ABSTRACT

General program synthesis has become an important application area for genetic programming (GP), and for artificial intelligence more generally. Code Building Genetic Programming (CBGP) is a recently introduced GP method for general program synthesis that leverages reflection and first class specifications to support the evolution of programs that may use arbitrary data types, polymorphism, and functions drawn from existing codebases. However, neither a formal description nor a thorough benchmarking of CBGP have yet been reported. In this work, we formalize the method of CBGP using algorithms from type theory. Specially, we show that a functional programming language and a Hindley-Milner type system can be used to evolve type-safe programs using the process abstractly described in the original CBGP paper. Furthermore, we perform a comprehensive analysis of the search performance of this functional variant of CBGP compared to other contemporary GP program synthesis methods.

CCS CONCEPTS
• Software and its engineering → Genetic programming.

KEYWORDS
automatic programming, genetic programming, inductive program synthesis

ACM Reference Format:
Edward Pantridge, Thomas Helmuth, and Lee Spector. 2022. Functional Code Building Genetic Programming. In Genetic and Evolutionary Computation Conference (GECCO ‘22), July 9–13, 2022, Boston, MA, USA. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3512290.3528866

1 INTRODUCTION

A variety of genetic programming (GP) methods have been developed for general program synthesis applications, in which the goal is to automatically produce programs of the kind that humans routinely write [25]. Such programs usually make use of a range of data and control structures.

Among the most successful recent approaches are those using a multi-type stack-based language called Push [12, 26] and those rooted in the use of grammars, such as Grammatical Evolution (GE) [14] and Grammar Guided Genetic Programming (G3P) [5, 6].

In the Push approach, evolved programs are expressed in a novel language that was designed to allow for the use of arbitrary data and control structures without imposing constraints on program syntax. This is accomplished by arranging for the passing of arguments between instructions to take place via typed data stacks. The result is that Push programs (or, more commonly, linear "plushy" genomes that map in a straightforward way to Push programs) can be freely mutated and crossed over without risking syntax or execution errors, even when multiple types, conditionals, loops, and more sophisticated data and control structures are being used [18].

PushGP systems—GP systems that evolve Push programs—have produced some of the best general program synthesis results to date, but the interactions between Push programs, human programming practices, and human-written codebases leave something to be desired. For example, although it is possible to "wrap" arbitrary functions as Push instructions that can then be used in evolved Push programs, this is a cumbersome, manual process that becomes impossible with the introduction of polymorphic types and functions. If one wants to use an evolved Push program in software that is written in a conventional programming language, then one either has to translate the evolved Push program into the conventional language or to include a Push interpreter in the software. Because is difficult for humans to read, understand, debug, or modify Push programs, translation is not always straightforward, and inclusion of evolved Push programs in production software may raise quality and reliability concerns.

By contrast, grammar-based approaches support general program synthesis by embracing the syntactic specification of data and control structures, and by employing methods that always generate syntactically valid programs in a conventional programming language. This provides a better interface to human programming practices and codebases. However, polymorphism is still problematic because grammars do not fully capture type information, which limits the possibly of real-world applications. In addition, these methods do not yet appear to be able to solve benchmark software synthesis problems quite as reliably as PushGP.

Code building genetic programming (CBGP) is a recently developed method that takes some inspiration from both of these approaches. It uses linear genomes that are similar to PushGP’s plushy genomes, and which are compiled in a manner similar to Push program execution [18]. With CBGP, however, this compilation process produces code in a conventional programming language, which can be read and understood by humans, and which can also make use of arbitrary (and possibly polymorphic) functions. As noted in the original CBGP paper [19], this appears to provide
benefits both in terms of problem-solving power and potential for interaction with human programming practices.

The original CBGP paper, however, was a preliminary report, which provided neither a formal description of the algorithm nor a thorough benchmarking of CBGP relative to other methods. In this paper we fill those gaps. We formalize CBGP using a functional, Hindley-Milner type system, and we describe our new functional CBGP system on the basis of this formalization. We then perform a comprehensive analysis of the search performance of this system, comparing it to other contemporary GP program synthesis methods.

