Impact of types on essentially typeless problems in GP

Nicholas Freitag McPhee **Nicholas J. Hopper**
Mitchell L. Reierson
Division of Science and Mathematics
University of Minnesota, Morris
Morris, MN USA-56267
(320) 589-6300
{mcphee, hoppernj, reiersml}@mrs.umn.edu
http://www.mrs.umn.edu/{~mcphee, ~hoppernj}

ABSTRACT

Several researchers have shown type systems to be valuable in extending the range of problems (conveniently) addressed by Genetic Programming. There are other possible benefits of type systems, however, that derive from the new kinds of structural representation they make possible, and the effects that this has on the performance of recombination operators like the crossover operator. In this paper we compare the performance of Standard (untyped) Genetic Programming (SGP) and Hindley-Milner (typed) Genetic Programming (HMGP) on a suite of problems where an untyped representation (satisfying the closure property) is quite natural. We find that on several problems HMGP significantly outperforms SGP, while on other problems the performance of SGP and HMGP are essentially the same. We also suggest an intermediate representation that should provide many of the benefits of HMGP on these problems without requiring the complexity of a powerful type system.

1 Introduction

1.1 History and background

All versions of Genetic Programming (GP) involve modifying portions of expressions (often represented by their parse trees) in hopes of yielding more fit individuals. Most of these involve replacing some subtree of a parent with a new subtree (e.g., taken from another parent in crossover, or generated anew in mutation). In order to prevent spending large amounts of time creating and processing nonsense trees (e.g., a tree that adds a *boolean* and a *real*), some sort of mechanism is useful to ensure that the new subtree "makes sense" in its new position.

In the version of Genetic Programming presented in [Koza 1992], which we will refer to as Standard Genetic Programming (SGP), Koza requires that all trees and subtrees have the same type (the "closure property"). The closure property is one simple and effective way of ensuring that crossover and mutation yield valid trees. The downside is that many problems don't easily conform to the closure property. While types aren't strictly necessary (many typeless systems are Turing complete), many problems are more naturally represented with types.

Several researchers have explored alternatives to the closure property where a type system is used to constrain operators like crossover so they don't create "meaningless" trees. Major examples include the work of Montana [Montana 1995], Haynes [Haynes *et al.* 1995], and Yu [Clack and Yu 1997]; Koza's work on syntactically constrained GP [Koza 1992], Langdon's work on evolving Abstract Data Types [Langdon 1995], and Wong and Leung's work on Inductive Logic Programming [Wong and Leung 1995] can also be seen as examples of constraining the manipulation of trees through types or type-like mechanisms. Most of this research has focussed on demonstrating the ability of typed GP to evolve solutions to problems that are not (easily) represented in SGP due to the closure property. [Montana 1995], for example, shows that Montana's Strongly Typed GP can be used to solve several interesting problems on matrices and lists, all of which would be very awkward to represent in SGP. Note that in these cases the primary result is not that typed GP can solve problems that one can't easily solve with SGP; that result follows immediately from the difficulty of representing these problems in a way that respects the closure property. The primary result is that typed GP can solve typed problems at all, i.e., that the typed representation and the typed recombination operators (like crossover) interact in a way that allows one to solve interesting problems whose natural representation involves types in some significant way.

These are important positive results, for they show that we have tools that can be applied effectively to certain problems that have a strong type component. Note, however, that they provide little in the way of a direct comparison between typed GP and SGP, because SGP can't even begin to (easily) address these problems. Many of these typed GP systems (e.g., both Montana's and Haynes') would, in fact, behave exactly as SGP on problems where the closure property naturally holds (such as the problems used in the experiments reported in this paper).

Some type systems, however, allow or require different structural representations than that used in SGP. These structural differences are often interesting in their own right, and the interaction between these new representations and operators like crossover deserve study. In this paper we examine the impact of the representation typically used with the Hindley-Milner type system. This type system is the basis for the type systems in modern functional programming languages such as ML, Miranda, and Haskell. Yu has shown that this system can be effectively combined with GP to evolve several interesting higher order list functions that would extremely difficult to represent (let alone evolve) in SGP [Clack and Yu 1997].

It would seem valuable, then, to compare the performance of SGP and HMGP on a suite of problems that can easily be made to satisfy the closure property. Note that the HMGP representation necessitates some sort of type system (to prevent the swapping of expressions such as "$*$" and "$(x + y)$") even if all the data in the problem is essentially of the same type (as is the case in the examples in this paper).

1.2 Structural differences between typed and untyped trees

Before moving to a discussion of the details of the experiments, it's important to realize that the typical representation of expression trees based on the Hindley-Milner type system differs in significant ways from the representation of expression trees in SGP. Consider, for example, the expression `(* (+ x y) x)`. Figure 1 shows the standard tree representation of this expression in most GP systems (which we will call the SGP representation). Table 1 lists the (sub)expressions represented by each of the nodes, i.e., the expressions available to operators like crossover. Note that the types of all the (sub)expressions are the same, i.e., the closure property holds.

