Functional Genetic Programming with Combinators

Forrest Briggs <fbriggs@gmail.com> and Melissa O'Neill <oneill@acm.org>

Harvey Mudd College, Claremont, CA, U.S.A.

Abstract. Prior program representations for genetic programming that incorporated features of modern programming languages solved harder problems than earlier representations, but required more complex genetic operators. We develop the idea of using combinator expressions as a program representation for genetic programming. This representation makes it possible to evolve programs with a variety of programming language constructs using simple genetic operators. We investigate the effort required to evolve combinator-expression solutions to several problems: linear regression, even parity on N inputs, and implementation of the stack and queue data structures. Genetic programming with combinator expressions compares favorably to prior approaches, namely the works of Yu [37], Kirshenbaum [18], Agapitos and Lucas [1], Wong and Leung [35], Koza [20], Langdon [21], and Katayama [17].

1 Introduction

Genetic programming is the use of genetic algorithms to write programs through evolution. It is a powerful technique with many applications in artificial intelligence, such as controlling robotic soccer players [23]. For more about these subjects, see De Jong [10], Goldberg [12], Koza [20] and Langdon [22].

A central issue in genetic programming is how to represent programs. A representation must be powerful enough to naturally express solutions to the problems that we wish to solve, and it must also support a sufficient collection of genetic operators (typically random program creation, mutation, and crossover) [27].[1]

```
fun sum list =
  case list of
    nil    => 0
  | h :: t => h + (sum t)
```

Fig. 1. Code to sum a list.

Prior program representations for genetic programming that incorporated features of modern programming languages solved harder problems than earlier representations, but required more complex genetic operators (see Section 9, Related Works).

A statically typed functional program representation provides expressive power and valuable constraints on how programs can evolve. Consider the Standard ML [25] code in Figure 1, which sums a list of integers. In this example, type constraints disallow a mutation of `h + (sum t)` to `h + t` because addition requires two numbers, but `t` is a list, not a number.

[1] It is not strictly necessary to have mutation or crossover, but some kind of genetic operators are necessary.

Programs in functional languages can be simplified to λ-expressions [16], which are equivalent to *combinator expressions* [8, 16, 30]. Combinator expressions consist of only built-in values (functions and constants), and function applications (see Section 3, Combinator Expressions). Because they are simple in structure, but can represent arbitrary functional programs, combinator expressions are an ideal representation for functional genetic programming.

```
fun sum list =
  let fun sum2 list2 =
      case list2 of
        nil      => 0
      | h2 :: t2 => h2 + (sum2 t2)
  in case list of
        nil      => 0
      | h :: t   => h + (sum2 t) end
```

Fig. 2. Unrolling the loop from Figure 1.

The code in Figure 1 can be written in combinator form as

```
Y (B (C (C case 0)) (C (B B plus)))
```

The functions `Y`, `B`, and `C` are combinators. One possible mutation of the above combinator expression is

```
B (C (C case 0)) (C (B B plus))
  (Y (B (C (C case 0)) (C (B B plus))))
```

This mutation duplicates the underlined portion of the original expression. Figure 2 shows the SML code equivalent to this mutated combinator expression—the mutation unrolled the loop one iteration. In the combinator representation, the mutation involves only code duplication, but viewed in the higher-level SML representation, it defines a new local function. Thus, a mutation that is simple at the combinator level corresponds to a more complex mutation in higher-level code.

The major contribution of this paper is to show that combinator expressions are a useful program representation for genetic programming. In particular,

– We observe that combinator expressions can represent programs that introduce local variables, but the genetic operators to manipulate combinator expressions do not require special cases for variables;
– We give genetic operators to evolve statically typed combinator expressions;
– We show that evolving combinator expressions is efficient compared to the works of Yu [37], Kirshenbaum [18], Agapitos and Lucas [1], Wong and Leung [35], Koza [20], Langdon [21], and Katayama [17] on several problems: linear regression, even parity on N inputs, and devising representations and implementations for stacks and queues.

2 The Problem With Variables

Most programming languages that people use have variables. In functional programming languages, there are at least three common ways to introduce local variables: `let`-expressions, λ-abstractions, and `case` expressions.[2] Earlier program representations required special cases in their genetic operators to deal with λ-abstractions and `let`-expressions (see Section 9, Related Works).

Figure 3 shows one example of the difficulties that variables cause for genetic operators. A naïve crossover operator could produce the code in Figure 3(c) by replacing the subexpression `x * 7` from Figure 3(b) with the subexpression `x + y` from Figure 3(a). This result is invalid because `foobar` does not define `y`.

One approach to this problem is to use complex genetic operators that attempt to address all the problems that variables give rise to [19], but an alternative is to avoid the problems of variables by avoiding variables themselves. Because every function with variables has an equivalent combinator expression without variables, a combinator-based approach offers expressive power without complex genetic operators.

```
fun foo x =
    let val y = 3
    in  x + y end
```

(a) Parent 1

```
fun bar x =
    x * 7 + 1
```

(b) Parent 2

```
fun foobar x =
    x + y + 1
```

(c) Child

Fig. 3. Bad crossover.

