Functional Genetic Programming and Exhaustive Program Search with Combinator Expressions

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

August 19, 2007

Abstract

Using a strongly typed functional programming language for genetic programming has many advantages, but evolving functional programs with variables requires complex genetic operators with special cases to avoid creating ill-formed programs. We introduce combinator expressions as an alternative program representation for genetic programming, providing the same expressive power as strongly typed functional programs, but in a simpler format that avoids variables and other syntactic clutter. We outline a complete genetic-programming system based on combinator expressions, including a novel generalized genetic operator, and also show how it is possible to exhaustively enumerate all well-typed combinator expressions up to a given size. Our experimental evidence shows that combinator expressions compare favorably with prior representations for functional genetic programming and also offers insight into situations where exhaustive enumeration outperforms genetic programming and vice versa.

1 Introduction

Genetic programming is a powerful technique for evolving programs, with many applications in artificial intelligence [9, 17, 19], from controlling robotic soccer players [20] to modeling biological processes [29]. At its heart, a genetic-programming system is a genetic algorithm, where programs form the genome of the organisms in the evolutionary system. Because our choice of program representation influences both the overall design of the evolutionary system and the scope of the programs that can be evolved, choosing a good program representation is important for successful genetic programming.

Typed, functional program representations are well suited for genetic programming and other methods of program search [24, 39, 14, 25]. A strong type system can eliminate a very large number of ill-formed programs from the search space, and the expressiveness of functional programming languages makes it possible to represent complex programs concisely. But a high-level language intended for human programmers also presents some challenges to a genetic-programming system.

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

(a) Parent 1

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

(b) Parent 2

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

(c) Invalid Child

Figure 1: An example of invalid crossover.

Most programming languages, including functional languages, provide variables, but variables add complexities to a genetic programming system. Figure 1 shows one example of the difficulties that variables cause for genetic operators. A naïve crossover operator could produce the code in Figure 1(c) by replacing the subexpression `x * 7` from Figure 1(b) with the subexpression `x / y` from Figure 1(a). This result is invalid because `foobar` does not define `y`.

In functional programming languages there are at least three common ways to introduce local variables: `let`-expressions (which define local variables), function definitions (where variables represent function parameters), and `case` expressions (which decompose structured types into their constituent parts). If we build a genetic-programming system using these syntactic forms, its genetic operators must correctly handle each kind of variable. Doing so complicates the genetic operators, requiring strategies for introducing (and possibly eliminating) variables (and named functions), as well as strategies to handle or avoid ill-formed programs such as our example [16]. An alternative to this approach is to avoid the problems of variables by avoiding variables themselves.

In this paper, we show that *combinator expressions* are a useful program representation for genetic programming, because they provide the expressive power of high-level strongly-typed functional programming languages while avoiding the problems of variables *by eliminating them*. Specifically,

- 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 that can evolve statically typed combinator expressions;
- We show that generating combinator expressions is efficient compared to the works of Yu [39], Kirshenbaum [15], Agapitos and Lucas [1], Wong and Leung [37], Koza [17], Langdon [18], and Katayama [14] on several problems: even parity on N inputs, and devising representations and implementations for stacks and queues;

```
let fun double x = x + x
in  double (double 2)
end
```

Figure 2: A simple functional program.

- We compare the effort required by genetic programming and exhaustive program enumeration to solve six problems using the combinator-expression program representation.

Combinator expressions are not new—their origins date back to the early history of computer science. Our contribution is recognizing that they are well suited for genetic programming and developing a framework in which they may be used for that purpose.

2 Background: Combinator Expressions

Programs in functional languages such as Standard ML [23] and Haskell [27] can be simplified to λ-expressions [26], which, in turn, correspond to *combinator expressions* [7, 26, 31]. For example, the functions `foo` and `bar` from Figure 1(a) and (b) can be written in combinator form as

```
foo  ≡  S' (S plus) div (C plus 3)
bar  ≡  C' plus (C times 7) 1
```

In this representation, crossover between expressions avoids the complexities of variables; for example, the expressions `S' (S plus) plus (C times 7)` and `S' (S plus) (C times) (C' plus (C times 7) 1)` are two possible ways we might combine `foo` and `bar`.

In this section, we provide the background to explain how the above combinator expressions can actually be equivalent to the functions from Figure 1(a) and (b), and lay the groundwork for understanding how genetic programming on this representation can be implemented.

2.1 λ-Expressions

λ-Expressions are recursively defined as expressions that match one of the following forms, where x, y, and z are arbitrary variables and M, N, and O are arbitrary λ-expressions:

- $\lambda x.M$ — A λ-abstraction (i.e., an unnamed function, which introduces the variable x);
- x — A variable (used in λ-abstraction bodies to represent arguments);
- $M\,N$ — The function M applied to N.

In practice, the pure λ-calculus above is usually augmented with constants (e.g., integer constants) and built-in functions (e.g., math primitives).

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

Figure 3: Standard ML code to sum a list.

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

Figure 4: Code to sum a list using the `Y` combinator.

Programs written in a human-readable language such as Standard ML can be translated to their more baroque λ-calculus equivalent by following transformation rules.[1] For example, the program shown in Figure 2 has a λ-calculus equivalent: Given a built-in `plus` function to perform addition, we can define an anonymous function that doubles its input with the λ-expression $\lambda x.$`plus` x x. Additionally, we can replace the `let` expression in Figure 2 with a λ-abstraction, allowing us to write our entire program as $(\lambda d.d\ (d\ \texttt{2}))\ (\lambda x.\texttt{plus}\ x\ x)$.

We can also eliminate `if` and `case` statements. For `case` statements on lists, we translate `case` statements to calls to a `listcase` built-in function as follows:

$$\begin{array}{l}\texttt{case}\ L\ \texttt{of} \\ \quad\ \texttt{nil}\ \texttt{=>}\ E_1 \\ \ \ \mid\ \ v_1\ \texttt{::}\ v_2\ \texttt{=>}\ E_2\end{array} \quad \rightarrow \quad \texttt{listcase}\ L\ E_1\ (\lambda v_1.\lambda v_2.E_2)$$

The `listcase` built-in takes three (curried) arguments: the list to match, the value to compute if the list is empty, and a function to call (passing in the head and tail of the list) if the list is nonempty. We can use a similar transformation to turn `if` statements into calls to a built-in `cond` function.

Finally, in the λ-calculus there is no explicit recursion, but all recursive programs can be written nonrecursively by calling a helper function, the `Y` *combinator*. Figure 4 adapts the code from Figure 3 to use the `Y` combinator—notice that `sum_nr` is a nonrecursive function (it does not call itself, it calls `sum_rec`, which is its first argument). Figure 5 defines the behavior of the `Y` combinator. The `Y` combinator itself can actually be defined in the pure λ-calculus as $\lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))$.[2]

[1] We present this transformation to show the correspondence between the λ-calculus forms and their more human-readable equivalent forms. In our genetic-programming system, we form combinator expressions directly, and thus do not require this transformation. We do, however, use this transformation in ancillary parts of our system, such as transforming a user-specified fitness function into combinator form for execution.