In the next section we review the results demonstrated in the original CBGP paper and flesh out the goals for the present work. In the subsequent section we present tools from type theory that we will use in our formalization. We then present functional CBGP in detail. This is followed by a section on the design of our experiments, and then by a section on our experimental results. We conclude with a discussion of the implications of our results and of avenues for future work.

2 CODE BUILDING GENETIC PROGRAMMING

CBGP is a method of evolving programs using a stack-based compiler which folds variable length, linear, genomes into type-safe abstract syntax trees (AST). The introduction of CBGP in 2020 included demonstrations of novel capabilities such as:

- Type safe programs that call polymorphic functions.
- Anonymous functions defined inside evolved programs.
- Higher order functions for control flow.
- Use of user-defined data types, such as classes.
- Dynamic generation of the function set (aka genetic source [11]).

In addition, the evolved programs are generated as source code in a conventional programming language, a feature which was previously only possible using grammar based GP methods. These features were demonstrated on a small set of simple benchmark problems [19].

This paper focuses on a subset of these capabilities and aims to clarify and standardize the implementation by leveraging properties of functional programming and type theory. In particular, the functional CBGP system described in this paper uses parametric polymorphism, anonymous function creation, and higher order functions. We only consider a finite set of data types common to other GP program synthesis systems such as PushGP, GE, and G3P. Also, the function set used in this version of CBGP was manually implemented and curated to be as similar as possible to the function sets found in other contemporary GP program synthesis systems.

The "host language" of a program synthesis system is the language in which synthesized programs are expressed. The version of CBGP proposed in this paper requires a functional programming language, and Clojure was selected because the LISP syntax makes it trivial to convert ASTs into source code and vice versa. The choice of host language is an implementation decision and the algorithms detailed in this paper are applicable to any functional host language.

3 TOOLS FROM TYPE THEORY

Since the invention of simply typed lambda calculus by Church in 1940, the field of type theory has been developing algorithms for analyzing the validity of functional programs based solely on their structure [2]. One relatively modern manifestation of these ideas is the Hindley-Milner (HM) type system, which is commonly used by compilers of modern functional programming languages [15, 17]. It analyses programs represented as abstract syntax trees (AST) to prove the program is type safe. Furthermore, the HM system can provide the (possibly polymorphic) data type returned by the AST. We briefly describe the kinds of AST expressions the HM type system can analyze in the following paragraphs.

The atomic expressions, or leaves of the ASTs, are literals (aka constants) and variables. A variable is a symbol that will be translated into a known value when the program is executed. It is assumed the data type of a literal is trivially known and the data type of a variable is given by a mapping called the "type environment" denoted as $\Gamma$.

Larger expression trees can be created by composing literals and variables with function abstraction, function application, and 1et binding expressions. A function abstraction expression denotes the creation of a function with some fresh (new and unique) variables as arguments and an expression tree representing the body. A function application expression denotes the calling of a function on some arguments, each given by a child expression. A 1et binding expression defines a local variable which will take the value of a given child expression.

Figure 1a and 1b show the source code and AST of an example program that the HM type system is capable of analysing.

3.1 Types

The primitive, atomic, types in the HM system are called "ground" types. The set of ground types typically includes: Boolean, Int, Double, Char, and String although there is some variation between implementations. In our work, we consider Nil a ground type.

A type constructor is an operator that builds a type from one or more other types. Collection types such as Sequence, Set, and Map are type constructors that must be provided the data type of their elements to build the full type. For example, we denote a set of strings as Set[String]. The function type constructor takes one or more argument types and a return type. For example, the function type of the string length function is (String) $\rightarrow$ Int.

The Hindley-Milner type system supports abstract types using a form of parametric polymorphism called type "schemes" [3, 15]. The canonical example of a type scheme is the identity function. The argument type of the identity function can be any type, but the return type will always match the argument type. The type of the identity function is $\forall \alpha. \alpha \rightarrow \alpha$ which is read as "for all possible types $\alpha$, the function type from $\alpha$ to $\alpha$".