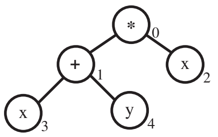

Figure 1 The SGP representation of `(* (+ x y) x)`. The nodes have been numbered for later reference.

The typical representation used in the implementation of most modern functional programming languages (which we will call the HMGP representation) would have only one kind of internal node, the application (or APP) node, which applies a function subtree to a single argument subtree. Functions are applied to multiple arguments through the use of multiple APP nodes. Figure 2 shows the HMGP representation of our sample expression, using '@' to indicate APP nodes.[1] Table 2 lists the (sub)expressions represented by each of the nodes, i.e., the expressions available to operators like crossover.

Node #	Expression	Type
0	`(* (+ x y) x)`	*real*
1	`(+ x y)`	*real*
2	`x`	*real*
3	`x`	*real*
4	`y`	*real*

Table 1 The (sub)expressions of the expression represented in Figure 1. The Node Numbers refer to the numeric labels in Figure 1. The type of each expression is also given.

Node #	Expression	Type	SGP node
0	`(* (+ x y) x)`	*real*	0
1	`(* (+ x y))`	*real* $\rightarrow$ *real*	
2	`x`	*real*	2
3	`*`	*real* $\rightarrow$ *real* $\rightarrow$ *real*	
4	`(+ x y)`	*real*	1
5	`(+ x)`	*real* $\rightarrow$ *real*	
6	`y`	*real*	4
7	`+`	*real* $\rightarrow$ *real* $\rightarrow$ *real*	
8	`x`	*real*	3

Table 2 The (sub)expressions of the expression represented in Figure 2. The Node Numbers refer to the numeric labels in Figure 2. The type of each expression is given, and nodes that correspond to nodes in the untyped representation (Figure 1) are listed in "SGP node".

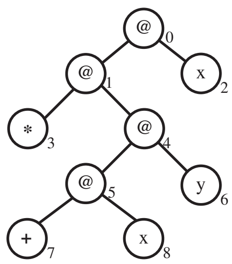

Figure 2 The HMGP representation of `(* (+ x y) x)`. The nodes have been numbered for later reference.

[1] In this APP node representation function application associates to the left, so our sample expression is actually a simplification of the fully parenthesized `((* ((+ x) y)) x)`.

One major difference between these two representations is that the HMGP representation has more nodes than the SGP representation, providing more potential points to apply recombination operators. Unlike the SGP representation, however, these nodes do not obey the closure property, with (sub)expressions having one of three different types. Leaves like the $+$ operator (node 7 in Figure 2) are functions that take two *real* values and return a *real* value, written $real \to real \to real$.[2] This representation also allows currying, or partial function application, so expressions like `(+ x)` (node 5 in Figure 2) are meaningful, representing the function that takes a value and adds `x` to it; this expression has type $real \to real$.

Thus the HMGP representation contains nodes that correspond directly to all of the nodes in the SGP representation (see Table 2), plus several nodes with no direct counterpart in the SGP representation. For example, in the HMGP representation the functions have been moved from being internal nodes to being leaves like the terminals. This is potentially significant because it now allows crossover to change an operator without affecting the arguments. Such a change is difficult for crossover in SGP, especially for operators at or near the root of the expression tree. As an example, in the HMGP representation a simple application of crossover could change the root operator ('*' in our example) to a different operator having a compatible type (such as '+') much more easily than in the SGP representation.

In problems that are highly sensitive to the choice of operators at or near the root, such as Gathercole's MAX problem [Gathercole and Ross 1996], it seems plausible that the HMGP representation might perform quite differently from the SGP representation. Note, however, that it's not obvious that the HMGP representation will in fact be preferable, since the increase in the number of nodes can potentially increase the size of the search space. In SGP, for example, there are 25 possible crossovers of our example expression with itself. In HMGP, there are 33 possible crossovers if we require that the swapped sub-expressions have the same type (25 crossovers of *real* with *real*, 4 crossovers of $real \to real$ with $real \to real$, and 4 crossovers of $real \to real \to real$ with $real \to real \to real$). [3]

[2] In this representation of types $\to$ is right associative, so the given type expression is shorthand for $real \to (real \to real)$.

[3] It has often been said (occasionally by one of the authors this paper) that one of the benefits of type systems is that they restrict the search space by constraining operations like crossover to compatibly typed subtrees. Such statements should be examined carefully. As mentioned earlier, many of the published applications of type systems are to problems where there is no straightforward untyped representation, making such claims highly suspect since they beg the question of what search space the typed space is smaller than, and in what way. For problems where a direct comparison between SGP and typed GP is meaningful, many of the types GP systems will behave exactly like SGP, i.e., there will be no reduction in the size of the search space. With HMGP, as our example shows, the type system can in fact increase the size of the search space. While it's clear that incorporating type systems into GP can often allow us to solve problems we can't easily address with SGP, claims that types restrict the search space are more suspect and depend crucially on the specifics of the type system and problem.

