Genetic Programming with Statically Scoped Local Variables

Evan Kirshenbaum

Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
kirshenbaum@hpl.hp.com

Abstract

This paper presents an extension to genetic programming to allow the evolution of programs containing local variables with static scope which obey the invariant that all variables are bound at time of use. An algorithm is presented for generating trees which obey this invariant, and an extension to the crossover operator is presented which preserves it. New genetic operators are described which abstract sub-expressions to variables and delete variables. Finally, extensions of this work to iteration and functional constructs are discussed.

1 BACKGROUND

The promise of genetic programming (Koza 1990, Banzhaf, et al. 1998) is the automatic discovery of computer programs that solve arbitrary problems. In practice, however, the ingredients of such programs are typically severely restricted. In the original formulation (Koza 1990, 1992) they were limited expressions formed by applying operators drawn from a fixed set to expressions, with each operator being required to be able to take as an argument the value returned by any available operator.

Later extensions brought to the field more of the tools that human programmers rely on. These include strong typing (Montana 1995), user-defined functions (Koza 1994, 1994b), data structures (Langdon 1996), and restricted forms of loops and iterations (Koza, et al. 1999). One very important tool used by human programmers is the notion of the "variable" both as means of naming semantically important values which may be used multiple times and also as a location for recording intermediate results.

The first of these uses does not appear to have received much attention within the genetic programming community (with the exception of zero-argument automatically defined functions.) The second has been addressed largely by the provision of global variables or data structures. In early work (Koza 1992), global scratchpad registers could be written with dedicated storage operators (`SET-X` *val*). This was later extended to allow selection from an arbitrary set of registers, indexed memory (Teller 1994), and finally a generalized "automatically defined store" (Koza, et al. 1999).[1]

These approaches suffer from three drawbacks. First, with the exception of automatically defined stores, the number of such storage locations is typically fixed as a parameter of the run. Second, the program must evolve to ensure that the storage is written to before it is read, and behavior must be defined for situations in which this is not the case. Third, since the only way to store a value is to compute it and write it to the storage location, it is possible for one part of a calculation to destroy a stored value needed by another part. While such cross-talk is at times beneficial (and may be seen as an advantage of evolved programs over designed programs), it can also get in the way.

The canonical solution to this problem in programming languages is the *local variable*. This is a variable whose visibility is bounded to a given scope (e.g., block or expression), and which typically receives its (initial) value at the same time as it becomes available. Local variables are typically introduced by a construct such as Lisp's `LET`:

```
(LET ((R0 (+ X (* Y Y))))
  (* 4 (* R0 R0)))
```

which computes the function $4(x+y^2)^2$. Using `LET`, the variables defined in the bindings are accessible only to the body form, and their definitions may be based on any (and only) variables visible to the `LET` expression itself.

A more radical approach is the dataflow-like Cartesian Genetic Programming described in (Miller 1999). Representing the program as a directed acyclic feed-forward graph (encoded by a linear genome), this formalism has many of the advantages of the statically scoped local variables described here (at arguably considerably less cost), but the choice of representation would appear to preclude taking advantage of types or performing other than constant-time computation.

Another approach that introduces lexically scoped variables are the Automatically Defined Functions of

[1] A different sort of "memory" is presented in (Brave 1996), in which a "mental model" in the form of a graph is built up during one phase of execution and used during a second phase.

(Koza 1992, 1994). In these, the function parameters may only be used within the scope of the function itself. The invariant described in section 2 is maintained under this approach by restricting subtrees selected for crossover to coming from "the same" function or at least functions with identical signatures and therefore identically named and typed parameters.

In this paper, we present a mechanism for evolving programs containing locally scoped, LET-bound variables. We discuss the generation of the initial population, evaluation of trees containing local variables, modifications to standard genetic operators to handle local variable bindings, and new genetic operators that introduce and delete such bindings.

Other operators may also introduce local variables. These include iteration operators, which introduce variables for indices and elements, and function-defining operators such as LAMBDA, which introduce variables for parameters. Iteration operators will be discussed in section 9.1 and function-defining operators will be touched on in section 9.2**.**

The paper is organized as follows. In section 2, we introduce an invariant that needs to be maintained. In sections 3 through 8, we describe algorithms which generate trees which meet this invariant (section 3), evaluate expressions containing local variables (section 4), and preserve the invariant during crossover (section 5) and mutation (section 6), as well as during the application of variable-specific genetic operators (section 7), finishing up with a summary of run parameters introduced (section 8). In section 9 we discuss the use of local variables in iteration and function abstraction, and we finish up with a discussion of our findings (section 10) and conclusions (section 11).

2 PROBLEM

The main problem faced when adding local variables to genetic programming is that they introduce the following invariant that must be maintained:

> **INVARIANT:** any use of a local variable must be properly contained within a form that binds it.

