PREPRINT VERSION

To appear in *Genetic Programming Theory and Practice IX*. Springer. 2011.

# WHAT'S IN AN EVOLVED NAME? THE EVOLUTION OF MODULARITY VIA TAG-BASED REFERENCE

Lee Spector[⋆†], Kyle Harrington[‡], Brian Martin[⋆], Thomas Helmuth[†]

[⋆] *Cognitive Science, Hampshire College, Amherst, MA, 01002 USA.*

[†] *Computer Science, University of Massachusetts, Amherst, MA, 01003 USA.*

[‡] *Computer Science, Brandeis University, Waltham, MA, 02453 USA.*

**Abstract**

Programming languages provide a variety of mechanisms to associate names with values, and these mechanisms play a central role in programming practice. For example, they allow multiple references to the same storage location or function in different parts of a complex program. By contrast, the representations used in current genetic programming systems provide few if any naming mechanisms, and it is therefore generally not possible for evolved programs to use names in sophisticated ways. In this chapter we describe a new approach to names in genetic programming that is based on Holland's concept of tags. We demonstrate the use of tag-based names, we describe some of the ways in which they may help to extend the power and reach of genetic programming systems, and we look at the ways that tag-based names are actually used in an evolved program that solves a robot navigation problem.

**Keywords:** genetic programming, modularity, names, tags, stack-based genetic programming, Push programming language, PushGP

## 1. Programming with Names

It would be hard to imagine a modern computer programming language that did not allow programmers to name data objects and functional modules. Indeed, even the simplest assembly languages allow programmers to label defined code and data and then, later, to use those labels to refer to the defined items. The binding of names is also a fundamental operation in the interpretation of lambda expressions, which provide the semantic foundations of programming language theory.

Certainly for any task that lends itself to hierarchical, modular decomposition, a programmer will want to assign names to modules and to use those names to allow modules to refer to one another. Because we can expect most complex real-world problems to fit this description to some extent—as Simon put it, hierarchy is "one of the central structural schemes that the architect of complexity uses" (Simon, 1969)—we can understand why naming is such an important element of programming language design.

Indeed, naming is so important that language designers accommodate it even though it complicates both language and compiler design in a variety of ways. The grammars of most programming languages fail to be fully context free because name definitions and uses must match, and special measures must be taken to ensure program validity and the proper handling of name references in the context of language-specific rules for name scope and extent.

The concept of a "name" is not, however, completely straightforward. A great many subtleties in the use of names have been noted by philosophers of language and logic over millennia, with seminal contributions to the theory of names having been made by Aristotle, John Stuart Mill, Gottlob Frege, Bertrand Russell, and others. Many of these theories consider names to be abbreviated descriptions, a characterization that might apply to binding forms in declarative languages such as Prolog, but which does not seem to apply to names as used in imperative or functional programming languages. Arguably the most significant recent work was done by Saul Kripke, whose *Naming and Necessity* breaks from the tradition of description-based theories of names and presents a causal theory that accounts for proper names as "rigid designators" (Kripke, 1972). While natural languages and programming languages are quite different, aspects of Kripke's theory do seem to apply to names in imperative and functional programming languages as well. The key insight

is that a name is associated with a referent through an act of "dubbing" or "baptism," after which the name will designate the referent.[1]

How does this concept of naming apply to programming practice? In most programming languages the programmer can use definitions to perform the "dubbing" action, after which the defined names can be used to refer to the values provided in the definitions. Names may also occur within named values, of course, and these will in turn refer to values that were associated with those names in previous definitions. The use of a name that has not been defined is generally an error that halts compilation or execution; this has implications for the evolution of name-using programs, as discussed below. Additional complications are introduced when the same name is defined more than once; different programming languages deal with this issue in different ways, expressed through the languages' rules for scope and extent.

What about names in genetic programming? In the simplest traditional genetic programming systems, such as that described in (Koza, 1992), no explicit facilities for naming are provided. In systems with automatically defined functions, such as that described in (Koza, 1994), evolving programs define and use names for functions and their arguments but the numbers and types of names are specified in advance by the human who is using the system; they do not arise from the genetic programming process. In systems with "architecture-altering operations," like that described in (Koza et al., 1999), the number and types of names do indeed emerge from the evolutionary process but only through the use of considerably more complicated program variation operators that must ensure that calls to functions match their definitions and that all names are defined before they are used. Furthermore, the programs produced by these systems can only use names in certain predefined ways; for example, they cannot dynamically redefine names as they run, as many programs written by humans do.