In the remainder of this paper we will first discuss some of the changes to the initial creation of individuals and to the crossover of individuals necessitated by the use of the Hindley-Milner type system. We will then outline a suite of problems chosen to represent a variety of different GP applications, each of which can be easily made to satisfy the closure property. We then present the results of applying both SGP and HMGP to these different problems, showing that HMGP significantly outperforms SGP on several problems without performing substantially worse on any of the problems. We conclude by suggesting several avenues for further study, including an intermediate representation that could provide much of the benefit of the HMGP representation without requiring the fairly complex machinery necessary to support the full Hindley-Milner type system.

2 Creation and crossover of typed individuals

The incorporation of any type system necessitates changes to the algorithm for the creation of new individuals and the algorithm for crossover. In this section we briefly outline the key changes made in our use of the Hindley-Milner type system.[4]

Before going into the details of the algorithms, however, a few general notes about the Hindley-Milner type system are in order. As discussed in Section 1, in HMGP there is no distinction between terminals and non-terminals. All internal nodes are application nodes (represented by '@' in the example in Figure 2) and what are traditionally called function and terminal nodes are all leaves in the HMGP representation. Since the traditional distinction between terminals and non-terminals breaks down here, we will refer to all the entities that can be combined to form a tree (functions, variables, constants, etc.) as *primitives*.

Each primitive has a type, such as *real* or $real \to real \to real$. Some primitives are *higher order*, meaning they take functions as arguments or return functions as results. An example is the function `ant_compose` from the artificial ant problem (see Section 3) which has type

$$(state \to state) \to (state \to state) \to (state \to state).$$

This is a function that takes two state modifying functions (like `move` or `turn_left`) and composes them into a new state modifying function that acts by applying the two functions one after the other. Note that due to currying, functions that would not normally be seen as higher order, such as $+$, are in fact higher when partially applied. When applied to a single argument, for example, we are treating + as a function which takes a *real* and returns a function of type $real \to real$.

Since we will be comparing SGP and HMGP, it's important to ensure that depth limits affect both GP systems in the same way. This requires that we modifies the definition of tree

[4] Our system is very similar to that used by Yu in [Clack and Yu 1997]. There are some differences, but most only apply when using polymorphic functions, which doesn't happen in any of the experiments described here.

depth for HMGP so that it computes the depth the tree would have it were an SGP tree instead of its actual depth (including all the extra APP nodes).

As well as supporting this new representation and curried function application, the Hindley-Milner type system also supports a very powerful version of *polymorphic* (or generic) functions which further complicates the picture. One major level of complexity polymorphism introduces is the need for type unification. Instead of being able to simply talk about types *matching* one needs to determine whether two types can be *unified*. While our system supports the use of polymorphic functions, we used none in the experiments reported here, so we'll limit our discussion of polymorphism and type unification. For more information on using these powerful tools in GP see [Clack and Yu 1997].

The structural differences induced by the use of APP nodes and the presence of high order functions (including curried versions of "normal" functions) lead to some significant changes in the way we create new individuals and perform crossover.

2.1 Initial creation of typed individuals

2.1.1 Basic concepts

When building individuals from scratch the key difference between typed and untyped GP is that in typed GP one specifies a desired type when asking the system to build a new (sub)tree. In most typed systems (such as Montana's [Montana 1995]) one can either choose a terminal having the desired type, or choose a non-terminal whose return type is the desired type.

In HMGP, however, the APP node representation and support for currying (partial function applications) complicates things. The key difference is that since there are a number of different "result types" for a given function, depending on how many arguments it is applied to, there are a number of different ways in which a primitive could be said to match the target type. Take as an example the function `and` having type $bool \to bool \to bool$. As well as directly matching a target type of $bool \to bool \to bool$, `and` can be used to generate an expression of type $bool \to bool$ by applying it to an expression of type *bool* (e.g., `(and true)`). It can also be used to generate an expression of type *bool* by applying it to two expressions of type *bool* (e.g., `(and true x)`, assuming `x` has type *bool*).

In the process of creating a tree having type t, then, one needs to find a primitive `p` whose type s contains t as a (not necessarily proper) suffix, i.e.,

$$s = \alpha \to \beta \to \cdots \to t$$

where α, β, etc., are possibly complex types.[5] The list of types α, β, etc., then represent the types of arguments that `p` needs to be applied to to yield an expression of type t.