The example from the prior section satisfies the invariant, but the following do not:[2]

```
(+ R0 3)

(LET ((R0 (* R0 7)))
  (+ R0 X))

(LET ((R0 5)
      (R1 (* R0 0)))
  (+ R0 R1))
```

In all of these, the bold-faced R0 is not properly contained within a form that binds it. Note in particular that the variables introduced by the LET operator are only considered to be bound within the final, result-producing subexpression.

The existence of the invariant has two implications for an implementation. First, all initially-generated trees must meet it, and second, it must be preserved by the application of genetic operators. Each of these will be covered in the following sections.

3 GENERATING CORRECTLY SCOPED TREES

When generating trees in strongly-typed genetic programming, one is given a target type *T* and a set of available operators *O*. The traditional algorithm is roughly[3]

1. Select from *O* an operator *Op* whose return type is a subtype of *T*.
2. For each argument of the operator, generate a tree of the appropriate type over the same set of operators *O*.
3. Construct a tree from the selected operator and generated argument trees.

To handle local variables, it helps to generalize the input from a set of operators to a set of operator *schemata*, where each schema can be thought of as a template capable of returning concrete operators when presented with a given context, which consists of a desired return type and a set of available schemata. So, for example, the LET schema represents *any* LET operator and can, for instance generate "the LET operator binding I25 and returning a Boolean". Similarly, a single IF schema can suffice to generate IF operators typed to return any particular type, and the so-called "ephemeral constants" can also be considered as schemata which simply choose a different constant each time they are invoked. All concrete operators, including terminals and local variables, are trivially operator schemata, which return themselves.

For the new algorithm, we assume the following functions:

> For a set *S* of operator schemata, *bound(S)* is the subset of *S* consisting of local variables.
>
> For an operator *Op*, *bindings(Op, i)* is the set of local variables introduced by *Op* that are considered bound in the i^{th} argument form.

bound represents the set of local variables available to us at a given point, which, if the invariant is being maintained will be those which are statically bound at this point in the tree. When generating the topmost tree, there

[2] In this paper, R0, I0, … represent local variables, while X, Y, … represent global quantities such as program parameters.

[3] Ignoring complications such as target depth or bushiness.

should be no local variables in the set of available schemata, so *bound* will be empty.[4]

The algorithm for generating a tree over a set S of operator schemata returning a type T then becomes

1a. Select from S an operator schema capable of returning an operator whose return type is a subtype of T.

1b. Use the schema to obtain an operator Op given S and T.

2. For each argument i of Op, generate a tree of the appropriate type, given $S \cup bindings(Op, i)$.

3. Construct a tree from the selected operator and generated argument trees.

If the local variables are the only schemata that can return local variables when invoked, it is clearly true that the only way a local variable v can be introduced as the operator of a node in the tree is if the node is a descendent of the j^{th} child of a node with operator O, where $v \in bindings(O, j)$.

For the specific case of the `LET`-constructing schema, every time it is invoked with a context of S and T, it

1. Determines a number n of bindings to introduce,

2. Creates an n-element set of types for which trees can be constructed over S,

3. For each type, obtains a local variable over that type, and

4. Constructs an $n+1$-argument `LET` operator, for which *bindings(i)* is the set of introduced variables for $i = n$ and is empty for all other i. The operator's first n arguments are of the selected types, and its last argument is of type T.

The careful reader will have noticed that three of the four steps in the above algorithm contain rather vague phrases. This is because it is not yet apparent to us what the "correct" approach is. We now briefly discuss possible approaches to each.

3.1 CHOOSING THE NUMBER OF VARIABLES TO INTRODUCE

The first decision to make is the number of variables that the `LET` form will bind. The most straightforward answer is to say that each `LET` binds a single variable. Since the framework on which this was implemented generates depth-bounded trees, it was thought that doing so would prove too limiting in, on the one hand, the number of variables that could be introduced over a given scope and, on the other hand, the size of the resulting scope.

Another option would be to choose uniformly between one and some maximum specified as a run parameter. Rather than use a hard maximum, we instead decided to use an exponential fall-off, repeatedly adding variables to the set with a run-parameter-specified probability.

In early runs it became apparent that this approach resulted in most of the variables introduced being very near the leaves of the tree—since in any bushy tree *most* nodes are near the leaves—resulting in very narrow scope. In an attempt to push the variable bindings further up the tree, we introduced a discount factor of $1/2^{depth}$ on the probability, where the root of the tree is at $depth = 0$. To make it more likely that local variables would be created even when the set of available operators was large, the `LET` schema itself is chosen with this probability. (In retrospect, this may have been an unnecessary and perhaps even detrimental "optimization", as the discount factor quickly makes it *less* likely to introduce local variables as one goes down the tree.)