However, even in the simplest traditional genetic programming systems one can include functions in the function set that provide name-like capabilities. Perhaps the most straightforward of these is "indexed memory," in which "write" and "read" functions store and retrieve values by integer indices (Teller, 1994). In a sense, indexed memory provides a naming scheme in which names are integers and in which undefined names refer to a default value (e.g. zero). The standard indexed memory scheme does not permit the naming of code, but one could imagine simple extensions that would allow this.

[1]Kripke's theory is considerably more complicated than this, and is expressed in terms of his "possible world semantics." We will not be concerned with these complications here.

Of course, there are now many other forms of genetic programming, and different opportunities for naming may arise in each of them. Any system that permits the evolution of modules of any kind must have some way of referring to the modules that have been defined, and such systems will therefore have to have some capabilities or conventions for naming (Koza, 1990; Koza, 1992; Angeline and Pollack, 1993; Kinnear, Jr., 1994; Spector, 1996; Bruce, 1997; Racine et al., 1998; Roberts et al., 2001; Li et al., 2005; Jonyer and Himes, 2006; Hornby, 2007; Hemberg et al., 2007; Walker and Miller, 2008; Shirakawa and Nagao, 2009; Wijesinghe and Ciesielski, 2010). Some other systems use forms of reference based on pattern matching (Ray, 1991), which bear some similarities to the tag-based naming scheme that we present below. However, we do not believe that any of these systems have demonstrated the full gains that we should expect from capabilities for the flexible evolution of complex modular architectures.

In this chapter we describe work on a new technique for naming that is based on "tags" as described by Holland (Holland, 1993; Holland, 1995). Tags have been used in a variety of contexts, most notably in work on the evolution of cooperation (e.g. (Riolo et al., 2001; Spector and Klein, 2006; Soule, 2011)), but their use for naming in genetic programming is new. We will argue that this new technique provides powerful new capabilities.

Although tag-based names can conceivably be used in many kinds of genetic programming systems, our work to date has been conducted with PushGP, a genetic programming system that evolves programs expressed in the Push programming language. We will therefore next present the core concepts of Push, including the facilities for naming that have long existed in Push but which have never been shown to be useful for program evolution. We will then introduce tags and describe how they may be used to name modules in various kinds of genetic programming systems, with a focus on their implementation in PushGP. Following the description of the technique we discuss recent results that demonstrate the utility of tag-based modularity on standard problems, and we look in detail at the ways that tags are actually used in evolved programs. We conclude with a number of directions for future work.

## 2. Push and Push Names

The Push programming language was designed for use in evolutionary computation systems, as the language in which evolving programs are expressed (Spector, 2001; Spector and Robinson, 2002a; Spector et al., 2005). It is a stack-based language in which a separate stack is used

for each data type, with instructions taking their arguments from—and leaving their results on—stacks of the appropriate types. This allows instructions and literals to be intermixed regardless of type without jeordardizing execution safety. Instructions act as "no-ops" (that is, they do nothing) if they find insufficient arguments on stacks. 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.[2]

In Push "code" is itself a data type. A variety of powerful capabilities are supported by the fact that Push programs can manipulate their own code on the "exec" stack, which stores the actual execution queue of the program during execution, and on the "code" stack, which stores code but is otherwise treated as any other data stack. Push programs can transform their own code in ways that produce the effects of automatically defined functions (Koza, 1992; Koza, 1994) or automatically defined macros (Spector, 1996) without pre-specification of the number of modules and without more complex mechanisms such as architecture-altering operations (Koza et al., 1999). Push supports the evolution of recursion, iteration, combinators, co-routines, and novel control structures, all through combinations of the built-in code-manipulation instructions. PushGP, the genetic programming system used for the work presented below, was designed to be as simple and generic as possible (e.g. it has no automatically defined functions, strong typing mechanisms, or syntactic constraints) aside from using Push as the language for evolving programs, but it can nonetheless produce programs with all of the features described above because of the expressive power of the Push language itself.

Push supports several forms of program modularity that do *not* involve naming. For example, a Push program can use standard stack-manipulation instructions on the exec stack (e.g. `exec.dup`, which duplicates the top element of the exec stack, and `exec.rot`, which rotates the top three items on the exec stack) to cause sub-programs to be executed multiple times, providing a form of modular code reuse. More exotic forms of nameless modularity can be implemented by manipulating code in arbitrary ways prior to execution, using Push's full suite of code manipulation instructions that were inspired by Lisp's list manipulation functions (including versions of `car`, `cdr`, `list`, `append`, `subst`, etc.). Push also includes versions of the $K$, $S$ and $Y$ *combinators* in combinatory logic (Schönfinkel, 1924; Curry and Feys, 1958), each of which

[2] http://hampshire.edu/lspector/push.html

performs a specific transformation on the exec stack to support the concise expression of arbitrary recursive functions. These mechanisms have been shown to support the evolution of programs with 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 (Spector et al., 2005)) Push's code manipulation mechanisms had been shown to automatically produce modular solutions to parity problems (Spector and Robinson, 2002a) and to induce modules in response to a dynamic fitness environment (Spector and Robinson, 2002b). More recent work, using Push 3, has produced solutions to a wide range of standard problems (including list reversal, factorial regression, Fibonacci regression, parity, exponentiation, and sorting (Spector et al., 2005)) along with the production of human-competitive results in quantum circuit design (Spector, 2004) and pure mathematics (Spector et al., 2008).

