Code Factoring and the Evolution of Evolvability

Terry Van Belle
Department of Computer Science
University of New Mexico
Albuquerque, New Mexico, USA
vanbelle@cs.unm.edu
505-277-7833

David H. Ackley
Department of Computer Science
University of New Mexico
Albuquerque, New Mexico, USA
ackley@cs.unm.edu
505-277-9149

Abstract

Evolvability can be defined as the capacity of a population to evolve. We show that one advantage of Automatically Defined Functions (ADFs) in genetic programming is their ability to increase the evolvability of a population over time. We observe this evolution of evolvability in experiments using genetic programming to solve a symbolic regression problem that varies in a partially unpredictable manner. When ADFs are part of a tree's architecture, then not only do average populations recover from periodic changes in the fitness function, but that recovery rate itself increases over time, as the trees adopt modular software designs more suited to the changing requirements of their environment.

1 EVOLUTION OF EVOLVABILITY

Evolutionary biologists have explored the sometimes controversial notion that beyond merely producing organisms adapted to their environments, the forces of evolution may operate to improve the adaptation process itself (Dawkins 1989, for example). The idea is that over relatively long time periods populations can become *more capable of adaptation* in the face of future environmental change, a process described as "the evolution of evolvability."

While it may appear only logical that a 'more adaptable' population would ultimately outcompete and displace some other population that is less so, it seems a good deal less obvious when imagining how that competition would actually have to play out. If two individuals are equally fit in their environment there will be no direct pressure favoring the survival of one over the other, even if they vary drastically in how well-suited their designs would be for evolutionary adaptation to future changes. Even worse, if there is any near-term fitness cost associated with being adaptable for the future, we would expect such adaptability to dwindle rather than proliferate.

1.1 EVOLVABILITY IN ARTIFICIAL LIFE

Artificial life researchers have developed models in which the capacity of populations to adapt improves. Some approaches have involved mechanisms such as encoding the mutation rate inside of the genotype (Fogel, Fogel & Atmar 1991), using 'locking bits' to turn on and off the mutability of individual data bits (Turney 1999), and allowing variable gene ordering to encourage modularity (Pepper 2000). Although not providing experimental results, (Altenberg 1994) suggested that genetic programming (GP) experiments can exhibit an increase in evolvability through the proliferation of favorable blocks of code.

On the other hand, given an unchanging fitness function most typical genetic algorithms will converge to a small number of genotypes. As that happens, the average fitness of the population rises, but the ability of the population to adapt further declines, because less and less of the overall solution space remains easily reachable by applications of the genetic operators. In such cases, shorter-term fitness optimization is antagonistic to longer-term maintenance of evolvability.

1.2 CODE FACTORING

In software engineering, *code factoring* refers to the process of reorganizing the code within a program to improve its 'internal structure' in some manner, generally without changing what the program actually does (Fowler, Beck, Brant, Opdyke & Roberts 1999). Code factoring is used, for example, to merge duplicated pieces of code, and to separate more volatile code elements from a more stable code base. In this paper we explore a model that displays an increase in evolvability based on evolutionary code factoring.

We applied genetic programming to a simple symbolic regression task with a repeated term. Unlike typical GP experiments, we periodically varied the value of this repeated term over the course of a run, thus changing the GP population's environment. We report on three experiments comparing trees that contained an ADF (Koza 1994) with trees that did not. We found that while both populations only adapted in temporarily constant environments, the ADF population also evolved forms capable of adapting more quickly to a changed environment—the evolvability of the ADF population evolved in a positive direction.

2 BACKGROUND

2.1 BIOLOGY AND SOFTWARE

(Kirschner & Gerhart 1998) point out how some biological mechanisms—such as versatile protein elements, weak linkage, compartmentation, redundancy, and exploratory behavior—can improve the evolvability of multicellular organisms. Such mechanisms can reduce the interdependence between components, allowing functionally independent traits to vary without adversely affecting each other.