3.2 CHOOSING THE VARIABLE TYPES

Once the number of variables has been chosen, the next question to be answered is how to choose the types of the variables. For simplicity, in our prototype, we made the set of allowed local variable types a run parameter, but it would be reasonable to automatically determine it based on the sets of types that can be returned by and accepted as arguments by the operators generable by the available schemata.

3.3 OBTAINING THE VARIABLES

Finally, given a set of types one must actually choose the variables. The first method we attempted for the prototype kept track of all variables created for each type, indexed by their order of creation, and allocated the lowest numbered variable not in *bound(S)* where S is the set of schemata passed in to the `LET` schema, subject to the restriction that no variable may be used more than once in a given `LET`.

This resulted in the outermost real-valued variable(s) in every tree being `R0`, the next being `R1`, etc. After running this way for a while, we began to suspect that this resulted in competition among subpopulations over what the lowest numbered variables were supposed to *mean*. We therefore switched to a scheme in which variables are selected uniformly from a set whose size is specified as a run parameter, with the question of whether a variable may be rebound within a given scope (in the initial population) also controlled by a run parameter. More testing needs to be done to determine whether one method is better than the other.

The variables themselves are operators of type *MemCell*<*T*>, which is a proper subtype of both T and *Lvalue*<*T*>.[5] The assignment operator `:=` takes an *Lvalue*<*T*> and modifier operators, such as `++` and `*=`, take *MemCell*<*T*>s, so these variables are available for

[4] Alternatively, the program inputs can be treated as implicitly bound local variables.

[5] The GPLab framework used to implement the prototype supports contravariant lvalue types, so this is also a subtype of *Lvalue*<*U*> for any subtype U of T.

modification when such operators are thrown into the mix. It would be possible at this point to distinguish between normal variables, as described, and read-only variables of type *T*, perhaps controlled by a run parameter specifying the probability of each being chosen.

4 EVALUATING EXPRESSIONS

4.1 BASIC EXECUTION

In the absence of locally defined functions (section 9.2), each local variable can be thought of as maintaining a stack of values of the appropriate type. When a LET form is entered, each variable definition argument is evaluated, and then each value is pushed onto the stack of the appropriate variable.[6] Then the body form is evaluated, and all of the bound variables have their stacks popped. When a local variable is evaluated, the value at the top of its stack is returned.

(Strictly speaking, this implements *dynamic scoping* rather than *static scoping*, and so when locally defined functions are added, a free variable within a function will get the value most recently associated with that named variable rather than the value of the lexically enclosing binding. The variable evaluation mechanism will have to be changed to support true static scoping.)

4.2 OPTIMIZATIONS

When running with local variables, it quickly becomes evident that the system spends a good deal of its time evaluating the definitions of variables that are never used in their scope. In the absence of operators which produce side-effects, a simple optimization can recover nearly all of this time.

The first step is to compute, for each node *n* in a tree, *vars-needed(n)*, which is the set of local variables whose bindings are actually needed in order to compute *n*'s value. To compute this set, we first assume the existence of two functions:

> *free-vars(Op)* computes the set of local variables that the operator itself requires to be bound. For most operators, this is the empty set. For local variables, it is the singleton set containing the operator itself.
>
> *bindings(Op, i)* computes the bindings introduced by *Op* that cover its i^{th} argument. For most operators, this is the empty set for all *i*. For LET, it is the empty set for all arguments but the last, for which it is the set of variables bound.

vars-needed(n) can then be computed as follows:

1. Start with *free-vars(Op).*
2. For each argument *i*, add to the set the set difference between *vars-needed(argument(n, i))* and *bindings(Op, i)*.

For LET forms, each this is process is modified to only add *vars-needed(argument(n, i))* if the i^{th} variable is in *vars-needed(body-arg)*.

When evaluating a LET, a defining form is evaluated and a variable bound only if the corresponding variable is contained within *vars-needed(body-arg)*.

This scheme must be elaborated a bit when side-effect–producing operations are present, as it may be necessary to evaluate a variable definition for its side-effects even though the value will be unused. The above algorithm can be straightforwardly extended to consider two sets associated with each node: the variables needed for evaluation and the variables needed for side-effect, along with a flag indicating whether the node should be evaluated for side-effect. In such a case, a definition is evaluated only for side-effects if its binding is unneeded by the body form, but the definition contains a side-effect.

It should be apparent that if these sets were computed at run-time the expense would swamp any savings, so they should be computed at tree-construction time and cached in the node itself. This may seem expensive, but the expense is mitigated by the fact that most of these sets are (1) empty, (2) singletons, which can be represented by the variables themselves, or (3) identical between parent and one or more children, so a considerable degree of sharing is possible.

