function takes two arguments, and the code provided as arguments is substituted for the symbols `arg0` and `arg1` in the retrieved code. This idea can be extended for any number of arguments; while it requires the user to specify the maximum number of allowed arguments, and hence is not as fully automatic as the evolution of architectures with tags in Push, the numbers of modules of each arity that are actually used will be determined by evolution. Because it is possible that `arg` symbols will sometimes occur in contexts where they are not removed by substitution, they will sometimes be evaluated; we envision them returning default values in this case.

Another alternative would be to use tags themselves to simulate arguments, but to do so in a way that is more parsimonious and evolvable than the "pseudo-argument" strategy described above. For example we might implement an ephemeral random function generator that produces one-argument tag-reference functions that include two tags, one of which will be used to retrieve the code for execution and the other of which will be used to tag the code that appears in the argument position (or its value) before executing the retrieved code. This scheme suffers from some of the same complexities as the pseudo-argument strategy for passing values to tagged modules—for example, similar tag values would have to appear in the function name and in the tagged code—but it avoids the need for `arg` symbols in the method presented above.

The ideas for passing arguments outlined here are probably not ideal, but they may be promising first steps toward a better approach.

5.2 Unbounded recursion

When examining failures of the simple technique presented here on the even-4-parity problem, we noticed that a relatively large fraction of programs in the initial, random generations—often in the neighborhood of 10%—contained unbounded recursion that caused the program to hit the execution step limit and therefore to receive the severe fitness penalty. The number of offending programs would drop off sharply in subsequent generations, but the effect of the penalty would not only be to discourage non-termination but also to discourage tag usage more generally. Since tag usage often produces non-termination, and since non-termination is severely punished, it is difficult to search the space of tag-using programs. Even worse, to the extent that tags are "dangerous" because of non-termination-related penalties, the presence of tags may "pollute" the population more generally, reducing the number of programs that are immune to the penalty and hence slowing the search even for solutions that do not involve tags.

Aspects of this problem are significantly ameliorated in our implementation of the lawnmower problem because we were able to avoid fitness penalties for hitting the execution step limit there. This may help to explain why tags were so much more beneficial for lawnmower than for even-4-parity.

Again, the Push-based systems have an advantage with respect to this issue because Push programs can almost always deliver valid results even when they are terminated for hitting the execution step limit—one can simply declare the results to be whatever is on the stacks at the time of termination—and the imposition of any penalty is optional. A Push user may penalize hitting the limit if she has reason to do so, but she may instead allow programs to hit the limit without penalty if that is more beneficial for evolutionary search.

The primary approach to this problem that we have explored for tree-based tagging involves steps that one can take to prohibit or eliminate unbounded recursion before an execution limit is hit. For example, in one approach we *un-tag* code upon retrieval, so that embedded code retrievals will not be able to retrieve the same overall code block while it is being executed, and then re-tag the code after the retrieved code has completed execution. When one does this there are at least two options for the "un-tagging" operation: In one version of the idea the retrieved code is executed in an environment that simply excludes the tag/value pair from which it was retrieved, which means that embedded references to the same (or similar) tags will retrieve *different* code. In another version of the idea the retrieved code is executed in an environment in which the tag/value pair is temporarily replaced by a pairing of the tag with a default value.

Depending on the implementation of these ideas and on other features of the function set it may not be possible to completely eliminate unbounded recursion. But a variety of methods may serve to dramatically decrease the prevalence of non-termination in programs that use tags, and hence to decrease the "dangerousness" of tags in tree-based genetic programming. Some methods will, however, have costs insofar as they may make it harder to find programs that use recursion in valid and/or terminating ways.[3]

5.3 Program size and depth

In the simple technique presented here the tagging of a subtree increases the size and depth of the overall program tree by one. In many situations the addition of tagging operations will be initially neutral with respect to fitness, because no tag-based retrieval functions are present or because those that are present retrieve code tagged with other tags, and this will allow tags to proliferate. Presumably this

[2] Alternatively, one could first evaluate the code that appears in the argument position and then substitute the resulting value for `arg0`. This may produce different results in some circumstances, with the difference being similar to that between macro and function calls in many programming languages.

[3] Note, however, that valid and/or terminating recursion may be rare or impossible to achieve in the context of the limited function sets like those that we have explored here. In the context of more expressive function sets—for example function sets that include conditionals—this will be more of an issue.

is beneficial in many respects, since once there are many tagged subtrees it is possible for additional genetic changes to introduce useful references to these subtrees. But the proliferation of tags will also add to the size and depth of the trees, and particularly in the case of depth—which is often limited to a number less than 20—it will lead programs to hit the depth limit much earlier. Many genetic programming systems perform poorly when large numbers of programs reach the depth limit, for example because many mutations are discarded for exceeding the limit.

The Push-based approach also has advantages with respect to this issue. Tagging in Push increases the size of a program but not its depth, and the presence of tags is unlikely to be a major factor in the hitting of limits. But with the simple version of the tree-based tagging technique that we have explored here the depth limit may be reached quickly, and indeed we have observed this in several runs that we have examined.

We have considered two approaches to this problem, but neither yet appears to be fully satisfactory. In one approach we count tagging functions differently from other functions when determining whether a tree exceeds the depth limit, and we set independent limits for total depth and for depth of nested tagging. In the other approach we modify the syntax of program trees so that tagging occurs within other function calls and therefore does not increase the overall depth of the tree. There is some complexity involved in the implementation of both of these approaches and the effects that they will have on program size, program depth, and problem-solving performance are not yet clear.

