Facilitating Function Application in Code Building Genetic Programming

Thomas Helmuth
Hamilton College
Clinton, New York, USA
thelmuth@hamilton.edu

Jayden Fedoroff
Hamilton College
Clinton, New York, USA
jfedorof@hamilton.edu

Edward Pantridge
Spindle Health
Boston, Massachusetts, USA
eddie@spindlehealth.com

Lee Spector
Amherst College, UMass Amherst
Amherst, Massachusetts, USA
lspector@amherst.edu

ABSTRACT

Code Building Genetic Programming (CBGP) is a method for general inductive program synthesis that uses a genetic algorithm and a formal type system to evolve linear genomes that are compiled into type-safe programs in a host language. Prior work showed that CBGP can evolve programs that use arbitrary abstractions from existing codebases along with higher-order functions and polymorphism. In tests on benchmark problems, however, the problem solving capabilities of CBGP have been mixed. One hypothesized explanation for weak performance on some problems is that many functions encountered during the compilation process are typically not applied. Here we propose two modifications to the compilation algorithm, both of which make it more likely that functions will be applied when composing programs. The first modification changes how frequently CBGP attempts to perform function application, while the second allows the construction of function applications to backtrack. While both modifications increase solution rates on benchmark problems, the backtracking modification shows more promise with a modest increase in computational cost and no additional configuration requirements. We argue that this modification should be considered the new standard compilation algorithm for CBGP systems.

CCS CONCEPTS

• **Software and its engineering → Genetic programming**.

KEYWORDS

automatic programming, genetic programming, inductive program synthesis, function application

ACM Reference Format:
Thomas Helmuth, Jayden Fedoroff, Edward Pantridge, and Lee Spector. 2024. Facilitating Function Application in Code Building Genetic Programming. In *Genetic and Evolutionary Computation Conference (GECCO '24), July 14–18, 2024, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3638529.3654068

1 INTRODUCTION

A variety of methods have been proposed for *inductive program synthesis*, the task of automatically producing programs that meet requirements given in the form of input/output examples. Among those that show promise are several that employ evolutionary computation processes in which random variation and selection operate on populations of candidate solutions. Koza's genetic programming (GP) [15], in which populations of program parse trees are mutated and recombined in a generation-based search algorithm, spawned much of the early work in the field. More recently, the best performance on standard program synthesis benchmark problems [25] has been obtained using other methods such as grammar-guided GP [9], linear GP [7], and stack-based GP [13].

Code Building Genetic Programming (CBGP) is an evolutionary approach to program synthesis that was developed relatively recently [20–22], and which has several features that are attractive in principle. The CBGP approach uses linear genomes that can be generated and varied using generic algorithms that are independent of the syntax or semantics of the target programming language, and yet the system always produces syntactically correct, type-safe programs in that language. Furthermore, the programs produced by CBGP can make use of arbitrary data types, data structures and specifications from existing codebases, along with sophisticated programming techniques that leverage higher-order functions and polymorphism. All of this is enabled by a compilation process that translates the linear genomes into type-safe computational graphs, and thereafter into programs in the target language. A bonus of this approach is that the compiled programs generally execute more quickly than the interpreted program representations used in many other genetic programming systems. This allows programs to be evaluated faster, even over large training data sets. Because program evaluation is often the most time-consuming step of the genetic programming process, the compilation performed in CBGP can speed up the overall genetic programming process considerably.

The first published work on CBGP described the general ideas behind the approach and demonstrated its use on a variety of benchmark problems [22]. Subsequent work formalized CBGP using a functional programming language and a Hindley-Milner type system, while also providing an analysis of the search performance of CBGP in comparison to other GP-based program synthesis approaches [21]. More recent work has focused on the capability of CBGP to produce programs that use generic data types in conjunction with higher order functions and functions with polymorphic type signatures [20].

While many of the features that are uniquely provided by CBGP make it a promising candidate for advancing the state of the art of program synthesis, its performance on standard program synthesis benchmarks has so far been mixed. For example, when CBGP was run 100 times on each of 14 problems from the PSB1 program synthesis benchmark suite [13] it performed excellently on seven of the problems and reasonably well on two others. However, it found zero solutions for the remaining five, all of which were solved at least some of the time by at least one of the competing methods [21].

Subsequent analysis of data produced during CBGP runs revealed that a great deal of the program structure produced during the compilation process is typically not actually used in the final programs produced as output from the compiler. As described below, many times a genome will produce many small abstract syntax trees (ASTs) without combining them into larger trees through function application. These function applications, which produce the core of any program written in a functional programming language, are required to build more complex programs. Since CBGP only conducts function application when an explicit application gene appears in the genome, and since this application is limited in which functions and arguments it can select, function application does not occur successfully as often as may be beneficial. After noticing this behavior, we hypothesized that this lack of function application was part of the explanation for CBGP's weak performance on some problems.

