Language Representation Progression in Genetic Programming

Astro Teller
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
astro@cs.cmu.edu

Abstract

The signal-to-symbol problem is the task of converting raw sensor data into a set of symbols that Artificial Intelligence systems can reason about. We have developed a method for directly learning and combining algorithms that map signals into symbols. This new method is based on Genetic Programming (GP). Previous papers have focused on PADO, our learning architecture. We showed how PADO applies to the general signal-to-symbol task and in particular the positive results it brings to natural image object recognition. Originally, PADO's programs were written in a Lisp-like language formulated in (Teller 1994b). PADO's programs are now written in a very different language. Using this new language, PADO's performance has increased substantially on several domains including two vision domains this paper will mention. This paper will discuss these two language representations, the results they produced, and some analysis of the performance improvement. The higher level goals of this paper are to give some justification for PADO's specific language progression, some explanation for the improved performance this progression generated, and to offer PADO's new language representation as an advancement in GP.

Introduction

PADO (Parallel Algorithm Discovery and Orchestration) is a system that has been designed and built to learn arbitrary signal understanding tasks for any signal size or type. The heart of PADO's architecture is an extension of Genetic Programming (GP) to the space of algorithms. This extension of GP to the space of algorithms required initially a small change to the traditional s-expression language. A natural question to ask is "Now that we're evolving algorithms instead of functions, is a more radical change to the language called for?"

The Architecture Section starts by giving a very short overview of PADO's architecture. This is given to situate the reader, not to give a full explanation of how PADO operators. The next two Sections describe two different PADO languages for evolution. The first is PADO's original language, inspired by the s-expressions of traditional GP (Koza 1992). The second is PADO's new language, specifically designed to be more appropriate for evolving algorithms. The Related Work Section gives some relation to other GP work that has informed PADO's evolution. Results from two previous papers will be compared in the Results Section to give empirical basis for the claim that PADO's new language representation is an improvement. Finally the Discussion Section will give some motivation behind this new language design and some intuition about why this new language has such a marked positive effect on PADO's performance.

The PADO Architecture

The goal of the PADO architecture is to learn to take signals as input and output correct labels. When there are $\mathcal{C}$ classes to choose from, PADO starts by learning $\mathcal{C}$ different *systems*. System$_{\mathcal{I}}$ is responsible for taking a signal as input and returning a confidence that class $\mathcal{I}$ is the correct label. Clearly, if all $\mathcal{C}$ systems worked perfectly, labeling each signal correctly would be as simple as picking the unique non-zero confidence value. If, for example, system $\mathcal{J}$ returned a non-zero confidence value, then the correct label would be $\mathcal{J}$. In the real world, none of the $\mathcal{C}$ systems will work perfectly. This leads us to the recurring two questions of the PADO architecture: "How does PADO learn good components (systems or programs)?" and "How does PADO orchestrate them for maximum effect?" First, let's touch on how one of these systems is built.

System$_{\mathcal{I}}$ is built out of several programs. Each of these programs does exactly what the system as a whole does: it takes a signal as input and returns a confidence value that label $\mathcal{I}$ is the correct label. The reason for this seeming redundancy has been justified and discussed in (Teller & Veloso 1995a).

System$_{\mathcal{I}}$ is built from the $\mathcal{S}$ programs that best (based on the training results from that generation) learned to recognize objects from class $\mathcal{I}$. The $\mathcal{S}$ responses that the $\mathcal{S}$ programs return on seeing a partic-

ular image are all weighted, and their weighted average of responses is interpreted as the confidence that $\text{System}_{\mathcal{I}}$ has that the image in question contains an object from class $\mathcal{I}$. How the responses are obtained in each language will be explained later. PADO does object recognition by orchestrating the responses of the $\mathcal{C}$ systems. On a particular test case, the function F (e.g., MAX) takes the weighted confidences from each $\text{System}_{\mathcal{I}}$ and selects one class as the image object class. See (Teller & Veloso 1995a) for how these weights are obtained. Figure 1 pictures this orchestration learning process.