3 Combinator Expressions

As we saw in the introduction, functional programs can be written in a form that is free of variables using simple combinator functions [8, 16, 30]. Figure 4 lists the definitions of a set of combinators that are useful for implementing functional programming languages [16].[3]

A combinator expression is either a built-in value (which can be a combinator function, another built-in function such as addition, or a constant) or the application of one combinator expression to another.

$$
\begin{aligned}
\texttt{I}\ x &= x\\
\texttt{K}\ c\ x &= c\\
\texttt{S}\ f\ g\ x &= f\ x\ (g\ x)\\
\texttt{B}\ f\ g\ x &= f\ (g\ x)\\
\texttt{C}\ f\ g\ x &= f\ x\ g\\
\texttt{S'}\ c\ f\ g\ x &= c\ (f\ x)\ (g\ x)\\
\texttt{B*}\ c\ f\ g\ x &= c\ (f\ (g\ x))\\
\texttt{C'}\ c\ f\ g\ x &= c\ (f\ x)\ g
\end{aligned}
$$

Fig. 4. The definitions of several useful combinators.

For example, `S add I` is equivalent to the function `fn x => add x x`.[4] The expression `S add I` does not contain any variables, but it represents a function that introduces a local variable named `x`.

3.1 Running Combinator Expressions

Genetic programming systems need to run the programs they generate. *Combinator reduction* [33, 16] is a simple and efficient technique for executing combina-

[2] The code on the first page shows that `case` expressions that perform pattern matching on lists introduce local variables. To evolve combinator expressions that represent programs with `case` statements, it is only necessary to make an appropriate `case` function available.

[3] Strictly speaking, we can encode every functional program using only `S` and `K` (for example `S K K` $\equiv$ `I`), but, in practice, the larger set shown in Figure 4 allows programs to be expressed more compactly.

[4] Parentheses are implicitly left-associative, so `S add I` is the same as `((S add) I)`.

tor expressions. Although this technique has been described at length elsewhere, we can cover its essence here.

The rules given in Figure 4 provide the basis for implementing the necessary reduction machine. If a built-in function does not yet have all its arguments, it cannot yet be reduced. But once the function has been applied to the all the arguments it needs, we can perform a reduction. Thus, `K` and `K 7` cannot yet be reduced, but `K 7 3` can be reduced to `7`, using to the rule for `K` given in Figure 4. To run a combinator expression, repeatedly consider the outermost function application and attempt to reduce it. Figure 5 shows an example of running `S add I 7`.

```
   S add I 7
⇒ add 7 (I 7) – rule for S
⇒ add 7 7     – rule for I
⇒ 14          – rule for add
```

Fig. 5. Running a combinator expression.

3.2 Type System

Many of the expressions that we could form by applying built-in functions to each other are not meaningful; for example, the expression `add I I` is meaningless, because the `add` function adds numbers, not identity functions. By not constructing such meaningless expressions, we can greatly narrow the search space of our genetic algorithm. A type system can impose constraints that prevent these kinds of obvious errors.

The most natural type system for a functional language, even a simple one based on combinator expressions, is the Hindley-Milner type system [14, 24, 9], which forms the basis for the type systems of most modern functional languages, such as Standard ML [25] and Haskell [15]. The key ideas behind this type system are *parametric types* and *type inference* via *unification.*

Table 1. Types of `S`, `I`, `add` and `7`

Value	Type
`S`	$(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$
`I`	$\delta \to \delta$
`add`	`Int` $\to$ `Int` $\to$ `Int`
`7`	`Int`

Every type in the Hindley-Milner system has zero or more types as parameters. Types such as `Int` and `Bool` are types with zero parameters, whereas types such `List` take a single parameter indicating the kinds of objects stored in the list; thus we would write `List(Bool)` to denote a list of booleans. For brevity, we write the type of the function from `X` to `Y` as `X` $\to$ `Y`, which is short for `Function(X,Y)`.

Types can include type variables. For example, the type of the `length` function is `List`$(\alpha) \to$ `Int`. Here α is a type variable.[5] Type variables are implicitly universally quantified (i.e., `List`$(\alpha) \to$ `Int` is short for $\forall\alpha$, `List`$(\alpha) \to$ `Int`).

The Hindley-Milner system infers the type of an expression using unification [5, 16, 28]. Two types unify if there is a way in which all of their type variables can be assigned such that after substituting the assigned values for the type

[5] Greek letters denote type variables.

variables, the two types are equal. For example, `List`(α) unifies with `List(Int)` if $\alpha =$ `Int`. `List`(α) does not unify with `Bool`, because there is no way to assign α that makes the types equal.

The details of type inference are beyond the scope of this paper, but an example is useful. Let us infer the type of `S add I`, given the types in Table 1. The only consistent way to assign type variables for `S` applied to `add` is to set $\alpha = \beta = \gamma =$ `Int`, inferring the type of `S add` as (`Int` $\rightarrow$ `Int`) $\rightarrow$ `Int` $\rightarrow$ `Int`. Similarly, from the types of `S add` and `I`, the type of `S add I` results in $\delta =$ `Int` and an inferred type for `S add I` of `Int` $\rightarrow$ `Int`.

4 Genetic Operators

In order to evolve combinator expressions, it is necessary to generate random expressions, and to have genetic operators that are analogous to mutation and crossover. This section discusses an implementation of genetic operators for combinator expressions.