5 INVARIANT-PRESERVING CROSSOVER

Once invariant-observing trees have been generated and evaluated, the next step is to reproduce them using genetic operators modified to preserve the invariant. The most important such operator is the crossover operator, which takes a subtree from one tree (the "father") and uses it to replace a subtree from another tree (the "mother").

We assume extensions of the *free-vars* and *bindings* functions from operators to nodes, with *free-vars* being defined (without exception) as is the default definition of *vars-needed* above and *bindings* being the union of the parent node's *bindings* for the argument index of the node in question and the node's operator's *bindings* for the argument index in question.

The obvious invariant-preserving modification to crossover would simply be to declare two fragments to be incompatible if the *free-vars* of the father's subtree was not a subset of the *bindings* of the argument slot covering the mother's subtree. While this would work, it would also rule out a large number of crossover points and was felt to be too restrictive and unlikely to allow the merging of two "almost good" programs.

What we do instead is to declare that *any* two otherwise-compatible crossover points remain compatible, with the

[6] It is important to do this in two passes, or the definitions for the later-bound variables will be incorrectly evaluated in the context of the earlier bindings, yielding, in Lisp terminology, LET* rather than LET.

resulting tree taking definitional material from father's tree as needed to produce a correctly scoped child. The basic notion is that if the mother does not have semantics for a particular local variable, those of the father are used as if they were part of the father's fragment. Rather than simply extending the father's fragment, however, the definitions are distributed along the spine of (the copy of) the mother's tree from the root to the crossover point, resulting in a child whose bindings are a blend of those of the two parents.

5.1 CROSSOVER EXAMPLE

To illustrate the invariant-preserving crossover, we will consider the father to be

```
(+ …
   (LET ((R17 (* X Y))
         R2 X))
     (* 5
        (LET ((R4 Z)
              (R9 -2)
              (R2 (* 3 R17)))
          (- 5
             (* X
               (LET ((R1 R2))
                 (+ (- R2 R17) R9))
              ))))))
```

with the fragment itself highlighted, and the mother to be

```
(LET ((R2 …)
      (R7 …))
  (* … (LET ((R6 …))
         (- Y 2))))
```

The free variables of the father's subtree are R2, R9, and R17, and the bindings enclosing the mother's subtree are R2, R6, and R7.

5.2 FATHER'S BINDINGS

The first step is to collect a list of the bindings that are in effect at the father's subtree. This can be done by walking down the spine of the tree and storing in an array <*variable*, *definition*> pairs for each binding introduced, keeping track of where each level's definition starts. For the example, this results in the following list:

R17	(* X Y)
R2	X
R4	Z
R9	-2
R2	(* 3 R17)

Note that R2 is in the list twice (once for each binding) and that R1 is not in the list at all, since its definition is properly contained within the fragment.

5.3 COUNTING BINDINGS

The next step is to start with the list of free variables in the father's subtree and increment a counter associated with the lowest corresponding row of the table. Whenever a counter for a row is incremented, this step is repeated recursively with the list of free variables from that row's definition, considering only the table above that row. When this step is completed, the table looks like

R17	(* X Y)	2
R2	X	0
R4	Z	0
R9	-2	1
R2	(* 3 R17)	1

R17 has two uses (one from the fragment and one from the second definition of R2), R9 and the second definition of R2 each have one (from the fragment), and the other two definitions have none. Only those rows of the table with non-zero counts will be considered for incorporation.

The table also contains a variable identifying the last unprocessed row.

5.4 MERGING

Five functions are then used to effect the merge:

copy-patched walks down the mother's spine copying nodes until it reaches the crossover point, at which point it inserts the father's fragment.

capture-vars looks to see whether variables in the father's tree are (or should be made to be) available in the mother's scope.

replace replaces all free occurrences of one variable by another in a tree.

remove-binding recursively decrements the counter associated with variables free in unneeded definitions.

wrap-node probabilistically wraps a form in a LET form with bindings taken from the binding list.

5.4.1 copy-patched

copy-patched is called initially with the mother's tree, the father's subtree (as the *fragment* to be inserted), the list of bindings, a depth of zero, and information needed to determine the path from the root of the mother's tree to the insertion point (or "hole").

When at the insertion point itself, *copy-patched* calls *capture-vars* on the fragment, and passes the result to *wrap-node*, whose value it returns.

When above the insertion point, *copy-patched* duplicates the current node, calls itself recursively on the next child down in the path (with a depth one greater), patches the

result in as the appropriate argument to the constructed node, and passes the resulting node to *wrap-node*, whose value it returns.

5.4.2 capture-vars

capture-vars walks through the list of free variables associated with a node. For each variable, it checks to see whether the variable is already bound at the current point on the mother's spine. If not, it probabilistically checks to see whether it should replace the variable with some variable that is bound at the current point on the mother's spine. If either condition is met, *remove-binding* is called to update the table to reflect that a binding is no longer needed. If a replacement is mandated, *replace* is called with a randomly selected variable in scope.

