Evolving Control Structures with Automatically Defined Macros
by Lee Spector

Full citation:
Spector, L. 1995. Evolving Control Structures with Automatically Defined Macros. *Working Notes of the AAAI Fall Symposium on Genetic Programming*. The American Association for Artificial Intelligence. pp. 99-105.

Evolving Control Structures with Automatically Defined Macros

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

Abstract

Koza has previously shown that the power of a genetic programming system can often be enhanced by allowing for the simultaneous evolution of a main program and a collection of automatically defined functions (ADFs). In this paper I show how related techniques can be used to simultaneously evolve a collection of automatically defined macros (ADMs). I show how ADMs can be used to produce new *control structures* during the evolution of a program, and I present data showing that ADMs sometimes provide a greater benefit than do ADFs. I discuss the characteristics of problems that may benefit most from the use of ADMs, or from architectures that include both ADFs and ADMs, and I discuss directions for further research.

Introduction

Modern programming languages support the production of structured, modular programs through several mechanisms including subroutines, coroutines, and macros. Koza has shown that the power of a genetic programming system can often be enhanced by allowing for the simultaneous evolution of a main program and a collection of subroutines (Koza 1994a). Further studies have investigated factors underlying the performance of Koza's automatically defined functions (ADFs) along with alternative techniques for the automatic generation of subroutines (Kinnear 1994).

While subroutines promote program modularity and code reuse, they do not normally provide programmers with the tools needed to produce new *control structures* or to otherwise enhance the structure of their programming languages. Many languages provide an alternative mechanism, macros, to support this need (Kernighan & Ritchie 1988) (Steele 1990).

In this paper I show how a genetic programming system can simultaneously evolve a program and its control structures; that is, I show how a genetic programming system can simultaneously evolve a main program and a collection of automatically defined macros (ADMs). Using a variant of Koza's obstacle-avoiding robot problem, I show that ADMs are in some cases more useful than ADFs. For other problems, however, ADFs are more useful than ADMs. ADFs and ADMs serve different purposes — as do functions and macros in common programming languages — and we can therefore expect architectures that include both ADFs and ADMs to provide the best support for the evolution of programs in certain complex domains.

In the following pages I briefly describe the genetic programming framework and the use of ADFs in genetic programming. I then discuss the use of macros to define new control structures and show how a genetic programming system can simultaneously evolve a main program and a set of ADMs that are used by the main program. I present data from case studies, discuss the results, and describe directions for future research.

Genetic Programming

Genetic programming is a technique for the automatic generation of computer programs by means of natural selection (Koza 1992). The genetic programming process starts by creating a large initial population of programs that are random combinations of elements from problem-specific function and terminal sets. Each of the programs in the initial population is assessed for fitness. This is usually accomplished by running each program on a collection of inputs called fitness cases, and by assigning numerical fitness values to the output of each of these runs; the resulting values are then combined to produce a single fitness value for the program.

The fitness values are used in producing the next generation of programs via a variety of genetic operations including reproduction, crossover, and mutation. Individuals are randomly selected for participation in these operations, but the selection function is biased toward highly fit programs. The reproduction operator simply selects an individual and copies it into the next generation. The crossover operation introduces

variation by selecting two parents and by generating from them two offspring; the offspring are produced by swapping random fragments of the parents. The mutation operator produces one offspring from a single parent by replacing a randomly selected program fragment with a newly generated random fragment.

Over many generations of fitness assessment, reproduction, crossover, and mutation, the average fitness of the population may tend to improve, as may the fitness of the best-of-generation individual from each generation. After a preestablished number of generations, or after the fitness improves to some preestablished level, the best-of-run individual is designated as the result and is produced as the output from the genetic programming system.

Automatically Defined Functions

Koza has shown that the performance of a genetic programming system, as measured by the number of individuals that must be processed to produce a solution with a probability of 99%, can often be improved by allowing for the simultaneous evolution of a main program and a collection of subroutines (Koza 1994a). He implements the evolution of subroutines by considering one part of an evolved program to be a main program (or "result producing branch") while other parts are treated as definitions for automatically defined functions (ADFs). Each ADF may have its own function and terminal set, and hierarchical references between ADFs are allowed.