[2] Or alternatively, slightly more briefly as $\lambda f.(\lambda x.x\,x)(\lambda x.f(x\,x))$.

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

Figure 5: The definitions of several useful combinators.

Applying all of the preceding rules, our `sum` function can be written as an anonymous function in the λ-calculus as

$$\texttt{Y}\ (\lambda s.\lambda l.\texttt{listcase}\ l\ \texttt{0}\ (\lambda h.\lambda t.\texttt{plus}\ h\ (s\ t)))$$

If we were to use λ-expressions as our program representation, the regular form of λ-expressions would make all aspects of the system implementation simpler as compared to using a more typical functional-programming language. In particular, there would be only one mechanism by which variables are introduced. But we can simplify our format for expressions further yet.

2.2 Eliminating Variables

All λ-expressions that represent evaluatable expressions can be translated to combinator expressions. Combinator expressions represent programs entirely by function application alone; they contain no λ-abstractions and no variables. The job of variables, namely directing function arguments to their intended destinations, and storing values that are to be used in multiple places, is instead performed by a small set of built-in functions, such as the ones shown in Figure 5 (only `S` and `K` are strictly necessary, but the other combinators allow a more compact translation). An arbitrary λ-expression can be translated into combinator form using the mechanical translation rules given in Figure 6.

Continuing our `sum` example, applying the **Trans** transformation to the λ-expression from the previous section yields

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

Because all λ-expressions can be translated to combinator expressions, and programs written in functional languages such as Standard ML can be translated to λ-expressions, combinator expressions can represent arbitrary functional programs.

One downside of using combinator expressions is that the meaning of an expression, such as `C B (S plus I)`, may not be as clear to a human reader as the function $\lambda f.\lambda x.f\ (\texttt{plus}\ x\ x)$ in the λ-calculus. However, it is possible

$$
\begin{array}{rcll}
\mathbf{Trans}[\![\, x \,]\!] & = & x & \\
\mathbf{Trans}[\![\, M\, N \,]\!] & = & (\mathbf{Trans}[\![\, M \,]\!])(\mathbf{Trans}[\![\, N \,]\!]) & \\
\mathbf{Trans}[\![\, \lambda x.x \,]\!] & = & \texttt{I} & \\
\mathbf{Trans}[\![\, \lambda x.\lambda y.M \,]\!] & = & \mathbf{Trans}[\![\, \lambda x.\mathbf{Trans}[\![\, \lambda y.M \,]\!] \,]\!] & \\
\mathbf{Trans}[\![\, \lambda x.M \,]\!] & = & \texttt{K}\ (\mathbf{Trans}[\![\, M \,]\!]) & \textbf{if } x \notin \mathbf{FV}[\![\, M \,]\!] \\
\mathbf{Trans}[\![\, \lambda x.N\, x \,]\!] & = & \mathbf{Trans}[\![\, N \,]\!] & \textbf{if } x \notin \mathbf{FV}[\![\, N \,]\!] \\
\mathbf{Trans}[\![\, \lambda x.N\, O \,]\!] & = & \texttt{B}\ (\mathbf{Trans}[\![\, N \,]\!])\ (\mathbf{Trans}[\![\, \lambda x.O \,]\!]) & \textbf{if } x \notin \mathbf{FV}[\![\, N \,]\!] \\
\mathbf{Trans}[\![\, \lambda x.N\, O \,]\!] & = & \texttt{C}\ (\mathbf{Trans}[\![\, \lambda x.N \,]\!])\ (\mathbf{Trans}[\![\, O \,]\!]) & \textbf{if } x \notin \mathbf{FV}[\![\, O \,]\!] \\
\mathbf{Trans}[\![\, \lambda x.N\, O \,]\!] & = & \texttt{S}\ (\mathbf{Trans}[\![\, \lambda x.N \,]\!])\ (\mathbf{Trans}[\![\, \lambda x.O \,]\!]) & \\
 & & & \\
\mathbf{FV}[\![\, x \,]\!] & = & \{x\} & \\
\mathbf{FV}[\![\, M\, N \,]\!] & = & \mathbf{FV}[\![\, M \,]\!] \cup \mathbf{FV}[\![\, N \,]\!] & \\
\mathbf{FV}[\![\, \lambda x.M \,]\!] & = & \mathbf{FV}[\![\, M \,]\!] - \{x\} &
\end{array}
$$

Figure 6: Translating λ-expressions to combinator expressions.

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

Figure 7: Running a combinator expression.

to transform combinator expressions into λ-expressions by an algorithm that is analogous to the one in Figure 6, but in reverse.[3]

2.3 Running Combinator Expressions

Genetic-programming systems need to run the programs they generate. *Combinator reduction* [35, 26] is an efficient technique for executing combinator expressions. Although this technique has been described at length elsewhere, we can cover its essence here.

The rules given in Figure 5 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 all of 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 the rule for `K` given in Figure 5. To run a combinator expression, we repeatedly consider the outermost function application and attempt to reduce it. Figure 7 shows an example of running `S add I 7`.

It is worth noting that combinator reduction is a *lazy-evaluation* strategy. Function arguments are only evaluated when they are needed. Lazy evaluation is an advantageous technique in genetic programming because more programs terminate under lazy evaluation than under strict evaluation.

[3]We used such an algorithm to find the human-readable equivalents of evolved combinator expressions.

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`

2.4 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 [13, 22, 8], which forms the basis for the type systems of most modern functional languages, such as Standard ML [23] and Haskell [27]. The key ideas behind this type system are *parametric types* and *type inference* via *unification*.

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 as `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 (which we will denote with greek letters). For example, the type of the `length` function is `List`(α) $\to$ `Int`. Here α is a type variable.Type variables are implicitly universally quantified (i.e., `List`(α) $\to$ `Int` is short for $\forall\alpha$, `List`(α) $\to$ `Int`).

The Hindley–Milner system infers the type of an expression using unification [4, 26, 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 variables, the two types are equal. For example, `List`(α) unifies with `List(Int)` if $\alpha =$ `Int`. `List`(α) cannot 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` $\to$ `Int`) $\to$ `Int` $\to$ `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` $\to$ `Int`.

3 Genetic Operators

In order to evolve combinator expressions, it is necessary to generate random expressions, and to have genetic operators that recombine parent expressions to make similar children (usually mutation and crossover). This section discusses an implementation of genetic operators for combinator expressions.

3.1 Generating Combinator Expressions

Whether we are creating a new combinator expression from scratch (random creation), or a new expression based on existing expressions (mutation and crossover), our needs are similar—to assemble expression fragments into a single well-typed expression. We use a single function, generate, to provide this facility, where generate(τ, L) assembles a well-typed expression of type τ using function application to connect *phrases* taken from the library L. A phrase is a value of a known type, either a simple built-in value, or a larger prebuilt expression.