4.1 Random Creation

Genetic programming systems require an algorithm to generate random expressions. Let generate(τ, L) be the algorithm to generate a well-typed expression of type τ using a library L, where a library consists of *phrases* (a phrase is either a simple built-in value, or a larger prebuilt expression) and their associated types. We define generate(τ, L) as follows:

1. Find a value in L with a type that matches our desired type, τ, or a function in L that can return such a value if given suitable arguments.
2. Recursively use the generate algorithm to find values for any necessary arguments. If no suitable arguments can be found, repeat Step 1 to find a different starting point.

For example, if we wanted a function of type `Int` $\rightarrow$ `Int`, using Table 1 as our library, one possibility is for generate to choose `I`, as `I` has a type that matches ($\alpha \rightarrow \alpha$ matches if $\alpha =$ `Int`).

There are more possible ways to make an `Int` $\rightarrow$ `Int` function, however. In functional programming, multiargument functions are usually curried [8] and can be partially applied. Thus, `add` can be seen as both a function of two arguments and as a function with one argument that returns an `Int` $\rightarrow$ `Int` function. Hence generate should consider all possible partial applications of the functions in its library. Including such partial applications, observe that we can not only use `add` to make an `Int` $\rightarrow$ `Int` function (provided we can come up with an `Int` to pass to `add`), but that we can also use `S`, provided that we can come up with two arguments for `S` of type `Int` $\rightarrow \beta \rightarrow$ `Int` and an `Int` $\rightarrow \beta$. Using `add` as the first argument defines $\beta =$ `Int`, leaving us seeking an `Int` $\rightarrow$ `Int` value for the second argument; `I` is an acceptable choice. Thus, `S add I` is another possible result.

In essence, the generate algorithm is a backtracking search algorithm, and is, in fact, similar to a simple theorem prover [2]. In practice, we limit the amount of time the algorithm may spend by limiting the number of pieces it may assemble to form an expression. We define two parameters, *max-expression-size* and *max-phrases* to control this limit.

4.2 Generalized Genetic Operator

Genetic algorithms typically require genetic operators for mutation, crossover, and random creation. The generate algorithm can serve as the basis of all three. When we wish to mutate an expression or combine expressions, we can do so by constructing a library for generate that includes subexpressions from the parent expressions. In other words, we

1. Make a phrase for every subexpression in each parent;
2. Construct a list of phrases consisting of the phrases from all of the parent subexpressions and phrases for built-in values;
3. Use the list of phrases with generate to produce a new expression of the required type.

This algorithm can make any new expression that the standard point mutation or crossover algorithms [20] can make, but it can also produce other results. For example, given the expression `(x (y z))` as a parent and the built in values {`x`, `y`, `z`, `w`}, it could produce the new expression `(w (x (y z))`. Thus, whereas point mutation can only replace a subtree in the input expression, the result in this example embeds the parent tree as a subtree of a new root.

5 Experimental Setup

In Section 6, we discuss the results of several experiments that measure the effort required to evolve combinator expressions to solve various problems. This section describes the details of our experimental setup.

5.1 Genetic Algorithm

The basic idea behind a genetic algorithm is to simulate a population of evolving organisms that represent possible solutions to a problem. In this case, the organisms are combinator expressions. There are many variants of genetic algorithms. Like Langdon [22], our genetic algorithm uses tournament selection and steady-state replacement.[6]

Our genetic algorithm draws inspiration from Langdon [22] and Yu [37] by attempting never to evaluate the same expression twice. Whenever a genetic

[6] We include the details of our genetic algorithm only to provide a complete description of our experiments. Different optimization systems could also benefit from representing programs as combinator expressions.

operator produces a new expression, if the expression is not different from every other expression so far, the algorithm tries again as many as *tries-to-be-unique* times to get a unique expression. If it takes more than *tries-to-be-unique* attempts to get a unique expression, the algorithm accepts the next new expression, regardless of whether it is unique.

The parameters of the genetic algorithm are *population-size*, *tournament-size*, and *num-iterations*. The genetic algorithm works in the following way:

1. Generate and evaluate the fitness of *population-size* unique expressions.[7] If any of these expressions is a correct solution, stop immediately.
2. Choose the best amongst *tournament-size* randomly selected expressions in the population as a parent. Choose a second parent in the same way.
3. Apply the genetic operator to these parents to produce a new expression. Evaluate the fitness of new expression. If it is a correct solution, stop immediately.
4. If the new expression has a better fitness than the most unfit expression in the population, randomly choose one of the expressions in the population tied for most unfit, and replace it with the new expression.[8]
5. If the algorithm has iterated less than *num-iterations* times, go back to step 2. Otherwise, stop.

Our preliminary experiments showed that randomly choosing one of the expressions that is tied for worst to replace helps to prevent the population from stagnating. The problems in Section 6 have fitness functions with few distinct values. Thus, the behavior of the population when all expressions have the same fitness is important.

5.2 Brute Force

Sometimes brute-force enumeration of every correctly typed expression in order of size is more efficient than evolution [17]. We implemented a brute-force algorithm for combinator expressions and compared the effort required to find combinator expressions with evolution and with brute force.

5.3 Comparing Effort

A standard measure of effort for genetic programming is the minimum number of evaluations necessary to achieve a 99% likelihood of finding a correct solution [20]. Each time the genetic algorithm runs, there is some chance that it will find a correct solution. This probability is approximately S/C, where S is the number of trials that succeed out of C, the total number of trials. Let $P_{suc}(n)$ be the approximate probability of succeeding after evaluating n expressions. The number of times that the genetic algorithm must run to achieve a 99% likelihood

[7] In the context of a genetic algorithm, evaluate means "find the fitness of."