Koza showed that the use of ADFs allows a genetic programming system to better exploit regularities of problem domains, improving system performance. Further studies have investigated factors underlying the performance of Koza's ADFs along with alternative techniques for the automatic generation of subroutines (Kinnear 1994).

Macros

I use the term "macro" to refer to operators that perform source code transformations. Many programming languages provide macro definition facilities, although the power of such facilities varies widely. For example, C provides substitution macros by means of a preprocessor (Kernighan & Ritchie 1988), while Common Lisp allows the full power of the programming language to be used in the specification of macros (Steele 1990). A macro "call" is textually transformed into new source code prior to compilation or interpretation; this process is often called *macro expansion.*

Macros, like subroutines, can assist in the modularization of complex programs and in the exploitation of domain regularities. In certain circumstances macros can be more useful than subroutines.[1] In particular, one can implement new *control structures* with macros. One does this by writing macros that expand into code fragments that include the arguments to the macro call, unevaluated, in one or more places. If the bodies of code that appear as arguments to the macro work by side effect or are sensitive to their calling contexts, then the macro call can produce an effect not obtainable with subroutines. For example, consider the following Common Lisp definition for a macro called do-twice[2]:

```
(defmacro do-twice (code)
  `(progn ,code ,code))
```

Assume that the (beep) function works by producing a side effect. The call (do-twice (beep)) will expand into (progn (beep) (beep)), causing the side effect to be produced twice. Do-twice could not have been implemented as a Common Lisp *function* because the arguments in a function call are evaluated before being passed. A do-twice function would, in this context, receive only the *result* of (beep), not the code; it therefore could not produce the effect of two calls to (beep).

More generally, the utility of macros stems in part from the fact that they control the evaluation of their own arguments. This allows one to use macros to implement control structures that perform multiple evaluation or conditional evaluation of bodies of code.

One often builds new macros that leverage the utility of pre-existing macros or built-in control structure syntax. For example, one could use an existing if structure to build an arithmetic-if control structure, and one could use an existing while structure, along with problem-specific operators, to build a while-no-obstacles control structure.

There are many domains in which problem-specific control structures are useful. In a robot control domain one might want to use a control structure that causes an action to be repeated until a condition in the world becomes true. For example, the following macro causes the robot to turn until the given sense-expression returns non-nil, and returns the value of the given value-expression in its final orientation:

```
(defmacro where-sensed
     (sense-expression value-expression)
  `(progn (while (not ,sense-expression)
                 (turn))
          ,value-expression))
```

[1] A good discussion of related issues can be found in (Graham 1994).

[2] The "backquote" syntax used in this definition is documented in (Steele 1990).

This macro would be most useful when the bodies of code specified for sense-expression and value-expression depend on the orientation of the robot. If the domain provides many opportunities for the construction of such expressions then where-sensed may provide a useful behavioral abstraction that could not be obtained with ordinary subroutines.

Automatically Defined Macros

It is possible to simultaneously evolve a main program and a set of Automatically Defined Macros (ADMs) that may be used by the main program. In the experiments described below I use only the simplest sort of *substitution* macros; each macro specifies a template for the code into which macro calls will expand. Each ADM definition is treated as if it was defined with defmacro, with its body preceded by a backquote, and with an implicit comma before each occurrence of a parameter. Conceptually, macro expansion is performed by replacing the call with the template, and by then replacing occurrences of the macro's parameters with the bodies of code that appear in the call. While languages such as Common Lisp allow one to write macros that specify more general code transformations, substitution macros can nonetheless be quite useful.[3]

In practice one can avoid full expansion of substitution macros by expanding the ADMs incrementally during evaluation. Incrementally expanded substitution ADMs are easy to implement through minor modifications to Koza's publicly available ADF code (Koza 1994a). Koza's ADF functions are defined to fast-eval the evolved ADF code trees, after first binding the parameter symbols to the (evaluated) values passed to the ADF. For ADMs one can change fast-eval so that it treats ADMs like pseudo-macros, passing them unevaluated code trees as arguments. One can then define the ADM functions to substitute the unevaluated code trees for the parameter symbols, and then to fast-eval the result:

```
(defun adm0 (a0)
  (fast-eval (subst a0 'arg0 *adm0*)))
```