At the same time, sometimes decreasing flexibility can be advantageous. (Dawkins 1996) cites bilateral symmetry as an example of a constraint that can improve evolvability. If longer legs, say, would improve fitness, a developmental process structured so that both legs lengthened equally as the result of a single mutation could have an advantage over another ontogeny that required a separate mutation for each leg to occur simultaneously. The former approach sacrifices the added flexibility of differing limb lengths, but we imagine that would be generally detrimental anyway. Evolvability is inherently about placing both flexibility and constraint where they are likely to help more than hurt, and involves betting on how the future is likely to be different from the present. If there are patterns in the environmental changes, evolvability may be able to gain traction.

Similar issues arise in software design (Altenberg 1994, for example). The environment to which the software must adapt includes the changing requirements of the software users (Nehaniv 2000, Stiemerling & Cremers 2000). Such changes are not completely random, and good software design tries to anticipate them.

For example, when designing code for a graphical user interface that contains some clickable buttons, one possible approach would be to write a completely separate module for each button. This allows great flexibility in that everything about the appearance and behavior of one button could be changed without any effect on the others, but essentially no software designer would even consider such an approach. Buttons in GUIs generally behave in largely similar fashions, so the code responsible for each button can be very similar as well. Software designers separate attributes that distinguish the buttons—location, label, and action, for example—and provide them as parameters modifying the behavior of a single piece of code. The designer can then maintain the common code for all buttons simultaneously.

Although both solutions—lots of nearly duplicated code versus parameterized common code—could precisely satisfy the immediate behavioral needs, one solution is likely to be more evolvable than the other. Designers of durable systems strive not only to satisfy current requirements, but also to be adaptable along the dimensions in which they believe the requirements will vary in the future. This, we hypothesize, is a key part of the distinction between a program which merely solves a problem and one which solves it with *good design*: The latter is more evolvable.

In both biological and computational systems, the environment within which an organism or piece of software must function is likely to be more constant along some dimensions, and more variable along others. Systems with high evolvability will be adaptable along the variable dimensions of the environment without disrupting those design elements that have adapted to the environment's constant dimensions.

2.2 MODULARITY IN GP

(Koza 1994) introduced a variant to genetic programming, known as "Automatically Defined Functions" (ADFs), to introduce modularity into genetic programming, which previously had typically involved only a single block of code per organism. With ADFs, each genotype contains multiple blocks of code. One is designated the "Result Producing Branch" (RPB), and is the code that calculates the tree's result. The RPB can make use of special non-terminals that make calls to other blocks of code—the ADFs—that are by convention identified `ADF0`, `ADF1`, ... Each ADF contains a pre-set number of arguments, usually defined by the experimenter, which it can access through special argument terminal nodes, conventionally labelled `ARG0`, `ARG1`, ... The function $(x-1)^2+(x-1)$, for example, could be represented by:

```
ADF0:  (- X 1)
RPB:   (+ (* ADF0 ADF0) ADF0)
```

In this example, `ADF0` takes no arguments and computes $x-1$; the RPB computes the final function by calling `ADF0` several times. The combination of `ADF0` and the RPB constitutes a single genome.

(Koza 1994) demonstrated that introducing ADFs produced a near-universal improvement both in computational effort and in code size over equivalents without ADFs, provided the problem complexity exceeds a certain threshold. In this paper we show that ADFs can also improve a GP genome's evolvability over time.

3 EXPERIMENTAL FRAMEWORK

To investigate evolvability, we need both a dynamic environment and some evolutionary organisms. Here we present the models we used for each.

3.1 A DYNAMIC ENVIRONMENT

The environmental task was to perform symbolic regression on the function

$$y = A\sin(Ax) \tag{1}$$

where A is a constant, and x ranges from -1 to 1. Although A was held constant during fitness evaluations, it was changed periodically on a longer time scale, causing the fitness function to vary along a single dimension, while keeping all other dimensions constant. As software designers—seeing that A appears twice in the objective function and armed with the foreknowledge that A will vary over evolutionary time—we can readily conclude that it would be advantageous to factor out the computation of A and reuse that code; the question was whether 'blind evolution' would be able to see that as well.

Every generation, 200 x values were generated uniformly at random from the interval $[-1,1)$, and the corresponding y values were obtained from Equation 1 using the current A. A tree's fitness was calculated by evaluating it on the 200 current x's, and summing the absolute values of the differences between the correct y value and what the tree produced. For display purposes, we also computed the number of *hits* of a tree—the number of sample points where the calculated result was within 0.1 of the correct y value. Any tree scoring 200 hits was considered a *correct* tree, regardless of how it accomplished that performance.