The probability of replacement is a run parameter.

In the example R2 from the father's fragment is already bound by the mother, but R9 and R17 are not. They could, however, be replaced by the mother's R2, R6 or R7.

5.4.3 replace

replace simply recursively walks a tree, diving into any subtree which has its target variable free and replacing all occurrences of the target by the replacement.

5.4.4 remove-binding

remove-binding is passed in a variable and a row in the binding list. It searches for the row of the list at or above its parameter row which matches its parameter variable and decrements the count for that variable. If the count becomes zero (signifying that there is no further need to consider copying this binding), *remove-binding* calls itself recursively on all free variables in that row's definition, with a row parameter pointing just above the next highest level boundary.

After calling *remove-binding* for R2, the table now looks like

R17	(* X Y)	1
R2	X	0
R4	Z	0
R9	-2	1
R2	(* 3 R17)	0

The count for R17 has been decreased by one, but there is still one reference from the father's fragment itself.

5.4.5 wrap-node

wrap-node collects zero or more bindings from the end of the binding list and creates a LET form wrapping its tree parameter with the collected bindings. (If zero bindings are collected, the tree parameter is simply returned.)

To decide whether to collect a binding, the binding list is walked in reverse order, starting from the list's (current) last row. If the row under consideration has a count of zero, the binding is unneeded, and the row is skipped. If the count is non-zero, it is chosen for inclusion with a probability $1/(depth+1)$, where *depth* is the current depth in the tree being constructed. This probability was chosen for two reasons. First, when there is a single binding, it makes the probability of the binding being inserted at any level equal, and second, it makes the probability one at top level, ensuring that all needed bindings eventually get copied.

The algorithm as described needs a minor modification to ensure that a let form is not constructed which tries to combine two variables, the definition of the second of which contains the first as a free variable. To prevent that from happening, the set of free variables in all collected definitions is kept. Whenever the decision is made to collect a variable which is in that set, the LET form is constructed with the current collected set, and the result is passed recursively to *wrap-node*, with the same depth. This results in two LET expressions, one wrapping the other.

5.5 EXAMPLE RESULTS

Given the example father and mother trees shown above, the following are all valid results of crossover, with the father's contribution highlighted:

```
(LET ((R2 …)(R7…))
  (LET ((R17 (* X Y)))
    (* … (LET ((R9 -2))
           (LET ((R6 …))
              (LET ((R1 R2))
                (+ (- R2 R17) R9))
            )))))

(LET ((R2 …) (R7 …))
  (LET ((R17 (* X Y))
        (R9 -2))
    (* … (LET ((R6 …))
           (LET ((R1 R2))
             (+ (- R2 R17) R9))
         ))))

(LET ((R9 -2))
  (LET ((R2 …) (R7 …))
    (* … (LET ((R6 …))
           (LET ((R1 R2))
             (+ (- R2 R2) R9))
        ))))
```

Note that in each of these the free variable R2 has been captured by the binding on the mother's side. In the second example, the bindings for R17 and R9 have been lumped together into a single LET form, even though they were widely spaced in the father. And in the final example, the binding for R17 has disappeared, with the free instance of that variable replaced by a variable bound on the mother's side (in this case, R2).

6 INVARIANT-PRESERVING MUTATION

Since mutation in genetic programming is typically implemented by choosing a node within a tree and growing a new tree to replace it, the algorithm for growing the initial trees will suffice to preserve the invariant. To allow local variables to occur in the new portion, it is simply necessary to ensure that all variables bound at the chosen point are included in the set of available operator schemata.

For "point mutation", in which a random node is chosen and its operator alone is replaced by a randomly chosen operator having a compatible signature, there is again no problem, with the caveat that local variables (of the appropriate type) bound above the chosen node should be considered when the node is a leaf.

7 VARIABLE-SPECIFIC GENETIC OPERATORS

In addition to the normal genetic operators, there are several that deal directly with local variables.

7.1 LET ABSTRACTION

In *LET abstraction*, a node is selected randomly from a tree and that node is replaced by a local variable, with the tree rooted at the original node serving as the definition for the binding of the variable, this definition being distributed somewhere up the tree.

As an example, if the original tree is

```
(LET ((R0 5))
  (* … (LET ((R7 …))
         (+ Y (- X R0)))))
```

with the chosen subtree highlighted, possible results of LET abstraction include

```
(LET ((R0 5))
  (LET ((R42 (- X R0)))
    (* … (LET ((R7 …))
           (+ Y R42)))))

(LET ((R0 5))
  (* … (LET ((R15 (- X R0)))
         (LET ((R7 …))
           (+ Y R15)))))

(LET ((R0 5))
  (* … (LET ((R7 …))
         (+ Y (LET ((R1 (- X R0)))
                R1)))))
```

A straightforward way to implement LET abstraction is as a "self crossover" with the chosen subtree *Foo* replaced by `(LET ((`*var Foo*`)) `*var*`)`, as in the last enumerated possible result. With the crossover algorithm described in section 5, the "replacement" may be merely conceptual, as it suffices to simply add a new row to the binding list and treat the variable itself as the inserted fragment.