Zongker's C-based lil-gp system provides an even simpler way to implement substitution ADMs (Zongker 1995). One can simply use the evaluation function type EVAL_EXPR (rather than EVAL_DATA, which is used for normal ADFs), and ADM semantics will result. This is achieved by changing the runtime interpretation of a parameter symbol to branch to the code tree passed as an argument; actual expansion of the macro call is thereby avoided altogether.

[3]Examples of macros that perform more exotic code transformations can be found in (Graham 1994).

More complex implementation strategies are required for ADMs that perform non-substitutional transformations of their arguments. In some cases it may be necessary to fully expand the ADMs prior to evaluation; this may lead to long macro-expansion delays and to very large expansions. All of the experiments described in this paper used substitution ADMs and therefore took advantage of one of the simple implementation techniques described above.

Koza has previously made limited use of macro expansion in genetic programming; he used it as a means for deleting elements of programs during the simultaneous evolution of programs and their architectures (Koza 1994b). The present work argues for the more general use of macro expansion through the evolution of ADMs.

While ADMs and ADFs are in fact compatible and may be used together, the present study highlights the differences between ADFs and ADMs by using each to the exclusion of the other, and by contrasting the results. The simultaneous use of ADFs and ADMs is briefly discussed at the end of the paper.

The Dirt-Sensing, Obstacle-Avoiding Robot

Among the domains in which we expect ADMs to have utility are those that include operators that work by producing side effects, operators that are sensitive to their calling contexts, or pre-existing macros. Koza's obstacle-avoiding robot problem (hereafter "OAR") has all of these elements. The problem as expressed by Koza is quite difficult, and he was only able to solve it by using an unusually large population (4000). I have produced a somewhat simpler version of OAR by adding an additional sensor function (IF-DIRTY); with this change the problem can easily be solved with a population as small as 500.

The goal in the dirt-sensing, obstacle-avoiding robot problem (hereafter "DSOAR") is to find a program for controlling the movement of an autonomous floor-mopping robot in a room containing harmless but time-wasting obstacles. The problem is an extension of OAR, which Koza describes as follows:

> In this problem, an autonomous mobile robot attempts to mop the floor in a room containing harmless but time-wasting obstacles (posts). The obstacles do not harm the robot, but every failed move or jump counts toward the overall limitation on the number of operations available for the task.
>
> ...the state of the robot consists of its location in the room and the direction in which it is facing. Each square in the room is uniquely identified by

a vector of integers modulo 8 of the form (i, j), where $0 \leq i, j \leq 7$. The robot starts at location (4,4), facing north. The room is toroidal, so that whenever the robot moves off the edge of the room it reappears on the opposite side.

Six non-touching obstacles are randomly positioned in a room laid out on an 8-by-8 grid. ...

The robot is capable of turning left, of moving forward one square in the direction in which it is currently facing, and of jumping by a specified displacement in the vertical and horizontal directions. Whenever the robot succeeds in moving onto a new square (by means of either a single move or a jump), it mops the location of the floor onto which it moves. (Koza 1994a, p. 365)

OAR uses terminal sets consisting of the 0-argument function (MOP), the 0-argument function (LEFT), random vector constants modulo 8 ($\Re_{v8}$), and the names of arguments for ADFs. The same terminal sets are used in DSOAR.

(MOP) moves the robot in the direction it is currently facing, mops the floor at the new location, and returns the vector value (0,0). If the destination contains an obstacle then the robot does not move, but the operation still counts toward the limit on the number of movement operations that may be performed in a run. (LEFT) turns the robot 90° to the left and returns the vector value (0,0).

OAR uses a function set consisting of the operators IF-OBSTACLE, V8A, FROG, PROGN, and the names of the ADFs. DSOAR uses the same function set (substituting the names of ADMs when appropriate) and adds one new operator, IF-DIRTY.