[8] Lower fitness scores are better. A fitness of 0 corresponds to a correct solution.

of finding a correct solution is $R(P_{suc}(n)) = \lceil \ln(1 - 0.99)/\ln(1 - P_{suc}(n)) \rceil$ [20]. If the algorithm stops at n evaluations, the number of evaluations necessary to have a 99% likelihood of finding a correct solution is $E(n) = R(P_{suc}(n)) \times n$. There is some value for n that minimizes $E(n)$. We refer to this minimum effort as E.[9]

Different authors used fitness functions with different numbers of test cases for some of the problems in Section 6. Therefore, it is meaningful to compare effort in terms of the number of test cases that must be evaluated to have a 99% chance of finding a correct solution. This number is $E \times T$, where T is the number of test cases per evaluation.

Minimum effort, E, is an appropriate way to measure effort for a genetic algorithm, but we also need comparable measure for brute force. We could use the number of expressions that brute force tries before finding a correct solution. However, the order in which brute force examines expressions of the same size differs, depending on the order in which it stores built-in values. In the worst case, the correct solution is the last expression that it tries of a particular size. Therefore, we think that the number of expressions with size less than or equal to the size of the smallest correct expression is a fair measure of effort for brute force.

The brute-force algorithm has a run time that is exponential in the size of the expressions it searches. As the algorithm looks for progressively larger expressions, it takes longer to find each expression. If it takes more than 10 minutes for brute force to produce a new expression, we stop and declare that brute force is infeasible.[10]

5.4 Problem Specification

Like Katayama [17], to specify a problem, we parse the built-in values and the fitness function, and infer their types from code in a programming language, rather than coding them directly into the system. We specify the built-in values and fitness function in a subset of SML that we call Mini-ML.

Several kinds of errors can occur when evaluating a combinator expression, such as taking the `head` of an empty list, dividing by 0, and causing integer overflow. If an expression causes a run-time error, we stop evaluating it immediately and give it the worst possible fitness score (∞). Expressions that make more than 1000 recursive functional calls are deemed nonterminating, which we count as a run-time error.

In addition to any problem-specific built-in values, we include all the combinators from Figure 4 (except `K`) in the set of built-in values for all problems. Without `K`, combinator expressions cannot represent λ-expressions that introduce unused variables.

[9] Authors who use a generational genetic algorithm call this minimum effort $I(M, z)$, but as we use a steady-state genetic algorithm, this notation does not apply.

[10] We ran our experiments on an Apple MacBook with dual 2.0 GHz processors and 1 GB DDR2 SDRAM, running Mac OS X 10.4.6. Power saving options were disabled.

6 Experiments

In this section, we investigate the effort required to evolve combinator-expression solutions to several problems: linear regression, even parity on N inputs, and implementation of the stack and queue data structures. We also contrast those results with prior techniques.

6.1 Linear Regression

Let us first look at a simple problem that requires little effort for evolution, but for which brute force is infeasible: regression of the function $g(x) = 6x+6$. Genetic programming finds g by minimizing the sum-squared error between a possible solution and g over the inputs 0, 1, 2, 3, and 4. Tables 2(a) and 2(b) list the parameters and built-in values for the linear-regression problem. Figure 6 lists the Mini-ML code for the fitness function.

```
fun fitness f =
  let fun sqr x = x * x
  in  sqr((f 0) - 6) +
      sqr((f 1) - 12) +
      sqr((f 2) - 18) +
      sqr((f 3) - 24) +
      sqr((f 4) - 30)
  end
```

Fig. 6. The fitness function for the linear-regression problem.

One of the evolved solution is `B (times (C (C (C times)) (inc (inc 1)) (inc 1))) inc|`, which is essentially equivalent to the SML expression `fn x => times (times 2 3) (inc x)`.

If the genetic algorithm stops at 2141 evaluations, 60 out of 60 trials find a correct solution, so $P_{suc}(2141) \approx 60/60 = 1.0$ and 1 run is necessary for a 99% probability of success. The minimum effort is $1 \times 2141 = 2141$ evaluations for a 99% probability of success. Brute force is infeasible for this problem.

6.2 Even Parity

Koza [20] established the even-parity problem as a benchmark for genetic programming. The problem is: given a list of boolean values, return `true` if there are an even number of `true` values in the list, and `false` otherwise. The type of the `evenParity` function on N inputs is `List(Bool)` $\rightarrow$ `Bool`. Like Yu [37], we use 12 test cases, which comprise every list of 2 or 3 boolean values. The

Table 2. The parameters and built-in values for the linear-regression problem.

(a) Parameters

Parameter	Value
max-expression-size	20
max-phrases	7
population-size	5000
num-iterations	50,000
num-trials	60
tries-to-be-unique	50

(b) Built-in Values

Value	Type
`0`	`Int`
`1`	`Int`
`add`	`Int` $\rightarrow$ `Int` $\rightarrow$ `Int`
`times`	`Int` $\rightarrow$ `Int` $\rightarrow$ `Int`
`inc`	`Int` $\rightarrow$ `Int`

fitness of a potential solution to this problem is the number of test cases that it fails.[11] For the purpose of comparison, we use the same built-in values (except combinators), *population-size*, and *num-trials* as Yu [37]. Tables 3(a) and 3(b) list the parameters and built-in values for the even-parity problem.

