The role of Hierarchies and Reuse in Genetic Programming in Achieving Scalable Automatic Programming

John R. Koza
Biomedical Informatics Program
Department of Electrical Engineering
Stanford University
Stanford, California
koza@stanford.edu

Anyone who has ever looked at a floor plan of a building, a corporate organization chart, a musical score, a protein molecule, a city map, or a large electrical circuit diagram will be struck by the massive reuse of certain basic substructures within the overall structure. Indeed, complex structures are almost always replete with modularities, symmetries, and regularities. Reuse avoids reinventing the wheel on each occasion requiring a particular sequence of already-learned steps. We believe that reuse is essential preconditions for scalability in automatic problem solving.

Automatically defined functions (ADFs) (subroutines) are a very important mechanism by which genetic programming can automatically reuse code—either exactly or with different instantiations of dummy variables or formal parameters ((Koza 1990, Koza and Rice 1991, Koza 1992, Koza 1994).

The 1994 book *Genetic Programming II: Automatic Discovery of Reusable Programs* (Koza 1994a) made the key point that the reuse of code is a critical ingredient to scalable automatic programming. The book discusses scalability in terms of the rate (e.g., linearly, exponentially) at which the computational effort required to yield a solution to differently sized instances of a particular problem (e.g., n^{th}-order parity problem, lawnmower problem for an $n \times m$ lawn) changes as a function of problem size. The book demonstrated that one way of achieving scalability is by reusing code by means of subroutines (automatically defined functions) and iterations (automatically defined iterations). The book's eight main points are:

- Automatically defined functions enable genetic programming to solve a variety of problems in a way that can be interpreted as a decomposition of a problem into subproblems, a solving of the subproblems, and an assembly of the solutions to the subproblems into a solution to the overall problem (or that can alternatively be interpreted as a search for regularities in the problem environment, a change in representation, and a solving of a higher-level problem).
- Automatically defined functions discover and exploit the regularities, symmetries, homogeneities, similarities, patterns, and modularities of the problem environment in ways that are very different from the style employed by human programmers.
- For a variety of problems, genetic programming requires less computational effort (fewer fitness evaluations to yield a solution with a satisfactorily high probability) with automatically defined functions than without them, provided the difficulty of the problem is above a certain relatively low break-even point.
- For a variety of problems, genetic programming usually yields solutions with smaller overall size (lower average structural complexity) with automatically defined functions than without them, provided the difficulty of the problem is above a certain break-even point.
- For the three problems in *Genetic Programming II* for which a progression of several scaled-up versions is studied, the average size of the solutions produced by genetic programming increases as a function of problem size at a lower rate with automatically defined functions than without them.
- For the three problems in *Genetic Programming II* for which a progression of several scaled-up versions is studied, the number of fitness evaluations required by genetic programming to yield a solution (with a specified high probability) increases as a function of problem size at a lower rate with automatically defined functions than without them.
- For the three problems in *Genetic Programming II* for which a progression of several scaled-up versions is studied, the improvement in computational effort and average structural complexity conferred by automatically defined functions increases as the problem size is scaled up.
- Genetic programming is capable of simultaneously solving a problem and selecting the architecture of the overall program (consisting of the number of automatically defined functions and the number of their arguments).

In addition, *Genetic Programming II* demonstrated that it is possible to

- automatically create multibranch programs containing an iteration-performing branch as well as a main program and subroutines (e.g., the transmembrane identification problem and the omega loop problem),
- automatically create multibranch programs containing multiple iteration-performing branches and iteration-terminating branches (e.g., the look-ahead version of the transmembrane problem), and

- automatically determine the architecture for a multibranch program in an architecturally diverse population by means of evolutionary selection.

Automatically defined iterations (ADI's), automatically defined loops (ADL's), and automatically defined recursions (ADR's) provide additional ways by which genetic programming can automatically reuse code (Koza 1994; Koza, Bennett, Andre, and Keane 1999). Automatically defined stores (ADS's) provide a way by which genetic programming can automatically reuse the results produced by the execution of code. In genetic programming, the architecture, hierarchy, size, and content of the evolved computer program is part of the output produced by genetic programming—not part of the input supplied by the system's human user. Genetic programming uses architecture-altering operations (Koza, Bennett, Andre, and Keane 1999) to automatically determine program architecture in a manner that parallels gene duplication, and the related operation of gene deletion, in nature.

The genome of the simplest currently-known living organism manufactures only 470 different proteins, whereas the human genome manufactures about 100,000 proteins. Since mutation and crossover modify only a preexisting gene, the question arises as to how do new genes—that is, new biological functions—originate in nature? Occasionally, a gene may be duplicated—thus creating two places on the chromosome that manufacture the same protein. After such a gene duplication, one of the two initially identical genes may remain intact and continue to manufacture the original protein—thus performing the gene's presumably survival-related function. Meanwhile, over many generations, the second gene may harmlessly accumulate changes and diverge. Eventually the second gene may come to manufacture a new protein with a new function. Thus, new biological functions—that is, new genes and proteins—emerge in nature as part of the evolutionary process.

The subroutine duplication operation duplicates a preexisting subroutine in an individual program, gives a new name to the copy, and randomly divides the preexisting calls to the old subroutine between the two. This operation changes the program architecture by broadening the hierarchy of subroutines in the overall program. As with gene duplication in nature, this operation preserves semantics when it first occurs. The two subroutines typically diverge later—sometimes yielding specialization.

The argument duplication operation duplicates one argument of a subroutine, randomly divides internal references to it, and preserves overall program semantics by adjusting all calls to the subroutine. This operation enlarges the dimensionality of the subspace on which the subroutine operates.

The subroutine creation operation can create a new subroutine from part of a main result-producing branch (main program), thereby deepening the hierarchy of references in the overall program, by creating a hierarchical reference between the main program and the new subroutine. The subroutine creation operation can also create a new subroutine from part of an existing subroutine, further deepening the hierarchy of references. The subroutine creation operation can also create a new subroutine from part of an existing subroutine by creating a hierarchical reference between a preexisting subroutine and a new subroutine and a deeper and more complex overall hierarchy.

The subroutine deletion operation deletes a subroutine from a program thereby making the hierarchy of subroutines either narrower or shallower.

The argument deletion operation deletes an argument from a subroutine thereby reducing the amount of information available to the subroutine — a process that can be viewed as generalization.

Other architecture-altering operations add and delete iterations, loops, recursions, and memory.

These architecture-altering operation quickly create an architecturally diverse population containing programs with different numbers of subroutines, arguments, iterations, loops, recursions, and memory and, also, different hierarchical arrangements of these elements. Programs with architectures that are well-suited to the problem at hand will tend to grow and prosper in the competitive evolutionary process, while programs with inadequate architectures will tend to wither away under the relentless selective pressure of the problem's fitness measure.

Bibliography

Koza, John R. 1990. *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems.* Stanford University Computer Science Department technical report STAN-CS-90-1314. June 1990.

Koza, John R., and Rice, James P. 1991. Genetic generation of both the weights and architecture for a neural network. In *Proceedings of International Joint Conference on Neural Networks*, *Seattle, July 1991*. Los Alamitos, CA: IEEE Press. Volume II. Pages 397-404.

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

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

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1999. *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco, CA: Morgan Kaufmann.