One consequence of treating LET abstraction as crossover is that the variable introduced may safely migrate up the tree past free variables contained in its definition, pulling anything it needs up with it. Thus

```
(LET ((R0 5))
  (LET ((R19 (- X R0)))
    (LET ((R0 5))
      (* (LET ((R7 …))
           (+ Y R19))))))
```

is also a valid LET abstraction.

Note that the results of LET abstraction are almost always, but not always, value preserving. The value can be different if there are operators that produce side effects, because the order of evaluation is changed, and it is also possible to abstract expressions out of never-evaluated or conditionally-evaluated branches. It is also possible that if the introduced binding is raised above a free variable used in the definition, the variable may be bound in the outer scope to a different value.

7.2 VARIABLE DELETION

The flip side of LET abstraction is *variable deletion*. This operator chooses a random node in the tree and replaces within it a randomly chosen free variable with the variable's definition. An alternative form of this operator limits itself to LET expressions and completely removes one of the variables bound at the chosen one.

7.3 VARIABLE CAPTURE

One of the drawbacks with the LET abstraction operator is that the introduced variable is only used a single time. The *variable capture* operator introduces (potentially) extra uses of variables. It chooses a random node and replaces it by a variable bound at the node. This is, needless to say, not value-preserving.

7.4 UNUSED VARIABLE DELETION

As discussed previously, one of the consequences of introducing local variables is that many variables are defined but never used. While the performance lost due to evaluating unneeded definitions can be mostly recovered by the optimization outlined above, another consequence is that a great deal of the crossovers take place between subtrees that are never evaluated and do not contribute to the program's fitness.

The presence of such code, roughly analogous to the unexpressed *introns* or *junk DNA* of animal biology, has its good points and its bad points. On the plus side, it can be a storehouse of genetic variability, lessening or removing the need for mutation. Also, when the fitness landscape is subject to change, it can increase the gene pool's adaptability by keeping around solutions that "used to be needed". When its organism is relatively good, its sheer size can make it less likely that the organism will be harmed by a mutation or crossover. On the minus side, however, it also has the effect of slowing the evolutionary

rate by making many genetic operations non-functional. This is precisely what is needed (in nature, at least) when a population is very close to the local optimum, but it can impose a large performance penalty when searching for a solution from an initially-random population. The benefit or penalty of such code is a topic currently under much debate and investigation. (Soule, et al. 1996, Nordin and Banzhaf 1995)

The *unused variable deletion* genetic operator attempts to work against this trend by selecting a `LET` form from within a tree and snipping out one or more variables which are unused in the body (as asserted by the *vars-needed* function). If all of the variables are removed the `LET` form itself may be replaced by its body.

8 SUMMARY OF PARAMETERS

As described above, a run of a genetic programming experiment in the presence of statically scoped local variables is parameterized by several factors:

- the probability that a local variable is added to the tree during generation,
- the permitted types of local variables for generated trees,
- the initial number of local variables per type,
- whether "shadowing" bindings should be allowed in generated programs,
- the probability that a free variable in the father's contribution is replaced by a bound variable from the mother's contribution, and
- the probabilities of applying the genetic operators `LET` abstraction, variable deletion, variable capture, and unused variable deletion.

9 OTHER OPERATORS

Perhaps the most important benefit of the addition of statically scoped local variables is that it makes it straightforward to add a raft of new operators corresponding to more sophisticated control structures and higher-level functions. In this section we will briefly discuss bounded iteration constructs, which we have prototyped, and locally defined functions, which we have not.

9.1 BOUNDED ITERATION

One of the primary limitations of genetic programming, as typically implemented, is that the evolved programs can only implement constant-time algorithms. This is due to the fact that the operators implemented do not allow either recursion or iteration. For problems that require iteration for their solutions, the experimenter must set up the experiment harness to repeatedly invoke the program and combine the results in some ad hoc manner.

Human programmers, by contrast, have access to a wide array of iteration constructs, which can be invoked at any time and which can nest arbitrarily. It would be nice to be able to take advantage of them in evolved programs.

Unbounded iteration constructs, such as C++'s `while` are problematic because poor programs which use them can easily fail to terminate. Much of the use of iteration by humans, however, involves iterating over data structures such as a strings, vectors, or lists or iterating over numbers between a minimum and a maximum. Such *bounded* iteration constructs are more well-defined. Unfortunately, other than a simple "repeat *n* times" operator, they are difficult to implement in standard genetic programming, as each iteration requires access to the current element of the data structure or the current value of the index.