As an example, consider using the set of primitives for the Artificial Ant problem (listed, along with their types, in Table 4) to generate an expression having target type $state \to state$. One could choose the primitive `move` since its type is exactly the desired type. One could also, however, choose to build an expression using the function `ant_compose` having the type

$$(state \to state) \to (state \to state) \to state \to state$$

since the target type matches a suffix of `ant_compose`'s type. In particular we can write the type of `ant_compose` as

$$\alpha \to \beta \to (state \to state)$$

where

$$\begin{aligned} \alpha &= state \to state \\ \beta &= state \to state. \end{aligned}$$

In this case the problem of constructing an expression having type $state \to state$ has been transformed into the problem of constructing two subexpressions having types α and β respectively, and then constructing the expression which applies `ant_compose` to these subexpressions. We might, for example, construct the following subexpressions

- `move`
- `(ant_compose move turn-left)`

having the required types, yielding the final expression

```
(ant_compose
    move
    (ant_compose move turn-left))
```

2.1.2 The algorithm

To understand the changes necessary in the creation algorithm consider the standard algorithms for the *grow* and *full* methods for creating initial individuals [Koza 1992]. At various points in these algorithms leaves are chosen from the set of terminals, and internal nodes (functions) are chosen from the set of non-terminals. Thanks to the closure property, whenever such a choice is necessary we know that *any* member of the appropriate set (terminals or non-terminals) is acceptable. In the case of HMGP, however, we need to indicate the target type t we want the resulting expression to have, and we need to restrict the choices made so that the resulting expressions are well-typed.

In the case where we are choosing what would be a terminal in SGP, it is sufficient to uniformly choose a primitive `p` from the set of primitives having type s matching the target type t.[6]

[5] Since we're not using polymorphic functions, it is sufficient to require that t simply match a suffix of s. In the presence of polymorphic functions, however, this needs to be generalized to say that t unifies with some suffix of s. It's also worth noting that if there are no polymorphic functions the target type t can match at most one suffix of s. If s is a polymorphic type, however, they may be multiple suffixes of s that unify with t, meaning that a function having type s can used in multiple ways to generate an expression having type t.

[6] Or unifying with t if using polymorphic functions.

In the case where we are choosing what would be a non-terminal in SGP, we need to choose a primitive p having type s such that t matches a *proper* suffix of s. The match must be against a proper suffix to ensure we get a non-terminal instead of a terminal. Assume that the type of p has the form

$$s = a_0 \to a_1 \to \cdots \to a_{n-1} \to t.$$

Then p can be seen as a function of n arguments having types $a_0, a_1, \ldots, a_{n-1}$. We then recursively construct n new trees $\mathtt{x}_0, \mathtt{x}_1, \ldots, \mathtt{x}_{n-1}$ having types $a_0, a_1, \ldots, a_{n-1}$ respectively. We can then construct the desired expression of type t:

$$(\mathtt{p}\ \mathtt{x}_0\ \mathtt{x}_1\ \ldots\ \mathtt{x}_{n-1}).$$

2.2 Crossover on typed individuals

As well as affecting the creation of new individuals, using a type system requires changes to the recombination operators to ensure that they produce well typed trees. In the experiments reported here we only use the crossover operator, so only changes to it will be discussed. Similar changes obviously have to be made to implement other operators such as mutation.

As in most GP systems crossover acts on two parents P and Q and generates a child C.[7] The process begins by choosing a subtree A from P; in these experiments we restrict the choice to a non-leaf 90% of the time to reduce leaf swapping.

Now, instead of choosing any subtree B from the other parent Q, we need to restrict ourselves to subtrees that have the same type as A.[8] We again restrict the choice of B to be a non-leaf 90% of the time. We relax this, however, if the only subtrees with compatible types turn out to be leaves, and allow B to be a leaf in this case.

3 Experiments

For comparison purposes, we ran several problems using both SGP and HMGP:[9]

- The MAX problem
- The Artificial Ant problem with the Santa Fe trail
- 4-bit even parity
- Symbolic regression with a quartic polynomial as the target function
- Sequence induction on a quartic polynomial

[7]One minor difference between our system and many other GP systems is that our crossover operator takes two parents and only produces a single child. This is necessary when using the Hindley-Milner type system with polymorphic functions because there are situations where subtree A can be legally substituted for subtree B, but the reverse is not true.

[8]As before, in the presence of polymorphic functions we look for subtrees whose type unifies with the type of A instead of just matching it.

[9]These runs were performed using the *Sutherland* EC framework. For more information see [McPhee *et al.* 1998] or `http://www.mrs.umn.edu/~mcphee/Research`.

Population size	500, steady state
Selection Method	Tournament selection Tournament size t=10
Maximum Generation	200
Generative Method	Ramped half-and-half, max initial depth 4
Maximum created depth	10
Internal crossover rate	90%
Mutation rate	0
Reproduction rate	0

Table 3 Common parameters for all runs

Except where noted, the parameters are those in Table 3 ,[10] and the function and terminal sets were the same for both SGP and HMGP.

For all but the MAX problem, fitnesses were tuples compared element wise, that is, the comparison operator was applied to the first element of each and if they were equal, then applied to the second, and so on. For these non-MAX problems we also implemented a parsimony pressure by setting the last element of the fitness tuple to the number of nodes in the individual (with fewer being better). This always gives preference to an individual with better performance on the problem, but selects the more parsimonious individual among two individuals with equal performance. To discourage early convergence, the size was normalized by setting this component to 0 if it was less than a certain threshold arbitrarily determined for each problem.