One of the evolved solutions is `foldl (C' I (C (S (C' (C S') (C' I nand) and) or))) true`, which is equivalent to the SML expression `foldl (fn x => fn y => and (nand y x) (or x y)) true`.

If the genetic algorithm stops at 2269 evaluations, 10 out of 60 trials find a correct solution, so $P_{suc}(2,269) \approx 10/60 = 0.16$ and 26 runs are necessary for a 99% probability of success. The minimum effort is $26 \times 2269 = 58,994$ evaluations for a 99% probability of success. Brute force finds a solution of size 7. There are 13,006 total expressions of size ≤ 7. Thus, the effort required for brute force to find a solution is 13,006 evaluations.

Table 3(c) lists the effort required to find a solution to the even-parity problem using combinator expressions with brute force and evolution ("Brute Force" and "Combinators" in Table 3(c)), PolyGP [37], Kirshenbaum's GP with iteration [18], Generic Genetic Programming [35], Object Oriented Genetic Programming [1], and Genetic Programming with *automatically defined functions* [20] ("GP with ADFs" in Table 3(c)).[12] In Table 3(c), "Evals" is the minimum effort to solve the problem (E) and "Fitness Cases" is $E \times T$, where T is the number of test cases per evaluation (see Section 5.3, Comparing Effort). Smaller numbers are better.

Table 3. The parameters, built-in values, and results for the even-parity problem.

(a) Parameters

Parameter	Value
max-expression-size	11
max-phrases	9
population-size	500
num-iterations	50,000
num-trials	60
tries-to-be-unique	50

(b) Built-in Values

Value	Type
`true`	`Bool`
`false`	`Bool`
`and`	`Bool` $\rightarrow$ `Bool` $\rightarrow$ `Bool`
`or`	`Bool` $\rightarrow$ `Bool` $\rightarrow$ `Bool`
`nor`	`Bool` $\rightarrow$ `Bool` $\rightarrow$ `Bool`
`nand`	`Bool` $\rightarrow$ `Bool` $\rightarrow$ `Bool`
`head`	`List`$(\alpha) \rightarrow \alpha$
`tail`	`List`$(\alpha) \rightarrow$ `List`(α)
`foldl`	$(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow$ `List`$(\alpha) \rightarrow \beta$

(c) Results Comparison

Representation	Evals	Fitness Cases
Brute Force	13,006	156,072
PolyGP	14,000	168,000
Combinators	58,994	707,928
Kirshenbaum	60,000	6,000,000
Generic GP	220,000	1,760,000
OOGP	680,000	8,160,000
GP with ADFs	1,440,000	184,320,000

[11] If a solution fails 0 test cases, it is correct.

[12] Koza solved many different incarnations of the even-parity problem. The effort listing for Koza [20] is the same one that Yu [37] listed.

```
fun fitness (push, pop, emptyStack, isEmpty, top) =
  let fun test er b = if b then er else er + 1
      val er = 0
      val s1 = emptyStack
      val er = test er (isEmpty s1)
      val s2 = push s1 3
      val er = test er (not (isEmpty s2))
      val v1 = pop s2
      val er = test er (isEmpty v1)
      val v2 = top s2
      val er = test er (v2 = 3)
  in  er end
```

Fig. 7. The fitness function for stack.

Table 3(c) appears to be divided into two groups of approaches; those with implicit recursion require less effort, and other approaches require more effort.

6.3 Stack Data Structure

Langdon [21, 22] showed that genetic programming can evolve implementations of the stack and queue data structures. Genetic programming with combinator expressions finds implementations of the stack and queue with less computational effort than Langdon.

The interface for a stack includes five members: `push`, `pop`, `top`, `emptyStack`, and `isEmpty`. To implement a stack, we need to find expressions for each of these functions and values (`emptyStack` is not necessarily a function). Langdon [21] evolved each of these expressions separately (see Section 9, Related Work). With generate, no special cases in the genetic operators are necessary to evolve each part of the solution; since generate can create an expression of an arbitrary type, rather than making five separate expressions (like Langdon [21]), we request a product containing five elements, each of the appropriate type.

To enable generate to make such a product, we provide it with a function that takes each of the product's elements as arguments and returns the product. For example, `product3 1 false true` is an expression with type `Product(Int, Bool, Bool)`. Similarly, `product5` makes a product with five elements.

Figure 7 lists the fitness function for the stack problem, which takes the form of a brief unit test in Mini-ML. The fitness function starts with an empty stack, pushes an `Int` onto it, then pops that `Int` off of the stack. At the beginning of the fitness function is the definition of the function `test`, which returns its input `Int` incremented by one if its input `Bool` is false, and its input unchanged otherwise. The fitness of a stack implementation is `er`, the number of tests that it fails. For the purpose of calculating effort, each call to `test` is a test case within an evaluation. There are four test cases in this fitness function. Table 4(a) and Table 4(b) list the parameters and built-in values for the stack problem.

The type system infers that the types of `push`, `pop`, `emptyStack`, `isEmpty`, and `top` are $\alpha \rightarrow$ `Int` $\rightarrow \alpha$, $\alpha \rightarrow \alpha$, α, $\alpha \rightarrow$ `Bool`, and $\alpha \rightarrow$ `Int`, respectively.