IF-OBSTACLE is a 2-argument conditional branching operator that evaluates its first argument if an obstacle is immediately in front of the robot; it evaluates its second argument otherwise. IF-OBSTACLE is implemented as a pseudo-macro. V8A is a 2-argument vector addition function that adds vector components modulo 8. FROG is a 1-argument movement operator that jumps the robot to the coordinate produced by adding (modulo 8) its vector argument to the robot's current location. FROG acts as the identity operator on its argument, and fails in the same way as MOP when the destination contains an obstacle. IF-DIRTY is a 2-argument conditional branching operator that evaluates its first argument if the square immediately in front of the robot is dirty; it evaluates its second argument if the square has been mopped or if it contains an obstacle. IF-DIRTY is implemented as a pseudo-macro.

Two fitness cases, shown in Figure 1, are used for the DSOAR problem. Each program is evaluated once for each fitness case, and each evaluation is terminated prematurely if the robot executes either 100 (LEFT) operations or 100 movement operations ((MOP) and FROG operations combined). The raw fitness of each program is the sum of the squares mopped over the two fitness cases. A program is considered to have solved the problem if, over the two fitness cases, it successfully mops 112 of the total of 116 squares that do not contain obstacles.

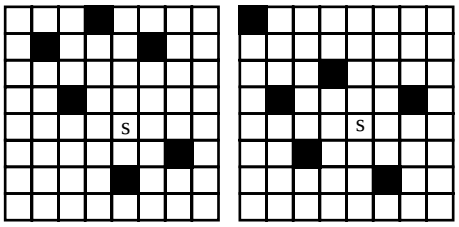

Figure 1: The fitness cases used for the DSOAR problem.

Results

I performed 200 runs of a genetic programming system on this problem, half with ADFs and half with ADMs. Each program in each population had an architecture consisting of a result-producing branch and two modules (either ADF0 and ADF1 or ADM0 and ADM1). ADF0 and ADM0 each took 1 argument[4] and ADF1 and ADM1 each took 2 arguments[5]. ADF1 and ADM1 could call ADF0 and ADM0, respectively, and result producing branches could call both of the automatically defined modules. I used a population size of 500 and ran each run for 51 generations. Tournament selection was used with a tournament size of 7.

Figure 2 shows a summary of the results as a graph of P(M,i), the probability of success by generation. The probability of producing a solution to this problem using a population size M=500 by any given generation is generally greater with ADMs than with ADFs; this can be seen by noting that the ADM line rises faster than the ADF line. Figure 3 shows a summary of the results as a graph of I(M,i,z), the number of individuals that must be processed to produce a solution with probability greater than z=99%. The number of individuals that must be processed is lower for ADMs than for ADFs; this can be seen by noting that the ADM line falls faster than the ADM line, and that it reaches a lower minimum. The minimum, defined by Koza as the "computational effort" required to solve the prob-

[4]Koza used a 0-argument ADF0 for OAR.

[5]Koza used a 1-argument ADF1 for OAR.

lem, is 26000 when ADFs are used; the computational effort is 21000 when ADMs are used instead.

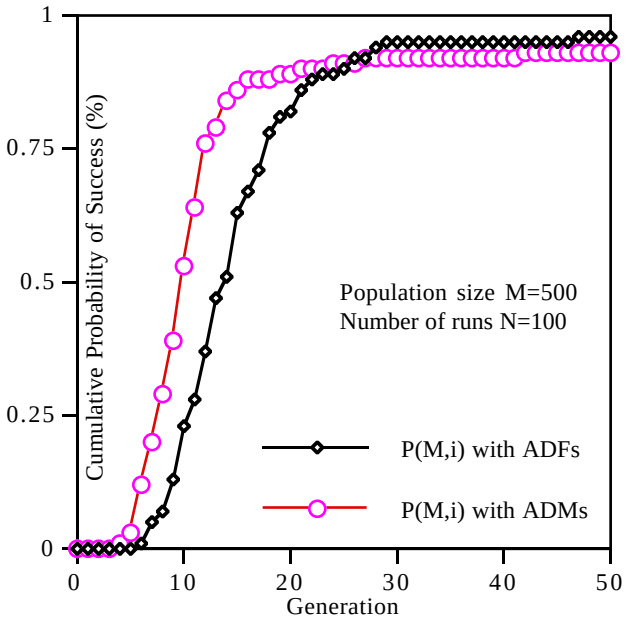

Figure 2: P(M,i) for ADFs and ADMs on the DSOAR problem.