Each run consisted of 1000 generations, evenly divided into *epochs* of L generations each. At the end of each epoch, a new value for A was selected uniformly at random from the range $[0,6)$.[1] Fitness of the best of generation was reported at the start of each epoch (immediately after A had been selected), and at the end of each epoch (after L generations of evolution had elapsed). Figure 1 summarizes the procedure used for the dynamic environment.

[1]Since $-A\sin(-Ax) \equiv A\sin(Ax)$, negative values of A would not add any problem complexity.

1. Update A
2. Generate 200 random sample points
3. Calculate fitnesses at the start of the epoch
4. Loop for L generations:
   - 4.1. Evolve for a generation
   - 4.2. Regenerate 200 random sample points
5. Calculate fitnesses at the end of the epoch
6. Calculate evolvability
7. Go to step 1

Figure 1: Experimental framework for a dynamic environment. See text for details.

3.1.1 Evolvability Defined

Within that environment, we defined the evolvability of the population at each epoch to be the following:

$$E = \frac{F_e - F_s}{L} \tag{2}$$

where F_s was the population's best fitness (measured in hits) just after A was changed and F_e was the best fitness at the end of L generations of evolution beyond that point.

3.2 EVOLUTIONARY ARCHITECTURES

We compared a 'monolithic' tree architecture containing no ADFs with an 'ADF' tree architecture that provided a single zero-argument ADF. In Experiment 1, below, the terminal node 'x' was not allowed in the ADF, so the ADF could only produce a constant value. In the monolithic configuration, a single branch calculated the entire function.

We used `lil-gp 1.1` (Punch & Goodman 1995), to run the experiments. A summary of the details can be found in Table 1. The percentage indicated for each operator gives the probability it will be used to produce an offspring. The 'Best' operator simply reproduces the best genome from the previous generation, then the second best, and so on for as many times as it is called, providing a kind of probabilistic elitism.

4 RESULTS

For each experimental configuration, we collected statistics such as the fitness (measured in hits) at the beginning and end of each epoch, the evolvability E, and the sizes of the various branches in numbers of nodes. All values reported

Population Size	1000
Generations	1000
Function	$y = A\sin(Ax)$
Fitness	sum of absolute value of error for 200 points
Fitness Reported	number of hits (max 200)
Operators	Crossover (80%), Mutation (10%), Reproduction (5%), Best (5%)
Crossover	Branch Typing
Selection	Generational, Tournament, Tournament Size = 7
Non-terminals	$\sin, \cos, \log, \exp, +, -, *, /$
Terminals	Random constant in $[-1,1]$. RPB of ADF: x, ADF0. Monolithic: x.
Architecture	Monolithic vs. one 0-arg ADF
Update of A	Uniform random from $[0,6)$
Epoch Length	$L = 5$ generations (except in Section 4.3)

Table 1: Details of Experiment 1

were measured from the best of generation, and averaged over 100 runs.

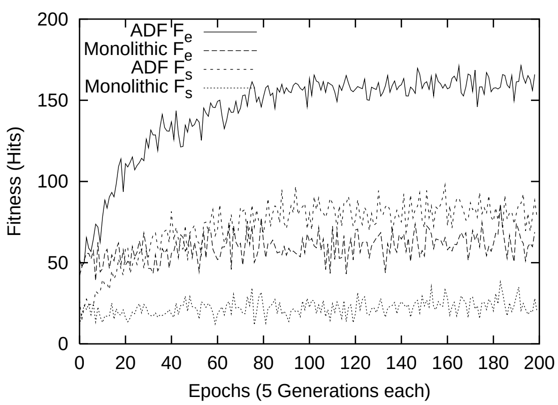

Figure 2: Average fitness values at the start (F_s) and end (F_e) of each epoch when regressing to $y = A\sin(Ax)$. A is selected at the start of each epoch uniformly from the range $[0,6)$.

4.1 ADFS AND EVOLVABILITY

The first experiment asked if ADFs can increase average evolvability of a population compared to their absence. The results can be seen in Figures 2 through 5. The x axis is measured in epochs.