The problem addressed by the generate algorithm is strongly analogous to theorem proving in the intuitionistic propositional calculus [31]. The correspondence between computer programs and mathematical proofs, known as the Curry–Howard isomorphism, can be described as follows for our problem: Suppose we have the functions $f : \alpha \to \beta$ and $g : \beta \to \gamma$. If we interpret the types of f and g, $\alpha \to \beta$ and $\beta \to \gamma$, as given theorems, we can prove the theorem $\alpha \to \gamma$. Suppose α is true; by rule f, β is true; by rule g, β implies that γ is true; and, therefore, $\alpha \to \gamma$. Analogously for combinator expressions, given a value x of type α, $f\,x$ yields a value of type β, and $g\,(f\,x)$ yields a value of type γ.

The needs of the generate function differ from those of a conventional theorem prover in two significant ways. Whereas short proofs are desirable, short expressions need not be. It is generate's task to return a randomly chosen expression from those that are possible, rather than a single "best" expression.[4] In addition, in theorem proving we usually desire a proof if one exists and will wait for it, but in genetic programming, we may be willing to trade completeness for performance. If generate does not generate some very complex expressions for performance reasons, the chances are that no harm will be done, because complex expressions can usually be evolved over subsequent generations rather than produced by generate in a single step.

Our relatively simple implementation of generate(τ, L) operates 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` $\to$ `Int`, using Table 1 as our library, one possibility is for generate to choose `I`, as `I` has a type that matches ($\alpha \to \alpha$ matches if $\alpha =$ `Int`).

[4] Or, in the case of exhaustive enumeration, *all* possible expressions.

There are more possible ways to make an `Int` → `Int` function, however. In functional programming, multiargument functions are usually curried [7] 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` → `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` → `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` → β → `Int` and an `Int` → β. Using `add` as the first argument defines β = `Int`, leaving us seeking an `Int` → `Int` value for the second argument; `I` is an acceptable choice. Thus, `S add I`, which corresponds to $\lambda x.\texttt{add}\, x\, x$, is another possible result.

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.

3.2 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;[5]
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 point-mutation or crossover operators [17] can, as well as some expressions that those operators would be very unlikely to make. Many genetic-programming systems have distinct mutation and crossover operators, which are associated with free parameters that determine how often they occur. Using only the generalized genetic operator eliminates these parameters (but adds a new parameter, *max-phrases*). There is still some probability that the output of the generalized operator will be an expression that crossover and/or point mutation could make, but these probabilities are implicit in the behavior of the algorithm, rather than explicit parameters.

[5]For an expression with n nodes, there are exactly $2n - 1$ subexpressions. For a linearly structured expression, the average subexpression length is $O(n)$, but more typical expressions have a tree structure resulting in an average subexpression length of $O(\log n)$. Regardless, through sharing, storing these expressions actually requires only $O(n)$ space. In our system, n is always less than *max-expression-size*, which is 20 in our experiments.

Suppose we apply the generalized genetic operator to the parent expressions `add (mult 2 3) 4` and `sub 7 9`, with the set of built-in values {`add`, `sub`, `mult`, `2`, `3`, `4`, `7`, `9`}.

First, we make a phrase for every subexpression in each parent. The nine phrases produced by the first parent are `2`, `3`, `4`, `mult`, `mult 2`, `mult 2 3`, `add`, `add (mult 2 3)` and `add (mult 2 3) 4`. The five phrases produced by the second parent are `7`, `9`, `sub`, `sub 7` and `sub 7 9`. To make a new child, we call generate with the union of both parent's phrase lists and the built-in values.

In this example, we seek to generate an `Int`. The first step in generate is to randomly pick a phrase that is an `Int` or that returns an `Int` when applied to one or more arguments. Suppose we choose `add`, which returns an `Int` when given two `Int` arguments. To provide those arguments, we recursively call generate. For the first argument, we may choose `mult 2 3`, because it is an `Int`. For the second argument, we may choose `sub 7`, which requires an integer argument, which we randomly generate as `4`. This process gives a complete expression, `add (mult 2 3) (sub 7 4)`. In this outcome, the generalized operator reproduces the effect of crossover, applying `sub 7` from the second parent to the subtree rooted at `4` in the first parent. But it could also have generated `add (mult 2 3) (mult 2 2)`, which would be equivalent to a point mutation at `4`.

The generalized operator can also make children that the point operators would be very unlikely to make. Suppose that we choose `sub` applied to two arguments as the root expression. For the first argument, let us choose the phrase `add (mult 2 3) 4`, and for the second, let us choose `3` (because these are whole phrases, there isn't any further recursion in generate). Now we have the expression `sub (add (mult 2 3) 4) 3`. Point mutation and crossover would not be likely to make this expression, because it embeds the first parent as a subtree of a new root.[6]

4 Exhaustive Enumeration

Genetic programming is not always the best way to automatically generate programs—sometimes, particularly in the case of *small* programs, an exhaustive enumeration of every correctly typed expression is more efficient than evolution [14]. Although, in principle, exhaustive enumeration of all valid programs can be applied to any program representation, the simple structure of combinator expressions makes them well suited for this task, and the algorithms we have developed for genetic programming can be easily adapted for this purpose.

Our depth-first enumeration algorithm for combinator expressions uses the same generate function as our generalized genetic operator. Normally, generate is asked to produce a single, randomly chosen, expression of a given type. In the case of exhaustive search, we instead enumerate *all* available

[6] A point mutation might affect a node near the root of a tree, but only by randomly generating the mutated subtree. Our approach differs in that the mutated subtree can be replaced by a phrase that consists of a multinode subtree from a parent.

expressions matching a given type and size constraint.[7]

Even with type constraints, the number of possible expressions usually grows exponentially with the size of the expression, thus exhaustive enumeration is usually only practical for relatively small expressions. By using essentially the same generate function for both exhaustive enumeration and our genetic algorithm, we can explore the trade-offs between evolutionary programming and exhaustive search in the context of automatically generating programs.

5 Experimental Setup

Before we can provide experimental data to substantiate our claim that combinator expressions are a useful program representation for genetic programming and exhaustive search, we must complete our description of our genetic-programming system by detailing our genetic algorithm and other aspects of our experimental setup. Our genetic algorithm is not intended to be novel—we provide the details only to give a complete account of our experiments.

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 [19], our genetic algorithm uses tournament selection and steady-state replacement [33].

Our genetic algorithm draws inspiration from Langdon [19] and Yu [39] by attempting never to evaluate the same expression twice. Whenever a genetic 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.[8] 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.

[7] The difference in generate for exhaustive enumeration is that after Step 2, it always starts over until there are no further possibilities.

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

Table 2: The parameters for the genetic algorithm.

Parameter	Value
max-expression-size	20
max-phrases	9
population-size	500
tournament-size	4
num-iterations	20,000
num-trials	60
tries-to-be-unique	50

3. Apply the genetic operator to these parents to produce a new expression. Evaluate the fitness of the new expression. If the expression 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.[9]

5. If the algorithm has iterated less than *num-iterations* times, go back to Step 2. Otherwise, stop.

We used the same parameters for the genetic algorithm in all experiments. Those parameters are listed in Table 2.