With local variables, such operators are simple to define, from a general `(for (I17` *`min max`*`)` *`body`*`)`, in which `I17` is available within *`body`*, to forms that iterate over specific data structures. Useful examples of the latter include

```
(foreach (v vector) body)
```

which sets *`v`* to successive elements of the value returned by (the vector-returning expression) *`vector`* while executing *`body`*,

```
(sum (v vector) body)
(product (v vector) body)
(map (v vector) body)
(select (v vector) pred)
```

which do the same thing, but return as values the sum or product of the values returned by *`body`*, combine the values into a new vector, or select those elements satisfying a predicate, and the general

```
(accumulate ((v1 vector) body)
            (((v2 init) v3) comb))
```

which binds *`v1`* within *body* for each element and combines the result (as *`v3`*) and the previous result (as *`v2`*) by evaluating *`comb`*(inator).

The self-contained nature of such forms allows for their easy reuse, and early results of experiments indicate that they make the use of variable-sized data structures quite tractable.

As an example, a symbolic regression from a vector of real numbers to the mean of the squares of the values turned out to be trivial given (in addition to the vector itself, the canonical arithmetic operations and integer constants between –10 and 10), a `sum` operator that mapped each element of a vector to a real number and summed the result. Running on a population of 5,000 for a maximum of 50 generations over only ten training cases, validated solutions were found on 21 of 30 runs. Surprisingly, half (11) of the solutions were found within the first 4 generations, including three in the initial random generation. The expected effort for this problem to reach 99% confidence is 210,000 evaluations (21 runs

to generation 1). An example of a correct solution to this problem guessed in the initial generation is

```
(/ (+ (sum (R6 v)(- R6 R6))
      (sum (R5 v)(* R5 R5)))
   (sum (R9 v)(+ (- R9 R9)
                 (/ R9 R9))))
```

This program walks the input vector three times. In the numerator, it is walked twice, once to compute a constant zero and again to compute the sum of squares. In the denominator, it is walked once, its body form returning a constant one, which when summed yields the length of the vector.

There are two implementation considerations when adding support for such iteration constructs. The first relates to the handling of such forms during crossover (section 5), since some of the variables have no overt definition. When an expession is abstracted out of the body of an iterator, these variables may be free. In our prototype implementation, when a LET binding is introduced for such a variable, we simply grow a new definition (in the scope of the prevailing bindings) by mutation.

The second consideration is that even though bounded iterations are guaranteed to terminate, since they may nest arbitrarily, they may take an unacceptably long time to do so. To get around this, we implemented a *computation budget*, specifying the maximum number of operations a program could execute on a given case before being declared to have infinitely bad fitness.[7] Clearly, such a decision is an indication of little more than human impatience, although it may well be worthwhile when using loops to consider the number of operations evaluated in the fitness measure. One obvious optimization is to note that, in the absence of side-effects, any body form in which none of the loop variables are free must return the same value for the length of the loop and so need only be evaluated once.

Aside from repeatedly calling a candidate program from within an experiment harness, Koza, et al. (1999) introduced the notions of *automatically defined loops* and *automatically defined iterations*, as named constructs by analogy with automatically defined functions. To get around the complexity issues discussed above, these are each evaluated a single time, before the main, result-producing form is evaluated. In the case of iterations, the iteration is over a fixed, experiment-specific data structure. Any result computed must be stashed away in a global store. Because of the way these are defined, it appears to be impossible to evolve programs that require nested loops.

9.2 LOCALLY DEFINED FUNCTIONS

Another intriguing idea, which we have not yet explored, is to use a process similar to that described in (Koza 1994b) to extract a LAMBDA expression, which can be named by a LET binding and replaced by a call to the named variable. The variables within the LAMBDA form become local variables whose scope extends over the body of the form. Since function calls would be on named functions,[8] the mechanism described in the paper would serve to prevent (or explicitly allow) recursion, as a function could only name those other functions bound in the scope of its definition. Another benefit is that it will be straightforward for functions to share variables simply by the functions' being bound within the scope of the shared variables. A final advantage is that such named functions can easily be passed in as parameters to other functions, allowing the evolution of higher-order functions and operators such as REDUCE.

Of course, the addition of functional values (which has some support in GPLab already) raises the issue of what happens if a functional value is returned to a scope in which the variables it references are not bound. While this can easily be prevented, the solution—full closures—may be worth investigating but will necessitate changes in the way local variables are implemented.

10 DISCUSSION

The mechanisms described to support local variables have been added to GPLab, a flexible framework for genetic programming developed and used for data mining research at Hewlett-Packard Laboratories. We are currently in the process of identifying and evaluating classes of problems in which such a capability makes a significant improvement in the likelihood of finding a solution.

