Tacit Programming Code Synthesis and Optimization with Genetic Algorithms

Adam McDaniel
Tickle College of Engineering
University of Tennessee, Knoxville
Knoxville, Tennessee
amcdan23@vols.utk.edu

Colby Smith
Tickle College of Engineering
University of Tennessee, Knoxville
Knoxville, Tennessee
csmit402@vols.utk.edu

***Abstract*—Evolutionary algorithms and genetic programming techniques have been widely employed for code generation and optimization.[8] In this paper, we apply evolutionary algorithms to the tacit programming paradigm, where instructions act as combinators that can be flexibly restructured and reordered into pipelines. As a part of this exploration, we propose a novel specialized Turing machine-based architecture that is particularly well-suited for genetic programming. Furthermore, a compiler has been developed that targets this instruction set, enabling the application of our evolutionary algorithm techniques to compiler-generated code.**

***Index Terms*—component, formatting, style, styling, insert**

CONTENTS

I Introduction 1

II Methods 2
- II-A SKI Combinator Calculus 2
  - II-A1 Overview 2
  - II-A2 Implementation 2
  - II-A3 Genome Representation . 2
  - II-A4 Evolutionary Algorithm . 3
- II-B Sage Compiler and Virtual Machine 3
  - II-B1 Compiler Overview . . . 3
  - II-B2 Virtual Machine Overview 4
  - II-B3 Instruction Set 4
  - II-B4 Genome Representation . 4
  - II-B5 Program Synthesis Evolutionary Algorithm . . . 4
  - II-B6 Program Optimization . . 4

III Results 5
- III-A SKI Combinator Calculus 5
- III-B Sage Virtual Machine Code Synthesis 5
- III-C Sage Virtual Machine Code Optimization 5

IV Discussion 7
- IV-A Expression Composition and Highly Disruptive Mutations 7
- IV-B Specialized Turing Machine Architecture Suitable for Evolutionary Algorithms 7

V Conclusion 7

VI Future Work 7

I. INTRODUCTION

a) Evolutionary Algorithms and Applications to Code Synthesis and Optimization: Existing techniques for applying genetic algorithms to code synthesis or optimization are predominantly centered on evolving programs under traditional architectures such as x86[7], LLVM[5], or various high level source languages like LISP or C. The code expressed in these languages typically revolves around creating expressions which recursively compose the results of sub-expressions. This expression tree representation of a program can have side effects: the compositional nature of these expressions means that slightly different expression trees exhibit very distinct behavior. When these trees are used as genetic representations, mutations on these trees are highly disruptive as a result of this compositional nature. Highly disruptive mutations worsen the evolutionary algorithm's ability to converge on a solution or refine an existing one to be more optimal.

b) Applying Evolutionary Algorithms to Tacit-Programming Paradigm: Our first goal was to explore evolutionary algorithms to new architectures which might be well suited to genomes. Tacit programming seemed like a great option because it promotes a code structure very reminiscent of DNA: each of the instructions is just a predefined combinator that can be inserted into a given pipeline. This structure is also easy to represent and manipulate as a genome: the program is represented as a list of instructions, and each instruction is denoted by a unique number. This organization of code structure proved to be inherently better suited to a genetic algorithm approach compared to languages that support binding values to variables. Code structures that support variables can cause programs that are very similar in terms of instructions to be very different in terms of functionality. Evolutionary algorithms work best with genomes that can be tweaked by small amounts that are reflected in the genome's capabilities or functionalities.

c) SKI Combinator Calculus: Our first attempt at evolving code was through SKI-combinator calculus because its code representation seemed to be well suited

to genetic representation. A genome represented in SKI combinator calculus is very simple: it is a combination of 3 functions applied to one another in a successive fashion. This fit the requirements that we wanted: A program that can be represented as a list of instructions and mutated at random to produce different genomes. This representation did not turn out to be successful, for reasons that we detailed above: due to the compositional nature of λ calculus, from which SKI-combinator calculus is derived[4]. That is, recursively evaluating sub-expressions that get reused and reshaped, and reapplied to other sub-expressions. Small deviations in the organization of the tree made huge changes to the behavior of the program. For the reasons detailed above, it was impossible to get programs to converge on any solutions or improve an existing program. To improve an existing program would mean massively rewriting the function applications to be perfect, and any slight deviation would lead to a completely non-functional but closely related variant.