The calculations of P(M,i) and I(M,i,z) were performed according to the discussion on pages 99 through 103 of (Koza 1994a). P(M,i) is calculated by dividing the number of runs that succeed in each generation by the total number of runs, and by then calculating a running sum of the quotients across generations. I(M,i,z) is calculated from P(M,i) according to the following formula:

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

The results show that for DSOAR, with the given parameters, ADMs are somewhat more useful than ADFs; the number of individuals that must be processed to produce a solution using ADMs is lower than the number that must be processed to produce a solution using ADFs.

The Lawnmower Problem

I have also obtained a related but negative result on Koza's 64-square Lawnmower problem. The Lawnmower problem is identical to OAR (described above) except that there are no obstacles and the IF-OBSTACLE operator is not used. Note that this domain includes no pre-existing macros and no operators that are sensitive to the robot's environment. It does, however, include operators that work by side effect, so one might expect ADMs to be more useful than ADFs.

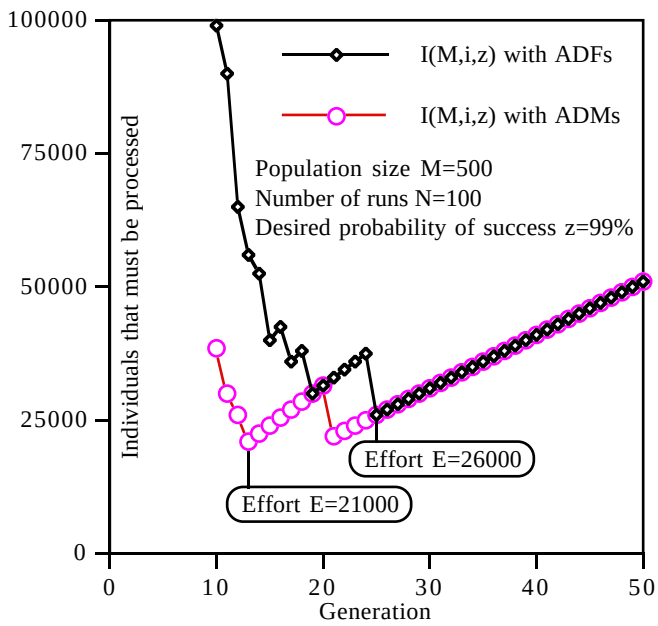

Figure 3: I(M,i,z) for ADFs and ADMs on the DSOAR problem.

I performed 200 runs of a genetic programming system on the 64-square Lawnmower problem, half with ADFs and half with ADMs. Aside from the switch to ADMs for half of the runs, I used the same parameters as did Koza (Koza 1994a). The results are shown in Figures 4 and 5. Figure 4 shows P(M,i) for ADFs and for ADMs on this problem; it is not obvious from casual inspection of this graph that either type of module provides greater benefit. Figure 5 shows I(M,i,z), from which it is clear that ADMs are more of a hindrance than a help on this problem; the computational effort (minimum for I(M,i,z)) is 18000 for ADMs, but only 12000 for ADFs.

When Are ADMs Useful?

The semantics of ADFs and of substitution ADMs are equivalent when all operators in the domain are purely functional. The only difference between an ADF and an ADM in such a case is that the ADM may take more runtime to evaluate, since the code trees appearing as arguments to the ADM may have to be evaluated multiple times. It is therefore clear that substitution ADMs can only be more useful than ADFs in environments that include operators that are not purely functional.