In each problem the HMGP representation was chosen to be as close to the SGP representation as possible. The one problem where the representation differs in any significant way is the Artificial Ant problem (see Section 3.2 for details).

3.1 The MAX problem

The MAX problem was first described in [Gathercole and Ross 1996]. It consists of the set of problems MAX-depth-D-{Function Set}-{Terminal Set} where the depth is restricted to D and the goal is to find the optimal tree, where fitness is measured as the value of the tree. In these experiments, the MAX-depth-7-{*,+}-{0.5} problem was used. The ramped half-and-half generative method was used with the maximum initial depth set to 5. We performed 24 runs using HMGP and 31 using SGP for this problem.

3.2 The Artificial Ant problem

The Artificial Ant problem evolves a program which navigates a near-sighted and motion-constrained artificial ant along an irregular trail of food [Koza 1992]. There are several well-known trails used for this problem; we used the one known as the Santa Fe trail, which lies on a 32×32 toroidal grid and contains 89 pieces of food. The ant is constrained to 400 moves, where a move is one of turn-

[10] The parameters described here were not tuned for optimal performance on the test problems; they simply provide a convenient reference point when comparing HMGP and SGP.

Function	Type
`if-food-ahead`	$\mathit{state} \to (\mathit{state} \to \mathit{state}) \to (\mathit{state} \to \mathit{state}) \to \mathit{state}$
`ant_compose`	$(\mathit{state} \to \mathit{state}) \to (\mathit{state} \to \mathit{state}) \to (\mathit{state} \to \mathit{state})$
`move`	$\mathit{state} \to \mathit{state}$
`turn-right`	$\mathit{state} \to \mathit{state}$
`turn-left`	$\mathit{state} \to \mathit{state}$
`ant_state`	state

Table 4 HMGP artificial ant primitives

Fitness measure	(food eaten, size)
Size normalization threshold	30
HMGP runs	61
SGP runs	113

Table 5 Artificial Ant parameters

ing left, turning right, or moving straight ahead. For SGP, the available non-terminals were `if-food-ahead` and `begin2`. `if-food-ahead` takes two arguments and evaluates the first if there is food ahead and the second otherwise. `begin2` takes two arguments and evaluates them sequentially. The terminals were the side-effecting thunks `(move)`, `(turn-right)`, and `(turn-left)`.

The SGP representation of the Artificial Ant problem depends crucially on side-effects. HMGP uses a side-effect free semantics, however, so several changes to the representation were necessary when moving to HMGP. A data type *state* was introduced which represented the state of an ant problem (the position of the pieces of food on the grid, and the ant's current position and direction). The functions used are listed in Table 4. `move`, `turn-right`, and `turn-left` all function in essentially the same way as their untyped equivalents, each taking a problem state as an argument and returning the new state resulting from their action.

In the HMGP representation `if-food-ahead` takes three arguments: a *state* `s` and two $\mathit{state} \to \mathit{state}$ functions `f` and `g`. If there is food ahead in `s`, `(f s)` is returned, otherwise `(g s)` is returned. Note that `s` may be a state resulting from a computation, so that the HMGP expression `(if-food-ahead (computation) f g)` is equivalent to the SGP expression `(begin2 (computation) (if-food-ahead f g))`. In the HMGP representation `ant_compose` takes two $\mathit{state} \to \mathit{state}$ functions `f` and `g` and returns their composition.

For both systems, an individual's program was iterated until all 89 pieces of food were found or the number of moves exceeded 400. In the HMGP version it is possible for an individual to return the state unchanged (i.e., without turning or making a move) which would lead to infinite computations; such cases were caught and treated the same as exceeding 400 moves. The additional problem-specific parameters for the Artificial Ant problem are listed in Table 5.

Population size	1000, steady state
Function set	{`and, or, nand, nor`}
Terminal set	{`a0, a1, a2, a3`}
Fitness cases	all 16 possible combinations of 4 bits
Fitness measure	(hits, size)
Size threshold	50
HMGP runs	54
SGP runs	61

Table 6 4-bit even parity parameters

Function set	{`+, -, *, %, sin, cos, exp, rlog`}
Terminal set	{`x`}
Fitness cases	20 evenly spaced points from the interval [-1, 1)
Fitness measure	(difference, hits, size)
Size normalization threshold	100
HMGP runs	77
SGP runs	75

Table 7 Symbolic regression with quartic polynomial as target

3.3 4 bit even parity problem

The n-bit even parity problem is the problem of determining whether a sequence of n input bits has an even number of 1's [Koza 1992]. We tested runs of the 4-bit even parity problem using the parameters listed in Table 6. The fitness component `hits` represents the number of cases correctly classified by the individual.