d) Sage: Specialized Turing Machine Architecture: To mitigate this issue, we used an architecture consisting of a straightforward collection of 24 core instructions that can be arranged into programs. This also fit the requirements, while also not having the flaw of drastically changing the genome's functionality from a minor mutation. It is designed to be easily implementable, and flexible enough to represent high-level concepts. This makes it suitable for genetic programming. By adopting the tacit programming approach, we depart from the traditional method of evolving syntax trees, which involves complex expressions recursively composed of sub-expressions. Instead, we employ a more "flattened" approach to computation, utilizing longer sequences of simpler operations. This helps avoid drastic mutations that arise from altering the relationships between branches and leaves in the expression tree. As a result, mutated programs within this instruction-set exhibit less divergence from their parent program in terms of function. This enhances the effectiveness of our genetic algorithm by promoting gradual changes and speeds up convergence by minimizing large changes to the genome through disruptive mutations.

This citation is about how we set up our genetic algorithm with our programs.[8]

II. Methods

A. SKI Combinator Calculus

1) Overview: SKI combinator calculus works by successive applications of the following λ functions.

$$S = \lambda x.\lambda y.\lambda z.(xz(yz))$$

$$K = \lambda x.\lambda y.x$$

$$I = \lambda x.x$$

Any λ calculus abstraction can be encoded by applications of these functions. Below is an example implementation of the Y combinator in SKI calculus.

$$Y = S(K(SII))(S(S(KS)K)(K(SII)))$$

The evolutionary algorithm represents applications of combinators by treating the application operation as another standalone instruction. This instruction is treated like any of the other encoded combinators in the sequence of tacit programming-based operations to execute.

2) Implementation: The library and test code for the SKI combinator calculus evolution is implemented in Rust and leverages the genevo library to perform crossover, mutation, tournament selection, and assist fitness evaluation. The library contains implementations for evolving programs composed of several other optional combinators in addition to the base SKI combinators. The intention of this was to make the evolution smoother by introducing a wider variety of possible genes which can simplify certain operations that an evolved program might benefit from.

$$W = \lambda x.xx$$

$$B = \lambda f.\lambda g.\lambda x.f(gx)$$

$$C = \lambda f.\lambda g.\lambda x.(fx)g$$

Programs encoded in these functions are randomly generated in their genome representation, converted into an internal representation of the combinator calculus program, and then evaluated with a custom interpreter. This interpreter performs tail recursion elimination in order to evaluate these massive call graphs without overflowing the stack: recursion and self reference are the only bits of bubble gum and duct tape holding this system of logic together.

3) Genome Representation: Evolved programs are represented as a vector of integers which represent each of the combinators. Genomes can be of arbitrary length. The genome is converted into a program by converting each of integers into their respective combinator. These are successively pushed onto a stack of combinators. When the application combinator is reached, it takes the top two combinators on the stack, applies them, and pushes the result back onto the stack. If an apply is used in an impossible manner, such as trying to apply combinators when the stack of expressions is empty, then the genome is rejected. Here is a table illustrating the encodings of each combinator.

Value in Genome	Combinator
1	S
2	K
3	I
4	W
5	B
6	C
7	Apply last two
*	Does not affect representation

Here is an example genome representation using the encoding from the table above and its corresponding combinator representation.

```
52062721768697870233745490429
44376134444433732749118924450
31135048130450399760632572237
79201948303814913766943323814
60099861144173104677583740703
15068671982961265954181281762
21541717478959377725047382135
00974611105365441100040263511
43545862158548166541913231880
69711973060946871620764947157
54031159783999467199652939425
15598325928356108800316838071
08828080521279694891747520156
73695824140522166212093880980
14399413051604335025506737436
68622445160628893330173836394
34389380735153213027342124006
4588490
```

Example genome representation for combinator calculus evolution.

```
( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( B K ) C
K ) K S ) C C ) K I ) I W ) B W ) W
K ) W W ) I C ) S I ) W W ) W W ) W
I ) I I ) K W ) S S ) K W ) W B ) I
S ) S I ) B W ) S I ) W
```

The combinator representation for the genome in the previous figure.

4) Evolutionary Algorithm: The following hyperparameters were found to be the best for evolution upon testing ranges of very low mutation rates to very high mutation rates. These hyperparameters were used to gather our results in our experiments.