But what about named modules? Push includes a data type for names, along with instructions for associating names with values and for retrieving named values, but the use of names has never been evident in evolved programs. One can write Push programs that use names by hand, but such programs have not readily emerged from genetic programming runs. This has been an issue of concern to the Push development team, and consequently the details of the ways in which Push handles names have been revised and refined several times over the history of the project. The current version of the language specification, for Push 3 (Spector et al., 2004), allows names to be defined and used quite parsimoniously.

For example, consider this program fragment from (Spector et al., 2005):

```
(times2 exec.define (2 integer.*))
```

Because `times2` is not a pre-defined Push instruction and does not yet have a defined value the interpreter will push it onto the name stack when it is processed. When `exec.define` is processed it will pop the name stack and associate the popped name (`times2`) with the item that is then on top of the exec stack (which will then be `(2 integer.*)`; the exec stack will also be popped). Subsequent references to `times2`, unless they are quoted, will cause the value `(2 integer.*)` to be pushed onto the exec stack, with the effect that `times2` will act as a named subroutine for doubling the number on top of the integer stack. Of course one could do the same thing with any arbitrarily complex module code.

However, even though the Push 3 naming scheme is both powerful and parsimonious we have rarely seen it used in significant ways in evolved programs. Why? Our most recent thinking on this question has focused

on the difficulty of matching name definitions and references, which depends in part on the number of names in circulation and in part on the details of the genetic programming system's code generation and variation algorithms. Concerning the former, one faces a dilemma because the best way to promote the matching of definitions and references is to provide only a small number of names, but doing so would also limit the complexity of the name-based modular architectures that can possibly evolve, which is antithetical to our research and development goals. A variety of approaches to this dilemma might be envisioned, and we have explored options such as gradually increasing the number of available names over the course of a genetic programming run. But the measures that we have taken have not been particularly effective. One can also envision a variety of modifications to code generation and variation algorithms that would encourage or enforce the proper matching of name definitions and references, for example by adding definition and reference expressions to programs only in matched pairs and by repairing mismatches after crossover. Such approaches deserve study, but we are more interested in approaches that allow name-use strategies to emerge more naturally from the evolutionary process, without putting constraints on the ways in which names can be used and without requiring the re-engineering of the overarching evolutionary algorithm.[3]

This is the context within which we turn to the concept of tags, to provide a mechanism for naming and reference that is more suited to use in genetic programming systems.

## 3. Tags

John Holland, in his work on general principles of complex adaptive systems, has presented an abstraction of a wide variety of matching, binding, and aggregation mechanisms based on the concept of a "tag" (Holland, 1993; Holland, 1995). A tag in this context is an initially meaningless identifier that can come to have meaning through the matches in which it participates. Holland provides several examples of tag-like mechanisms in human societies and in biological systems, including banners or flags used by armies and "the 'active sites' that enable antibodies to attach themselves to antigens" (Holland, 1995, p. 13). In some but not all of the examples of tag usage presented by Holland and others the

[3]It is particularly desirable to avoid constraints on code generation and variation in meta-genetic programming systems and autoconstructive evolution systems, in which code generation and variation algorithms are themselves subject to random variation and selection. But we think it is desirable to do this even in the context of standard genetic programming with hand-designed code generation and variation algorithms.

matching of tags may be inexact, with binding occurring only if the degree of match exceeds a specified threshold, or with the closest matching candidate being selected for binding. This inexact matching will be crucial for the application to genetic programming that we present below.

One of the areas in which the concept of tags has been applied by several researchers is the study of the evolution of cooperation. For example, in the model of Riolo, Cohen, and Axelrod, one agent will donate to a second agent if difference between the tags of the two agents is less than the "tolerance" threshold of the donor (Riolo et al., 2001). In this model, and the related models that have subsequently been developed by others (Spector and Klein, 2006; Soule, 2011), tags and tolerances are allowed to change over evolutionary time. These studies show that the existence of a tagging mechanism can have major effects on the features of the systems that evolve, sometimes permitting the evolution of cooperation in contexts in which it would not otherwise emerge.