5.2 Problem Specification

Like Katayama [14], 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 Standard ML that we call Mini-ML.

5.3 Combinator Library

In addition to any problem-specific built-in values, we provide generate with a library containing all the combinators from Figure 5, except for `Y` and `K`. For our experiments, `Y` is unnecessary—in some cases the evolved functions iterate using a provided function such as `foldl`, and in one case we derive `Y` from scratch. The `K` combinator is rendered largely redundant by the `B` and `C` combinators. The uses of the `K` combinator not subsumed by `B` and `C` effectively introduce an unused local variable—if ignoring a value is really required, it is usually possible for an evolved program to contrive a way, such as multiplying by zero or ingeniously using `foldl`.[10] By avoiding `K` we simply make it less easy to create expressions that contain ignored values.

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

[10] Specifically, `C' C (C' foldl (C C)) nil` ≡ `K`.

Finally, we also eliminate from our library the option to create *fully applied* variants of each of our combinators. Doing so eliminates the generation of redundant forms such as `I` e and `I (I` e`)`, where e is some arbitrary expression, which can be expressed more simply as e (because `I` is the identity function). Similarly, there is little point in generating `C` e_1 e_2 e_3 when we will can more directly generate the shorter equivalent expression e_1 e_3 e_2, and likewise for `B`.

5.4 Runtime Errors

Several kinds of errors can occur when evaluating a combinator expression, such as taking the `head` of an empty list, dividing by zero, 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.

5.5 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 [17]. 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 of finding a correct solution is $R(P_{suc}(n)) = \lceil \ln(1 - 0.99) / \ln(1 - P_{suc}(n)) \rceil$ [17]. 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.[11]

Different authors use 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.

5.5.1 Effort for Exhaustive Enumeration

We need a measure of effort for exhaustive enumeration that is comparable to E, as defined in the preceding section. We could use the number of expressions that our depth-first–search algorithm tries before finding a correct solution, but doing so risks a result that is overly specific to details of our implementation (such as the order in which it stores built-in values) rather than to exhaustive enumeration in general. Not only can different implementations enumerate expressions in different orders (while being otherwise

[11] 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.

equivalent), but a single randomized algorithm might cover the search space in different orders from run to run. Thus we use a metric that depends on the results of exhaustive enumeration (up to a given expression size), but not on their particular order.

First, let us define effort for non–size-first exhaustive enumeration in which all expressions up to some maximum size are listed in arbitrary order. If we find one solution in a search yielding all n expressions of size $\leq k$, we define the expected number of evaluations for a 99% likelihood of finding the correct solution in that search as $n \times 0.99$. But if those n expressions contain s solutions, the expected number of evaluations required to have a 99% chance of finding a solution is $n(1 - \sqrt[s]{1 - 0.99})$.

In size-first enumeration, we list all expressions of a given size before listing any expressions of the next largest size. Thus if the first solution is of size k, we must have iterated through all m expressions of size $< k$ before we stand any chance of finding that solution. If there are s solutions and n expressions of size k, the expected effort is $m + n(1 - \sqrt[s]{1 - 0.99})$

If our exhaustive enumeration algorithm did not find a solution after twelve hours, we stopped and declared the problem infeasible for exhaustive enumeration. (In contrast, our evolutionary algorithm completed all *sixty* trials well within this time limit for all problems.)

6 Experiments

In this section, we examine how well our genetic-programming system works in practice by presenting results from six experiments. Three of the experiments (the even-parity, stack, and queue experiments) are benchmark problems used by several authors to test genetic programming systems [1, 15, 18, 17, 37, 39]. The other three experiments (linear regression, the constrained-list problem, and the `Y`-combinator problem) are included to provide a better sense of the range of our system.

6.1 Linear Regression

Let us begin with a simple problem: linear regression from the data points $(0, 6)$, $(1, 12)$, $(2, 18)$, $(3, 24)$, $(4, 30)$. The fitness of a candidate solution is its sum-squared error on the data set. The line $y = 6x + 6$ goes directly through these data points, and thus the goal in this problem is to discover a function corresponding to this line, given these points. Table 3 lists the built-in values for this problem.

Figures 8 and 9 show three of the evolved solutions, in combinator and human-readable form (where `f` is the desired function), respectively. The first solution is the shortest one found by genetic programming, the other two are arbitrarily chosen and more representative. These results are revealing in several ways. First, output from genetic programming *can* be as short as human-written code, although it is more likely to contain some redundant

Table 3: Built-in values for the linear-regression problem.

Value	Type
`0`	`Int`
`1`	`Int`
`add`	`Int` → `Int` → `Int`
`times`	`Int` → `Int` → `Int`
`inc`	`Int` → `Int`

```
B (times (times (inc (inc one)) (inc one))) inc
```

```
C' (C' times (B* (C (S times) (inc (inc zero))) (B* I inc) add))
   (add one)
   zero
```

```
C' (S B* (S add)) (C times) (B (C times one) inc)
   (times one (inc one))
```

Figure 8: Three evolved solutions to the linear-regression problem.

```
fun f x = times (times (inc (inc one)) (inc one)) (inc x)
```

```
fun f x =
    let val y = inc (inc zero)
    in  times (times y (inc (add zero y))) (add one x)
    end
```

```
fun f x =
    let fun g y = times y (times one (inc one))
        val z   = times (inc x) one
    in g (add z (g z))
    end
```

Figure 9: A human-readable version of Figure 8.

code. Second, the latter two evolved solutions correspond to human-written code that defines local variables: the second evolved solution shown defines a value and uses it twice; and the third solution also defines and reuses a local function. Thus, although evolved combinator expressions contain no variables themselves, they correspond to expressions that use variables in meaningful ways.

If the genetic algorithm stops at 2866 evaluations, 60 out of 60 trials find a correct solution, so $P_{suc}(2866) \approx 60/60 = 1.0$ and 1 run is necessary for a 99% probability of success. The minimum effort is $1 \times 2866 = 2866$

Table 4: Built-in values for the even-parity problem.

Value	Type
`true`	`Bool`
`false`	`Bool`
`and`	$\texttt{Bool} \to \texttt{Bool} \to \texttt{Bool}$
`or`	$\texttt{Bool} \to \texttt{Bool} \to \texttt{Bool}$
`nor`	$\texttt{Bool} \to \texttt{Bool} \to \texttt{Bool}$
`nand`	$\texttt{Bool} \to \texttt{Bool} \to \texttt{Bool}$
`head`	$\texttt{List}(\alpha) \to \alpha$
`tail`	$\texttt{List}(\alpha) \to \texttt{List}(\alpha)$
`foldl`	$(\alpha \to \beta \to \beta) \to \beta \to \texttt{List}(\alpha) \to \beta$

```
fun foldl f accum list =
    case list of nil    => accum
               | h :: t => foldl f (f h accum) t
```

Figure 10: The `foldl` function for iterating over lists.

evaluations for a 99% probability of success.