Figure 2 shows fitnesses, while Figure 3 presents the data

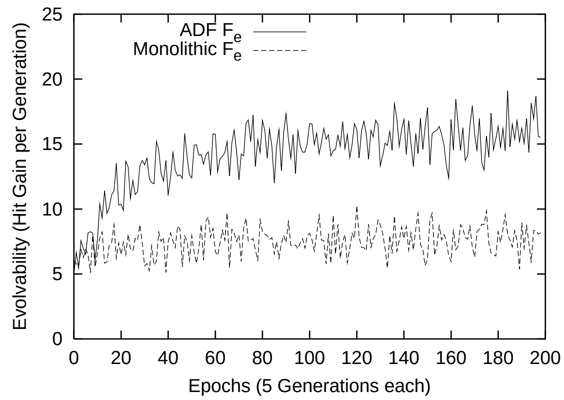

Figure 3: Average evolvabilities for each epoch, regressing to $y = A\sin(Ax)$.

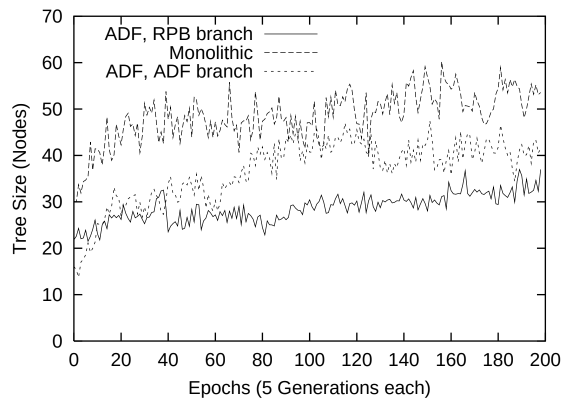

Figure 4: The average sizes of the three major branches. The ADF case shows sizes for both the RPB and the ADF; the monolithic size is the entire tree.

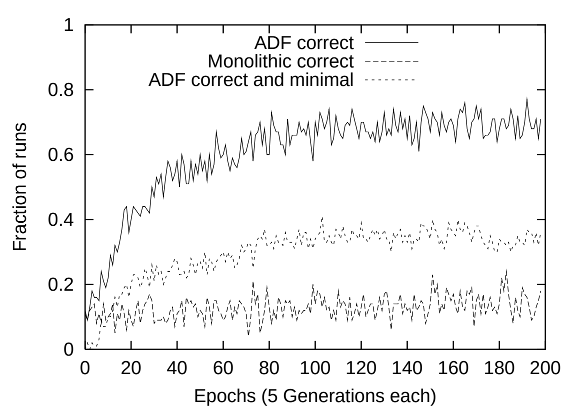

Figure 5: Fraction of runs, at each epoch, that contained correct solutions (hits = 200). Also plotted is the percent of ADF runs that contains a minimal (RPB size 6) correct solution—the tree size of Equation 3.

expressed as evolvabilities. The gap between the start (F_s) and end (F_e) fitnesses increases over time when ADFs are part of the tree architecture, but shows little to no increase when they are not; only when the trees contained ADFs did the best of generation average evolvability increase substantially over time.

An unexpected result visible in Figure 2 is that F_s increased over time when ADFs were part of the tree architecture. This is somewhat surprising, since F_s is calculated immediately after a new random value had been chosen for A, and before any evolution using that value has occurred. The ADF populations not only improved their capacity to track the changing environment, but as time went on, the populations became seeded with good solutions along the dimension of varying A.

Figure 4, showing the average tree sizes over time, suggests that the environmental change also had the effect of curbing any substantial tree size increases, so 'code bloat' (Blickle & Thiele 1994) was not a problem. The average sizes of all branches increased at most slowly, staying below 60 nodes.