In order to investigate this hypothesis we developed and tested two modifications to the CBGP compilation algorithm. Both of these modifications, which are independent of each other and can be used either separately or together, make it more likely that function ASTs will be successfully applied to argument ASTs to create larger ASTs. The first modification explores different "function application triggers," which determine when and how a function application happens during compilation. We implement and test multiple novel application triggers intended to decrease the number of function ASTs that are ignored due to scarcity of the dedicated function-application gene. The second modification, "application backtracking," allows for backtracking in the process by which the compiler makes decisions regarding the functions and arguments used in each attempt to apply a function. This makes it more likely that a compatible set of function and argument choices will be found, resulting in the successful composition of a function application AST.

To summarize, the research questions we will address in this paper are as follows:

RQ1. Function application triggers: Can changing how and when function application is triggered during compilation improve CBGP's problem solving performance?

RQ2. Backtracking: Can we improve performance by allowing the function application algorithm to backtrack after selecting functions and arguments?

RQ3. Combination: Do function application triggers and backtracking work together?

We conduct experiments on a suite of polymorphic benchmark problems, and find that both modifications increase solution rates. That said, the combination does not improve over simply using backtracking, leading to the conclusion that backtracking should be used without introducing new hyperparameters to tune.

In the remainder of this paper we first provide a more complete description of the process by which function applications are built into programs during CBGP compilation. Following this, we describe our proposed modifications to this process, the methods and experimental design by which we tested the modifications, and the results of our experiments. We conclude with a discussion of the results and related work, our recommendations regarding compilation in future CBGP systems, and suggestions for further research.

2 FUNCTION APPLICATION COMPILATION

CBGP uses an indirect encoding of programs that are compiled from a genome data structure into an executable phenotype in the form of an abstract syntax tree (AST). A genome is a variable length linear sequence of AST nodes and compilation directive genes. The compilation process composes these nodes into a tree structure to form a syntactically valid and type safe AST in some host language. The runtime of the host language is then able to further compile, or interpret, the AST into machine code during program execution. The full CBGP compilation algorithm for a lambda calculus based host language (in particular, a strongly typed functional language that supports parametric polymorphism) is described in [21] and briefly reiterated in this section. We give a detailed explanation of the previously published compilation process for function applications because the main contribution of this work is a modification to this procedure.

The compilation algorithm uses two ordered data structures: the genome sequence, and a stack-like data structure for holding ASTs. Initially the AST stack is empty. During each iteration of the algorithm, a subsequent gene in the genome is processed. If the kind of gene is an AST "leaf" node it is pushed to the top of the AST stack. Leaf nodes are either literals (which include function literals) or variables, both of which must have a known data type. Compilation directive genes trigger some method of AST composition by searching for ASTs on the stack and combining them into a larger AST which is pushed to the top of the stack. The three compilation directive genes that compose ASTs are:

- `APP` - **Application** - Calling a function AST on argument ASTs.
- `ABS` - **Abstraction** - Creating a lambda (anonymous) function AST.
- `LET` - **Let Binding** - Defining a new local variable bound to an AST.

Once all genes in the genome have been processed, the AST stack will hold zero or more type safe ASTs. The top-most AST with a type matching the target program's specified output type is returned as the corresponding phenotype, or "program". If the AST stack is

```
procedure App($ASTs$)                                   ▷ Given an AST stack.
    $functionASTs \leftarrow [a \in ASTs \mid \text{Type}(a) \text{ is a function}]$
    if $functionASTs$ is empty then
        return $NOOP$
    end if
    $functionAST \leftarrow$ top AST of $functionASTs$
    $ASTs.remove(functionAST)$
    $argASTs \leftarrow []$
    $typeVars \leftarrow \{\}$                            ▷ Mapping from type var to type.
    for all $t \in \text{ArgTypes}(functionAST)$ do
        $t' \leftarrow t.substitute(typeVars)$
        $unused \leftarrow [a \in ASTs \mid a \notin argASTs]$
        $safe \leftarrow [a \in unused \mid \text{Type}(a) \text{ is a } t']$
        if $safe$ is empty then
            return $NOOP$
        end if
        $arg \leftarrow$ top AST of $safe$
        $argASTs.append(arg)$
        $newBindings \leftarrow \text{Unify}(t', Type(arg))$
        $typeVars.addAll(newBindings)$
    end for
    $ASTs.removeAll(argASTs)$
    $ASTs.push(Apply(functionAST, argASTs))$
end procedure
```

Figure 1: The previously published procedure for compiling a function application gene in a CBGP system. The conditions that return *NOOP* revert the AST stack to its state prior to compiling the apply gene and no function application AST is created.

empty or does not have an AST of the correct type, the genome has no corresponding phenotype, and is typically considered ineligible for evaluation and selection.