Exhaustive enumeration finds 21 solutions from a total of 856,668 expressions of size 9 and no solutions in 143,188 expressions of size < 9. Thus the expected number of evaluations required for size-first enumeration to find a solution with 99% certainty is $143{,}188 + 0.20 \times 856{,}668 = 311{,}879$ (because $(1 - 0.20)^{21} \approx 1 - 0.99$).

6.2 Even Parity

Koza [17] 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)` $\to$ `Bool`. Like Yu [39], we use twelve test cases, which comprise every list of two or three boolean values. The fitness of a potential solution to this problem is the number of test cases that it fails.[12] Table 4 lists the built-in values for the even-parity problem. These values include the `foldl` function (sometimes also called `reduce`), which provides a mechanism to iterate over lists. An implementation of `foldl` is shown in Figure 10.

One of the evolved solutions is

```
B* (foldl (S' (S' and) or nand) true) I I
```

which is equivalent to the Standard ML expression

```
foldl (fn x => fn y => and (or x y) (nand x y)) true
```

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

Table 5: Results comparison for the even-parity problem.

Approach	Evals	Fitness Cases
Exhaustive Enumeration	9478	113,736
PolyGP	14,000	168,000
GP with Combinators	58,616	703,392
GP with Iteration	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

Our genetic algorithm found 23 solutions within the 20,000 iteration limit imposed by *num-iterations*. From our experimental data, we find that if the genetic algorithm stops at 431 evaluations, 2 out of 60 trials find a correct solution, so $P_{suc}(431) \approx 2/60 = 0.033$ and 136 runs are necessary for a 99% probability of success. The minimum effort is $136 \times 431 = 58{,}616$ evaluations for a 99% probability of success.

Exhaustive enumeration finds 4 solutions from a total of 11,114 expressions of size 7 and no solutions in 1878 expressions of size < 7. Thus the expected number of evaluations required for size-first enumeration to find a solution with 99% certainty is $1878 + 0.68 \times 11{,}114 = 9478$ (because $(1 - 0.68)^4 \approx 1 - 0.99$).

Table 5 lists the effort required to find a solution to the even-parity problem using combinator expressions with exhaustive enumeration and evolution (listed as "GP with Combinators" in the table). The table also shows the performance of PolyGP [39], GP with iteration [15], Generic Genetic Programming [37], Object Oriented Genetic Programming [1], and Genetic Programming with Automatically Defined Functions [17].[13] In the table, 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 5.5, Comparing Effort). Smaller numbers are better.

6.3 Stack Data Structure

Langdon [18, 19] showed that genetic programming can evolve implementations of the stack and queue data structures. The interface for a stack consists of four functions and a value: `push`, `pop`, `top`, `emptyStack`, and `isEmpty` (`emptyStack` is not a function—it is the stack containing nothing). To implement a stack, we need to find expressions for each of these parts.

[13] Koza solved many different incarnations of the even-parity problem. The effort listing for Koza [17] is the same problem incarnation that Yu [39] addressed. Where possible, we have used the same experimental parameters as Yu, but they do differ in one significant way—Yu had a *max-expression-depth* parameter, which was set to 4. We have no equivalent parameter; our closest equivalent, *max-expression-size*, was set to 20 for all our experiments.

```
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
```

Figure 11: The fitness function for stack.

Table 6: Built-in values for the stack problem.

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`
`nil`	`List`(α)
`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`$(\alpha) \to \beta$

Figure 11 lists the fitness function for the stack problem, which takes the form of a brief unit test in Mini-ML. The argument to the fitness function is a single value, a quintuple, containing the five required parts of a stack implementation. The fitness function starts with an empty stack, pushes an `Int` onto it, then pops that `Int` off of the stack. As it is performing these operations, it tests four conditions that will be true for a correct implementation of a stack, using a local function, `test`, to keep track of the number of failed tests. This helper function takes as input the number of tests failed so far and a boolean representing the success status of a new test, and returns the new number of failures. 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. Table 6 lists the built-in values for the stack problem.

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

To provide a values for the fitness function, our genetic-programming system must create quintuples of type

$$(\alpha \to \texttt{Int} \to \alpha) \times (\alpha \to \alpha) \times \alpha \times (\alpha \to \texttt{Bool}) \times (\alpha \to \texttt{Int})$$

(or, stated in the terminology of our type-system, `Product(`$\alpha \to$ `Int` $\to \alpha$`,` $\alpha \to \alpha$`,` α`,` $\alpha \to$ `Bool,` $\alpha \to$ `Int)`). To enable `generate` to make such a product, we include in our function library a function, `product5`, that takes five arguments and returns a quintuple of those argments.

Notice that this specification does not mandate any particular representation for 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 for a stack).

One of the evolved solutions is

```
product5 (C cons) tail nil isempty head
```

which is equivalent to the Standard ML expression

```
(fn x => fn y => cons y x, tail, nil, isempty, head)
```

The type signatures of all these functions show that through evolution and type constraints, the system found a `List(Int)` representation for the queue (i.e., $\alpha =$ `List(Int)`). 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 7 evaluations, 60 out of 60 trials find a correct solution, so $P_{suc}(7) \approx 60/60 = 1.0$ and one run is necessary for a 99% chance of success. The minimum effort for genetic programming is $1 \times 7 = 7$ evaluations for a 99% chance of success. Exhaustive enumeration finds exactly one solution of size 7 from a total of 2 expressions of size 7 and no smaller expressions, so the effort required for exhaustive enumeration to find a solution is 2 evaluations.

Table 7 lists the effort required to find a stack implementation by exhaustive enumeration, by evolving combinator expressions, and by evolution using Langdon's approach [18]. Exhaustive enumeration shows us that there are few correctly typed expressions of the minimum size necessary to represent a solution. Type constraints make this problem very easy for either exhaustive enumeration 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 [18].

Table 7: Results comparison for the stack problem.

Approach	Evals	Fitness Cases
Exhaustive Enumeration	2	8
GP with Combinators	7	28
Langdon	938,000	150,080,000

6.4 Queue Data Structure

Evolving an implementation of the queue data structure poses a greater challenge than evolving a stack, because a queue cannot be as trivially implemented. Like the stack, the fitness function for the queue is a short unit test, written in Mini-ML. It pushes four `Int`s onto the queue, then pops them back off. Interspersed between pushes and pops, it tests ten 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 `(isEmptyQ, enQ, headQ, deQ, emptyQ)`.

The shortest evolved solution is

```
product5 isempty
         (C' (B* (foldl cons nil)) cons (foldl cons nil))
         head tail nil
```

With the exception of `enQ`, all the operations in this solution mirror their counterparts in the stack experiment. The implementation of `enQ` is equivalent to the Standard ML function

```
fn x => fn y => foldl cons nil (cons x (foldl cons nil y))
```

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 4425 evaluations, 3 out of 60 trials finds a correct solution, so $P_{suc}(4425) \approx 3/60 = 0.05$ and 90 runs are neccessary for a 99% chance of success. The minimum effort for genetic programming is $90 \times 4425 = 398{,}250$ evaluations for a 99% chance of success. Exhaustive enumeration did not find a solution within twelve hours.