Evolutionary Algorithm Hyperparameter	Value
Population size	2000
Maximum epochs	20000
Individuals per parent	10
Selection ratio	0.3
Number of crossover points	10
Mutation rate	0.1
Reinsertion rate	0.6

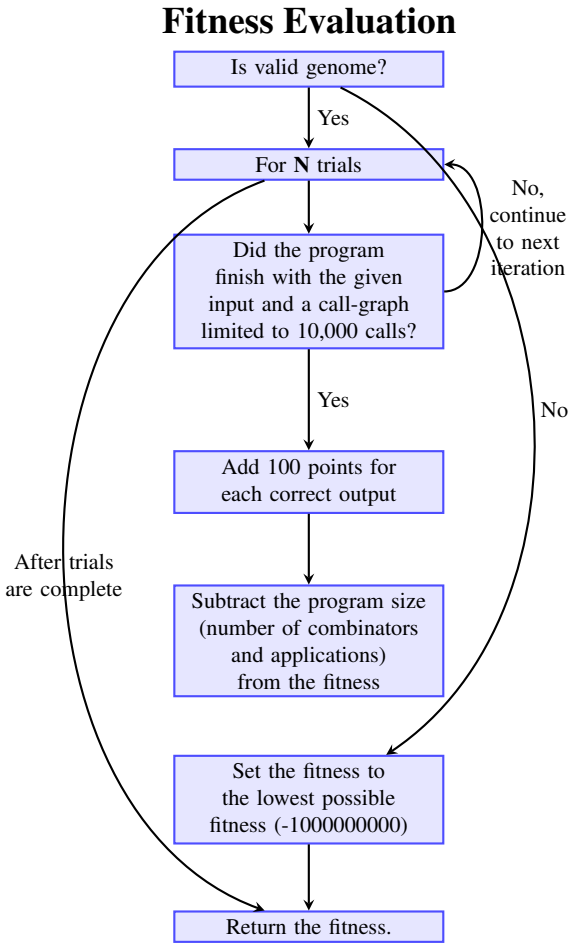
The genomes are evaluated by converting them into their combinator representation, and then grading the resulting expression by applying it to an input. Some attempted program evolutions involve logical XOR, OR, and the Church encoding for Boolean false given below. Genetic algorithms involving λ calculus and its subsets has been researched before.[10]

$$False = \lambda x.\lambda y.y$$

$$XOR = \lambda p.\lambda q.p(q(False)(p))(q(q)(p))$$

$$OR = \lambda p.\lambda q.p(p)(q)$$

The fitness evaluation function applies the evolved program to the input, checks for correctness, and also gives extra fitness points for smaller program size. The fitness function is described by the function below.



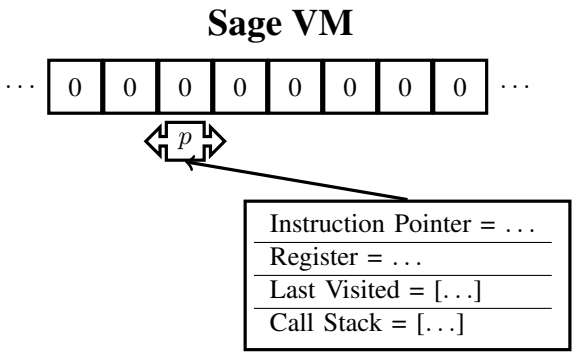
B. Sage Compiler and Virtual Machine

Sage targets a low level virtual machine architecture that supports very minimal instructions. These instructions are based on a very simple Turing tape and a read/write head. Only two of the 24 operators the virtual machine supports require operands, and these operands are constant integer literal values. Previous work has already been done on evolving Turing machines.[6][9][1]

1) Compiler Overview: The compiler takes source code that is roughly a C-like language. The source language compiles down to an assembly-like intermediate representation, which can also be written by the user to add certain functionalities that can't be achieved in the high level language. Such functionalities include implementing an allocator, or calling the "allocate" optional assembly

instruction. Examples of functions which implement bitwise operators and memory manipulation functions in the source code with this inline assembly can be seen below.

2) Virtual Machine Overview: The virtual machine is composed of a Turing tape, a read/write head, a single register, a stack of "last visited" positions on the tape, and a call stack. The stack of last visited positions is utilized by two operators to temporarily navigate to another portion of the tape and subsequently return to the previous position. The call stack is used to implement function calls, and allows the virtual machine to avoid implementing an arbitrary "jump" instruction to encode control flow. This allows evolution to be even smoother, as arbitrary jump instructions are difficult to control and could be highly disruptive.