This work is solely concerned with the process for composing function applications. Figure 1 provides pseudocode for the original compilation process of a function application gene. At a high level, function application finds the first function AST on the stack, then finds the first AST(s) that have the correct types for the function's parameters, creating a new AST composed of the function AST and argument ASTs as children to a function application AST node. This algorithm was introduced in [21] and is used in all CBGP implementations that leverage a functional type system prior to this work [20]. Other CBGP implementations not based on lambda calculus, such as [22] and [5], use a slightly different procedure for compiling function applications, but all known implementations only consider the top-most function AST and perform repeated searches through the AST stack for valid arguments. Crucially, all known implementations immediately *NOOP* if any argument cannot be satisfied. A *NOOP* does not modify the AST stack and does not create any new composite ASTs.

Figures 2a and 2b depict an example function application compilation on a given AST stack. The first function AST, + of type $(Int, Int) \rightarrow Int$, is chosen as the function to invoke. Subsequent passes through the AST stack find 2 *Int* type ASTs to serve as the arguments. Notice that if a subsequent function application is triggered on the stack in Figure 2b it will result in a *NOOP* under the original CBGP semantics. This is because the chosen function AST will be < of type $\forall\alpha.(\alpha, \alpha) \rightarrow Boolean$. The arguments to this function initially have free type variables, thus an AST of any type is considered valid and the AST on top of the stack will be chosen. In this case, the first argument AST will have type *Int* which substitutes all instances of α in the remaining argument types with *Int*. There are no other ASTs on the stack with type *Int* and therefore the function application compilation must *NOOP* because it cannot satisfy the second argument of the < function. This is true even though the *reverse* or *inc* functions would be able to be successfully applied, since the algorithm is locked into the first function that it finds.

3 MODIFICATIONS

In this section, we provide the motivation behind improving the compilation of function applications. We then describe the modifications to CBGP that form the basis of our proposal for a new canonical CBGP compilation algorithm.

Functional CBGP has a known bias towards synthesizing smaller programs. It has been hypothesized that this bias contributes to CBGP's inability to find solutions to some problems [21]. Similarly, it has been observed that the number of unique data types for the synthesized ASTs is large. A previous empirical study of CBGP showed the median number of unique data types seen across an evolutionary run ranged from 668 to 15,941 depending on the problem. Many of the more complex types are function types from ASTs that could be invoked if valid arguments were provided [20]. Encouraging these ASTs to be composed could be important for synthesizing larger, more sophisticated, programs.

3.1 Function Application Triggers

Before this work, CBGP only triggers the function application process in Figure 1 when an explicit APP gene is encountered while compiling the genome. Otherwise, no functions are applied. We hypothesize that this design likely results in smaller compiled programs, since many functions will not be applied unless the prevalence of APP genes is very high; in previous studies, they appear in genomes 15-20% of the time [20, 21]. We call this method *explicit function application.*

In order to increase the rate at which function application is triggered, we experiment with a variety of methods that change when this happens. *Implicit function application* triggers the function application process after every single time that an AST is pushed onto the AST stack, even ASTs that were just created through function application. We expect that implicit function application will increase the rate of successful function applications, building larger ASTs. However, we also expect that it will be too extreme, since there are functions that it would be better to not apply. Specifically, CBGP contains many higher-order functions that take a function as one or more of their arguments. With implicit function application, there may be few or no function ASTs to use as arguments when applying higher-order functions; a function would only not be applied if it is attempted to be applied, but is unable to find the correct arguments.

+	$(Int, Int) \rightarrow Int$
1	Int
"AZ"	$String$
2	Int
<	$\forall\alpha.(\alpha, \alpha) \rightarrow Boolean$
"b"	$String$
reverse	$String \rightarrow String$
inc	$Int \rightarrow Int$
AST	**Type**

(a) The state of the AST stack prior to processing an APP gene.

APP / | \\ + 1 2	Int
"AZ"	$String$
<	$\forall\alpha.(\alpha, \alpha) \rightarrow Boolean$
"b"	$String$
reverse	$String \rightarrow String$
inc	$Int \rightarrow Int$
AST	**Type**

(b) The AST stack after processing an APP gene using the procedure from Figure 1 on the stack from Figure 2a.

APP / \\ *reverse* "AZ"	$Boolean$
APP / | \\ + 1 2	Int
<	$\forall\alpha.(\alpha, \alpha) \rightarrow Boolean$
"b"	$String$
inc	$Int \rightarrow Int$
AST	**Type**

(c) The AST stack after processing an APP gene using function backtracking on the stack from Figure 2b.

APP / | \\ < "AZ" "b"	$Boolean$
APP / | \\ + 1 2	Int
reverse	$String \rightarrow String$
inc	$Int \rightarrow Int$
AST	**Type**

(d) The AST stack after processing an APP gene using argument backtracking on the stack from Figure 2b.