Table 8 lists the effort required to find a solution by evolving combinator expressions and by Langdon [18]. This problem requires less effort to solve with typed combinator expressions and functional lists than it does without a type system, using indexed memory.

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 easier by including `reverse` in the built-in values, but doing so allows us to solve the problem trivially using

Table 8: Results comparison for the queue problem.

Approach	Evals	Fitness Cases
GP with Combinators	398,250	3,982,500
Langdon	3,360,000	1,075,200,000
Exhaustive Enumeration	Infeasible	Infeasible

Table 9: Built-in values for the list problem.

Value	Type
`1`	`Int`
`inc`	`Int` $\rightarrow$ `Int`
`sqr`	`Int` $\rightarrow$ `Int`
`nil`	`List`(α)
`cons`	$\alpha \rightarrow$ `List`(α) $\rightarrow$ `List`(α)

exhaustive enumeration (with an effort of 2064 evaluations). Only without this helper function is the problem challenging enough to make using genetic programming necessary.

6.5 A Value that Satisfies a Simple Constraint

We have seen how genetic programming with combinator expressions can solve problems where the solution is a function (in the linear-regression and even-parity problems), or a data structure that consists of a collection of functions and values (in the stack and queue problems). We can also use combinator expressions to solve problems with solutions that are not functions. Consider the following fitness function:

```
fun fitness L = sqr((length L) - 3) + sqr((sum L) - 30)
```

Because `length` and `sum` are functions that act on values of type `List(Int)`, the type system infers that any solution to this problem must have type `List(Int)`. We can see that if `L` is a list with three elements, the first term in the fitness function is 0. If the elements of `L` sum to 30, the second term in the fitness function is 0. Thus, the solution to this problem must be a list with three elements that sum to 30. In this problem, we must evolve an expression that satisfies these constraints using only the functions and values given in Table 9. For brevity, we refer to this problem as "the constrained-list problem" in the rest of this paper.

An important distinction between the genetic operators in Section 3 and other genetic operators is that our operators can evolve any type of expression, not just functions or function bodies. Thus, we can solve this problem using exactly the same algorithms as we used to solve the linear-regression,

even-parity, stack and queue problems.[14]

One evolved solution is

```
cons (sqr (inc one))
  (cons one (cons (sqr (inc (sqr (inc one)))) nil))
```

which is equivelent to $[(1+1)^2, 1, ((1+1)^2+1)^2]$, or $[4, 1, 25]$.

If the genetic algorithm stops at 980 evaluations, 60 out of 60 trials find a correct solution, so $P_{suc}(980) \approx 60/60 = 1.0$ and one run is necessary for a 99% probability of success. The minimum effort is $1 \times 980 = 980$ evaluations for a 99% probability of success.

Exhaustive enumeration finds 6 solutions from a total of 5741 expressions of size 13 and no solutions in 4060 expressions of size < 13. Thus the expected number of evaluations required for size-first enumeration to find a solution with 99% certainty is $4060 + 0.54 \times 5741 = 7137$ (because $(1 - 0.54)^6 \approx 1 - 0.99$).

6.6 The Y Combinator

Section 2.1 introduced the `Y` combinator as the cornerstone of recursion in the λ-calculus. As we mentioned there, this combinator can be implemented as the λ-expression $\lambda f.(\lambda x.f\,(x\,x))(\lambda x.f\,(x\,x))$. In this section we address whether this combinator can be evolved.

Unfortunately, however, the `Y` combinator as we have just stated it in the λ-calculus would not be valid under the Hindley-Milner type system used by our system and by functional languages such as Standard ML and Haskell. If f has type $\alpha \to \alpha$, the subexpression $\lambda x.f(x\,x)$ has the infinite cyclic type $((((\ldots) \to \alpha) \to \alpha) \to \alpha) \to \alpha$, representing a function that when passed itself as an argument, yields an α. Although the type system prohibits this infinite cyclic type, it does allow recursive data types, which enable us to define the `Y` combinator using a type `Spiral(`α`)` to represent this infinite type, and helper functions, `wind` and `unwind`, of type

$$\begin{aligned} \texttt{wind} &: (\texttt{Spiral(}\alpha\texttt{)} \to \alpha) \to \texttt{Spiral(}\alpha\texttt{)} \\ \texttt{unwind} &: \texttt{Spiral(}\alpha\texttt{)} \to (\texttt{Spiral(}\alpha\texttt{)} \to \alpha) \end{aligned}$$

(operationally, `wind` and `unwind` are the identity function, they change the type without altering the underlying value). With these functions in place, instead of $\lambda x.f\,(x\,x)$, which would not type check, we write $\lambda x.f\,((\texttt{unwind}\,x)\,x)$, which has type `Spiral(`$\alpha$`)` $\to \alpha$ and is a suitable argument to `wind`. A full Standard ML nonrecursive implementation of the `Y` combinator is shown in Figure 12.[15] If you find this code nonobvious, you

[14] 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, such as lists of integers. If a combinator expression can represent the solution to a problem, then problem-specific genetic operators may be unnecessary.

[15] In Standard ML, the `Y` combinator would actually be a little more complex because Standard ML is strict rather than lazy. This issue need not concern us because we are developing `Y` for Mini-ML, which is lazy. A full implementation using these ideas for Standard ML can be found on the Internet in the comp.lang.ml FAQ.

```
datatype 'a spiral = Spiral of 'a spiral -> 'a
fun wind x = Spiral x
fun unwind (Spiral x) = x

val Y = fn f => (fn x => f ((unwind x) x))
                    (wind (fn x => f ((unwind x) x)))
```

Figure 12: The `Y` combinator in Standard ML syntax.

```
fun fitness ycomb =
  let val nats = ycomb (fn self => 0 :: map (fn x => x+1) self)

  in  sqr((head nats) - 0) +
      sqr((head (tail nats)) - 1) +
      sqr((head (tail (tail nats))) - 2) +
      sqr((head (tail (tail (tail nats)))) - 3)
  end
```

Figure 13: The fitness function for the `Y` combinator problem.

are not alone. Informally, we have observed that writing the `Y` combinator nonrecursively, even with `wind` and `unwind` provided, is unintuitive even for experienced functional programmers. Our question in this experiment is how difficult the problem is for a machine.

To determine whether the `Y` combinator can be evolved, we necessarily provide `wind` and `unwind`, as well as our usual set of combinators (which does not include `Y`), and ask our system to derive a function that will satisfy the fitness function shown in Figure 13. The fitness function requires a function that can serve the role of the `Y` combinator, which it uses to create an infinite list of the natural numbers, and then tests the first four elements.[16] The built-in values do not include any numeric functions at all, so its only hope is to successfully create the `Y` combinator.