A 'poster child' best-of-generation solution in the ADF case, taken from the end of the last epoch of run 10, looks like this:

```
RPB:  (* ADF0 (sin (* ADF0 X))
ADF0: (+ -0.52751 (- 0.03383
          (+ (sin -0.84486) -0.07376)))
```

This is an example of an ideally structured RPB. The calculation of the constant A has been moved to the `ADF0` branch, and the form of the RPB:

$$y = \mathrm{ADF0}() \cdot \sin(\mathrm{ADF0}() \cdot x) \tag{3}$$

matches that of Equation 1. The RPB contains six nodes (two multiplications, two ADF calls, one sin() call and one access to x), which is minimal for a correct general solution, given the set of terminals and non-terminals available. 36% of the runs ended with a solution of 200 hits and an RPB of size 6 at the end of the last epoch. Figure 5 shows that both the percent of correct solutions, and of correct minimal solutions rises substantially over the course of the ADF run, while staying fairly constant in the monolithic case.

Not all minimal correct trees will represent an ideal solution like Equation 3—it is possible to generate a non-general correct solution of size 6. However, manual examination of the last generation of the ADF case indicates that all but one of the runs was an ideal solution, or one of the form

```
RPB:  (/ (sin (/ X ADF0)) ADF0)
```

where the ADF calculates the reciprocal of A. The one exception had had the correct form at the end of the penultimate epoch, but had succumbed to an A-specific approximation during the last epoch.

By way of contrast, we ran 100 runs using a static fitness function, where the value for A never varied from 3 for the entire run. In this case, only 2 runs provided solutions of the ideal form *at any point* in the run.

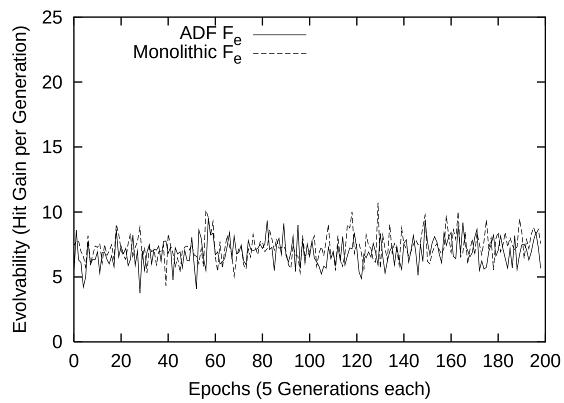

Figure 6: Average evolvabilities per epoch, regressing to $y = A\sin(Bx)$, with A and B varying independently. Compare to Figure 3.

We also performed a control experiment in which all conditions were identical to Experiment 1 except that the function being optimized was $y = A\sin(Bx)$, where A and B varied *independently* at random over precisely the same range. In that case there is no code reuse and thus should be no advantage to factoring the computation, and as Figure 6 shows, the ADF evolvability advantage completely disappeared in this case.

Experiment 1 strongly supports the idea that factoring the repeated and varying portion of the environment into a separate function aided the long-term successful populations by reducing the number of changes necessary to adapt to a new value of A. Though trees arose in ADF populations that did not perform such a factorization and still achieved correct solutions, such non-factored trees were less evolvable than the ones that factored, and so they and their descendents tended eventually to be out-competed.

4.2 EVOLVING CONSTANCY

Experiment 1 disallowed x in the body of the ADF, thus constraining the ADF to only produce constant values. Since that could bias solutions toward the 'more intuitive' factored representation, we also tested architectures without that constancy constraint.

In Experiment 2 the terminal node x was included in the ADF's terminal set, so the only remaining difference be-

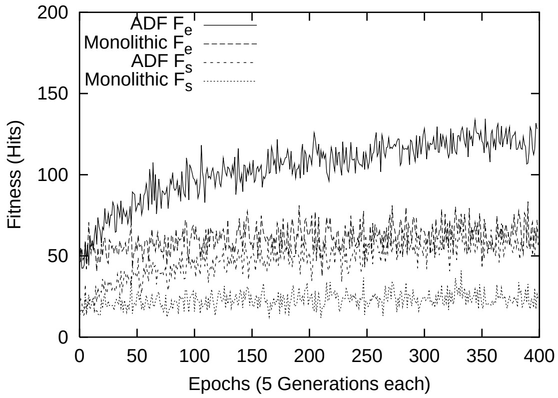

Figure 7: Experiment 2: Average F_s and F_s values when x is allowed in the ADF. The number of epochs is doubled over Experiment 1.