A more complex example is the polymorphic type of the map higher order function.

$$\forall \alpha, \beta. ((\alpha \rightarrow \beta), \text{Sequence}[\alpha]) \rightarrow \text{Sequence}[\beta]$$

We read this as: “For all possible types $\alpha$ and $\beta$, a function which takes a function from $\alpha$ to $\beta$ and a sequence of $\alpha$ and returns a sequence of $\beta.“$ The parameters of a scheme are referred to as “type variables”. In the examples above, $\alpha$ and $\beta$ are type variables.
Python
f = lambda x: max(0, x)
map(f, input)

Haskell
let f = \x -> max 0 x
in map f input

LISP (Clojure)
(let [f (fn [x] (max 0 x))] 
(map f input))

(a) The source code in a variety of languages.

Unification
The Hindley-Milner type system is often used by compilers and code analyzers to perform type checking and type inference of abstract syntax trees. A crucial component of these algorithms is unification; a procedure which produces a set of type substitutions that will bind free type variables to concrete types, if possible. Unification has roots in theorem proving and solving systems of symbolic equations [16, 22].

To illustrate the use of unification, suppose we have a function, \( f \), with the type \( \forall \alpha.\text{Sequence}([\alpha]) \rightarrow \alpha \), and an expression, \( e \), of type \( \text{Sequence}([\text{Int}] \rightarrow \text{Int}) \). To determine if a type safe expression can be created by applying \( f \) to \( e \), we unify the argument type of \( f \) with the type of \( e \). The unification algorithm will succeed and provide the substitution \( \alpha := \text{Int} \). Using this, we can replace all instances of \( \alpha \) with \( \text{Int} \) in the type of the new composite expression to determine that \( f(e) \) has type \( \text{Int} \).

If the unification of two types requires a type variable be bound to more than one concrete type, or if unification requires two different non-variable types to be the same, for example \( \text{Int} = \text{Boolean} \), the AST fails type checking.

4 FUNCTIONAL CODE BUILDING GP
In this section, we describe how functional CBGP uses HM types, schemes, and unification to produce type safe programs during evolution.

4.1 Genomes
CBGP uses a variable length linear genome, which is translated into a type-safe AST using a compilation process. Specifically, the

3The source code for our implementation is available at https://github.com/erp12/cbgp-lite.
Table 1: The distribution of gene categories for randomly generated genes and genomes, which is used to create the initial population and supply genes for mutation operators. Some literal genes are sampled from a discrete set dependent on the problem, while others were created via ephemeral random constant (ERC) generators [20]. This distribution was selected on the basis of human intuition and is potentially sub-optimal.

| Gene Category | Proportion |
|---------------|------------|
| Variable      | 0.2        |
| Local Variable| 0.15       |
| Literal       | 0.15       |
| Literal (via ERC) | 0.1      |
| Abstraction   | 0.15       |
| Application   | 0.15       |
| Open & Close  | 0.1        |

Figures and Excerpts:

Figure 3: The algorithm for finding ASTs which will be type-safe arguments to the function being applied by the APP stack instruction. The result is a list of ASTs with NIL values in the positions where no AST of a compatible type was found on the stack.

if the function type of the function AST selected by the APP instruction is polymorphic (has a type scheme with a type variable), the call to the unification algorithm (Figure 3, line 8) will bind the free type variables to the concrete type of the argument AST. These bindings are substituted in later argument types (Figure 3, line 4) to ensure that all instances of the type variable are bound to the same type. For example, when compiling an APP with the function `append` with type `∀α.Sequence[α] → Sequence[α]`, the first argument can be satisfied by any sequence type. If the top-most AST with a sequence type has the type `Sequence[Int]` then `α` will be bound to `Int` which will become the second argument type of this particular application of `append`.

When a nested push sequence is encountered, it is pushed to the chunk stack. Function abstraction instructions and LET instructions use these chunks to compile a body of code that may reference the local variables created by those constructs, as described below.