One evolved solution is `C' (B (S I wind)) B (S unwind I))`, which simplifies to `B (S I wind) (C B (S unwind I))` (which is also the expression discovered by exhaustive enumeration), and is equivalent to the alternate (shorter) `Y` combinator given in the footnote in Section 2.1.

If the genetic algorithm stops at 16 evaluations, 60 out of 60 trials find a correct solution, so $P_{suc}(16) \approx 60/60 = 1.0$ and one run is necessary for a 99% probability of success. The minimum effort is $1 \times 16 = 16$ evaluations for a 99% probability of success.

Exhaustive enumeration finds 7 solutions from a total of 48 expressions of size 9 and no solutions in 3 expressions of size < 9. Thus the expected number of evaluations required for size-first enumeration to find a solution with 99% certainty is $3 + 0.48 \times 48 = 27$ (because $(1 - 0.48)^7 \approx 1 - 0.99$).

[16] With lazy evaluation, only the first four elements of the list are actually created.

Table 10: Effort for genetic programming vs. exhaustive enumeration.

Problem	Effort for GP	Effort for EE
Stack	7	2
Y Combinator	16	27
Linear Regression	2866	311,879
List	980	7137
Even Parity	58,616	9478
Queue	398,250	Infeasible

7 Genetic Programming vs. Exhaustive Enumeration

Table 10 compares the effort required by genetic programming and exhaustive enumeration to solve each of the six problems that we investigated.

The stack and `Y`-combinator problems require very little effort to solve using either genetic programming or exhaustive enumeration, because type constraints limit the search space to only a few possible expressions. The genetic algorithm finds the solution while generating the initial population, so it can be viewed as a random search.

Type constraints do not restrict the search space in the linear-regression problem much, so the number of well-typed expressions increases rapidly as a function of expression size. The smallest possible solution is fairly large, so exhaustive enumeration must try many expressions before finding one that is correct. We speculate that genetic programming solves the problem with much less effort than exhaustive enumeration because the fitness landscape is smooth; children are likely to have similar fitness to their parents. Because of this characteristic, the genetic algorithm can start with a population of poor solutions, and makes small changes that gradually lead to a correct solution.

The constrained-list problem is more type-constrained than the linear-regression problem, but exhaustive enumeration still takes more effort to solve this problems than genetic programming. The solutions to both problems require nontrivial numeric expressions. We think that expressions which contain numbers are generally easier to find using evolution than exhaustive enumeration. To investigate this hypothesis, we ran a series of experiments with the built-in values `1`, `add`, `times`, and `inc`. The fitness function was `fitness x = sqr(x - N)`, where `N` = 1, 2, ..., 50 (so the goal is simply to build a number between 1 and 50 by adding, incrementing, and multiplying ones). Figure 14 shows the results for generating such expressions using genetic programming and exhaustive enumeration (using our usual effort metrics).[17] As the size of the solution increases, the effort for exhaustive

[17]Exhaustive search for expressions for 43 and 47 did not find a solution amongst the $1,209,974$ expressions of size ≤ 14; resource constraints prevented a search for expressions of size 15.

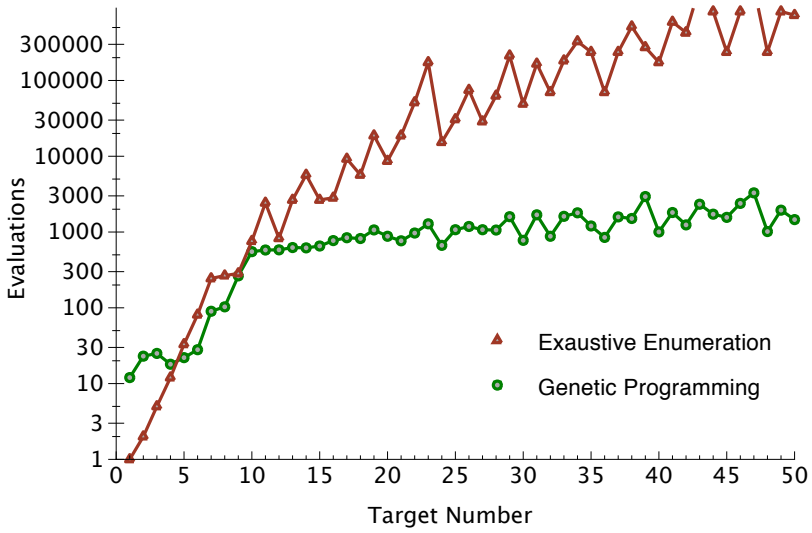

Figure 14: Effort to generate simple numeric values.

enumeration increases faster than for genetic programming.

The queue problem was infeasible for exhaustive enumeration, but feasible with genetic programming. Like the stack problem, the search space is heavily constrained by types, but unlike the stack problem, the solution is not one of the smallest valid expressions.

Exhaustive enumeration took less effort than genetic programming to solve the even-parity problem, in large part because this incarnation of the problem can be solved with a relatively short expression. Interestingly, a small change to this problem causes genetic programming to outperform exhaustive evaluation. The `if` ... `then` ... `else` variant of this problem (in which we remove the logical operators from the function set, provide a `cond` function, and keep all other aspects the same), requires an expression of size 10 rather than size 7. This change makes the problem much more difficult for exhaustive enumeration to solve, but makes a relatively small difference for genetic programming—our system solves this variant with an effort of 150,384 evaluations.

Despite our insights into when it is better to choose genetic programming over exhaustive program enumeration and vice versa, it may not be obvious which technique will be best for a new problem. But because it is straightforward to implement an exhaustive search system given a system for genetic programming with combinators, such a choice is a false dichotomy. It is practical and sensible to do both.

8 Conclusion

Combinator expressions are a useful program representation for genetic programming. They offer the power of fully general functional programs, but algorithms to manipulate combinator expressions do not require special cases to handle variables, because combinator expressions do not contain variables

(even though they can represent any expression that does contain variables).

The effort required to generate combinator-expression solutions to the even-parity problem on N inputs, and to find implementations of a stack and queue compares favorably with the works of Yu [39], Kirshenbaum [15], Agapitos and Lucas [1], Wong and Leung [37], Koza [17], Langdon [18], and Katayama [14].

Genetic programming with combinator expressions can find a solution to the constrained-list problem. This result demonstrates that combinator expressions are also a useful representation for problems with solutions that are not functions.

The `Y`-combinator experiment shows that there are some functions that can be found almost trivially by genetic programming and exhaustive enumeration, even though they are conceptually difficult for humans. It also demonstrates the applicability of genetic programming with combinator expressions to finding higher-order functions.

We hope we have inspired the reader to continue exploring combinator expressions as a program representation for genetic programming and other methods of program search. The following section gives a few possible directions for further research; and there are certainly many more.

9 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 included 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 Peyton Jones [26] listed these as an appropriate basis for the implementation of an efficient combinator-reduction machine. However, Katayama [14] used `S`, `B`, `C`, and a "list only" `K`.

With a compiler to transform code 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 [32, 6].