Even when a domain includes operators that work by

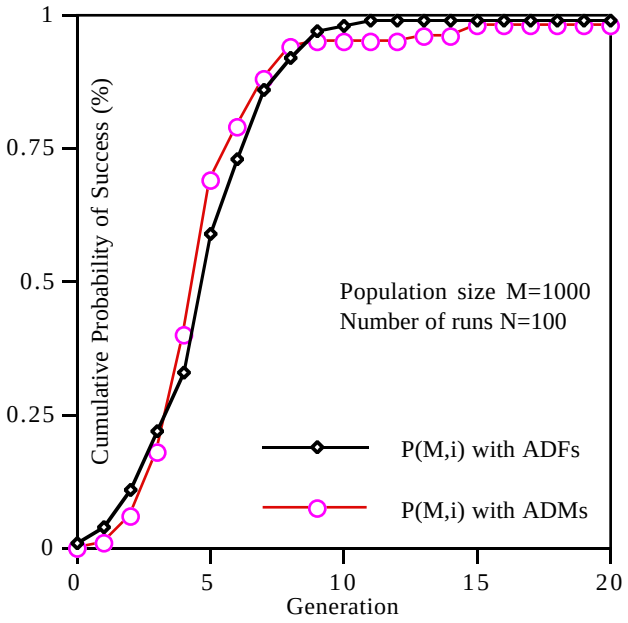

Figure 4: P(M,i) for ADFs and ADMs on the Lawnmower problem.

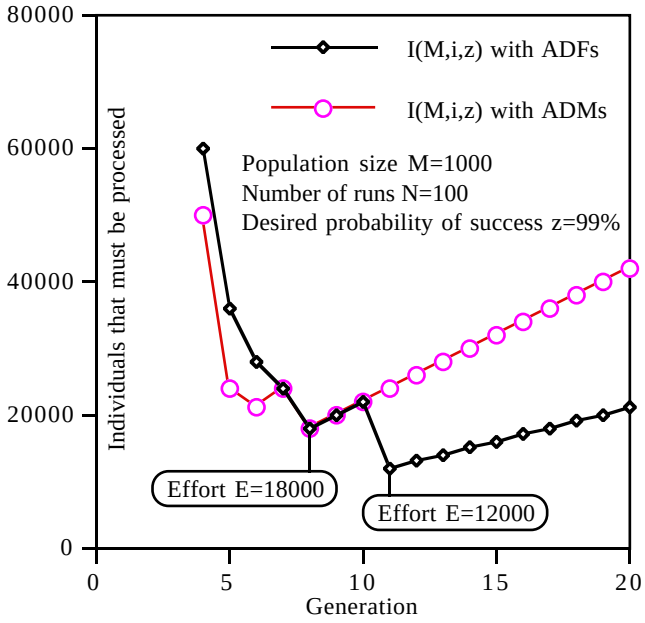

Figure 5: I(M,i,z) for ADFs and ADMs on the Lawnmower problem.

side effect, there is no guarantee that the use of ADMs will result in less individuals being processed than will the use of ADFs. The negative result on the Lawnmower problem is a testament to this fact. Informally we may speculate that the "less functional" a domain is, the more likely that ADMs will be useful in that domain. If context-sensitive and side-effecting operators play an important role in a given domain, then it is likely that new and/or problem-specific control structures will be useful; we can therefore expect a genetic programming system to take advantage of automatically defined macros to produce control structures that help in evolving a solution to the problem.

Although ADFs and ADMs have been contrasted above for expository purposes, they are in fact completely compatible with one another. Just as a human programmer may wish to define both new functions and new control structures while solving a difficult programming problem, it may be advantageous for genetic programming to define a collection of ADFs and a collection of ADMs. Functions may be most helpful for some aspects of a problem domain, while macros may be most helpful for others. Since the optimal number of ADFs and ADMs may not be clear from the outset, it may also be advantageous to simultaneously evolve programs and their macro-extended architectures, in the style of (Koza 1994b).