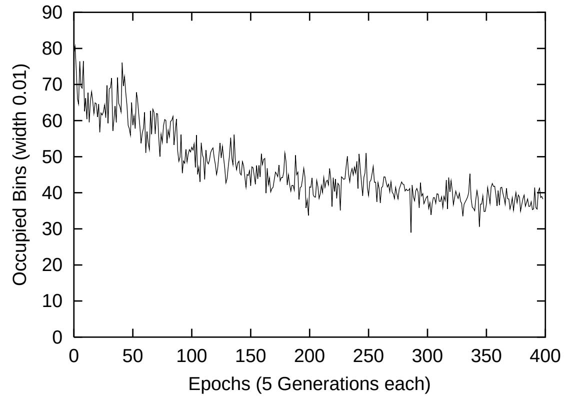

Figure 9: The average number of occupied bins returned by the evolved ADFs over time, given 200 random inputs. Lower numbers (minimum 1) imply 'more constancy'.

tween the RPB and the ADF was that the RPB could call the ADF but not vice-versa. The run length was doubled to 400 epochs over 2000 generations. The parameters were otherwise identical to Experiment 1.

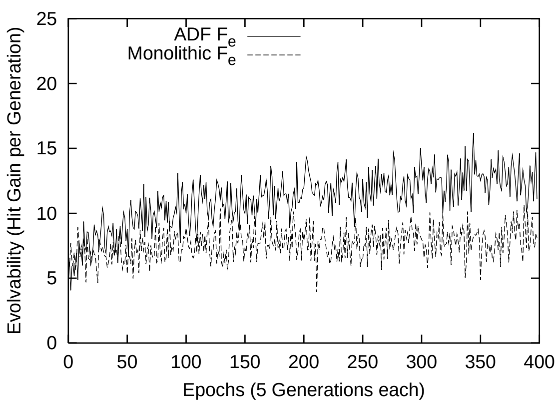

Figure 8: Average evolvabilities over time when x can be used in the ADF. The number of epochs is doubled over the first experiment.

The results can be seen in Figures 7 and 8. Increasing evolvability was still observed, though less pronounced than in Experiment 1. 22% of the runs produced ideal, minimal, correct solutions.

We wondered whether that increasing evolvability corresponded to increasing constancy in the ADF, but determining an ADF's constancy in a satisfying way is somewhat problematic. Simply counting the number of x's in the ADF's code fails because the x nodes may be contained inside of *introns*, non-functional blocks of code such as $0 \times x$, or $x - x$ (Angeline 1994).

Another approach might be to measure the variance of ADF(x) sampled over many values of x, but that tends to be highly sensitive to occasional large-magnitude outliers that skew the averages across runs. The procedure we eventually used is as follows: We called ADF(x) on 200 random points in the range $[-1, 1)$ and quantized the return values into bins of size 0.01, so that results were judged identical if they rounded to the same nearest hundredth. We could then simply count the number of distinct occupied bins as a crude measure of ADF constancy. A completely constant function would produce only one occupied bin.

Figure 9 shows that the average number of ADF occupied bins declined significantly over time, suggesting that the ADFs indeed tended towards constancy.

4.3 VARYING EPOCH LENGTHS

In general, there is every reason to expect evolvability effects to depend on the rate of change of the environment. At one extreme, there is no advantage to maintaining a flexible design for future change if no environmental change is ever forthcoming; at the other extreme if there too much change too frequently then no effective adaptation will be possible at all.

In our model, the length of an epoch provides a natural index of the rate of environmental change. In Experiment 3, the epoch length L was varied over the values L = 1, 2, 5, 10, 20, 50, 100, and 200 generations, resulting in runs that ranged from 5 to 1000 epochs. All other parameters were the same as in Experiment 1. In particular, the ADF was not allowed to use x.