Here, α means the type that internally represents a stack. For example, the `push` function takes a stack and an `Int`, and returns a new stack with the `Int` added to it. The way in which the fitness function uses `push` and `top` dictate that the stack holds `Int`s. Through evolution and type constraints, the system arrives at a representation for the queue that is based on a `List(Int)`. The fitness function does not specify how to represent the stack. Type inference determines that there is an unknown type, α that will represent the stack. Evolution must find an appropriate assignment of the type variable α (i.e., find an internal representation stacks). It does so, discovering that `List(Int)` is appropriate.

One of the evolved solutions is `product5 (C cons) tail nil isempty head`, which is equivalent to the SML expression `(fn x => fn y => cons y x, tail, nil, isempty, head)`. The fitness function specifies that the first element of the quintuple should be the implementation of `push`, that the second element should be `pop`, and so on. Thus, `push` is `fn x => fn y => cons y x` and `pop` is `tail`. An interesting part of the solution is that it could not just use `cons` for `push` because the arguments are in the wrong order. To flip them around, the genetic algorithm constructs the expression `C cons`.

If the genetic algorithm stops at 15 evaluations, 60 out of 60 trials find a correct solution, so $P_{suc}(15) \approx 60/60 = 1.0$ and 1 run is necessary for a 99% chance of success. The minimum effort for genetic programming is $1 \times 15 = 15$ evaluations for a 99% chance of success. Brute force finds a solution of size 8. There are 6 total expressions of size ≤ 8, so the effort required for brute force to find a solution is 6 evaluations.

Table 4. The parameters, built-in values, and results for the stack problem.

(a) Parameters

Parameter	Value
max-expression-size	30
max-phrases	9
population-size	500
num-iterations	20,000
num-trials	60
tries-to-be-unique	50

(b) Built-in Values

Value	Type
`product5`	$\alpha \to \beta \to \gamma \to \delta \to \epsilon \to$ `Product`$(\alpha,\ \beta,\ \gamma,\ \delta\ \epsilon)$
`0`	`Int`
`1`	`Int`
`true`	`Bool`
`false`	`Bool`
`cons`	$\alpha \to$ `List`$(\alpha) \to$ `List`(α)
`head`	`List`$(\alpha) \to \alpha$
`tail`	`List`$(\alpha) \to$ `List`(α)
`isEmpty`	`List`$(\alpha) \to$ `Bool`
`foldl`	$(\alpha \to \beta \to \beta) \to \beta \to$ `List`$(\alpha) \to \beta$

(c) Results Comparison

Representation	Evals	Fitness Cases
Brute Force	6	24
Combinators	15	60
Langdon	938,000	150,080,000

Table 4(c) lists the effort required to find a stack implementation by brute force, by evolving combinator expressions and by evolution using Langdon's approach [21]. Why do brute force and evolution with combinator expressions require so much less effort, than Langdon? Brute force shows us that there are few correctly typed expressions of the minimum size necessary to represent a solution. Evolution with combinators produces a result by random creation within 15 attempts (no genetic recombination occurs because the genetic algorithm finds a solution while initializing its population). Type constraints make this problem trivial for either brute force or random guessing with combinator expressions. Langdon's system had no such type constraints and relied on indexed memory to represent the stack rather than functional lists [21].

6.4 Queue Data Structure

Evolving an implementation of the queue data structure poses a greater challenge than evolving a stack, because type constraints do not mostly dictate the solution. The parameters and built-in values for the queue experiment are exactly the same as they are in the stack experiment. The only difference between the two experiments is the fitness function.

Like the stack, the fitness function for the queue is a short unit test, written in Mini-ML. It pushes three `Ints` onto the queue, then pops them back off. Interspersed between pushes and pops, it tests eight cases. The fitness of a queue implementation is the number of assertions it fails. The argument to the fitness function is a quintuple of the form `(enQ, emptyQ, QIsEmpty, deQ, front)`.

The evolved solution is `product5 (B (B (foldl cons nil)) (C (C' (C cons) (foldl cons nil)))) nil isempty tail head` which is equivalent to the SML expression `(fn x => fn y => foldl cons nil (cons y (foldl cons nil x)), nil, isempty, tail, head)`. The function `foldl cons nil` reverses the list to which it is applied, so this solution's `enQ` function could be written as `fn x => fn y => reverse(cons y (reverse x))`.

If the genetic algorithm stops at $10,046$ evaluations, 1 out of 60 trials finds a correct solution, so $P_{suc}(10,046) \approx 1/60 \approx 0.01667$ and 275 runs are neccessary for a 99% chance of success. The minimum effort for genetic programming is $275 \times 10,046 = 2,762,650$ evaluations for a 99% chance of success. Brute force is infeasible.

Table 5. A comparison of effort for the queue problem between combinators and Langdon [21].

Comparison to Other Approaches		
Representation	Evals	Fitness Cases
Combinators	2,762,650	22,101,200
Langdon	3,360,000	1,075,200,000
Brute Force	Infeasible	Infeasible

Table 5 lists the effort required to find a solution by evolving combinator expressions and by Langdon [21].

Evolving combinator expressions requires fewer evaluations to find a solution than Langdon's approach. This is a significant result, because this problem is neither trivial nor feasible for brute force.

Although the number of evaluations are fairly close, our experimental setup required many fewer fitness cases because we used eight test cases per evaluation, whereas Langdon used 320 fitness cases per evaluation. Langdon also reported an effort of $86,000,000$ evaluations to implement the queue if a particular function that is somewhat problem specific is not in the function set, and must be evolved as an *automatically defined function*. We could have made this problem nearly as easy as the stack problem by including `reverse` in the built-in values, but we did not because we wanted it to evolve.