Figure 2: Example AST stacks before (a) and after compiling a function application gene using original compilation semantics (b) and the proposed backtracking modifications (c and d).

In order to allow evolution to more finely tune when function applications trigger during compilation while still encouraging more of them than occur with explicit application, we designed two new methods. The first, *implicit with skips*, adds skip genes to implicit application. When compilation encounters a skip gene, the next gene in the genome will not trigger function application. Skip genes make up 10% of randomly generated genes. Implicit with skips does include APP genes in genomes to apply functions that otherwise are not applied, but at a much lower rate (2.5%) than with explicit application (20%).

Second, *gene-annotated application* associates with each gene in the genome a flag telling whether or not function application should be triggered after compiling that gene. In this way, the triggering of application is tied to specific genes instead of being a separate gene, as with explicit application. The application flag of each gene is set when the gene is created, either at genome initialization or mutation. The *application rate* determines how frequently the flag will be set to trigger application when each gene is created; for example, an application rate of 0.75 means that 75% of genes will have their flag set to trigger application. In theory, a custom mutation operator could be used to randomly toggle some of the application flags on a genome; however we did not explore such a method for this work.

3.2 Backtracking

In the context of this paper, *backtracking* refers to a modification to the function application compilation semantics from Figure 1 such that the *NOOP* outcome is delayed until all possible function ASTs and argument sets are considered, which allows more function applications to complete successfully. There are 2 ways which backtracking can occur: function backtracking and argument backtracking.

Function backtracking occurs when there are no argument sets on the AST stack which satisfy the function. In this case, our proposed modification is to consider the next function on the AST stack (searching in top-to-bottom order) and reattempt to find valid argument ASTs. Since different functions take different argument types, another function may be able to be applied after an attempt at applying the first fails. This processes is repeated until a composite AST can be constructed using one of the functions or all function type ASTs have been attempted, resulting in a *NOOP*. Type variable substitutions that were bound while attempting to find arguments for the previous function are not used to find arguments to the subsequent function.

For example, consider the state of the AST stack in Figure 2b. Without backtracking, function application in this state will fail and result in *NOOP*, as discussed in Section 2. With function backtracking, after the < fails to find arguments, the function application will backtrack to look for another function. At this point it will find the

Hyperparameter	Value
Population Size	1000
Max Generations	300
Parent Selection	Lexicase Selection [14]
Variation	UMAD [11]
Mutation Rate	0.1
Initial Genome Sizes	[50, 250]
Number of Training Cases	200
Number of Unseen Test Cases	2000
Runs per Problem	100

Table 1: The evolutionary hyperparameters used for the CBGP runs presented in this work, and the previously published baseline results from [20].

reverse function, and will look for a *String* argument for it, finding "AZ". Thus a new function application of *reverse* to "AZ" will be pushed onto the AST stack. The resulting state of the AST stack is given by 2c.

Argument backtracking occurs when a chosen argument AST binds a type variable in such a way that makes it impossible to find valid ASTs for one or more remaining arguments. In this case, we revert all argument ASTs chosen after the most recent type variable was bound and continue compilation with the state of the AST stack reverted to the point where the type variable was initially bound. From this point, we only allow the type variable to be bound to types to which it has not previously been bound in a backtracked fashion. This allows for all potential argument sets to be considered without an exhaustive search through all combinations of ASTs on the stack.

Again, we can consider the example AST stack given in Figure 2b. Still, the first function that application finds is <; however, something different happens when trying to apply <. After considering the *Int* AST on the top of the stack and not finding another *Int*-typed AST, instead of failing or backtracking to the next function, argument backtracking will revert the assignment of the type variable $\alpha := Int$, and keep searching for a different type assignment. It will then bind $\alpha := String$ when attempting to use "AZ" as an argument, and will be able to find a second *String* argument in "b", thus completing a function application of < to "AZ" and "b". The resulting state of the AST stack is given by 2d.