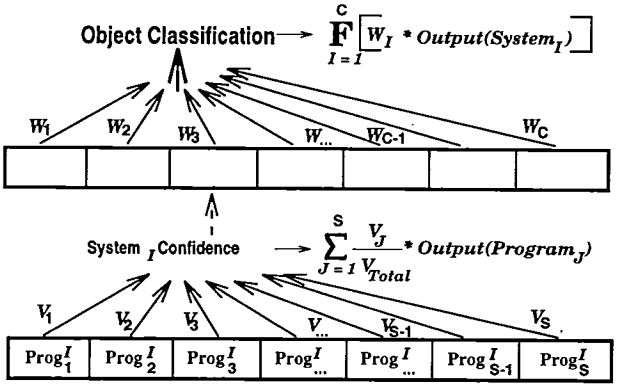


Figure 1: Weights **W** and **V** trained during orchestration.

Programs learned by PADO are written in an algorithmic language that is PADO-specific. During the training phase of learning, these programs are interpreted, not compiled. So like Lisp, the programs can be compiled or interpreted, but during the "construction" phase they are simply interpreted. At the beginning of a learning session, the main population is filled with $\mathcal{P}$ programs that have been randomly generated using a grammar for the legal syntax of the language. All programs in this language are constrained by the syntax to "return" a number that is interpreted as a confidence value between some minimum confidence and some maximum confidence.

At the beginning of a new generation, each program in the population is presented with $\mathcal{T}$ training signals and the $\mathcal{T}$ confidences it returns are recorded. Then the population is divided into $\mathcal{C}$ distinct groups of size $\mathcal{P}/\mathcal{C}$. The programs in group $\mathcal{I}$ are (within the size constraint) the programs that recognized class $\mathcal{I}$ better than any other class in the sense that they maximized a reward function **Reward** (see figure 2) when $K = \mathcal{I}$ (K is the class to which PADO is considering assigning program U).

On images that the program should return MaxConf for, the reward is multiplied by $\mathcal{C} - 1$ so that, even though this only happens once in $\mathcal{C}$ times, these images will account for half the reward. This seems reasonable since it should be as important to say **YES** when appropriate as to say **NO** when appropriate since these two cases are respectively coverage and accuracy.

```
int Reward(program U, class K, int Guess[ ])
    R = 0;
    Loop j = 1 to MaxResponses
    If (K = ObjectClass[j]) Then
        R = R + ((C − 1) * Guess[U][j]);
    Else
        R = R − Guess[U][j];
return R;
```

Guess[U][j] is the confidence program U returned for image j.
R is the reward.
C is the number of classes.
ObjectClass[j] is the object type that appears in image j.

Figure 2: PADO's Class-Relative Reward Function.

Each group is then sorted by increasing fitness and each program is ranked accordingly. C "mating pools" (temporary groups) are created by putting a copy of $Program_{\mathcal{J}}$ from Group_I into MatingPool_I with probability $2 * rank(\mathcal{J})/(P/C)$ (rank selection).

The **Libraries** are programs reference-able from all programs. After the division of the population, the libraries are updated according to how widely and how often they were used. These statistics are weighted by the fitnesses of the programs that called them (Teller & Veloso 1995a; 1995b). See (Angeline & Pollack 1993) for a similar concept.

Finally, 85 percent of the programs within each mating pool are subjected to crossover and another 5 percent are subjected to mutation. All crossovers take place between two programs in the **same** mating pool. That means they are both recognizers of the same class. Crossover in PADO is more complicated than its standard form in genetic algorithms or genetic programming.

In PADO two programs are chosen and given to a "SMART crossover" algorithm (Teller 1996). This algorithm examines the two programs and chooses two sub-parts in each. Then one subpart from each programs is exchanged. The new pairs of sub-parts are reconnected, creating two new programs. These two new programs replace the two old programs in the population. The "SMART Mutation" in PADO is also more complicated than the general case described in the previous section. One program is chosen and an "intelligently" chosen subpart is replaced with an "intelligently" generated new element. This changed program then replaces the old program in the new population. Both the SMART operators are co-evolved with the main population; their "smartness" evolves (Teller 1996). It is not preprogrammed.