3) Instruction Set: The Sage compiler requires the following "core" instructions to be implemented when compiling to a target architecture. These instructions perform basic operations like shifting the position of the read/write head, setting the value of the register, setting a label that can be "called", arithmetic operations performed on the register with data from the tape.

Move(Int)	Index	Where?
Deref	Refer	BitwiseNand
While	If	Else
Function	Call	Return
End	Add	Subtract
Multiply	Divide	Remainder
Set(Int)	Save	Restore
IsNonNegative?	Get	Put

These instructions implement additional features that are optionally supported by the compiler. The compiler allows the user to write inline assembly instructions (such as floating point math instructions) that utilize these Turing tape operations.

Allocate	Free	Set(Float)
ToInt	ToFloat	Power
Sine	Cosine	Tangent
ArcSine	ArcCosine	ArcTangent
Add	Subtract	IsNonNegative?
Multiply	Divide	Remainder
Peek	Poke	

Only the "allocate" and "free" instructions were used from the above optional instructions in the experiments; no floating point operations or foreign function interface via the "peek" and "poke" instructions were used. The evolutionary algorithm also accepts the use of three extra instructions. The "For" instruction performs a loop while the register is greater than zero. The remaining two are "IncrementRegister" and "IncrementTape", which do exactly as their names describe. These are not used when optimizing a compiled program, only for program synthesis.

4) Genome Representation: The genome representation is very similar to the representation for SKI combinator calculus: every instruction in the pool is assigned an integer value, and a list of random integers is generated.

5) Program Synthesis Evolutionary Algorithm: The following hyperparameters were found to be the best for evolution upon testing ranges of various low mutation rates to very high mutation rates. These hyperparameters were used for our results. The genevo library was not used to perform the evolutionary algorithm; techniques from past research was applied to implement the algorithm[8].

Evolutionary Algorithm Hyperparameter	Value
Population size	300
Maximum epochs	300
Individuals per parent	2
Selection ratio	0.1
Number of crossover points	Random # >1
Mutation rate	0.01
Reinsertion rate	0.1

a) Fill Tape with Ones: To evolve a program that writes ones to the virtual machine's memory, we used the following fitness function to guide the evolutionary algorithm.

$$Fitness = tape.count(1) * 1000 - sizeof(code) * 10$$

This rewards programs for writing ones to the tape, and also removes points in proportion to code size. As a result, the program should converge on a small program that maximizes ones on the tape.

6) Program Optimization: To evolve optimizations to existing programs, the output VM code is parsed and loaded into the interpreter instance in the evolutionary algorithm. The first generation is filled with copies of this program, and subsequent generations form mutations and recombinations of the programs. We utilized the optimizing evolutionary algorithm on two programs: one to compute factorial and another to compute Quicksort. To evaluate the fitness of a factorial program, the fitness function checks how many times the program produces

correct output, and divides by the program size. To evaluate the fitness of a quicksort program, the distance between an output list of numbers and the correct list of numbers needs to be measured. The distance between sorted lists is determined by randomly sampling pairs in the list and confirming that they are in the right order for some number of trials. The fitness of a Quicksort program is the sum of correctly sorted pairs divided by the program size.

III. Results

A. SKI Combinator Calculus

The results for the SKI combinator calculus implementation were less than desirable, likely due to the same principles that make syntax tree based approaches less effective in certain scenarios. The use of an absurdly high mutation rate of 10% is due to the fact that the evolutionary algorithm works more like a brute-force search rather than smooth evolution. The compositional nature of the λ-calculus system of logic that these combinators are built on leads to highly disruptive mutations, so slow incremental improvement is next to impossible[2]. The program was able to evolve a solution for the Church encoding for false, but this is just because the solution is the application of two combinators: SK. The evolutionary algorithm was unable to find solutions for any other functions. Even worse, the evolutionary algorithm struggled to generate programs that would get any correct outputs, making the fitness function always return zero.

B. Sage Virtual Machine Code Synthesis

Although the SKI combinator calculus portion of the exploration was not successful, synthesizing Sage VM code had much better results. Figure 1 depicts the best program randomly evolved in the first generation. Figure 2 depicts 15 generations later, by which point the fitness function has shrunk the program. Figure 3 depicts the final program that was selected for: an optimal solution that is an order of magnitude smaller and more effective than its ancestors. The first depicted solution haphazardly places a few ones on the tape, the second depicted solution places a few more, and the final result fills the tape (up until the virtual machine instance is halted by the evolutionary algorithm).