Figure 3 provides pseudocode for a complete CBGP function application compilation algorithm with backtracking. Although this algorithm has a higher time complexity compared to the original CBGP compilation algorithm, we have not observed a dramatic cost increase in practice. This is because argument backtracking only occurs on the order of the number of type variables which is always between [0,3] both in the function set used by our CBGP system and most "real world" code. Also, the architecture of CBGP only requires compilation to be performed once per genome, the cost of which is almost always dominated by running the compiled program on every training case.

```
procedure App($ASTs$)                              ▷ Given an AST stack.
    $functionASTs \leftarrow [a \in ASTs \mid Type(a)$ is a function$]$
    for all $functionAST \leftarrow functionASTs$ do
        $ASTs' \leftarrow [a \in ASTs \mid a \neq functionAST]$
        $types \leftarrow$ ArgTypes$(functionAST)$
        $argASTs \leftarrow$ FindAsts$(ASTs', types, \{\})$
        if $argASTs \neq BACKTRACK$ then
            $ASTs.remove(functionAST)$
            $ASTs.removeAll(argASTs)$
            $ASTs.push(Apply(functionAST, argASTs))$
            return $STOP$
        end if
    end for
    return $NOOP$
end procedure

function FindAsts($ASTs, types, typeVars$)
    $t \leftarrow$ First$(types).substitute(typeVars)$
    $safe \leftarrow [a \in ASTs \mid$ Type$(a)$ is a $t]$
    $attempted \leftarrow \{\}$
    for all $arg \in safe$ do
        if $Type(arg) \notin attempted$ then
            $newBindings \leftarrow$ Unify$(t', Type(arg))$
            $ASTs' \leftarrow [a \in ASTs \mid a \neq arg]$
            $types' \leftarrow$ DropFirst$(types)$
            if $types'$ is empty then
                return $[arg]$                     ▷ A list containing just $arg$
            end if
            $typeVars' \leftarrow typeVars.addAll(newBindings)$
            $restOfArgs \leftarrow$ FindAsts$(ASTs', types',$
                                            $typeVars')$
            if $restOfArgs = BACKTRACK$ then
                Add$(attempted, Type(arg))$
            else
                return Concat$([arg], restOfArgs)$
            end if
        end if
    end for
    return $BACKTRACK$
end function
```

Figure 3

4 EXPERIMENTAL DESIGN & METHODS

We present an empirical study into the problem solving performance of our modifications using a suite of benchmark problems for which CBGP has previous published results using the original compilation semantics from Figure 1. We perform our study using identical search methods to these previous CBGP results: a generation genetic algorithm with the same selection methods and genetic operators, only varying when and how function application happens. This section will provide an overview of our benchmark problems and the search algorithm used in our experiments.

The benchmark problems selected for this study are the 17 problems designed to assess a program synthesis system's capability of finding solution programs that require the use of parametric

				Gene-Annotated				
Problem	**Explicit**	**Implicit**	**Skips**	**0.25**	**0.375**	**0.5**	**0.625**	**0.75**
area-of-rectangle	59	**88**	**78**	68	**83**	**78**	**82**	**83**
centimeters-to-meters	92	97	98	97	97	98	**99**	98
count-true	100	100	100	100	100	100	100	100
filter-bounds	13	6	**35**	23	**34**	**31**	23	20
first-index-of-true	100	100	100	100	100	100	100	99
get-vals-of-key	12	**33**	**25**	10	21	**30**	**39**	**37**
max-applied-fn	24	24	**51**	**43**	**45**	**42**	**39**	32
min-key	31	**67**	32	37	40	**49**	**53**	**62**
set-cartesian-product	0	0	0	0	0	0	0	0
set-symmetric-difference	50	**83**	62	*18*	37	50	58	62
sets-with-element	4	0	3	0	0	1	0	0
simple-encryption	96	99	100	97	99	98	100	100
sum-2-vals	94	98	95	90	96	99	99	96
sum-2-vals-polymorphic	100	100	99	100	95	99	100	100
sum-2D	100	*90*	100	99	100	98	96	98
sum-vector-vals	16	5	13	21	24	21	26	16
time-sheet	2	**10**	3	3	3	5	9	6
Count sig better:	-	5	4	1	3	5	5	3
Count sig worse:	-	1	0	1	0	0	0	0

Table 2: The number of successes out of 100 GP runs. All results are compared to "Explicit", which was the CBGP default before this work; results that are significantly better are in bold, and results that are significantly worse are *underlined and italicized.* The "Explicit" and "Implicit" columns are for explicit and implicit function application. "Skips" is implicit function application with skip genes. We give five different sets of "Gene-Annotated" runs, with the column headers giving the application rates. The rows "Count sig better" and "Count sig worse" give the count of how many problems each method is significantly better/worse than Explicit respectively.

polymorphism and nested data types, hereby referred to as the *parametrically polymorphic program synthesis benchmark* (P3SB) suite [20]. We are not aware of any published results that use these problems with program synthesis methods besides CBGP (though there is nothing about the problems that is specific to CBGP), thus we do not make any claims to the relative performance of CBGP to other methods. Instead, we present evidence of an improvement over the original specification of CBGP.

The P3SB suite was selected because CBGP is known to have more variation of solution rates across individual problems compared to its performance on other genetic programming benchmarks. Studies using the PSB1 benchmark suite [13] showed that CBGP solves some problems 100% of the time, and others it never solves, with very few problems somewhere in between. [21] This makes statistical measurements of the changes to solution rates from our modifications difficult.