It seems that code bloat [36], introns, and neutrality [11, 40, 5] 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.[18]

Genetic programming and size-first search are not the only methods of automatically deriving functional programs [25]. Combinator expressions could be used in automatic programming systems that use other optimization algorithms.

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

10 Related Work

Church and Turing developed the idea that a Turing machine or λ-expression can compute any function that is computable [34, 2, 3]. Schönfinkel [30] developed the `S` and `K` combinators to eliminate the need for variables in logic. Curry and Feys [7] further developed the field, adding the `B` and `C` combinators. Curry and Feys [7], Turner [35], Peyton Jones [26], and Sorensen and Urzyczyn [31] 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 Peyton 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 [17], expressions satisfy the property of "closure."[19] 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 nonterminals can go together. The user of the system specifies which terminals and nonterminals can go together. Problem-specific genetic operators maintain these syntactic constraints.

In Montana's Strongly Typed Genetic Programming (STGP) [24], the user 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. Its 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. The genetic operators in STGP use a type-possibilities table that makes higher-order function types difficult to implement in a fully general way.

McPhee et al. [21] showed that typed genetic programming is close to or more efficient than Koza's original genetic programming on a set of problems that can be solved naturally without a type system.

Clack and Yu [4] introduced a new program representation called PolyGP, which Yu has since refined [38, 39]. 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 allow the body of a λ-abstraction to refer to any variable other than the one that it introduces, and can only perform crossover between λ-abstractions that represent the same argument of the same higher-order function, so it does not fully support evolving higher-order functions

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

(functions that return functions as their results). These limitations arise in PolyGP because it is difficult to apply genetic operators to expressions that introduce named variables. Our genetic operators in Section 3 avoid these restrictions on crossover. Combinator expressions can represent λ-expressions that would require closures (like the `Y` combinator). We think that PolyGP could not evolve the `Y` combinator, because its definition in λ-calculus requires a λ-abstraction with a body containing a variable bound in another λ-abstraction.

Kirshenbaum [16] 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 [39] and Kirshenbaum [16] 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 [14] presented a system for automatic program discovery by depth-first enumeration of all correctly typed expressions in order of size. Exhaustive enumeration 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 exhaustive enumeration, whereas others are easier to solve with evolution. Although derived independently, our generate algorithm has much in common with Katayama's enumeration algorithm.

Augustsson's *Djinn*[20] is similar to Katayama's enumeration system and our generate function, but finds a single expression that satisfies a given type constraint. Djinn uses a decision procedure for intuitionistic propositional calculus due to Dyckhoff [10], and will always (eventually) find an expression if one exists.

Langdon [18, 19] 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.

Haynes et al. [12] evolved expressions that represented sets of cliques in a graph with a strongly typed genetic-programming system. This work shows that evolving typed expression trees is an effective way to solve problems that require a data structure as the solution. We demonstrated that genetic programming with combinator expressions is applicable to such problems, by evolving a list represented as a combinator expression.

[20] Announced on the `haskell@haskell.org` mailing list in December, 2005.

References

[1] A. Agapitos and S. 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. Springer, 10–12 Apr. 2006.

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

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

[4] C. Clack and G. T. Yu. Performance enhanced genetic programming. In *Evolutionary Programming VI*, pages 87–100. Springer, 1997.

[5] M. Collins. Finding needles in haystacks is harder with neutrality. *Genetic Programming and Evolvable Machines*, 7(2):131–144, 2006. ISSN 1389-2576. doi: http://dx.doi.org/10.1007/s10710-006-9001-y.

[6] K. D. Cooper, P. J. Schielke, and D. 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. ACM Press, 1999.

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

[8] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982.

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

[10] R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57:795–807, Sept. 1992.

[11] M. Ebner, P. Langguth, J. Albert, M. Shackleton, and R. Shipman. On neutral networks and evolvability. In *Proceedings of the 2001 Congress on Evolutionary Computation (CEC 2001)*, pages 1–8. IEEE Press, 27-30 May 2001.

[12] T. D. Haynes, D. A. Schoenefeld, and R. L. Wainwright. Type inheritance in strongly typed genetic programming. In *Advances in Genetic Programming 2*, chapter 18, pages 359–376. MIT Press, 1996.

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

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

[15] E. Kirshenbaum. Iteration over vectors in genetic programming. Technical Report HPL-2001-327, HP Laboratories, Dec. 17 2001.

[16] E. Kirshenbaum. Genetic programming with statically scoped local variables. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 459–468. Morgan Kaufmann, July 2000. ISBN 1-55860-708-0.

[17] J. R. Koza. *Genetic Programming.* MIT Press, 1992.

[18] W. B. Langdon. Evolving data structures with genetic programming. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 295–302. Morgan Kaufmann, 1995.

[19] W. B. Langdon. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, volume 1 of *Genetic Programming.* Kluwer, 24 Apr. 1998.

[20] S. Luke. Genetic programming produced competitive soccer softbot teams for robocup97. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 214–222. Morgan Kaufmann, 22–25 1998.

[21] N. F. McPhee, N. J. Hopper, and M. L. Reierson. Impact of types on essentially typeless problems in GP. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 159–161. Morgan Kaufmann, July 1998. ISBN 1-55860-548-7.

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

[23] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML,* Revised edition. MIT Press, 1997.

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

[25] R. Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55–81, Mar. 1995.

[26] S. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[27] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, Apr. 2003.

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

[29] E. Sakamoto and H. Iba. Inferring a system of differential equations for a gene regulatory network by using genetic programming. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 720–726. IEEE Press, 27-30 2001. ISBN 0-7803-6658-1.

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

[31] M. H. Sorensen and P. Urzyczyn. *Lectures on the Curry–Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)*. Elsevier Science Inc., 2006. ISBN 0444520775.

[32] M. Stephenson, U.-M. O'Reilly, M. C. Martin, and S. Amarasinghe. Genetic programming applied to compiler heuristic optimization. In *Genetic Programming: 6th European Conference, EuroGP 2003*, volume 2610 of *Lecture Notes in Computer Science*, pages 231–280, 14–16 Apr. 2003.

[33] G. Syswerda. A study of reproduction in generational and steady state genetic algorithms. In G. J. E. Rawlins, editor, *Proceedings of Foundations of Genetic Algorithms Conference*, pages 94–101. Morgan Kaufmann, 1990.

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

[35] D. A. Turner. SASL language manual. Technical report, University of Kent, 1976.

[36] T. Van Belle and D. H. Ackley. Uniform subtree mutation. In *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 152–161. Springer-Verlag, 3-5 Apr. 2002.

[37] M. L. Wong and K. S. 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, 1996.

[38] G. T. Yu. Polymorphism and genetic programming. In *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*, pages 437–444, 2000.

[39] G. T. Yu. *An Analysis of the Impact of Functional Programming Techniques on Genetic Programming*. PhD thesis, University College, London, 1999.

[40] T. Yu, J. F. Miller, C. Ryan, and A. Tettamanzi. Finding needles in haystacks is not hard with neutrality. In *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *Lecture Notes in Computer Science*, pages 13–25. Springer-Verlag, 2002.