C. Sage Virtual Machine Code Optimization

The optimizations achieved by the evolutionary algorithm were the most impressive results achieved by the program. The tables below illustrate the improved metrics of each program.

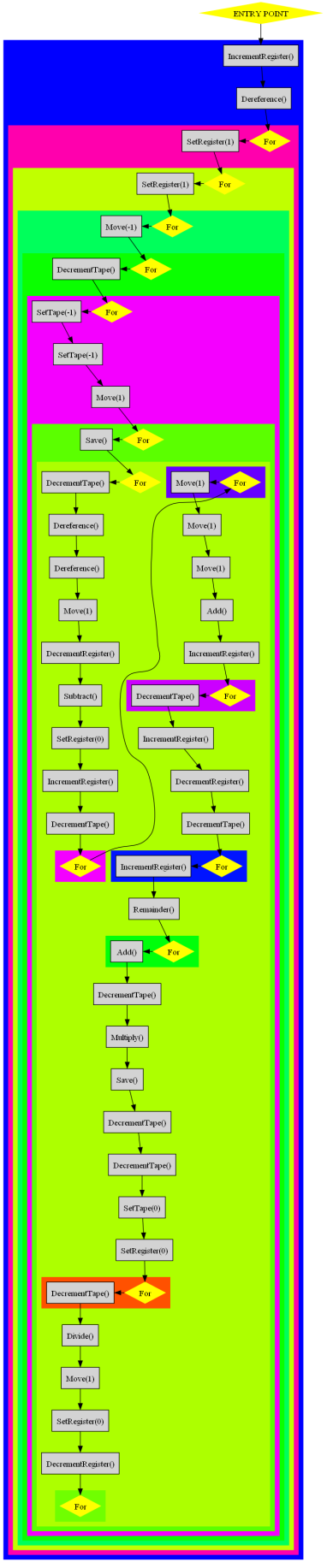


Fig. 1. This is the best program at epoch 0: evolving code to fill the VM memory with ones.

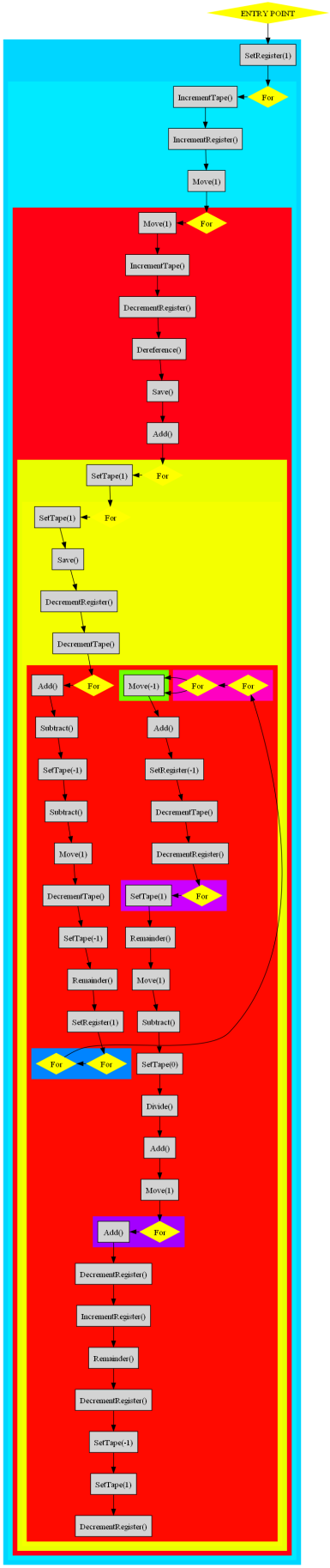


Fig. 2. This is the best program at epoch 15: evolving code to fill the VM memory with ones.

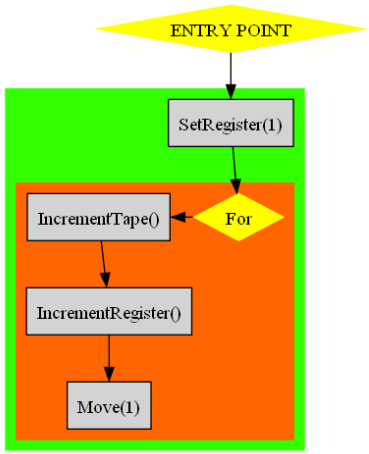


Fig. 3. This is the final program at epoch 25: an optimal solution to fill the memory with ones.