The expectation is that the addition of local variables will, like the addition of automatically defined functions, greatly simplify the discovery of solutions to problems in which it is profitable to make use of common subexpressions as well as those for which solutions can profit from the presence of scratchpad variables.

The added power will, however, be pitted against several factors which may serve to *increase* the effort required to find a solution. As discussed in section 7.4, the presence of unused bindings acts as a brake on the rate of evolution by encouraging crossover in sections of the program which do not affect fitness. Also, the addition of genetic operators such as LET abstraction and variable deletion, which are meaning preserving, can also be expected to retard the pace of evolution by creating fewer completely new children in each generation. Finally, of course, the mere addition of operators into the tableau increases the search space.

[7] The experiment described used a computation budget of 500 operations for vectors containing between five and twenty elements.

[8] Or statically nested LAMBDA expressions.

Very early runs of simple symbolic regression problems indicate that for problems that can tractably be solved *without* local variables, the mere addition of LET-bound local variables does not help and can indeed increase the required effort. (For problems which cannot tractably be solved either with or without local variables, it is difficult to say whether they help or hurt.)

On the other hand, as discussed in section 9.1, the use of local variables within iteration operators appears to be quite powerful.

One thing that has become apparent during this early testing phase—although we have not yet quantified it—is that there does not appear to be sufficient pressure to encourage bound variables to be used multiple times within their scope. With the exception of variables introduced by operators such as iterators, a local variable is truly useful only if it is used more than once. Further experimentation will be necessary to determine whether this is a property of the parameters so far explored or whether some new genetic operator will need to be added to further encourage variable reuse.

Finally, further work needs to be done to compare the use of local variables and (a sufficient number of) global variables. Each would seem to have advantages and disadvantages. The main advantage of global variables is that the state may be preserved between evaluations, allowing them to form a persistent memory. The apparent advantages of local variables over global variables include

- The fact that one definition can shadow another means that a local computation can usurp a variable without destroying its value,
- Unlike global variables, they can be profitably used even in the absence of side-effecting operators, and
- The number and types of variables needed can be determined by the evolutionary process.

The magnitude of the benefit (or penalty) still needs to be quantified. Note that data-structure–specific operators, such as those discussed in section 9.1, could make use of global, rather than local variables.

11 CONCLUSIONS

In this paper, we have shown that it is possible to evolve programs which use correctly scoped local variants. We have enunciated an invariant which needs to be maintained and outlined mechanisms for generating invariant-satisfying programs, evaluating programs, and maintaining the invariant over genetic operators. We have also presented several new genetic operators which add and remove local variables and discussed the use of local variables for purposes of iteration and functional abstraction.

While it appears intuitively obvious that such a facility, ubiquitous as it is in programming languages, should be part of any evolutionary arsenal, we have unfortunately not yet been able to demonstrate quantitatively the usefulness of this facility over any particular class of problem, although the ability to use them with bounded iteration operators appears to be extremely promising.

References

Banzhaf, Wolfgang; Nordin, Peter; Keller, Robert E.; and Francone, Frank D. 1998. *Genetic Programming: An Introduction*. San Francisco, CA: Morgan Kaufmann.

Brave, Scott 1996. The evolution of memory and mental models using genetic programming. In John R. Koza, et al., (eds.) *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28–31, 1996, Stanford University*. Cambridge, MA: MIT Press, pp. 261–266.

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. 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. 1994b. *Architecture-Altering Operations for Evolving the Architecture of a Multi-Part Program in Genetic Programming*. Stanford University Computer Science Department technical report STAN-TR-CS-94-1528. October 21, 1994

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

Langdon, W. B. 1996, Using data structures within genetic programming. In John R. Koza, et al., (eds.) *Genetic Programming 1996: Proc. of the First Annual Conference, July 28–31, 1996, Stanford University*. Cambridge, MA: MIT Press, pp. 141–149.

Montana, David J. 1995. Strongly typed genetic programming. *Evolutionary Computation* 3(2):199–230.

Nordin, Peter and Banzhaf, Wolfgang 1995. Complexity compression and evolution. In Eshelman, L.J. (ed.) *Proceedings of the Sixth International Conference on Genetic Algorithms*. San Francisco, CA: Morgan Kaufmann, pp. 310–317.

Soule, Terence; Foster, James A.; and Dickinson, John 1996. Code growth in genetic programming. In John R. Koza, et al., (eds.) *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28–31, 1996, Stanford University*. Cambridge, MA: MIT Press, pp. 215–223.

Teller, Astro 1994. The evolution of mental models. In Kinnear, Kenneth E., Jr. (ed.), *Advances in Genetic Programming*, Cambridge, MA: MIT Press, pp. 199–220.