At this point we merge the C mating pools back into a new total population and the next generation begins. To extract programs to use in the systems, we can pause the process after the evaluation step of a generation and copy out those programs that scored best or near best in each group $\mathcal{I}$. So this architecture is an *anytime* learning system: at any time we can

generate a system for signal classification using what we have learned so far.

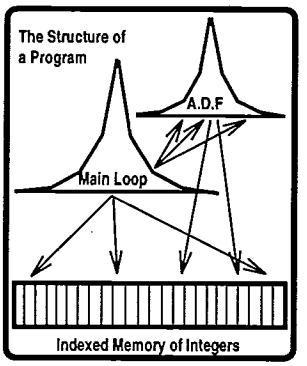
The Old PADO Program Language

Each PADO program was made up of three important parts: a main loop, an ADF, and an Indexed Memory. Both the main loop and the ADF (Automatically Defined Function) were written in a PADO-specific Lisp-like language. The main loop was repeatedly applied for a fixed time limit; there was no fixed number of iterations. A weighted average of the responses the program returned on each iteration (evaluation) was computed and interpreted as the answer. The weight of a response at time t_i was i. Later responses counted more towards the total response of the program. PADO's programs were guaranteed to *halt* and respond in a fixed amount of time.

Repeat

(READ (ADF (NOT (Library77 (Library4 (LEAST (WRITE 169 (Library16 189 125 147 27)) 91 192 (IF-THEN-ELSE (NOT 36) 66 67)) (SUB (EQUAL 228 56) 109) 181 59) (VARIANCE 139 (Library7 (NOT 152) (READ (READ 18)) 255 (ADF ...

Until *Time threshold*



The indexed memory is an array of integers indexed by the integers. Each program has the ability to access any element of its memory, either to read from it or to write to it (Teller 1994a). This memory scheme, in conjunction with the main loop described above has been shown to be Turing complete (Teller 1994b). Indexed memory can been seen as the simplest memory structure that can practically support all other memory structures. Indeed, indexed memory has been successfully used to build up complex data structures and mental models of local geography (Langdon 1995; Teller 1994a).

The ADF is a function definition that evolves along with the main loop (Koza 1994a). This ADF may be called as often as desired in the main loop but may not call itself. While each program has a private main loop, a private ADF, and a private indexed memory, there are a number of Library functions that may be called by the entire population of programs (Teller & Veloso 1995a; 1995b).

As a Lisp-like language, PADO programs were composed of nested functions. These functions acted on the results of the functions nested inside them and any **terminals** that made up their parameters. Terminals are the zero arity functions like constants and variables. In the main loop, the terminals are the integer values 0 thru 255. In the ADF and library functions, the terminals are the integer values 0 thru 255 plus the parameters X, Y, U, and V (see the Language Primitives Section).

The New PADO Program Language

Figure 3 sketches the structure of a PADO program. Each program is constructed as an arbitrary directed graph of nodes. As an arbitrary directed graph of N nodes, each node can have as many as N outgoing *arcs*. These arcs indicate possible flows of control in the program. In a PADO program each node has two main parts: an *action* and a *branch-decision*. Each program has an *implicit* stack and an indexed memory. All *actions* pop their inputs from this implicit stack and push their result back onto the implicit stack. These actions are the equivalent of GP's terminals and non-terminals. For example, the *action* "6" simply pushes 6 onto the parameter stack. The *action* "Write" pops *arg1* and *arg2* off the stack and writes *arg1* into Memory[*arg2*] after pushing Memory[*arg2*] onto the stack. Evaluating a GP tree is effectively a post-order traversal of the tree. Because there are many arcs coming into a particular node in the PADO language we evaluate a part of the graph (indeed, the whole graph) as a *chronological*, not structural, post-order traversal of the graph.

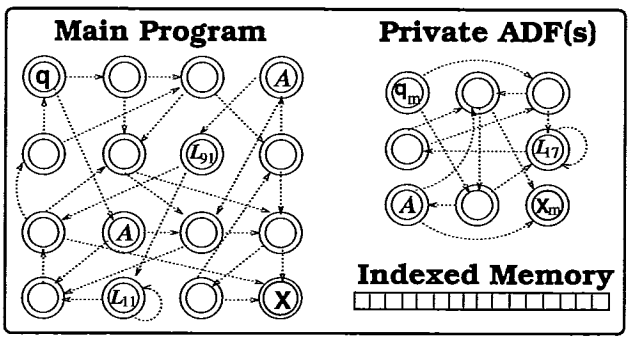


Figure 3: The general structure of a new PADO program.