Factorial Evolution Metrics

Epoch #	Metric	Value
0	Fitness	0.03816793893129771
	Code Size	394
100	Fitness	0.04285714285714286
	Code Size	350
200	Fitness	0.04437869822485207
	Code Size	338
300	Fitness	0.04789225458071856
	Code Size	314

Quicksort Evolution Metrics

Epoch #	Metric	Value
0	Fitness	0.07908264136022143
	Code Size	5058
100	Fitness	0.07974481658692185
	Code Size	5016
200	Fitness	0.08038585209003216
	Code Size	4976
300	Fitness	0.08100445524503848
	Code Size	4938

Overall, the factorial program saw a code size decrease from 394 instructions to 314 instructions: a 20.4% improvement. The quicksort program saw a code size decrease from 5058 instructions to 4938 instructions, which is a 2.4% improvement. The optimizations in these improvements are typically adjustments in how the program moves data around to improve locality. For example, instead of performing unnecessary operations to shift data around and clean up the stack after the factorial is recursively called and computed, the optimized programs streamline the read/write head's movements to go directly to where the factorial is computed at the end of the stack of calls and output the value. Finally, memory clean up operations added by the compiler at the end of the main function scope are discarded to shrink code size as

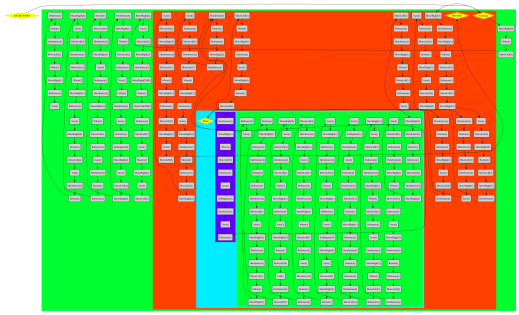


Fig. 4. This is the best factorial program at epoch 0

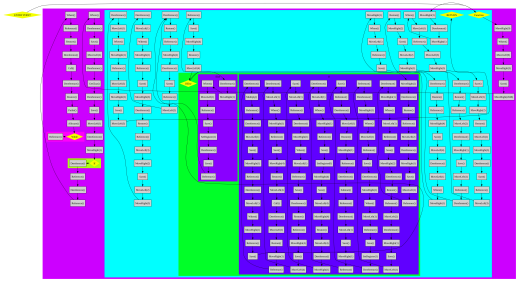


Fig. 5. This is the best program at epoch 300: evolving code to perform factorial with smaller code size (20.4% smaller)

well. Evolutionary algorithms are often thought to be the second-best way to solve a problem, and other, more rigid methods of optimizing quicksort have been researched, but this method proved successful.[3]

IV. Discussion

A. Expression Composition and Highly Disruptive Mutations

The SKI combinator calculus implementation's lack of success can likely be attributed to the highly compositional nature of λ calculus. Recursive functions have very large effects when haphazardly mutated into a λ calculus abstraction: they cause chain reactions that affect how subsequent calls are made further down the call stack. SKI combinator calculus still fundamentally relies on evaluating compositions of expressions and subexpressions, which seems to be the common theme for code representations that are not well suited to genetic programming.

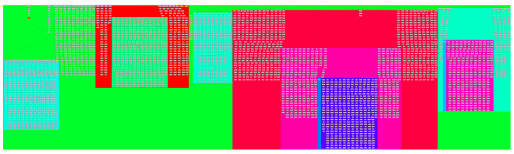


Fig. 6. This is the best quicksort program at epoch 0

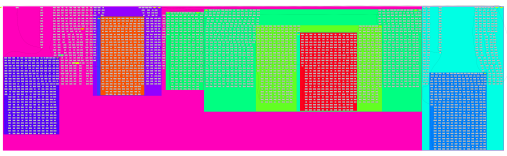


Fig. 7. This is the best program at epoch 300: evolving code to perform quicksort with smaller code size (2.4% smaller)

B. Specialized Turing Machine Architecture Suitable for Evolutionary Algorithms

The effects of the individual Turing tape based instructions are considered negligible in comparison. This makes the Turing tape based architecture much more suitable for evolution guided optimizations.