3.4 Quartic Polynomial

The quartic polynomial $x^4 + x^3 + x^2 + x$ [Koza 1992] was used as the target for a symbolic regression problem. The function set was {`+, -, *, %, sin, cos, exp, rlog`}, where `%` denotes the protected division operation and `rlog` the protected natural logarithm function. The terminal set was {`x`}, the independent variable. Fitness for both systems was the ordered triple `(difference, hits, size)`. `difference` was the sum of the absolute values of the differences between the actual value at each fitness case and the value of the individual at that point. `hits` was the number of fitness cases in which the difference between the actual value and the individual's value was less than 0.05. Other parameters for the Quartic Polynomial problem are listed in Table 7.

3.5 Sequence Induction

The sequence induction problem is an instance of the symbolic regression problem in which the domain of the independent variable is constrained to the natural numbers {0,1,2,3...}. We used the sequence $S_j = 5j^4 + 4j^3 + 3j^2 + 2j + 1$, as described in [Koza 1992]. The fitness measure used was the tuple `(difference, hits, size)`.

Function set	$\{+, *, -\}$
Terminal set	$\{j, 0, 1, 2, 3\}$
Fitness cases	$j \in \{0, 1, 2, ...20\}$
Fitness measure	(difference, hits, size)
Size normalization threshold	100
HMGP runs	72
SGP runs	52

Table 8 Sequence Induction parameters

Problem	Runs		Mean		Std. dev.	
	HM	S	HM	S	HM	S
MAX	24	31	3.7E9	2.1E9	1.3E8	2.1E9
Ant	61	113	45.95	39.81	14.41	9.73
Parity	113	123	14.14	14.56	1.34	1.08
Quartic	355	356	1.04	0.97	0.91	0.89
S.I.	332	338	3.8E5	5.3E5	4.4E5	4.7E5

Table 9 Number of runs, mean of the best fitness in each run, and standard deviation of the best fitness for each run for each problem using HMGP (HM) and SGP (S). For MAX the fitness is the tree value, with larger being better. For Artificial Ant the fitness component listed is `food eaten`, with larger being better. For 4 Bit Even Parity the fitness component listed is `hits`, with larger being better. For Quartic Polynomial and Sequence Induction the fitness component listed is `difference`, with smaller being better.

`difference` was the sum of the absolute values of the difference between the individual and the target over the cases $j \in \{0, 1, 2, ...20\}$, with the difference capped at 1,000,000 to prevent overflow errors. `hits` represented the number of cases where the individual's value exactly matched the value of the target sequence. Other parameters for the sequence induction problem are listed in Table 8.

4 Results

Each of the problems described was run multiple times using both HMGP and SGP, with the results summarized in Table 4.[11] For the MAX problem, the Artificial Ant problem, and the Sequence Induction problem the average best fitness over the various runs is substantially better when using HMGP. For the 4 Bit Even Parity problem and the Quartic Polynomial problem SGP did better than HMGP, but in each case the difference is minor (less than half a hit for Even Parity and an average per test error of less than 0.01 on Quartic Polynomial). The question, though, is whether any of these differences are statistically significant.

While t-test procedures are sufficient for determining significance in non-modal distributions, several of our problems produced highly modal distributions. To determine significance on these problems, we applied the Cochran-Mantel-Haenszel χ-square test with degrees of freedom = 1.

4.1 The MAX problem

The data for the MAX problem was quite modal, so the Cochran-Mantel-Haenszel method was used. This produced a statistic of 9.048, with p-value 0.003, indicating that it is highly unlikely that the types did not in fact influence the performance on this problem in a positive fashion.

4.2 The Artificial Ant problem

The data for the Artificial Ant problem was fairly evenly distributed, so a simple t-test was performed which indicated that again the HMGP results were significantly higher (probability of error less than 0.004). Further analysis of the data yielded a 95% confidence interval for the mean advantage of HMGP over SGP of between 2.481 to 9.793 pieces of food.

4.3 The 4 Bit Even Parity problem

The data for the 4 Bit Even Parity problem was fairly modal, so a Wilcoxon 2-Sample test was performed, yielding a p-value of 0.0196. This suggests that the difference in performance on this problem of approximately 0.42 cases was in fact statistically significant. The 95% confidence interval for the difference between HMGP and SGP means was (0.11, 0.73), which indicates that the difference between HMGP and SGP is unlikely to be more than .7 cases out of 16, or 4.3%.

4.4 The Quartic Polynomial problem

The data for the quartic polynomial symbolic regression problem presented a skewed normal distribution. A t-test revealed a p-value of 0.1255, which was validated by a Wilcoxon test which yielded a p-value of 0.1250. These results mean that if HMGP and SGP actually had the same performance on this problem, we could observe results like ours approximately 13% of the time. The 95% confidence interval for the difference between HMGP and SGP (HMGP - SGP) is (-0.1, 0.2), indicating that the best we can say is that the performances appear to be approximately equal on this problem, with SGP *slightly* outperforming HMGP.