After the action at node i is executed, an arc is taken to a new node. The branch-decision function at the current node makes this decision. Each node has its own branch-decision function that may use the stack top, the temporally previous node action type, the memory, and constants to pick an arc.

> If this new representation were allowed only one arc per node it would be a postfix notation of a function (with possible loops). Through this we can see that the new PADO language representation is a superset of the standard GP representation.

There are several special nodes shown in Figure 3. Node q is the start node. It is special in no other way than it is always the first node to be executed when a program begins. Node X is the stop node. When this node is reached, its action is executed and then

the program halts. When a program halts or is halted at the time-threshold, its response is considered to be the current value residing in some particular memory location (e.g., response = Memory[0]). If a program halts sooner than a pre-set time threshold, it is started again at its start node (without erasing its memory or stack) to give it a chance to revise its confidence value. A weighted average of the responses that the program gives on a particular execution is computed and interpreted as the answer. However, when no self-halting criteria is enforced by generation 100 only about 1% of the MAIN programs transition to their stop node even once during a particular execution.

Node A executes the private *ADF* program (starting at q_m) as its action. It then executes its branch-decision function as normal. The *ADF* programs associated with each *Main* program bear similarity to the concept of ADF's (automatically defined functions) (Koza 1994a). However, PADO ADFs do not take a specific number of arguments but evolve to use what it they need from the incoming argument stack. In addition, they have internal loops and recursion. The private ADF programs may be called at any point in the main program and they evolves along with the main program. ADF programs are in every way normal PADO programs; their size is not constrained to be smaller than the main programs. The *ADF* programs may recursively call themselves or the globally available *Library* programs, just like a main program may.

The *Library* programs (e.g., L_{91} in Figure 3) are globally available programs (public ADFs) that can be executed at any time and from anywhere just like the ADF programs. But unlike the ADF programs, where each ADF may be run only during the execution of the PADO program of which it is a part, the *Library* programs are publicly available to the entire population. The method by which these *Library* programs change can be seen in some detail in (Teller & Veloso 1995a; 1995b). While the creation and destruction of these *Library* programs is different from the module concept, the maintenance of such a pool of public encapsulations of code is not (Angeline & Pollack 1993).

The Language Primitives

Here is a brief summary of the language primitives and their effects. These primitives are the ones in the experiments mentioned latter, but the general language progression this paper presents is independent of any particular set of language primitives. For the old language, the evaluation result is simply "passed up" to be used by the primitive that called it. In the new language, the result is placed on the stack. This is a more flexible implementation of what is implicit in the old language.

Algebraic Primitives: {+ - * / NOT MAX MIN}
These functions allow basic manipulation of the integers. All values are constrained to the range 0 to 255. For example, DIV(X,0) results in 255 and NOT(X) maps {1..255} to 0 and {0} to 1.

Memory Primitives: {READ WRITE}
These two functions access the memory of the program via the standard indexed memory scheme (Teller 1994a). The memory is cleared (all positions set to zero) at the beginning of a program execution.

Branching Primitives: {IF-THEN-ELSE PIFTE}
In both cases the primitive takes 3 parameters (X,Y, and Z) and then returns either Y or Z (not both) depending on the value of X. For IF-THEN-ELSE the test is (X greater than 0). PIFTE is a probabilistic IF-THEN-ELSE that takes 3 arguments. A random number is chosen and if it is less than X then Y is returned, else Z is returned.

Signal Primitives: {PIXEL LEAST MOST AVERAGE VARIANCE DIFFERENCE}
These are the language functions that can access the signals. In order to demonstrate PADO's power and flexibility, the same primitives have been used for both image and sound data (Teller & Veloso 1995c)! PIXEL returns the intensity value at the point in the image specified by its two parameters. The other five "signal functions" each take 4 parameters. These four numbers are interpreted as (X1,Y1) and (X2,Y2) specifying a rectangle in the image. If the points specified a negative area then the opposite interpretation was taken and the function was applied to the positive area rectangle. LEAST, MOST, AVERAGE, VARIANCE, and DIFFERENCE return the respective functions applied to that region in the image. DIFFERENCE is the difference between the average values along the first and second half of the line (X1,Y1,X2,Y2).