How can tags be used to address the issues described in the previous section, concerning the use of names in genetic programming? We suggest that tags be incorporated into genetic programming systems by providing mechanisms that allow programs to tag values (including values that contain executable code), along with mechanisms that allow for the retrieval and possible execution of tagged values. Tags differ from names in this context because we can allow inexact tag matching, and because this can better facilitate the coordination of "name definitions" (now "tagging" operations) and "name references" (now tag-based retrieval operations, also called "tag references"). In particular, we can specify that a tag reference will always recall the value with the closest matching tag. This will ensure that as long as *any* value has been tagged *all* subsequent tag references will retrieve *some* value. As more values are tagged with different tags, the values retrieved by particular tag references may change because there may be different closest matches at different times.[4] But at almost all times—except when no tagging actions have yet been performed—each tag reference will retrieve some tagged value. If our current thinking about the evolutionary weakness of ordinary names is correct then tag-based names, with "closest match" retrieval, should make it easier to evolve programs that make significant use of named (actually tagged) modules.

[4]The set of tagged values will grow dynamically during program execution, as tagging instructions are executed. In addition, new tagging instructions may be added to programs over evolutionary time by mutation and by other genetic operators. Such events may cause a tag reference to refer to a different value than it referred to previously.

Versions of this general tagging scheme might be applied to a wide range of genetic programming systems. One implementation choice that must be made for any application concerns the representation of tags. In biology tags may be complex, structured objects, while in applications to the evolution of cooperation they have often been floating-point numbers with tag matching based on numerical differences.[5] In our work on tag-based modules in genetic programming we have used an even simpler representation in which tags are positive integers and tag matching is based on a ceiling function. We say that the closest match for a reference to tag $t_{ref}$, among all tags $t_{val}$ that have been used to tag values, is the tag $t_{val} \geq t_{ref}$ for which $t_{val} - t_{ref}$ is minimal if at least some $t_{val} \geq t_{ref}$, or the smallest $t_{val}$ if all $t_{val} < t_{ref}$. This means that if there is no exact match then we "count up" from the referenced tag until we find a tag that has been used to tag a value, wrapping back to zero if the referenced tag is greater than all of the tags that have been used.

Another implementation choice concerns the relation between tags, the instructions that tag values, and the instructions that use tags to retrieve values. It would be possible to treat tags as first-class data items, and to write tagging and tag reference instructions to take these data items as arguments. For the sake of parsimony, however, we have opted instead to embed tags within the tagging and tag reference instruction names so that, for example, an instruction such as `tag.float.123` would be used to tag a floating-point value with the tag "123" and an instruction such as `tagged.123` would be used to retrieve the value that has been tagged with the tag closest to "123."

In the context of these implementation decisions we can consider further steps required to provide tagging in various types of genetic programming systems. In the simplest traditional genetic programming systems, which represent programs as Lisp-style symbolic expressions and which do not include any facilities for automatically defined functions, we might support calls to one-argument functions of the form `tag.`$i$, which would act to tag the code in their argument positions with the tags embedded in their names (and presumably return some constant value or the results of evaluating their arguments), and also zero-argument functions of the form `tagged-`$i$, which would act as branches to the code with the closest matching tag (or presumably return some constant value if no code had yet been tagged). The system's code generation routines would have to be modified to produce such function calls, and the program execution architecture would have to be modified to support an execution

[5] In some of the work in this area tags are points in a multi-dimensional space, with matching determined by Euclidean distance (Spector and Klein, 2006).

step limit that prevents unbounded recursion. This particular implementation would only allow the tagging of zero-argument functions, but it might nonetheless have utility.

Systems that already support automatically defined functions and architecture altering operations present different opportunities, and we might begin by simply replacing function names with tags, both in definitions and in calls. This would allow us to drop many of the measures that must ordinarily be taken in such systems to ensure that all calls refer to defined functions, since calls would refer to the function with the closest matching tag (or to some default function if no function has yet been defined). Measures would still have to be taken to ensure that the numbers of arguments in calls match the numbers of parameters of definitions, but initial experiments might be conducted with settings that mandate a single, constant number of parameters for all automatically defined functions.

In the context of Push it is quite simple to define even more general and powerful tagging and tag-reference mechanisms. Each tagging instruction, of the form `tag.<type>.`$i$, pops the stack of the specified type and associates the popped value with the specified tag. As with all other Push instructions, if the needed value is not on the specified stack then the instruction does nothing. Tagging instructions specifying a type of "code" or "exec" can be used to tag bodies of code, which can be used as named (tagged) code modules that take any number of arguments and return any number of values. Instructions of the form `tagged.`$i$ retrieve the value with the closest matching tag and push it onto the exec stack. If the value is a literal of some type other than code then its execution will push it onto the appropriate stack, following the standard Push execution semantics. If the value is code then it will be executed. Additional instructions (not used here) of the form `tagged.code.`$i$ retrieve tagged values to the code stack rather than the exec stack, allowing for further manipulation prior to execution. And additional instructions (also not used here) of the form `untag.`$i$ can be used to remove tags from previously tagged values.