4.5 The Sequence Induction problem

For the sequence induction problem, we again found that the data was too modal for a t-test. A Wilcoxon 2-Sample Test revealed a p-value of 0.0001. This p-value reveals that our results would be *extremely* unlikely (0.01% chance) if the performances of HMGP and SGP on this problem were in fact similar. The 95% confidence interval for the difference between HMGP and SGP was (79000, 218000), indicating that it is highly likely that HMGP performs at least 20% better on average than SGP on the sequence induction problem.

5 Conclusions and future research

Our tests with HMGP and SGP indicate that HMGP outperforms SGP on several problems of interest, while doing about the same on the other test problems. While it would be nice to conclude that HMGP does as well or better than SGP on all problems, such a conclusion would clearly be premature

[11] The raw data these statistics are based on is available at `http://www.mrs.umn.edu/~mcphee/Research`.

given the small number of problems tested. The results do indicate that HMGP has potentially significant value, however, and warrants further study.

Beyond these basic conclusions, we provide some comments, speculation, and ideas for further research.

5.1 When (and why) will HMGP do better?

The obvious question is when will HMGP outperform SGP, and why? The most likely candidate is the ability of HMGP to more easily modify parts of the parse tree that would be near the root in the SGP representation, and therefore extremely difficult to change in SGP. In the remainder of this discussion, we will refer to these as "root" operators.

For the MAX problem, this conjecture is strongly supported by examining the best of run trees on the typed and untyped runs of that problem. The optimal SGP tree for this problem has nothing but "*" operators in the first four levels of the tree (starting at the root); "*" and "+" operators are interchangeable on the next level, and all operators on lower levels should be "+". One way, then, of measuring the ability of GP to develop the "root" structure necessary to maximize the tree's value is to determine the size of the largest tree starting at the root (in the SGP representation) containing only "*" operators. For the optimal tree this count should be at least 15, but trees with "+" operators in undesirable positions at or near the root will have lower counts. Out of 24 HMGP runs, 20 best of run individuals had counts of at least 15, i.e., maximal values. Over 31 SGP runs, however, only 14 best of run individuals had counts of at least 15, with 15 having counts no greater than than 7!

It would be nice if we could easily characterize desired "root" structure for the other problems; unfortunately this is not entirely straightforward. In the Sequence Induction problem, for example, the inclusion of "−" in the function set introduces the possibility of introns and makes a simple characterization of the resulting trees more difficult.

While the situation with the Artificial Ant problem is also more complex than in the MAX problem, there are some similarities that deserve mention. An examination of the trees (and their sizes) in the Artificial Ant runs suggests that SGP tended to try to "memorize" the trail. The tree sizes typically rose quickly and then levelled off, presumably as a result of the depth limit on the trees generated by crossover. The HMGP trees, however, tended to remain much smaller in size, suggesting that instead of taking the "memorize the trail" approach they were taking the "learn a strategy for following the trail" approach. One way to test this theory would be to test the best of run individuals on new trails. If SGP was indeed "memorizing the trail" then we would expect very poor performance by those individuals on new trials, and if HMGP was indeed "learning a strategy" then we would expect those individuals to perform reasonably well on new trails.

All this, however, begs the more interesting question of *why* HMGP did the "right" thing (strategy finding) more often than SGP. This may again be related to the ability of HMGP to more easily modify "root" operators. It would seem likely, for example, that a successful strategy would tend to avoid unconditional actions since they could cause the ant to lose track of the trail or miss nearby food. One might expect, however, that in the early stages of a run, unconditional trail following might do well compared to embryonic strategies. It's possible that the asymmetry in SGP between terminals and nonterminals (functions) makes it more difficult to later remove unconditional actions that had at one time been beneficial.

In 113 SGP runs of the Artificial Ant problem, for example, *all but three* best of run individuals start with one or more unconditional moves, whereas over half the best of generation individuals at generation 1 start with an `if-food-ahead`. In 61 HMGP runs of the Artificial Ant problem, however, 40 of the best of run expressions start with an `if-food-ahead` node, where only 27 of 61 runs had a best of generation individual at generation 1 that started with `if-food-ahead`. Note, however, that HMGP trees that start with `if-food-ahead` don't necessarily avoid unconditional actions, since the first argument to `if-food-ahead` could be an expression that involves unconditional actions. Note also, that expressions starting with an `ant_compose` node (of which there were 7 amongst the best of run individuals) could avoid unconditional actions depending on their arguments.

The size and complexity of the expressions involved (especially those from the SGP runs) made a detailed analysis impossible in the time available, but it seems likely that the proportion of unconditional moves increased over time in SGP while it decreased over time in the HMGP runs. Further analysis would clearly be valuable.