Each P3SB problem is an inductive program synthesis task, where the target program must be synthesized from training cases of program inputs and target outputs. We provide the CBGP system with 200 randomly generated training cases to evaluate individuals during evolution. Synthesized programs which pass all training cases, ending the evolutionary run, are evaluated on an additional 2000 randomly generated test cases to assess if the program is a generalizing solution. We compare the number of generalized solutions between sets of runs; statistical significance is determined using a chi-square test with a 0.05 significance level. The error functions used to evaluate programs are based on the output data type of the synthesized program, and are the same as those used in [20].

The CBGP implementation used here compiles genomes into native Clojure code.[1] Table 1 presents the hyperparameters used in our experimental runs. The function and literal sets (aka, the genetic source [12]) used to sample genes for mutation and the initial population vary by problem according to which data types are relevant. Prior research into CBGP on the P3SB problems [20] presented specific inclusion criteria for functions and literals which this work adheres to for the sake of comparison.

5 RESULTS

We present results of experiments for function application triggers and backtracking below. Unless stated otherwise, we use one or the other, not both.

5.1 Function Application Triggers

We present results for our function application triggers experiments in Table 2. All methods for increasing the number of function applications produced significantly better results on some problems, and were rarely significantly worse. The methods with most significant increases in success rate are implicit, implicit with skip genes, and gene-annotated with moderate application rates.

[1]https://github.com/erp12/cbgp-lite

Problem	None	F	F&A	F&A + GA
area-of-rectangle	59	67	73	62
centimeters-to-meters	92	98	**99**	95
count-true	100	100	100	100
filter-bounds	13	10	**26**	**28**
first-index-of-true	100	100	100	100
get-vals-of-key	12	**48**	**62**	**90**
max-applied-fn	24	35	**52**	33
min-key	31	**64**	**81**	**86**
set-cartesian-product	0	0	0	0
set-symmetric-diff	50	**91**	**89**	**87**
sets-with-element	4	3	7	5
simple-encryption	96	100	100	100
sum-2-vals	94	98	99	97
sum-2-vals-poly	100	100	100	100
sum-2D	100	97	95	97
sum-vector-vals	16	**55**	**71**	**43**
time-sheet	2	9	**14**	**15**
Count sig better:	-	4	8	6

Table 3: The number of successes out of 100 GP runs. All results are compared to "None"; results that are significantly better are in bold; no results were significantly worse. "None" uses CBGP without backtracking. "F" uses function backtracking, and "F&A" uses function and argument backtracking. "F&A + GA" uses function and argument backtracking, as well as gene-annotated function application triggering with a rate of 0.625. The row "Count sig better" gives the count of how many problems each method is significantly better than "None".

When considering the five rates we tested for gene-annotated function application, we found that at either extreme (0.25 or 0.75 application rate), performance decreased compared to using rates closer to 0.5. In particular, having too few or too many genes that conduct function application degrades performance. Interestingly, a rate of 0.75 performed worse than implicit function application, which could be thought of as a 1.0 application rate.

5.2 Application Backtracking

Table 3 presents the results of three different settings of application backtracking with the original compilation algorithm, which performs no backtracking. The first three columns use explicit function application triggers, and the fourth uses gene-annotated application triggering at a rate of 0.625. Doing just function backtracking gave significantly better results on 4 problems, while using function and argument backtracking was significantly better on 8 problems. Finally, the combined use of function and argument backtracking with gene-annotated application triggering gave significantly better results on 6 problems.

6 DISCUSSION

All of our function application trigger methods, which increase how often function application happens, improved performance across our benchmark problems. These results address **RQ1**: increasing function application is beneficial to CBGP's performance. While the particular method for increasing does have some effect on performance, all methods improve performance, meaning the previously-used explicit application likely does not provide a sufficient amount of function application to compile useful programs.

Of the methods we considered, implicit, implicit with skip genes, and gene-annotated with moderate application rates all improve significantly on 4 to 5 problems. While any of these may be a reasonable choice, implicit is significantly worse on one problem, and skips is not significantly better on as many problems as gene-annotated. We therefore recommend using gene-annotated function application, with a moderate application rate of 0.5 or 0.625.

In answer to **RQ2**, when considering backtracking methods, performing both function and argument backtracking is the clear winner compared to just doing function backtracking or doing neither. All of these methods use explicit function application, so they will not perform function application as frequently as the application trigger methods. Still, backtracking likely performs many more successful function applications, since the applications that are attempted are more likely to find functions and arguments that can be applied. Argument backtracking seems to have a real impact beyond function backtracking, producing the best results of any treatment in our experiments.

One important note is that function backtracking can lead to successful function applications using AST stacks that would otherwise never be able to successfully apply a function, no matter how many function application procedures were carried out. In particular, if a function or argument is buried on the stack under another function or argument, function application will never be able to find and apply it. Thus backtracking can produce qualitatively different compilations, where function application triggers can only change how often function application happens.