Old Routine Primitives: {ADF LIBRARY[i]}

- **(ADF X Y U V) :** This function was private to the individual that used it and could be called as many times as desired from the main loop (Koza 1994a). Each individual had exactly one ADF which evolved along with the main loop. The ADF differed from the main loop in two ways. It was not allowed to call ADF or Library functions, and it had 4 extra legal terminals: X, Y, U, and V. These extra terminals were local variables that took on the values of the four sub-expressions that were used in each particular call to ADF from the main loop.
- **(LIBRARY[i] X Y U V) :** There were 150 Library functions. The i is not really a parameter. Instead a call to a Library function from some program's main loop might have looked like (Library57 56 (ADD 1 99) 0 (WRITE 3 19)). All 150 Library functions were available to all programs in the population. How these Library functions were created and changed has been discussed in (Teller & Veloso 1995a; 1995b).

New Routine Primitives: {ADF LIBRARY[i]}
These are programs that can be called from the Main program. In addition, they may be called from each other. Because they are programs, and not simple functions, the effect they will have on the stack and

memory before completion (if they ever stop) is unknown.

- **ADF :** This ADF program is private to the MAIN program that uses it and can be called as many times as desired from this MAIN program. Because each ADF is an arbitrarily complex program, it has an arbitrary number of "parameters" which it can pull of the stack. In general, a MAIN program could have many ADF programs for its private use. The only distinctions between MAIN and ADF programs, other than the mechanism of referring to (i.e., calling) an ADF, is that ADF programs may become Library programs. MAIN programs currently may not.
- **LIBRARY[i] :** There are 150 library programs. The **i** is not really a parameter. Instead an *Action* calling a Library program from some program's MAIN, ADF, or from another Library program might look like **Library57**. Like the ADF programs, the Library programs take an unknown number of parameters by "popping" the parameters they want off the parameter "stack". All 150 Library programs are available to all programs in the population. How these library programs are created and changed has been discussed in (Teller & Veloso 1995a).

Genetic Programming Related Work

There has been previous work that involved graphs in evolutionary computation. This work has ranged from graph searches (e.g., (Tamaki 1994)) to building networks (e.g., (Sushil 1994)). Clearly, tree graphs and DAGs have been thoroughly investigated in GP. At least graphically, a PADO program in the new representation is reminiscent of Turing machines and Finite State Automata. The indebtitness of ours to the idea of graphs as representation for programs is wider and deeper than we can cite here. So far, however, the programming of completely general graph-structured programs by evolution has been largely unexplored.

One work worth mentioning as an example is Karl Simms' work on evolving arbitrary graph structures that represent the morphologies of virtual animals. In (Simms 1994), for example, the question of how to "crossover" two graphs is addressed and dealt with in a simple and static way. In (Teller 1996) we tackle this issue in search of more intelligent solutions.

While this particular conception of an evolvable program is new, the idea of using a stack to keep track of information in an evolving system is not (Keith & Martin 1994),(Perkis 1994). We mention this because the stack mentioned in the New Language Section is not in itself a departure from GP, but, like stack-based GP, is simply an easier way of telling the same story.

Issues related to halting and the effective use of resources in a limit time have been investigated (Siegel & Chaffee 1995). While PADO's Libraries and Angeline's Modules have several important differences, (Angeline & Pollack 1993) was an important influence on our work. Some GP experiments have evolved "programs" (e.g., within a context like sorting (Kinnear 1993)).

This paper has mentioned, though not detailed, PADO's SMART operators. The idea that evolvability is something our systems must evolve has been successfully argued as an important field of future research (Altenberg 1994). There have been some different implementations that attempt to respond to this challenge in a variety of ways. Tackett proposes brood selection as an alternative method for helping the evolutionary process to hill-climb more effectively (Tackett 1995). Koza's work on operators has also been influential in this area (Koza 1994b).