7 Conclusion

Combinator expressions are a powerful program representation for genetic programming. Genetic programming with combinator expressions finds solutions to a linear-regression problem, the even-parity problem on N inputs, and implementations of a stack and queue. The amount of effort that evolving combinator expressions required to solve these problems compares favorably with brute-force search and with prior approaches.

Clack and Yu [5] and Kirshenbaum [19] devised ways to evolve programs that introduce local variables. However, their genetic operators had variable-specific special cases. Combinator expressions can represent programs that introduce local variables, but the genetic operators to manipulate combinator expressions do not require special cases to deal with variables.

8 Future Work

There are many ideas, questions, and issues related to genetic programming with combinator expressions that might be fruitful areas for future research.

The impact of the set of combinators that are in the built-in values on the evolution of combinator expressions remains unexplored. We used the `I`, `S`, `B`, `C`, `S'`, `B*`, and `C'` combinators because Jones [16] listed these as an appropriate basis for the implementation of an efficient, lazy interpreter. However, Katayama [17] used `S`, `B`, `C`, and a "list only" `K`.

An important distinction between the genetic operators in Section 4 and prior genetic operators is that they produce expressions that are functions, rather than expressions that are the *bodies* of functions. These operators can evolve any type of expression, not just functions. Haynes et al. [13] evolved expressions that represented sets of cliques in a graph with a strongly typed genetic-programming system. Perhaps genetic programming with combinator expressions can solve other problems with answers that are type-constrained data structures.[13]

[13] Genetic algorithms solve problems with answers that are data structures, but the programmer usually needs to implement genetic operators that are specific to the solution representation. Our idea is that combinator expressions can represent a wide variety of data structures. If a combinator expression can represent the solution to a problem, then there is no need for problem-specific genetic operators.

With a compiler from an expressive functional programming language (such as Mini-ML) into combinator expressions, it would be possible to evolve populations of combinator expressions that include code written by humans. There may be applications of this idea to optimizing compilers [31, 7].

It seems that code bloat [34], introns, and neutrality [11, 38, 6] play important roles in the dynamics of evolving populations of combinator expressions. Combinator expressions may provide useful ways to investigate these phenomena. Partial evaluation on combinator expressions can remove some, but not all introns.[14]

9 Related Work

Church and Turing developed the idea that a Turing machine or λ-expression can compute any function that is computable [32, 3, 4]. Schönfinkel [29] developed the `S` and `K` combinators to eliminate the need for variables in logic. Curry and Feys [8] further developed the field, adding the `B` and `C` combinators. Curry and Feys [8], Turner [33], Jones [16], and Sørensen and Urzyczyn [30] give algorithms to convert a λ-expression to a combinator expression. The existence of these algorithms constitutes a proof of the equivalence of the two representations. Both Turner and Jones discussed implementations of functional programming languages that compile λ-expressions into combinator expressions and run them by combinator reduction.

In Koza's original genetic programming system [20], expressions satisfy the property of "closure."[15] For an expression to satisfy the closure property, all functions (non terminals) it contains must take arguments and return values of the same type, and all constants (terminals) must be of that type. Koza offered constrained syntactic structures as a way to evolve expressions that did not satisfy the closure property. When using constrained syntactic structures, only certain terminals and non terminals can go together. The user of the system specifies which terminals and non terminals can go together. Problem-specific genetic operators maintain these syntactic constraints.

In Montana's Strongly Typed Genetic Programming (STGP) [26], one supplies syntactic constraints implicitly through a static type system. Users specify the type of each function and constant that can be incorporated into an evolved program. STGP's genetic operators always produce correctly typed expressions. STGP's type system supports generic functions in the function set, but they are instantiated to a monomorphic type that does not change during evolution. STGP can evolve parametrically polymorphic functions through the use of type variables. STGP maintains type correctness with a type-possibilities table that makes higher-order function types difficult to implement in a fully general way.

[14] An intron is a piece of code that has no effect on the behavior of the program in which it resides.

[15] The property of closure in genetic programming should not to be confused with the functional programming languages concept of closures.

Clack and Yu [5] introduced a new program representation called PolyGP, which Yu has since refined [36, 37]. PolyGP combines Montana's static typing with functional programming concepts such as λ-abstractions, higher-order functions and partial application. In order to support higher-order functions, Yu replaced Montana's type-possibilities table with a unification algorithm. PolyGP does not support closures, so the body of a λ-abstraction can refer only to the variables that the λ-abstraction introduces. PolyGP can only perform crossover between λ-abstractions that represent the same argument of the same higher-order function. These limitations arise in PolyGP because it is difficult to apply genetic operators to expressions that introduce named variables. The genetic operators in Section 4 avoid these restrictions on crossover. Combinator expressions can represent λ-expressions that would require closures.

Kirshenbaum [19] provided several new genetic operators that enable genetic programming to evolve expressions that introduce statically scoped local variables through `let` expressions. In contrast, evolving combinator expressions does not require specialized genetic operators that deal only with variables. Speaking of closures, Kirshenbaum writes that they "may be worth investigating but will necessitate changes in the way local variables are implemented."