When a function abstraction instruction (ie. `ABS[Int]`) is encountered new local variables are created to serve as the function’s argument variables. No elements of the AST stack contain references to these new arguments names. The body of the function must be compiled from a push sequence within a new scope, or environment, that includes the new argument variables. To compile the body of the new function, the compilation algorithm makes a recursive call to itself with a new environment based on the ABS instruction’s argument types and passes a push sequence from the chunk stack. During this nested call, local variable genes may resolve to one of the new local variables included in the environment. If the recursive compilation call does not produce any ASTs, for example if the chunk is empty or does not contain any leaf nodes from which to build ASTs, subsequent items of the chunk stack are compiled until an AST is found or no chunks remain on the stack. The latter scenario triggers the ABS instruction to noop. Figure 4 details the process of compiling a function abstraction expression in pseudocode.
The genome in Figure 2 is artificially simple for demonstration purposes. During evolution, most genomes are much longer and contain genes that either noop or build additional ASTs which do not get selected as the program because they do not return the correct data type for the problem or are buried deep in the stack at the end of compilation.

4.3 Evolution

For this work, a standard generational genetic algorithm was used to evolve programs. An initial population of random genomes was produced using the method described in Section 4.1. Genomes are compiled into a type-safe ASTs (Section 4.2) which are executed identically to a native function in the host language. These programs are evaluated based on a set of training cases in the form of input-output pairs.

The program’s error on each training case is determined by a user provided error function. The collection of errors across all training cases is referred to as an individual’s “error vector”. If no AST with the problem’s target return type is produced after compilation the individual is given a penalty error on every training case. If the compiled AST produces a runtime error, such as “index out of bounds,” when called on a training case, it is given a penalty error. Parents are selected from the population of evaluated individuals on the basis of error vectors using Lexicase Selection [10, 13]. The next generation of genomes is produced through variation of parent genomes.

If an individual is found to have an error of zero on all training cases, or if the maximum number of generations is reached, evolution is stopped and the individual with the lowest total error, given by the sum of its error vector, is returned. If this individual has a total error of zero, it is called a “solution.”

4.4 Simplification

The best individual found during evolution is extracted for simplification. It has been shown that simplification acts as a form of regularization which improves the program’s generalizability to unseen data cases [8]. In addition, a simplified program may be easier for a human to understand. The best individual from evolution is simplified using a hill-climbing algorithm, as follows:

1. Create a new genome using an order-preserving random subset of the best individual’s genome.
2. Compile and evaluate the new genome to create a new individual.
3. If the total error of the new individual is equal to, or lower than, the current best individual it replaces the best individual.
4. If iteration limit is reached, return best individual. Otherwise, return to step 1.

The best individual’s program after all iterations of simplification is reported as the output of the evolutionary search. For our experiments, this is the program that is tested for generalization on an unseen set of test cases.

5 EXPERIMENTAL DESIGN

We assess the ability of CBGP to perform automatic program synthesis using a subset of 14 problems from the program synthesis
would supplement our experiments here. To allow for a wide range of requirements, such as data types and control flow, we will use a functional host language compared to Push or the other GP methods. In particular, much of the control flow in this implementation of CBGP is handled by the control flow in grammar-based programs of G3P and GE. In particular, much of the control flow in this implementation of CBGP is handled by higher-order functions that iterate over lists, such as map, filter, and reduce.

### Table 2: The evolutionary hyperparameters used for all runs of CBGP associated with the results presented in this paper.

| Hyperparameter               | Value                  |
|------------------------------|------------------------|
| Population Size              | 1000                   |
| Max Generations              | 300                    |
| Parent Selection             | Lexicase Selection [13]|
| Variation                    | UMAD [9]               |
| Mutation Rate                | 0.1                    |
| Simplification Steps         | 2000                   |
| Initial Genome Sizes         | [50, 250]              |
| Number of Training Cases     | 100                    |
| Number of Unseen Test Cases  | 300                    |

The description of the problems in PSB1 recommends not using every single available function for every problem [11, 12]. For example, including functions that manipulate strings when solving a problem that only relates to lists of integers would expand the search space unnecessarily. As such, we follow these recommendations by creating type-tuned genetic sources for each problem in the fashion recommended by PSB1: for each problem, only include functions that manipulate the data types deemed relevant by PSB1. This ensures that we do not cherry-pick instructions known to be useful for a problem, while not including instructions that have no bearing on it.