Results

So far this paper has explained and contrasted two consecutive language representations used by PADO. The summary of a few experiments will help bring our PADO language progression effort into focus. The experimental results below come from a pair of image classification experiments. This paper makes no pretense to explain or justify these experiments, but simply mentions them. For a detailed account of the experiments, see (Teller & Veloso 1995a; 1995b).

There are two sets of 256x256 8-bit greyscale images, Sets A and T. Each of these two sets contains images from seven different classes. This means that by guessing randomly, a classification rate of 14.28% could be achieved (as shown by the dotted line in Figure 4). Both image sets have been shown to be nontrivial classification domains (Teller & Veloso 1995a; 1995b).

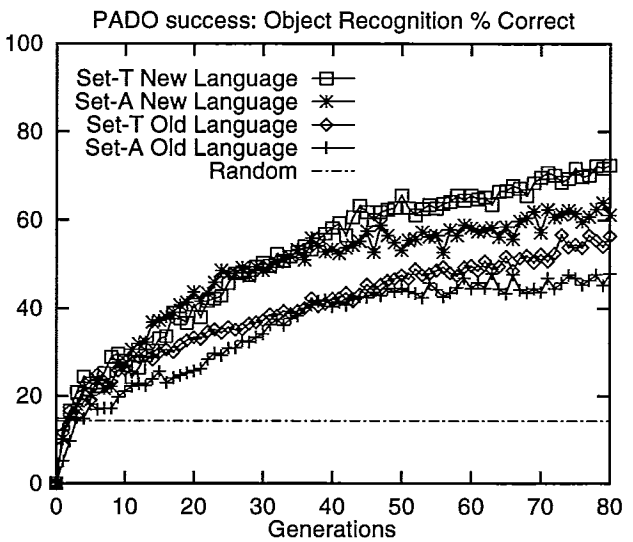


Figure 4: Object recognition rates on Sets A and T.

The curves in Figure 4 clearly show that PADO, using either language representation, can achieve classification rates far above random for both image sets. As with the details of these experiments, the justifi-

cation of these results as significant is tangent to this paper and is expanded upon in (Teller & Veloso 1995a; 1995b). The feature of these curves that make it relevant for this paper is the relative heights for the same image set and the alternate language representations. For both image sets, the new language representation yields a significant improvement in performance. The first purpose of this paper is to describe the alternate language representations (Sections 4 and 5). The second purpose of this paper is to discuss *why* the new language representations made such a difference.

Discussion

Because the rest of the PADO architecture and the *actions* were not changed, there must be something important about the language representation change. Is it possible, the reader may wonder, that on other domains, the new language is less of an improvement? The answer is that similar comparisons have been done on other domains and the difference was as noticeable. Another reasonable question might be "Is it the case that the new language does better on average (as shown in figure 4) but on some runs it does worse than the old language?" Again, the answer is no. The new language representation is very consistently better than the old representation. Is it possible that this new language representation is "just better" for expressing programs than the s-expression inspired representation? The answer may or may not be yes, but more explanation is certainly in order.

ADFs often help considerably in GP problem domains. This full graph representation allows an arbitrary number of "sub-parts" (densely connected subparts of the full program-graph) to be constructed and "used for different reasons" (entered from different arcs) and with different "parameters" (since the stack holds the "parameters" and is constantly changing). This could certainly be an advantage over the old representation which was decomposable and reusable only at the ADF level. Because these sub-parts can be reused through the internal loops in the arcs, an N node graph in the new representation can express a program that might take $2N$, N^2, or even more nodes to express in the old representation.

Graph-structured ADFs are full programs, where as the ADFs in the old representation were only functions. Because the PADO architecture turns the ADFs of highly fit programs into Library elements, the Library elements are now full programs (because the ADFs are now full programs) and may refer to other Library programs. This means that PADO's new language supports and evolves recursion and mutual recursion in the ADFs and Library elements. In the old representation, the ADFs were not allowed to refer to themselves or to Library elements. This difference is also a candidate reason for the noticeable performance change. ADFs take an arbitrary and dynamic number of parameters. Because this was not the case for the old representation's ADF's, this also stands as a potentially important change.