6. CONCLUSIONS AND FUTURE WORK

Tag-based modularity is an appealing concept because it can potentially allow programs with complex modular architectures to evolve without requiring users to specify those architectures in advance and without requiring much in the way of additional complexity in the genetic programming system. Previous work has shown that tag-based modularity can have significant utility in some situations, but the previous demonstrations have all been in the PushGP system which is unconventional in several respects. Here we explored a simple idea for incorporating tag-based modularity into tree-based genetic programming systems, following a suggestion published in the prior work [24].

Our results show that the simplest form of tag-based modularity in tree-based genetic programming does indeed have some utility, but while it is quite beneficial in some circumstances it is detrimental in others. We have identified several possible reasons for these detrimental effects, including the difficulty of passing arguments, problems related to unbounded recursion, and problems stemming from the effects of tagging on program size and depth. We described several approaches to the solution of these problems, although we have not yet tested them sufficiently to draw conclusions.

We believe that the potential promise of the technique justifies further work; we plan to conduct such work and we would encourage others to do so as well. The code that we used for the experiments presented here is freely available for others to use for their own experiments.[4]

Aside from the investigation of the hypotheses and alternative approaches proposed above, several other lines of future work seem to us to be potentially worthwhile. First, it should be informative to conduct experiments like those reported here on a much wider range of problems. This should provide data about where and when the problems do and don't arise, and hence it should also provide clues about which approaches to solving the problems are likely to be most fruitful. It would also be interesting to examine the tag usage that occurs in evolved programs in detail, to see what forms of tag usage are promoted by the simple mechanisms that we have already provided and what forms of tag usage may be helpful but are not yet accessible. In addition, it would be interesting to examine the benefits of tag-based modularity in tree-based programs relative to other modularity mechanisms for similar program representations, such as ADFs [10, 11]. Since ADFs require more pre-specification and provide less architectural flexibility than tag-based modules, this would not be an "apples to apples" comparison, but it would nonetheless be informative to see how the different modularity mechanisms fare on problems of increasing complexity.

Finally, we note that the difficulties that we have encountered in developing an approach to tag-based modularity in tree-based genetic programming do nothing to diminish the utility of the technique in PushGP, or indeed in any other form of genetic programming. The core idea of tagging and of tag-based (closest-match) retrieval of modules should be applicable in many other forms of genetic programming—for example, in machine code genetic programming [13], Cartesian genetic programming [4], and grammatical evolution [14]—and the problems that we have encountered here may not occur, or may not be as severe, in those contexts. It might turn out that the representational constraints of traditional, tree-based genetic programming are a poor match to the requirements for tag-based modularity. Nonetheless, tag-based modularity may be important for the future of genetic programming because of its utility in the context of other program representations.

[4]http://hampshire.edu/lspector/tags-gecco-2012

7. ACKNOWLEDGMENTS

Thanks to the anonymous reviewers who made several excellent suggestions for future work, to Emma Tosch, Omri Bernstein, and Kwaku Yeboah Antwi for discussions related to this work and comments on a draft, to Josiah Erikson for systems support, and to Hampshire College for support for the Hampshire College Institute for Computational Intelligence. Computational support was partially provided by the Brandeis HPC. 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.

8. 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, 25-26 Feb. 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, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[3] Hales, D. Altruism "For Free" using Tags. In *Paris ECCS'05 Conference, Nov. 2005*, 2005.

[4] S. Harding, J. F. Miller, and W. Banzhaf. Developments in cartesian genetic programming: self-modifying CGP. *Genetic Programming and Evolvable Machines*, 11(3/4):397–439, Sept. 2010. Tenth Anniversary Issue: Progress in Genetic Programming and Evolvable Machines.

[5] J. Holland. The effect of labels (tags) on social interactions. Technical Report Working Paper 93-10-064, Santa Fe Institute, Santa Fe, NM, 1993.

[6] J. H. Holland. *Hidden Order: How Adaptation Builds Complexity*. Perseus Books, 1995.

[7] 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, Lausanne, Switzerland, 30 Mar. - 1 Apr. 2005. Springer.

[8] 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.

[9] 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.

[10] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[11] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

[12] J. R. Koza, David Andre, F. H. Bennett III, and M. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufman, Apr. 1999.

[13] P. Nordin. A compiling genetic programming system that directly manipulates the machine code. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, 1994.

[14] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, Aug. 2001.

[15] 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, Paris, 14-15 Apr. 1998. Springer-Verlag.

[16] R. L. Riolo, M. D. Cohen, and R. Axelrod. Evolution of cooperation without reciprocity. *Nature*, 414:441–443, 2001.

[17] 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, Lake Como, Italy, 18-20 Apr. 2001. Springer-Verlag.

[18] 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, Ann Arbor, 13-15 May 2004.

[19] 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.

[20] L. Spector. Autoconstructive evolution: Push, pushGP, and pushpop. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 137–146, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[21] L. Spector, K. Harrington, B. Martin, and T. Helmuth. What's in an evolved name? the evolution of modularity via tag-based reference. In R. L. Riolo, E. Vladislavleva, and J. H. Moore, editors, *Genetic Programming Theory and Practice*, chapter 1, pages 1–16. Springer, 2011.

[22] L. Spector and J. Klein. Genetic stability and territorial structure facilitate the evolution of tag-mediated altruism. *Artificial Life*, 12(4):1–8, 2006.

[23] 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, Washington DC, USA, 25-29 June 2005. ACM Press.

[24] L. Spector, B. Martin, K. Harrington, and T. Helmuth. Tag-based modules in genetic programming. In Natalio Krasnogor, et al., editor, *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1419–1426, Dublin, Ireland, 12-16 July 2011. ACM.

[25] 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.

[26] J. M. Swafford, E. Hemberg, M. O'Neill, M. Nicolau, and A. Brabazon. A non-destructive grammar modification approach to modularity in grammatical evolution. In Natalio Krasnogor, et al., editor, *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1411–1418, Dublin, Ireland, 12-16 July 2011. ACM.