### 5.1 Comparison Methods

We compare our CBGP results with those of other GP representations: PushGP, G3P, and GE. We choose results from papers using comparable evolutionary hyperparameters as much as possible.

For PushGP, we use results from the paper introducing Uniform Mutation by Additions and Deletions (UMAD) [9]. Like this paper, our CBGP runs use UMAD as the only genetic operator, making a reasonable comparison.

We use the paper introducing grammar design patterns as the results for G3P [5]. The paper uses similarly type-tuned grammars to determine the instructions available to evolving programs.

Our reported GE results are taken from a paper exploring the use of domain knowledge and novelty in program synthesis [14]. Since neither of these ideas are used in our work here, we use the baseline control results reported in the paper.

### 6 RESULTS

Table 3 compares the success rates of CBGP to other GP representations on the 14 benchmark problems. CBGP performs quite well on 7 of the problems, producing success rates near or at 100. On all of these problems except number-ios, at least one of the other methods performs significantly worse than CBGP. On the other hand, CBGP performs significantly worse than at least one other method on the last 6 problems.

The last column in Table 3 gives the proportion of solutions on the training data that perfectly generalize to the unseen test set. Compared to the other three GP representations, which have typically produced low generalization rates on some, but not all, of these problems, the generalization rate of CBGP solutions is quite high across the board. For example, compare-string-lengths, last-index-of-zero, median, and negative-to-zero all produced generalization rates lower than 0.75, while almost no problem exhibited a generalization rate of 1.0, in a study of generalization using PushGP [8].

Table 4 presents the sizes of solution programs found for each problem solved by CBGP. Program size is measured in number of nodes in the Clojure S-expression representation of the program, which is identical to the number of nodes in the AST. We find that three of these problems have been solved by programs containing only 4 nodes, while the remainder have been solved by programs with 10 or fewer nodes. Interestingly, the mean solution sizes pre- and post-simplification tend to be quite close to the minimum sizes. This means that evolved solutions rarely have unnecessary code in the programs themselves. Note that genomes, on the other hand,
Table 3: Percentage of runs that found a generalized solution on each problem. Underlined values indicate the comparison method has a statistically significantly worse solution rate than CBGP according to a chi-squared test with a p-value of 0.05. Values in bold indicate a statistically significantly better success rate using the same test. The generalization rate column denotes the proportion of runs for which the program which solved all training cases also solved the unseen test data.

Table 4: Solution sizes for each problem that CBGP solved. Min gives the minimum size of any solution program, while Pre and Post give the mean sizes before and after applying automatic simplification to the solution genomes.

| Problem                | Min | Pre  | Post |
|------------------------|-----|------|------|
| smallest               | 7   | 7.55 | 7.18 |
| mirror-image           | 4   | 4.54 | 4.06 |
| number-io              | 4   | 4.92 | 4.03 |
| vectors-summed         | 4   | 4.25 | 4.00 |
| negative-to-zero       | 7   | 7.90 | 7.02 |
| median                 | 9   | 10.52| 10.03|
| vector-average         | 7   | 9.74 | 8.89 |
| compare-string-lengths | 10  | 12.34| 11.79|
| last-index-of-zero     | 8   | 12.42| 10.33|

6.1 Example Solution Programs

The supplementary materials to this paper include a file containing every solution evolved by CBGP. In Figure 5 we give some examples of those solution programs and note some of their interesting features below.

The solution to negative-to-zero interestingly maps the subtract function over two copies of the input vector, which produces a vector entirely made of zeros. It then maps the max function over the zeros and the input vector, changing every negative integer into 0 as required.

The vector-average solution behaves as expected. One thing to note is that it converts the length of the input vector to a Double, since the count function is typed to return an Int. Future work into allowing for subtyping or type classes could allow for all Int expressions to be considered valid Double expressions, but for now, the conversion must happen explicitly.

The smallest problem requires the program to find the minimum of four inputs. Instead of simply applying the min function 3 times, this solution unnecessarily defines a new function that finds the min of input4 and its argument, and then applies that function to input1.

The last-index-of-zero problem requires the program to find the last index where 0 appears in the input list. This solution reverses the input, finds the first index of zero, and then subtracts that from the decremented length of the input. This strategy is similar to some solutions to this problem that have been evolved in PushGP.