The average over 100 runs of the last-epoch F_s and F_e values can be seen in Figure 10 as a function of L. The largest F_s occurs at $L = 2$, while the maximum F_e occurs at $L = 5$. At present we are unsure why the ADF F_e rises from $L = 50$

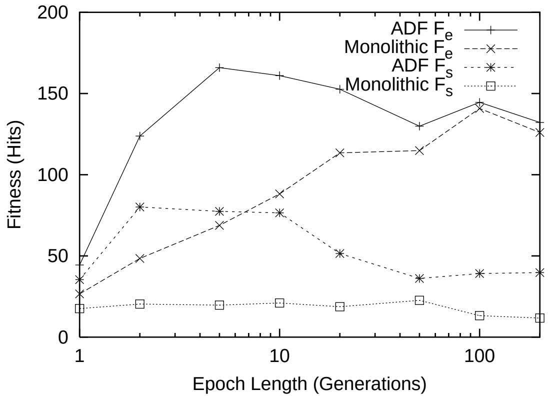

Figure 10: Average starting (F_s) and ending (F_e) fitnesses for the last epoch, for various epoch lengths.

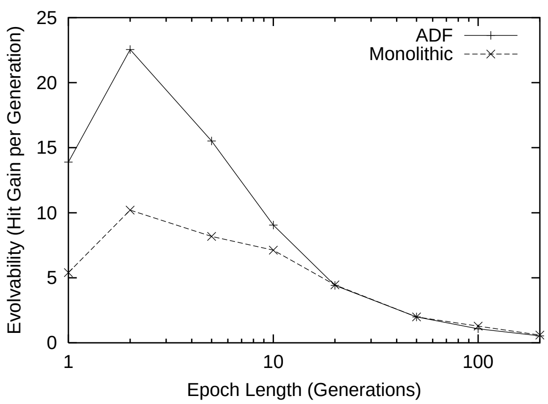

Figure 11: Average evolvability E for the last epoch, for various epoch lengths.

to $L = 100$. One possibility is that $L = 5$ is close to the optimal value when evolvable architectures are the norm, while $L = 100$ is close to optimal when evolvability is not an issue.

As epoch lengths increase, the monolithic F_e increases and F_s declines slowly, in line with expectations that the longer intra-epoch periods allow more evolution but then the inter-epoch changes are more disruptive. In both ADF and monolithic cases, the evolvability peaks at $L = 2$ (Figure 11), suggesting that diminishing returns set in rapidly when measured on a gain-per-generation basis.

Figure 12 shows the number of correct solutions for ADF and monolithic cases, as well as the correct and minimal solutions for the ADF case. The latter value is largest at $L = 5$, and is unaffected by the $L = 100$ resurgence that occurs for F_e and correct solutions, suggesting the $L = 100$ rise may not be due to improved evolvability.

As the epoch lengths increase, the tree sizes in the mono-

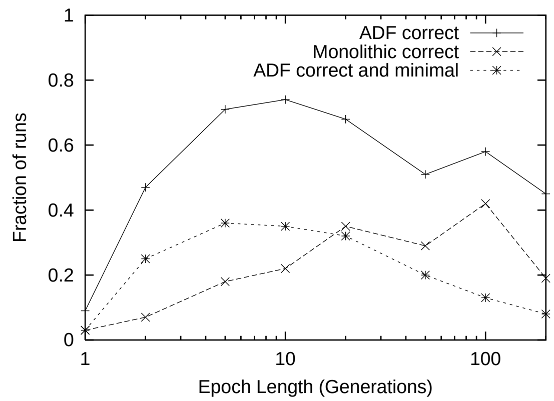

Figure 12: Fraction of ADF and monolithic runs that achieve correct (hits = 200) solutions at the end of the last epoch. Also plotted is the percent of ADF runs that are correct *and* have RPBs with a size of 6.

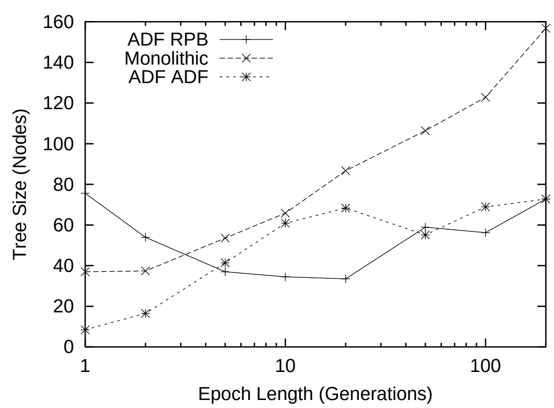

Figure 13: Average tree sizes for the last epoch, for various epoch lengths. Plotted are the size of the RPB in the ADF and monolithic configurations, as well as the ADF size in the ADF configuration.

lithic case increase steadily, but in the ADF case, the RPB average size reaches a minimum at $L = 20$ (Figure 13). The corresponding ADF size seems to vary inversely to that of the RPB; reasons for that effect are presently obscure.