In previous work (Spector et al., 2011) we have demonstrated that tag usage readily emerges in runs of PushGP and that the availability of tags allows PushGP to scale well on problems such as the lawnmower problem, which Koza used to show that systems with automatically defined functions scaled better than those without them (Koza, 1994). In fact, the results showed that PushGP with tags scaled better than PushGP with combinators and exec stack manipulation instructions, which are the PushGP mechanisms that had previously been most effective in supporting the evolution of modularity. It is therefore clear that we have

succeeded in our goal of developing a way to evolve programs using arbitrary named (tagged) values, and it is also clear that this capability does indeed enhance the power of genetic programming.

## 4. What's in an Evolved Name?

Our prior work showed that tags can support the evolution of modularity and that this capability can allow a genetic programming system to scale well on problems that have regularities that can be exploited by modular programs (Spector et al., 2011). But in the prior work we did not study the modules that were actually produced by genetic programming. In the remainder of this chapter we take this additional step.

One of the problems to which we applied PushGP with tags in our prior work is the "dirt-sensing, obstacle-avoiding robot problem" (originally described in (Spector, 1996)). This problem is much like the lawnmower problem, in which one seeks a program that causes a robotic lawnmower to mow the grass in all of the squares in a "lawn" grid, but with the following modifications: the metaphor is changed from lawn mowing to mopping, there are irregularly placed objects through which the robot cannot move, and sensors are provided for dirt and for obstacles. Each program is tested on two grids with differently placed obstacles. The details of the problem specification and system configuration are not relevant to our purposes here, but the interested reader may consult (Spector et al., 2011).[6]

In one PushGP run on this problem, on an $8 \times 12$ grid, we evolved a successful program that had the modular calling structure shown in Figure 1-1 for one of the two obstacle placements on which it was tested, and the modular calling structure shown in Figure 1-2 for the other obstacle placement. Each of these figures was produced by tracing tag references dynamically during a run of an automatically simplified version of the evolved program, with each unique retrieved value being represented as a distinct node in the graph and with Module 0 being added to the graph to represent the entire program (with which execution starts). The thicknesses of the node outlines indicate the size of the code in the corresponding modules. Arrows between nodes indicate that the module at the tail of an arrow initiates a call of the module at the head of the arrow, and the numbers by the arrows indicate the number of times that such calls are made in a single execution of the program.

There are many interesting things to note about these diagrams. First it is interesting to note that they are different, which means that the pro-

[6]See also the source code and erratum note at http://hampshire.edu/lspector/tags-gecco-2011.

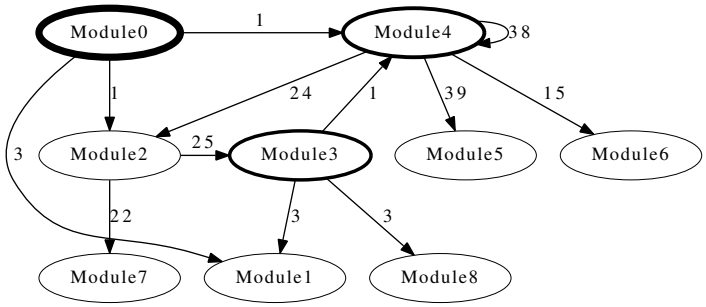


*Figure 1-1.* The modular calling structure of an evolved solution to the dirt-sensing, obstacle-avoiding robot problem when executed on one particular $8 \times 12$ grid.

gram executes different code, with altered modular architectures, when it confronts different environments. But there are also many commonalities between the two diagrams. In particular, each shows that module 4, which is relatively large and complex, is used as a main loop that makes many calls to Module 2 which in turn calls Module 3. Table 1-1 shows the actual code associated with each module, and while we have not presented enough details about the problem or instructions to explain this code completely, we can nonetheless see, for example, that Module 3 is the only one of the large and frequently called modules that performs obstacle sensing; we can therefore infer that it serves as something like a subroutine for obstacle avoidance.

In producing these diagrams we also noticed that this evolved program does not dynamically redefine any tagged modules after their first uses. That is, it is never the case that the same tag retrieves two different values at different times during the execution of the program. This is interesting in part because it is an emergent property that is not mandated by the tagging mechanism; programs that perform dynamic redefinition may be produced in other runs.