Given the Sequence Induction results it is somewhat surprising that the performance of HMGP and SGP were so similar on the Quartic Polynomial problem, with SGP performing marginally better. One interesting experiment would be to extend the range of test cases in the Quartic Polynomial problem beyond $[-1, 1)$. Over this range the target function is fairly flat and featureless, and it's clear from our results that there are many fairly simple expressions that approximate the given quartic polynomial reasonably well, many of which include "incorrect" functions such as `exp`, `rlog`, and `sin`. This may make the GP process fairly insensitive to the choice of "root" operators, limiting the advantage of HMGP. If we were to extend the range of the test cases, however, we might well capture more features of the quartic polynomial, which could make the choice of "root" operators more important, possibly giving HMGP an advantage over SGP.

5.2 Asymmetry among arguments

It appears that one possible advantage of the HMGP representation is that it breaks the asymmetry between functions and arguments, allowing either to be replaced with equal ease. Note, however, that it introduces an asymmetry between the arguments by binding earlier arguments more tightly to the function than later arguments. This could well affect the evolutionary process, especially when using functions that are not symmetric in their arguments, such as `if-food-ahead`. It would be interesting to experiment with different versions of such operators by rearranging the order of their arguments and see how this affects the the behavior of HMGP.

5.3 A simpler, but promising, representation

Many of the benefits of HMGP on these problems seem to stem from the representation of functions (internal nodes in SGP) as leaf nodes (primitives in HMGP) making it easier to recover from a poor choice of top-level operator in subsequent generations. Given this benefit, and the performance and complexity issues related to HMGP, it seems it might be beneficial to develop a simpler representation which still affords the opportunity to swap functions as leaves.

Such a representation could be implemented by treating functions as leaves and introducing a node type similar to the APP node described in Section 1, with the important difference that these nodes would be k-ary instead of binary and would not allow partial application. Given such a representation, the tree `(* (+ x y) x)` described in the introduction would be represented as shown in Figure 3

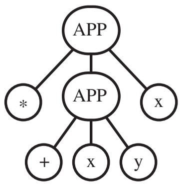

Figure 3 The expression from the Introduction as represented under the scheme described in Section 5.3

With this representation, tree generation would be similar to that of SGP, except that when a function is chosen, it is created as an APP node with the appropriate number of value-returning children. Crossover between two parents P and Q would uniformly choose a point C from among the possible points in the first parent P. If C is a value-returning node (i.e. an APP node or a terminal value such as a constant or variable), then crossover uniformly selects a value-returning node from among the value-returning nodes in Q; if C is a function, crossover selects a function of the same arity from Q.

It would be useful to compare runs using this representation with those with SGP and HMGP representations, as this would suggest what role, if any, currying (partial function applications) played in the HMGP results reported here.

5.4 Speed of HMGP vs. SGP

In this paper we present results about the performance of HMGP vs. SGP in terms of problem results; it should be noted, however, that the complexity of the tree generation and crossover operations as well as the need for more sophisticated evaluation procedures make HMGP runs significantly slower than SGP for equivalent population and tournament sizes. In the experiments performed here, SGP enjoyed an approximate 4:1 speed advantage over HMGP. However, the system used for these experiments was in its early stages and could significantly benefit from a variety of optimizations. (Note that even with these optimizations, we would expect the type system to be slower than untyped GP) Additionally, while a slowdown factor of 4 may seem significant today, given the current rate of hardware improvement this disadvantage should seem less significant given the passage of time.

Acknowledgments

Jon Anderson provided invaluable assistance with our statistical analysis.

Reierson's work was supported by the Morris Academic Partners program, and both Reierson and Hopper were supported by the Undergraduate Research Opportunities Program. A grant from the University of Minnesota Graduate School Grant-in-aid program program funded the purchase of computing equipment used in these experiments.

References

Clack, Chris and Tina Yu (1997). Performance enhanced genetic programming. In: *The Proceedings of the Sixth International Conference on Evolutionary Programming*.

Gathercole, Chris and Peter Ross (1996). An adverse interaction between crossover and restricted tree depth in genetic programming. In: *Genetic Programming 1996: Proceedings of the First Annual Conference*.

Haynes, Thomas, Dale Schoenefeld and Roger Wainwright (1995). Type inheritance in strongly typed genetic programming. In: *Advances in Genetic Programming 2* (Kenneth E. Kinnear Jr. and Peter J. Angeline, Eds.). MIT Press.

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

Langdon, William B. (1995). Data structures and genetic programming. In: *Advances in Genetic Programming 2* (Kenneth E. Kinnear Jr. and Peter J. Angeline, Eds.). MIT Press.

McPhee, Nicholas Freitag, Nicholas J. Hopper and Mitchell L. Reierson (1998). *Sutherland*: An extensible object-oriented software framework for EC. In: *Genetic Programming 1998: Proceedings of the Third Annual Conference* (John R. Koza, *et al*, Eds.).

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

Wong, Man Leung and Kwong Sak Leung (1995). Evolving recursive functions for the even-parity problem using genetic programming. In: *Advances in Genetic Programming 2* (Kenneth E. Kinnear Jr. and Peter J. Angeline, Eds.). MIT Press.