V. Conclusion

a) SKI Combinator Calculus and Highly Disruptive Mutations: The conclusion of this approach is that it failed. It proved to be too difficult to evolve code structures whose functionality drastically changed after minor mutations. We were able to evolve programs using this method, but it wasn't performing better than a completely random search. This means that evolution was not the driving force behind the programs' improvement.

b) Effectiveness of Tacit-Programming Evolutionary Algorithm Approach: This approach proved successful. Both synthesizing code from scratch and optimizing existing code were able to be done by the evolutionary algorithm. Synthesizing code and then optimizing the synthesized code was significantly more successful than optimizing existing code structures generated by a compiler. This can be explained by the code generated by a compiler after being written by a human is much better code than code generated by an evolutionary algorithm.

VI. Future Work

a) Applying Learned Techniques in Instruction Sets and High-Level Languages Not Based in Tacit-Programming: The techniques used in this paper can be applied to higher level languages by evolving code blocks consisting of point-free expressions that each individually have an incremental effect on the program state. Expressions that both depend on and affect the expressions around them at runtime (composing their behavior to be greater than the sum of their parts) are more difficult to evolve than expressions that perform statically predictable operations. Future work on applying these optimizations to higher level languages could explore evolving combinations of Turing-tape-inspired code segments in x86: treating the stack pointer as a Turing-tape read/write head and evolving these very simple operations.

b) Creating Virtual Machines and Languages Suited to Evolutionary Code Generation and Optimization: There is a lot more work to be done in exploring instruction set architectures that are specifically well designed for evolution guided optimization. Exploring more complex Turing-tape based systems seems promising, but these systems would likely have to be abstractions over existing architectures (LLVM; x86) to see adoption. It appears that creating virtual machine architectures particularly suited for genetic programming could give substantial performance gains to compilers that perform little or inefficient

optimizations (which can easily result from poor inlining heuristics).

References

[1] Mark Burgin and Eugene Eberbach. *Recursively Generated Evolutionary Turing Machines and Evolutionary Automata*, pages 201–230. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[2] Mohammad Hamdan. A dynamic polynomial mutation for evolutionary multi-objective optimization algorithms. *International Journal on Artificial Intelligence Tools*, 20(01):209–219, 2011.

[3] Md. Sabir Hossain, Snaholata Mondal, Rahma Simin Ali, and Mohammad Hasan. Optimizing complexity of quick sort. In Nirbhay Chaubey, Satyen Parikh, and Kiran Amin, editors, *Computing Science, Communication and Security*, pages 329–339, Singapore, 2020. Springer Singapore.

[4] Oleg Kiselyov. λ λ to ski, semantically: Declarative pearl. In *Functional and Logic Programming: 14th International Symposium, FLOPS 2018, Nagoya, Japan, May 9–11, 2018, Proceedings 14*, pages 33–50. Springer, 2018.

[5] William B Langdon, Afnan Al-Subaihin, Aymeric Blot, and David Clark. Genetic improvement of llvm intermediate representation. In *Genetic Programming: 26th European Conference, EuroGP 2023, Held as Part of EvoStar 2023, Brno, Czech Republic, April 12–14, 2023, Proceedings*, pages 244–259. Springer, 2023.

[6] Amashini Naidoo and Nelishia Pillay. Using genetic programming for turing machine induction. In Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Isabel Esparcia Alcázar, Ivanoe De Falco, Antonio Della Cioppa, and Ernesto Tarantino, editors, *Genetic Programming*, pages 350–361, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[7] Eric Schulte, Stephanie Forrest, and Westley Weimer. Automated program repair through the evolution of assembly code. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, page 313–316, New York, NY, USA, 2010. Association for Computing Machinery.

[8] Adam Slowik and Halina Kwasnicka. Evolutionary algorithms and their applications to engineering problems. *Neural Computing and Applications*, 32(16):12363–12379, Aug 2020.

[9] Julio Tanomaru. Evolving turing machines from examples. In Jin-Kao Hao, Evelyne Lutton, Edmund Ronald, Marc Schoenauer, and Dominique Snyers, editors, *Artificial Evolution*, pages 167–180, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[10] Xiaomeng Ye. *Evolving Lambda-calculus functions using genetic Programming*. PhD thesis, The College of Wooster, 2014.