5 CONCLUSION

In this paper we have presented a model, based on genetic programming, which demonstrates the evolution of evolvability when solving a symbolic regression task with a periodically changing fitness function. The successful solutions improved their evolvability by adopting forms that segregated the reused and variable portion of the fitness function (the A parameter), from the unitary and constant portion ($y = \diamond \sin(\diamond x)$). Many intriguing questions are open at this point, from detailed issues of the relative effects of redundancy and variability, to more fundamental goals such as the evolutionary emergence of other software engineering principles, and the scaling up of this research to real world problems.

Well-factored code is not strictly *required* to make a program operate correctly, and bold young programmers often use precisely that argument to resist such basic principles of ‘code hygiene’. We have demonstrated how effective code factorings can emerge from an evolutionary process under a variety of appropriate conditions, even though the fitness function guiding the evolution is—like the novice programmer—focused entirely on the external program behavior, and not at all on its internal structure.

Thus, we establish an experimental link between the evolution of evolvability experiments previously published, and the body of knowledge that forms conventional wisdom about good software design. Though the gap between these two fields is still large, this paper represents a step towards bridging that gap.

Acknowledgments

This research was supported in part by DARPA contract F30602-00-2-0584, and in part by NSF contract ANI 9986555.

References

Altenberg, L. (1994), The evolution of evolvability in genetic programming, *in* K. E. Kinnear, Jr., ed., ‘Advances in Genetic Programming’, MIT Press, pp. 47–74.

Angeline, P. J. (1994), Genetic programming and emergent intelligence, *in* K. E. Kinnear, Jr., ed., ‘Advances in Genetic Programming’, MIT Press, chapter 4, pp. 75–98.

Blickle, T. & Thiele, L. (1994), Genetic programming and redundancy, *in* J. Hopf, ed., ‘Genetic Algorithms Within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)’, Saarbrücken, Germany, pp. 33–38.

Dawkins, R. (1989), The evolution of evolvability, *in* C. G. Langton, ed., ‘Artificial Life: The Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems’, Vol. 6, Addison-Wesley, Redwood, CA, USA, pp. 201–220.

Dawkins, R. (1996), *Climbing Mount Improbable*, W. W. Norton and Company, New York.

Fogel, D. B., Fogel, L. J. & Atmar, J. W. (1991), Meta-evolutionary programming, *in* R. R. Chen, ed., ‘Proceedings of the 25th Asilomar Conference on Signals, Systems, and Computers’, Pacific Grove, CA, pp. 540–545.

Fowler, M., Beck, K., Brant, J., Opdyke, W. & Roberts, D. (1999), *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Boston, MA.

Kirschner, M. & Gerhart, J. (1998), ‘Evolvability’, *Proceedings of the National Academy of Science, USA* **95**, 8420–8427.

Koza, J. (1994), *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press, Cambridge, MA.

Nehaniv, C. L. (2000), Evolvability in biology, artifacts, and software systems, *in* ‘Artificial Life 7 Workshop Proceedings’, pp. 17–21.

Pepper, J. (2000), The evolution of modularity in genome architecture, *in* ‘Artificial Life 7 Workshop Proceedings’, pp. 9–12.

Punch, B. & Goodman, E. (1995), ‘lil-gp1.1 genetic programming system’. *http://garage.cps.msu.edu/software/lil-gp/ lilgp-index.html

Stiemerling, O. & Cremers, A. B. (2000), A paleontological perspective on designing adaptable software, *in* ‘Artificial Life 7 Workshop Proceedings’, pp. 26–29.

Turney, P. D. (1999), Increasing evolvability considered as a large-scale trend in evolution, *in* A. Wu, ed., ‘Proceedings of 1999 Genetic and Evolutionary Computation Conference Workshop Program (GECCO-99 Workshop on Evolvability)’, pp. 43–46.