On a more general level, the looping that happened in the old representation was at the tree-evaluation level (**Repeat** *Main Loop* **Until ...**) and was not explicitly controllable by the evolutionary process. In other words, most of the loops taking place in the old representation were semanticly derived from the memory, rather than corresponding directly to the syntactic shape of the program. The new representation allows arbitrary looping and it can be done at exactly the granularity desired. This representation change does not change the theoretical power of the language, but in practice it may make a big difference.

Another important reason, we believe, lies in the original motivation for changing the PADO language representation. The language was changed to better suit the needs of the co-evolved SMART operators that perform the genetic recombination in PADO (Teller 1996). The SMART operators in PADO examine one (Mutation) or more (Crossover) programs from the main population and decide how to change or recombine them in useful ways. Clearly, the language representation has a big effect not only on how these SMART operators act, but on how easy it is for them to understand and recombine the programs they examine.

One of the possible problems with the new representation is that it is much denser than the old representation (as mentioned above). This may make it harder for the SMART operators to examine the main population programs and understand what they do and how to change them most effectively. On the other hand, there are aspects of the new representation that are much more friendly to the SMART operators. For example, the looping is largely explicit in the new representation; it was all implicit in the old representation (as mentioned above). The explicitness of a PADO program's structure almost surely makes it easier for the SMART operators to suggest changes that have a better than random chance of producing highly fit offspring.

Another example of the new language's benefit to the SMART operators is in the program's method of specifying a response. Having the root of the "tree" return its value each time it is evaluated is explicit, but the value return must be done there and only there. In the new language, when the program writes its "Answer" to a memory position this write is independent of where it occurs in the program. This flexibility of expression gives many more options to a SMART operator trying to change one aspect of a program without disrupting others.

This same change in how the programs specify their responses certainly also has a positive effect directly on the main population. Simply, if a program has any sub-graph that computes a good idea (the nugget of its intended response) then all it has to do in the new rep-

resentation is (WRITE X <SUB-GRAPH>). This is much less likely to be disrupted than when some sub-tree in the old representation computes a good idea. This sub-tree is dependent on every node from its sub-tree-root up the path to the root of the whole tree. If any of these nodes change, the good idea is likely to be corrupted on the way up. This makes crossover much more destructive (without being more constructive) in the old representation. Now of course, this strategy of expecting the answer from a variable or memory position rather than the top of the tree could be implemented in a traditional tree-GP format. But since it wasn't in the old representation this is likely to be one of the contributing factors to the overall performance boost the new representations seems to provide.

A final point is that the SMART operators existed in both the old and new representations. Not only is there a difference for them because the syntax they have to examine changes between the two representations, but *their* syntax changes as well. So it is possible that some of the general performance improvement from the language progression comes from the SMART operators and that some of that improvement is due to the fact that the SMART operators are written in (and therefore evolve in) this apparently superior representation

New Language Representation Advantages

- Many, possibly interlocking, possibly nested, loops of various sizes
- Programs can update their "responses" at any time during arbitrary computation
- ADFs and Library's take arbitrary numbers of parameters
- ADFs and Library's are themselves full programs with recursion and mutual recursion.
- Arbitrary variety of sub-part reuse within a program
- Explicitness of the language (particularly looping structures) simplifies the SMART operators' job.
- If the new representation is advantageous, the SMART operators may improve their performance because they too are now written in and evolve in the new representation.

Conclusions

The goal of this paper has been to describe and justify the progression of PADO's language representation. Sections and described this representation progression. The Results Section gave empirical justification for this language progression; the results graphed in figure 4 show how identical experiments in two other papers differ as a result of the language change. Empirical justification is convincing, but seeking a theoretical foundation is also useful. The Discussion Section gave some insight into possible reasons why PADO's language progression caused a significant jump in performance, even across domains. A still more scientific, theoretical understanding of the relationship between language representation and evolution performance is part of our ongoing research.

What should the reader learn from this paper? The clearest message should be that the representation we choose has an important effect on how our evolutionary computation continues. Because PADO's new language works better then PADO's old language, we can assume that there are some positive lessons to be learned from this change. The Discussion section proposes some likely positive lessons, but without further investigations, these are proposals, not proofs.