We had hoped that a combination of a function application trigger method and backtracking would gain the benefits of both, producing better results. Unfortunately, combining one of the best application trigger methods (gene-annotated at a rate of 0.625) with function and argument backtracking produced slightly worse results than by simply using backtracking. This gives an initial answer to **RQ3**: adding application triggers to backtracking does not improve performance over backtracking alone.

Perhaps backtracking by itself is sufficient to produce a reasonable number of successful function applications, and performing more function applications is overkill. On the other hand, this combination gave the second best results of our experiments, and it would be more useful on problems requiring even larger solution programs. Additionally, an application trigger that produces fewer applications (such as gene-annotated with a lower rate) may be more beneficial when backtracking makes more of the function applications successful. Additional work is needed here to distinguish further.

In summary, all methods to encourage more successful function applications produced improvements. While function application triggers helped on some problems, backtracking of both functions and arguments was the most successful.

7 RELATED WORK

At the time of writing, techniques for general program synthesis using *large language models* (LLMs) are receiving a lot of attention based on impressive results and widespread use in real world applications [1, 3, 6, 23]. Some of these methods have been compared with or combined with GP [17, 18, 24]. Some practitioners may feel that non-LLM methods for program synthesis, such as genetic programming, are simply not contemporary enough to warrant further research; however there are important distinctions that highlight the respective benefits. LLMs and genetic programming typically use different program specifications. Genetic programming systems, such as CGBP, are usually *inductive* and leverage a sample of input-output training cases, while LLMs are provided a natural language "prompt" describing the target program. Another crucial distinction is cost. Training a LLM capable of general program synthesis requires a large, curated, corpus of code and a training procedure that is many orders of magnitude more expensive than a single genetic programming run. In contrast, genetic programming requires relatively few training samples, yet is more expensive than applying a pre-trained LLM to a new synthesis task. Therefore, the most efficient method will depend on the availability of an LLM pre-trained with suitable knowledge for the problem domain.

Other recent genetic programming methods have been motivated by some of the same issues (production of type-safe code, use of polymorphism) as CBGP. HOTGP [8], based on strongly-typed GP [19], produces code that can use higher-order functions and parametric polymorphism. Garrow compares evolved solution programs in Python and Haskell (which are closer to CBGP's programs), and finds that the Haskell solutions tend to be more likely to generalize to unseen data [10].

The concept of backtracking is borrowed from the fields of *constraint satisfaction* [2, 16] and *logic programming* [4]. Generally, our function and argument backtracking algorithm is searching for a function and its arguments that meet the constraints of type unification, backtracking when we discover that the constraints cannot be met. In logic programming languages such as Prolog, backtracking is used to repeatedly find unifications that instantiate variables in an attempt to satisfy a query. These unifications with values is very similar to our unifications with types, resulting in a very similar backtracking algorithm.

8 CONCLUSION

In this paper we propose a change to the canonical compilation procedure for Code Building Genetic Programming with the goal of facilitating the synthesis of more sophisticated ASTs through the use of function application. This proposal is supported by empirical results on a suite of benchmark problems that show all of our proposed changes have a net positive impact on the solution rates of CBGP. However, the modification of adding backtracking stood out as having the highest solution rates, needing the least configuration, and allowing the CBGP compilation process to synthesize more function applications beyond what is possible with an increase in number of attempted function applications without backtracking.

Although this research has presented a clear improvement to the problem solving capabilities of CBGP, additional study is required to determine how this modified system compares to other genetic programming and program synthesis methods. Furthermore, a more causal study into how changes to the CBGP compilation algorithm impact the size and shape of synthesized programs would likely yield insights that would inform additional improvements to CBGP, particularly on more difficult benchmarks.

ACKNOWLEDGMENTS

We thank Kien Tran, Anthony Hevia, and other members of the PUSH lab for discussions that improved this work. This material is based upon work supported by the National Science Foundation under Grant No. 2117377. 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.

REFERENCES

[1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs.PL]

[2] Sally C Brailsford, Chris N Potts, and Barbara M Smith. 1999. Constraint satisfaction problems: Algorithms and applications. *European journal of operational research* 119, 3 (1999), 557–581.

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[4] William F Clocksin and Christopher S Mellish. 2003. *Programming in PROLOG*. Springer Science & Business Media, Berlin.