One other interesting observation regarding this solution is that it treats module boundaries in unconventional ways. For example, Module 3 ends with `(if-dirty frog)` which is *part* of a conditional expression. If the robot is facing a dirty grid location then `frog` will be executed, but if it is not then the first expression in the code that *follows* the call to Module 3—in this case this will be `tagged.10`, in Module 2—will be executed. Furthermore, if the robot is facing a dirty grid location then not only will `frog` be executed but also the call to `tagged.10`, which is not textually within the same module as the `if-dirty` conditional, will be skipped. This arrangement clearly has utility but it is quite unusual from the perspective of ordinary human programming practice.

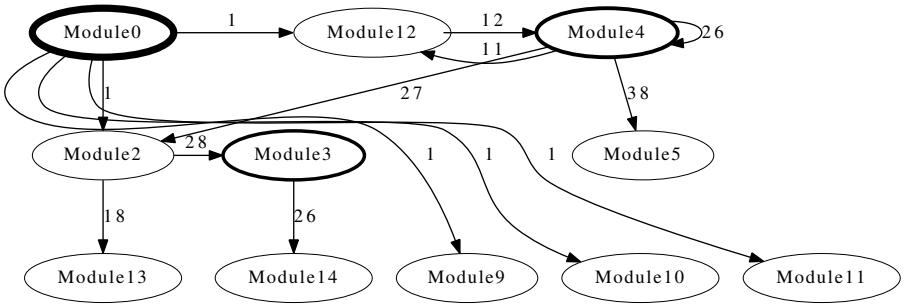


*Figure 1-2.* The modular calling structure of the same evolved solution to the dirt-sensing, obstacle-avoiding robot problem that produced the structure diagrammed in Figure 1-1 but, in this case, executed on a different $8 \times 12$ grid.

*Table 1-1.* Code modules retrieved by tag references in an automatically simplified version of an evolved solution to the $8 \times 12$s dirt-sensing, obstacle-avoiding robot problem. See (Spector et al., 2011) for full descriptions of the instructions used for this problem.

| Module # | Tag # | Value |
|---|---|---|
| 0 | [main] | [omitted to conserve space] |
| 1 | 242 | mop |
| 2 | 608 | (tag.exec.489 frog left tagged.220 tagged.10) |
| 3 | 238 | ((left) (if-obstacle (if-obstacle) (mop if-dirty if-obstacle left mop) (if-obstacle (left tagged.243 tagged.239 left)) tag.exec.770) (tagged.343) (if-dirty frog)) |
| 4 | 360 | ((mop mop (mop mop left mop tag.exec.143 v8a if-dirty if-dirty left tagged.435) (if-dirty) (tagged.662 tag.exec.91) (if-dirty mop)) (if-dirty if-dirty tagged.580) (tagged.336)) |
| 5 | 489 | frog |
| 6 | 695 | if-obstacle |
| 7 | 143 | v8a |
| 8 | 245 | left |
| 9 | 200 | left |
| 10 | 920 | left |
| 11 | 258 | mop |
| 12 | 770 | tagged.343 |
| 13 | 90 | mop |
| 14 | 248 | frog |

## 5. Conclusions and Future Work

We have described a new technique for evolving programs that use name-like designators to refer to code and data modules, and we have situated this new mechanism within the literature of related mechanisms that have been used in genetic programming. In previous work we demonstrated that this new mechanism, which is based on Holland's concept of tags, supports the evolution of modular programs and allows genetic programming to scale well on certain benchmark problems. Here we also exhibited the actual modular architectures produced by the system for the dirt-sensing, obstacle-avoiding robot problem. We believe that we have demonstrated that tags can support the evolution of programs with complex, modular architectures, and that they may therefore play an important role in the future application of genetic programming to complex, real-world problems.

We have outlined ways in which tags could be implemented genetic programming systems that evolve Lisp-like symbolic expressions, but we have only experimented with tag-based modularity in PushGP so far. An obvious area for future work is to implement and test versions of the technique in other kinds of genetic programming systems.

Among the other next steps that we would like to take are the tracing and analysis of tag usage over the course of evolutionary runs. We would expect tag usage to grow incrementally, and we would expect to see evolutionary transitions when new tags arise and "steal" references from pre-existing tags. We would also like to explore variations of the technique. For example, preliminary experiments in PushGP indicate that tag-based modules may arise even more readily if tagging instructions do not pop their arguments from their stacks, so that the insertion of a tagging instruction into a program is more likely to leave the functionality of the program unchanged until a tag reference instruction is later added; we would like to explore this variation more systematically. More ambitiously, would also like to investigate tag reference mechanisms that support more sophisticated notions of scope and extent, along with tag matching schemes in which the conditions for matching are themselves subject to variation and natural selection.[7]

## Acknowledgments

Nathan Whitehouse, Daniel Gerow, Jaime Dávila, and Rebecca S. Neimark contributed to conversations in which some of the ideas used

[7]As suggested by John Holland (personal communication).

in this work were refined. Thanks also Jordan Pollack and the Brandeis DEMO lab for computational support, to the GPTP reviewers and discussion participants, and to Hampshire College for support for the Hampshire College Institute for Computational Intelligence.