7 DISCUSSION AND FUTURE WORK

CBGP has demonstrated that it can readily find solutions to some problems, but on others the solution rate of CBGP quickly drops to zero. These trends correlate somewhat to the problems found difficult by other GP representations; however, there are some problems that CBGP solves readily that others do not and vice versa. When initially tested on the PSB1 benchmark problems, the other genetic programming systems saw similar trends, and have since increased performance as the methods mature through continued research. We hope to see a similar rise in the search performance of CBGP in the future.

The large variety in which problems each system finds easier or harder points to the importance of program representation for search performance. This area is not well understood in GP, and we hope that CBGP can help better illuminate important differences in representation. We suggest this as an area of future research.
such that we can understand what makes problems difficult under a given representation.

One hypothesis regarding different representations producing wildly different results on some problems is the impact of representations on the size of programs needed to form a particular computation. Solutions to problems with high solution rates tend to be smaller than solutions to problems with a low success rate, regardless of representation. The problems that CBGP solves most readily have small solution programs, and similar results have been shown for PushGP on the same problems [12]. We do not know if similarly small solution programs are possible for the problems that CBGP did not solve, and it simply did not find them, or if they require larger programs and therefore are more difficult to find in the search space. Further research into CBGP solutions to these problems could help us understand whether it is simply the size of the solution programs preventing them from being solved, or whether CBGP has issues traversing the search space effectively regardless of solution size for some problems.

The exceptionally high generalization rate of CBGP is not easily explained. When considered in combination with the inability to find solutions on harder problems, this may be an indication that CBGP cannot fall back on memorizing or bloating the program into something that overfits the training data. In CBGP, an increase in genome size does not necessarily cause an increase in program size because additional genes may simply result in more ASTs being left on the stack after compilation, rather than larger ASTs. This hypothesis is further supported by the minimum and average program sizes of solutions found by CBG. The problems with high solution (and generalization) rates are solved by small programs.

When looking through solution programs, we found very few instances of programs that define and use anonymous functions effectively. Defining such functions is an integral part of functional programming for human programmers. Thus one piece of important future work is to try to assess why CBGP is not making use of function definition, and considering ways to encourage this behavior.

One insight from this research that may be helpful to the wider research fields of genetic programming and program synthesis is the value of introducing formalisms, such as type theory, into our systems. The body of work accumulated in fields of theoretical computation provide the program synthesis community with tools to guide synthesis towards programs with desirable properties, such as type safety.

Functional CBGP can, in theory, represent programs using any data type or language construct supported by the type system, and the unification algorithm in particular. A valuable direction of future research is to implement the common extensions to the Hindley–Milner type system which add support for function overloading, sub-types, and variadic functions [1, 4, 21, 23]. The primary benefit of these extensions would be the ability to represent programs using all the features of a modern functional programming language and possibly approach any programming task that can be well-specified by types. Another benefit of supporting additional kinds of polymorphism is the ability to use a smaller genetic source with considerably fewer, more general, functions which could dramatically reduce the search space and improve solution rates on complex problems.

8 CONCLUSION
In this paper we present functional Code Building Genetic Programming and show how it leverages type theory to ensure synthesized programs are type safe while also allowing polymorphic functions, anonymous functions, and higher order functions to be expressed. We report on empirical benchmarks that show CBGP can find solution programs more consistently than other contemporary GP methods on some problem, while it struggles to find any solutions on others. Investigations into solution programs show repeated use of polymorphic functions and higher order functions, but little use of anonymous function definitions.

When CBGP does find a solution on training data, we observe an exceedingly high rate of generalization to unseen test data. This phenomenon is in contrast to the comparatively low generalization rates of all other GP systems included in our comparison [5, 8, 24]. Furthermore, the solution programs found by CBGP are small, even without the use of typical genome simplification techniques.

Finally, we direct future research towards a deeper utilization of type theory in general program synthesis systems. We also suggest the genetic programming field perform broader studies into the impact of representation on problem difficulty.

ACKNOWLEDGMENTS
This material is based upon work supported by the National Science Foundation under Grant No. 1617087. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.