[5] Li Ding, Edward Pantridge, and Lee Spector. 2023. Probabilistic Lexicase Selection. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Lisbon, Portugal) *(GECCO '23)*. Association for Computing Machinery, New York, NY, USA, 1073–1081. https://doi.org/10.1145/3583131.3590375

[6] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1469–1481. https://doi.org/10.1109/ICSE48619.2023.00128

[7] Austin J. Ferguson, Jose Guadalupe Hernandez, Daniel Junghans, Alexander Lalejini, Emily Dolson, and Charles Ofria. 2019. Characterizing the effects of random subsampling and dilution on Lexicase selection. In *Genetic Programming Theory and Practice XVII*, Wolfgang Banzhaf, Erik Goodman, Leigh Sheneman, Leonardo Trujillo, and Bill Worzel (Eds.). Springer, East Lansing, MI, USA, 1–23. https://doi.org/doi:10.1007/978-3-030-39958-0_1

[8] Matheus Campos Fernandes, Fabrício Olivetti de França, and Emilio Francesquini. 2023. HOTGP–Higher-Order Typed Genetic Programming. *arXiv preprint arXiv:2304.03200* (2023).

[9] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2018. Towards effective semantic operators for program synthesis in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Kyoto, Japan) *(GECCO '18)*. Association for Computing Machinery, New York, NY, USA, 1119–1126. https://doi.org/10.1145/3205455.3205592

[10] Fraser Garrow, Michael A. Lones, and Robert Stewart. 2022. Why functional program synthesis matters (in the realm of genetic programming). In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Boston, Massachusetts) *(GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 1844–1853. https://doi.org/10.1145/3520304.3534045

[11] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2018. Program Synthesis Using Uniform Mutation by Addition and Deletion. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Kyoto, Japan) *(GECCO '18)*. ACM, New York, NY, USA, 1127–1134. https://doi.org/10.1145/3205455.3205603

[12] Thomas Helmuth, Edward Pantridge, Grace Woolson, and Lee Spector. 2020. Genetic Source Sensitivity and Transfer Learning in Genetic Programming. In

Artificial Life Conference Proceedings. MIT Press, 303–311. https://doi.org/10.1162/isal_a_00326

[13] Thomas Helmuth and Lee Spector. 2015. General Program Synthesis Benchmark Suite. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation* (Madrid, Spain) *(GECCO '15)*. Association for Computing Machinery, New York, NY, USA, 1039–1046. https://doi.org/10.1145/2739480.2754769

[14] Thomas Helmuth, Lee Spector, and James Matheson. 2015. Solving Uncompromising Problems with Lexicase Selection. *IEEE Transactions on Evolutionary Computation* 19, 5 (Oct. 2015), 630–643. https://doi.org/doi:10.1109/TEVC.2014.2362729

[15] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA.

[16] Vipin Kumar. 1992. Algorithms for constraint-satisfaction problems: A survey. *AI magazine* 13, 1 (1992), 32–32.

[17] Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O. Stanley. 2024. *Evolution Through Large Models.* Springer Nature Singapore, Singapore, 331–366. https://doi.org/10.1007/978-981-99-3814-8_11

[18] Vadim Liventsev, Anastasiia Grishina, Aki Härmä, and Leon Moonen. 2023. Fully Autonomous Programming with Large Language Models. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Lisbon, Portugal) *(GECCO '23)*. Association for Computing Machinery, New York, NY, USA, 1146–1155. https://doi.org/10.1145/3583131.3590481

[19] David J. Montana. 1995. Strongly Typed Genetic Programming. *Evolutionary Computation* 3, 2 (06 1995), 199–230. https://doi.org/10.1162/evco.1995.3.2.199 arXiv:https://direct.mit.edu/evco/article-pdf/3/2/199/1492842/evco.1995.3.2.199.pdf

[20] Edward Pantridge and Thomas Helmuth. 2023. Solving Novel Program Synthesis Problems with Genetic Programming using Parametric Polymorphism. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Lisbon, Portugal) *(GECCO '23)*. Association for Computing Machinery, New York, NY, USA, 1175–1183. https://doi.org/10.1145/3583131.3590502

[21] Edward Pantridge, Thomas Helmuth, and Lee Spector. 2022. Functional Code Building Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Boston, Massachusetts) *(GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 1000–1008. https://doi.org/10.1145/3512290.3528866

[22] Edward Pantridge and Lee Spector. 2020. Code Building Genetic Programming. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference* (Cancún, Mexico) *(GECCO '20)*. Association for Computing Machinery, New York, NY, USA, 994–1002. https://doi.org/10.1145/3377930.3390239

[23] Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2021. Multi-modal program inference: a marriage of pre-trained language models and component-based synthesis. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–29.

[24] Dominik Sobania, Martin Briesch, and Franz Rothlauf. 2022. Choose Your Programming Copilot: A Comparison of the Program Synthesis Performance of Github Copilot and Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Boston, Massachusetts) *(GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 1019–1027. https://doi.org/10.1145/3512290.3528700

[25] Dominik Sobania, Dirk Schweim, and Franz Rothlauf. 2023. A Comprehensive Survey on Program Synthesis With Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation* 27, 1 (2023), 82–97. https://doi.org/10.1109/TEVC.2022.3162324