There are a plethora of open questions in language representation for GP. This paper is not a general scheme for improving representations. The new PADO representations has much to recommend it, but is surely not the best language for all GP tasks. One of the main contributions of the PADO work so far is, we believe, the creation of this new representation. The new PADO representation has been specifically designed for the evolution of algorithms and the co-evolution of intelligent genetic recombination operators. We believe that this new language will prove to be of value to the general GP community in the near future. Whether still better languages can be made (or learned) and what they will look like is a story for another day.

References

Altenberg, L. 1994. The evolution of evolvability in genetic programming. In Kenneth E. Kinnear, J., ed., *Advances In Genetic Programming.* MIT Press. 47–74.

Angeline, P., and Pollack, J. 1993. Evolutionary module acquisition. In Fogel, D., ed., *Proceedings of the Second Annual Conference on Evolutionary Programming,* 154–163. Evolutionary Programming Society.

Keith, M. J., and Martin, M. C. 1994. Genetic programming in c++: Implementation issues. In Kenneth E. Kinnear, J., ed., *Advances In Genetic Programming.* MIT Press. 285–310.

Kinnear, K. 1993. Generality and difficulty in genetic programming: Evolving a sort. In *Proceedings of the Fifth International Conference on Genetic Algorithms.* San Mateo, CA: Morgan Kaufmann.

Koza, J. 1992. *Genetic Programming.* MIT Press.

Koza, J. 1994a. *Genetic Programming II.* MIT Press.

Koza, J. R. 1994b. Architecture-altering operations for evolving the architecture of a multi-part program in genetic programming. Technical Report STAN-CS-TR-94-1528, Computer Science Department, Stanford.

Langdon, W. 1995. Evolving data structures with genetic programming. In Forrest, S., ed., *Proceedings of the Sixth International Conference on Genetic Algorithms.* Morgan Kauffman.

Perkis, T. 1994. Stack-based genetic programming. In *Proceedings of the First IEEE International Conference on Evolutionary Computation,* 148–153. IEEE Press.

Siegel, E., and Chaffee, A. 1995. Evolutionary optimization of computation time of evolved algorithms. Unpublished report, Computer Science Department, Columbia.

Simms, K. 1994. Evolving virtual creatures. In *Proceedings of the 21st International SIGGRAPH Conference.* ACM Press.

Sushil, L. 1994. Using genetic algorithms to design structures. In *Proceedings of the 7th annual FLAIRS*, 120–127. IEEE Press.

Tackett, W. A. 1995. Greedy recombination and genetic search on the space of computer programs. In Whitley, L., and Vose, M., eds., *Proceedings of the Third International Conference on Foundations of Genetic Algorithms*, 118–130. Morgan Kauffman.

Tamaki, H. 1994. A comparison study of genetic codings for the traveling salesman problem. In *Proceedings of the First International Conference on Evolutionary Computation*, 1–6. IEEE Press.

Teller, A., and Veloso, M. 1995a. PADO: A new learning architecture for object recognition. In Ikeuchi, K., and Veloso, M., eds., *Symbolic Visual Learning.* Oxford University Press.

Teller, A., and Veloso, M. 1995b. PADO: Learning tree structured algorithms for orchestration into an object recognition system. Technical Report CMU-CS-95-101, Department of Computer Science, Carnegie Mellon Unversity.

Teller, A., and Veloso, M. 1995c. Program evolution for data mining. In Louis, S., ed., *The International Journal of Expert Systems. Third Quarter. Special Issue on Genetic Algorithms and Knowledge Bases.* JAI Press.

Teller, A. 1994a. The evolution of mental models. In Kenneth E. Kinnear, J., ed., *Advances In Genetic Programming.* MIT Press. 199–220.

Teller, A. 1994b. Turing completeness in the language of genetic programming with indexed memory. In *Proceedings of the First IEEE World Congress on Computational Intelligence*, 136–146. IEEE Press.

Teller, A. 1996. Evolving programmers: The co-evolution of intelligent recombination operators. In Kinnear, K., and Angeline, P., eds., *Advances in Genetic Programming II.* MIT Press.