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.

## References

Angeline, Peter J. and Pollack, Jordan (1993). Evolutionary module acquisition. In Fogel, D. and Atmar, W., editors, *Proceedings of the Second Annual Conference on Evolutionary Programming*, pages 154–163, La Jolla, CA, USA.

Bruce, Wilker Shane (1997). The lawnmower problem revisited: Stack-based genetic programming and automatically defined functions. In Koza, John R., Deb, Kalyanmoy, Dorigo, Marco, Fogel, David B., Garzon, Max, Iba, Hitoshi, and Riolo, Rick L., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 52–57, Stanford University, CA, USA. Morgan Kaufmann.

Curry, H.B. and Feys, R. (1958). *Combinatory Logic*, 1.

Hemberg, Erik, Gilligan, Conor, O'Neill, Michael, and Brabazon, Anthony (2007). A grammatical genetic programming approach to modularity in genetic algorithms. In Ebner, Marc, O'Neill, Michael, Ekárt, Anikó, Vanneschi, Leonardo, and Esparcia-Alcázar, Anna Isabel, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 1–11, Valencia, Spain. Springer.

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

Holland, J. H. (1995). *Hidden Order: How Adaptation Builds Complexity.* Perseus Books.

Hornby, G.S. (2007). Modularity, reuse, and hierarchy: measuring complexity by measuring structure and organization. *Complexity*, 13(2):50–61.

Jonyer, Istvan and Himes, Akiko (2006). Improving modularity in genetic programming using graph-based data mining. In Sutcliffe, Geoff C. J. and Goebel, Randy G., editors, *Proceedings of the Nineteenth International Florida Artificial Intelligence Research Society Conference*,

pages 556–561, Melbourne Beach, Florida, USA. American Association for Artificial Intelligence.

Kinnear, Jr., Kenneth E. (1994). Alternatives in automatic function definition: A comparison of performance. In Kinnear, Jr., Kenneth E., editor, *Advances in Genetic Programming*, chapter 6, pages 119–141. MIT Press.

Koza, J. (1990). 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.

Koza, John R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

Koza, John R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts.

Koza, John R., Andre, David, Bennett III, Forrest H, and Keane, Martin (1999). *Genetic Programming 3: Darwinian Invention and Problem Solving*. Morgan Kaufman.

Kripke, S. A. (1972). *Naming and Necessity*. Harvard University Press.

Li, Xin, Zhou, Chi, Xiao, Weimin, and Nelson, Peter C. (2005). Direct evolution of hierarchical solutions with self-emergent substructures. In *The Fourth International Conference on Machine Learning and Applications (ICMLA'05)*, pages 337–342, Los Angeles, California. IEEE press.

Racine, Alain, Schoenauer, Marc, and Dague, Philippe (1998). A dynamic lattice to evolve hierarchically shared subroutines: DL'GP. In Banzhaf, Wolfgang, Poli, Riccardo, Schoenauer, Marc, and Fogarty, Terence C., editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 220–232, Paris. Springer-Verlag.

Ray, Thomas S. (1991). Is it alive or is it GA? In Belew, Richard K. and Booker, Lashon B., editors, *Proceedings of the Fourth International Conference on Genetic Algorithms (ICGA'91)*, pages 527–534, San Mateo, California. Morgan Kaufmann Publishers.

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

Roberts, Simon C., Howard, Daniel, and Koza, John R. (2001). Evolving modules in genetic programming by subtree encapsulation. In Miller, Julian F., Tomassini, Marco, Lanzi, Pier Luca, Ryan, Conor, Tettamanzi, Andrea G. B., and Langdon, William B., editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 160–175, Lake Como, Italy. Springer-Verlag.

Schönfinkel, M. (1924). Über die bausteine der mathematischen logik. *Mathematische Annalen*, 92:307–316.