Yu [37] and Kirshenbaum [19] used their GP systems to address several problems that depend on iteration or recursion, including the even-parity problem on N inputs. Yu showed that PolyGP could evolve polymorphic recursive functions, such as `map` and `length`.

Katayama [17] presented a system for automatic program discovery by brute-force enumeration of all correctly typed expressions in order of size. Brute force requires less effort than PolyGP to find many of the same functions (including `map` and `length`). Our experiments show that some problems are easier to solve with brute force, whereas others are easier to solve with evolution. Although derived independently, our generate algorithm has much in common with Katayama's enumeration algorithm.

Langdon [21, 22] used genetic programming with indexed memory to evolve the stack and queue data structures. In Langdon's representation, each of the functions in the implementation of a data structure is a separate expression tree. Crossover could only take place between corresponding trees. In contrast, we evolve a single expression that contains all parts of a data structure.

References

[1] Alexandros Agapitos and Simon M. Lucas. Learning recursive functions with object oriented genetic programming. In *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 166–177, Budapest, Hungary, 10 - 12 April 2006. Springer.

[2] C.-L. Chang and R.C.-T. Lee. *Symbolic Logic and mechanical Theorem Proving.* Academic Press, 1973.

[3] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33:346–366, 1932.

[4] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[5] Chris Clack and Gwoing Tina Yu. Performance enhanced genetic programming. In *Evolutionary Programming VI*, pages 87–100, Berlin, 1997. Springer.

[6] M. Collins. Finding needles in haystacks is harder with neutrality. In *GECCO*, pages 1613–1618. ACM, 2005.

[7] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *LCTES '99: Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, pages 1–9, New York, NY, USA, 1999. ACM Press.

[8] H. B. Curry and R. Feys. *Combinatory Logic.* North-Holland, Amsterdam, 1958.

[9] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982.

[10] Kenneth De Jong. Learning with genetic algorithms: An overview. *Machine Learning*, 3:121, 1988.

[11] Marc Ebner, Patrick Langguth, Juergen Albert, Mark Shackleton, and Rob Shipman. On neutral networks and evolvability. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 1–8, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 May 2001. IEEE Press.

[12] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison Wesley, 1989.

[13] Thomas D. Haynes, Dale A. Schoenefeld, and Roger L. Wainwright. Type inheritance in strongly typed genetic programming. In *Advances in Genetic Programming 2*, chapter 18, pages 359–376. MIT Press, Cambridge, MA, USA, 1996.

[14] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. AMS*, 146:29–60, 1969.

[15] S. Peyton Jones and et al, editors. *Haskell 98 Language and Libraries, the Revised Report.* CUP, April 2003.

[16] Simon P. Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[17] Susumu Katayama. Power of brute-force search in strongly-typed inductive functional programming automation. In *PRICAI*, volume 3157 of *Lecture Notes in Computer Science*, pages 75–84. Springer, 2004.

[18] Evan Kirshenbaum. Iteration over vectors in genetic programming. Technical Report HPL-2001-327, HP Laboratories, December 17 2001.

[19] Evan Kirshenbaum. Genetic programming with statically scoped local variables. In *GECCO*, pages 459–468, 2000.

[20] John R. Koza. *Genetic Programming.* MIT Press, 1992.

[21] William B. Langdon. Evolving data structures with genetic programming. In *Proc. of the Sixth Int. Conf. on Genetic Algorithms*, pages 295–302, San Francisco, CA, 1995. Morgan Kaufmann.

[22] William B. Langdon. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, volume 1 of *Genetic Programming*. Kluwer, Boston, 24 April 1998.

[23] Sean Luke. Genetic programming produced competitive soccer softbot teams for robocup97. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 214–222, University of Wisconsin, Madison, Wisconsin, USA, 22-25 1998. Morgan Kaufmann.

[24] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[25] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML,* Revised edition. MIT Press, 1997.

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

[27] Tim Perkis. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 148–153, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[28] Benjamin Pierce. *Types and Programming Languages.* The MIT Press, 2002.

[29] Moses Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924.

[30] M. Sørensen and P. Urzyczyn. Lectures on the curry-howard isomorphism, 1998.

[31] M. Stephenson, U. O'Reilly, M. Martin, and S. Amarasinghe. Genetic programming applied to compiler heuristic optimisation, 2003.

[32] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Procedings of the London Mathematical Society*, 42 (2):230–265, 1936.

[33] David A. Turner. SASL language manual. Technical report, University of Kent, Canterbury, U.K., 1976.

[34] Terry Van Belle and David H. Ackley. Uniform subtree mutation. In *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 152–161, Kinsale, Ireland, 3-5 April 2002. Springer-Verlag.

[35] Man Leung Wong and Kwong Sak Leung. Evolving recursive functions for the even-parity problem using genetic programming. In *Advances in Genetic Programming 2*, chapter 11, pages 221–240. MIT Press, Cambridge, MA, USA, 1996.

[36] Gwoing Tina Yu. Polymorphism and genetic programming. In *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*, pages 437–444, Las Vegas, Nevada, USA, 2000.

[37] Gwoing Tina Yu. *An Analysis of the Impact of Functional Programming Techniques on Genetic Programming.* PhD thesis, University College, London, 1999.

[38] Tina Yu, Julian F. Miller, Conor Ryan, and Andrea Tettamanzi. Finding needles in haystacks is not hard with neutrality. In *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 13–25, Kinsale, Ireland, 2002. Springer-Verlag.