Future Work

A reasonable speculation is that architectures that include both ADFs and ADMs will be particularly useful in application areas that have traditionally made use of exotic control structures. One such application area is "dynamic-world planning" or "reactive planning," in which the goal is to produce systems that integrate planning and action in complex environments. Computational architectures for dynamic-world planning tend to use multiple control "levels," blackboard-based opportunistic control structures, "monitor" processes, and other complex and unusual control structures (Dean & Wellman 1991, Spector 1992). A genetic programming system with ADMs should be capable of evolving and refining such control structures to suit particular problem environments. One clear avenue for future work therefore is to apply ADM-based techniques to dynamic-world planning problems; preliminary work on the "wumpus world" environment (Russell & Norvig 1995) has produced results similar to those reported for DSOAR above, indicating that ADMs are indeed useful for such applications.

Even when ADMs decrease the number of individuals that must be processed to solve a problem, the runtime costs of ADMs may cancel any savings in problem solving time. Such costs include the time spent on redundant re-evaluation of purely functional code fragments, and, depending on the techniques used to im-

plement ADMs, macro-expansion costs. Further studies must be conducted on the trade-offs involved.

The ADMs considered in this paper are all substitution macros, but it should also be possible to evolve more powerful code transforming operators. Macros may also be useful in more ways than were sketched above; for example, they can be used to establish variable bindings or, like `setf` in Common Lisp, to implement "generalized variables." These uses of macros, and the benefits of richer macro definition facilities, should also be explored.

Conclusions

The human programmer's toolkit includes several module-building tools, each of which can be useful in certain circumstances. Genetic programming systems should have access to a similar toolkit. In particular, they should have access to macro-definition facilities so that they can evolve control structures appropriate to particular problems. Automatically Defined Macros can improve the performance of a genetic programming system, but more work must be done to refine our understanding of the conditions under which ADMs, or combinations of ADFs and ADMs, are likely to be helpful. Perhaps the best strategy, given sufficient computational resources, is to simultaneously evolve programs and their macro-extended architectures (the number of ADF/Ms and the number of arguments that each takes). Architectures that include both ADFs and ADMs may prove to be particularly useful in domains that have traditionally made use of exotic control structures; for example, dynamic-world planning.

Acknowledgments

The idea for this work emerged from a conversation with William E. Doane in which he suggested that we simultaneously evolve chromosomes and protein expression mechanisms. The comments of an anonymous reviewer lead to several significant improvements in this work.

References

Dean, T.L. and M.P. Wellman. 1991. *Planning and Control.* San Mateo, CA: Morgan Kaufmann Publishers.

Graham, P. 1994. *On Lisp: Advanced Techniques for Common Lisp.* Englewood Cliffs, NJ: Prentice Hall.

Kernighan, B.W. and D.M. Ritchie. 1988. *The C Programming Language.* Second Edition. Englewood Cliffs, NJ: Prentice Hall.

Kinnear, K.E. Jr. 1994. Alternatives in Automatic Function Definition: A Comparison of Performance. In K.E. Kinnear Jr., Ed., *Advances in Genetic Programming.* pp. 119–141. Cambridge, MA: The MIT Press.

Koza, J.R. 1992. *Genetic Programming.* Cambridge, MA: The MIT Press.

Koza, J.R. 1994a. *Genetic Programming II.* Cambridge, MA: The MIT Press.

Koza, J.R. 1994b. Architecture-altering Operations for Evolving the Architecture of a Multi-part Program in Genetic Programming. Computer Science Department, Stanford University. CS-TR-94-1528.

Russell, S.J., and P. Norvig. 1995. *Artificial Intelligence, A Modern Approach.* Englewood Cliffs, NJ: Prentice Hall.

Spector, L. 1992. Supervenience in Dynamic-World Planning. Ph.D. diss., Dept. of Computer Science, University of Maryland.

Steele, G.L. Jr. 1990. *Common Lisp.* Second Edition. Digital Press.

Zongker, D. and B. Punch. 1995. *lil-gp 1.0 User's Manual.* Available via the Web at http://isl.cps.msu.edu/GA/software/lil-gp, or via anonymous FTP to isl.cps.msu.edu, in the directory "/pub/GA/lilgp".