Shirakawa, Shinichi and Nagao, Tomoharu (2009). Graph structured program evolution with automatically defined nodes. In Raidl, Guenther, Rothlauf, Franz, Squillero, Giovanni, Drechsler, Rolf, Stuetzle, Thomas, Birattari, Mauro, Congdon, Clare Bates, Middendorf, Martin, Blum, Christian, Cotta, Carlos, Bosman, Peter, Grahl, Joern, Knowles, Joshua, Corne, David, Beyer, Hans-Georg, Stanley, Ken, Miller, Julian F., van Hemert, Jano, Lenaerts, Tom, Ebner, Marc, Bacardit, Jaume, O'Neill, Michael, Di Penta, Massimiliano, Doerr, Benjamin, Jansen, Thomas, Poli, Riccardo, and Alba, Enrique, editors, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1107–1114, Montreal. ACM.

Simon, H.A. (1969). The architecture of complexity. In Simon, Herbert A., editor, *The Sciences of the Artificial*, pages 84–118. The MIT Press.

Soule, T. (2011). Evolutionary dynamics of tag mediated cooperation with multilevel selection. *Evolutionary Computation*, 19(1):25–43.

Spector, Lee (1996). Simultaneous evolution of programs and their control structures. In Angeline, Peter J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 7, pages 137–154. MIT Press, Cambridge, MA, USA.

Spector, Lee (2001). Autoconstructive evolution: Push, pushGP, and pushpop. In Spector, Lee, Goodman, Erik D., Wu, Annie, Langdon, W. B., Voigt, Hans-Michael, Gen, Mitsuo, Sen, Sandip, Dorigo, Marco, Pezeshk, Shahram, Garzon, Max H., and Burke, Edmund, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 137–146, San Francisco, California, USA. Morgan Kaufmann.

Spector, Lee (2004). *Automatic Quantum Computer Programming: A Genetic Programming Approach*, volume 7 of *Genetic Programming*. Kluwer Academic Publishers, Boston/Dordrecht/New York/London.

Spector, Lee, Clark, David M., Lindsay, Ian, Barr, Bradford, and Klein, Jon (2008). Genetic programming for finite algebras. In Keijzer, Maarten, Antoniol, Giuliano, Congdon, Clare Bates, Deb, Kalyanmoy, Doerr, Benjamin, Hansen, Nikolaus, Holmes, John H., Hornby, Gregory S., Howard, Daniel, Kennedy, James, Kumar, Sanjeev, Lobo, Fernando G., Miller, Julian Francis, Moore, Jason, Neumann, Frank, Pelikan, Martin, Pollack, Jordan, Sastry, Kumara, Stanley, Kenneth, Stoica, Adrian, Talbi, El-Ghazali, and Wegener, Ingo, editors, *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1291–1298, Atlanta, GA, USA. ACM.

Spector, Lee and Klein, Jon (2006). Multidimensional tags, cooperative populations, and genetic programming. In Riolo, Rick L., Soule, Terence, and Worzel, Bill, editors, *Genetic Programming Theory and Practice IV*, volume 5 of *Genetic and Evolutionary Computation*, chapter 15, pages –. Springer, Ann Arbor.

Spector, Lee, Klein, Jon, and Keijzer, Maarten (2005). The push3 execution stack and the evolution of control. In Beyer, Hans-Georg, O'Reilly, Una-May, Arnold, Dirk V., Banzhaf, Wolfgang, Blum, Christian, Bonabeau, Eric W., Cantu-Paz, Erick, Dasgupta, Dipankar, Deb, Kalyanmoy, Foster, James A., de Jong, Edwin D., Lipson, Hod, Llora, Xavier, Mancoridis, Spiros, Pelikan, Martin, Raidl, Guenther R., Soule, Terence, Tyrrell, Andy M., Watson, Jean-Paul, and Zitzler, Eckart, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1689–1696, Washington DC, USA. ACM Press.

Spector, Lee, Martin, Brian, Harrington, Kyle, and Helmuth, Thomas (2011). Tag-based modules in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2011)*. To appear.

Spector, Lee, Perry, Chris, Klein, Jon, and Keijzer, Maarten (2004). Push 3.0 programming language description. Technical Report HC-CSTR-2004-02, School of Cognitive Science, Hampshire College, USA.

Spector, Lee and Robinson, Alan (2002a). Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40.

Spector, Lee and Robinson, Alan (2002b). Multi-type, self-adaptive genetic programming as an agent creation tool. In Barry, Alwyn M., editor, *GECCO 2002: Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference*, pages 73–80, New York. AAAI.

Teller, Astro (1994). The evolution of mental models. In Kinnear, Kenneth E., editor, *Advances in Genetic Programming*, Complex Adaptive Systems, pages 199–220, Cambridge. MIT Press.

Walker, James Alfred and Miller, Julian Francis (2008). The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 12(4):397–417.

Wijesinghe, Gayan and Ciesielski, Vic (2010). Evolving programs with parameters and loops. In *IEEE Congress on Evolutionary Computation (CEC 2010)*, Barcelona, Spain. IEEE Press.