SuperHuman: Program Synthesis for the Game Human Resource Machine

Ruoyi Wang Jiaqi Li Jiaxun He
Simon Fraser University
{rwa81, jla470, vhe}@sfu.ca

1 Introduction

Human Resource Machine is a popular game and the players are asked to compose short programs to solve some specific problems. However, the players are only allowed to use limited instructions and memory space. Moreover, there is only one register. This makes the game interesting and attracting. However, it is not always easy to tackle the size challenge and speed challenge. Inspired by a paper about synthesizing SQL queries using I/O examples[9], we tried to adapt the techniques described in that paper so as to find the solutions to the problems in the game. We followed the approach of search-and-verification to program synthesis. We begin with a basic search algorithm, where we tried to enumerate all possible permutations of instructions. The verification is implemented with Sympy[2], a Python library for symbolic computation.

However, the basic search algorithm is very inefficient due to the vast search space, so some optimization techniques are required for the synthesis algorithm to terminate within reasonable time. Four optimization techniques prove to be very effective: pruning invalid outputs, avoiding invalid operations, merging same states and incorporate verification into the search process. The first three techniques are based on concrete I/O examples and have to use breadth-first search. The last technique are based on symbolic I/O constraints and can be implemented with depth-first search. Combining all these techniques together, we found a blazing fast synthesis algorithm for programs without branches or loops, which can produce a correct program for the hardest level without branches or loops within 5 seconds.

To find out how effective each optimization technique is, we conducted several experiments. We applied these algorithms to the problems in the game and got some observations. Our algorithms can solve up to 17 levels in a reasonable time. Moreover, we evaluated how effective the optimization technique is and found that merging states has the most significant effect in I/O examples-based synthesis, and incorporating verification into the search dramatically improves the efficiency of the search algorithm, making it possible to solve even the hardest problem almost instantly.

To solve problems that require branches, we implemented a complete but not very efficient algorithm. With careful preparation a correct program can be produced in several minutes, although the efficiency depends on how good the preparation is.

Many approaches we tried did not work or are incomplete, including Rosette, search based on three-address code, and using multiple I/O examples to further prune the search space. A clear plan for future works is also made, where we pointed out the key problems and possible solutions.

2 Background

Human Resource Machine is a game where the player is asked to write assembly programs. In each level the player is given a set of instructions, exactly one register, an input stream, an output stream, and limited memory space to solve the specified problem.

Each level has a size challenge, which requires the player to use limited number of instructions, and a speed challenge, which requires the program to terminate in limited steps. As the game progresses, the instruction set and the problems become increasingly complex and it becomes difficult even to write a right program. Our project aims at applying the techniques of program synthesis to automatically generate programs that solves the specified program and beats the size challenge.

The game can be roughly divided into 3 phases, each phase adds more instructions to the previous one, enabling more complex constructs. The first phase has no branch instructions. The second phase has branches. The third phase has pointers.

In this project, we focus on and solved every problem in the first phase, with some success on the second phase. The full instruction set for the first phase are described below:

add x add the data in memory location x to the register.

sub x subtract the data in memory location x from the register.

copyfrom x copy the data in memory location x to the register.

copyto x copy the data in the register to memory location x.

jump i jump to the i-th instruction and starts executing there.

inbox fetch the first data from the input stream into the register. If there is no more data in the input stream, halt the program.

outbox append the data in the register to the output stream, then clear the register.

Note that since there is only unconditional jump, branching is not possible without introducing dead code. Therefore the first phase cannot have branches (but can have loops).

The full instruction set of the second phase adds the following instructions to the first phase as described above:

bump+ x add the data in memory location x by 1, and copy the result to the register.

bump- x subtract the data in memory location x by 1, and copy the result to the register.

jump-if-zero i jump to the i-th instruction and start executing there if the data in the register is 0, otherwise continue executing at the next instruction.

jump-if-negative i jump to the i-th instruction and start executing there if the data in the register is negative, otherwise continue executing at the next instruction.

Note that with two conditional jumps available, branches are possible in the second phase and loops can be more complex. Also note that not every level in the same phase has access to the full instruction set of that phase.

3 Basics

This section describes the basics of our approach. We followed the basic search-and-verification approach to program synthesis. An abstract representation of the game, which serves as the foundation of further steps, is described in Section 3.1. The basic search algorithm is described in Section 3.2. The method to perform verification is described in Section 3.3.

3.1 Problem Abstraction

The first step to deal with the problem is to make an abstract model of the game, so that we can perform verification on a synthesized program. When a game state is modeled, the instructions can be modeled as functions from the old state to a new state. A straightforward model is to simply mirror the settings of the game, which means to model a game status as a tuple that consists of:

1. an input stream (a stream of integers)
2. a register (an integer)
3. the memory (an array of integers or null)
4. an output stream (a stream of integers)

Ideally, verification only needs to consider whether the synthesized program produces the desired output stream given any valid input stream. However, this is very difficult to achieve for the following reasons:

1. Streams are very difficult to verify and represent. Because the input stream does not have a predefined length, in order to verify them, we must use universal quantifiers ($\forall$). However, no known verification techniques work with universal quantifiers. The Grounding technique used in [6] cannot be easily applied because not all input streams are valid. For the same reason, it is hard to find a good representation of the input stream such that the inbox instruction always fetches a valid input.
2. This model makes the process of synthesis more complex. Most problems in the game are element-wise, such as 'do something for every two elements in the input stream'. The faithful model of the game introduced above would always require the program body to have a loop, which is hard to generate and verify.

Therefore, we used a slightly different model of the game, based on the observation that most problems in the game are element-wise. We do not try to synthesize the whole program. Instead, we construct the main loop for element-wise processing, and only synthesize the loop body. The main loop looks like:

1. preparation

2. loop body

3. jump to preparation

We assume that we can always find an optimal solution where all inbox instructions appear only in the preparation step. Therefore, we do not need to consider inbox instructions in the loop body, and as a result we no longer need to model the input stream. The memory model is also changed to only store allocated memory, because access-before-store is always an invalid operation so we prevent such operations in the model. After the program is synthesized, we need to map the memory address. The new model of our game is[1]:

1. a register (an integer)

2. the memory (an vector of integers)

3. an output stream (a stream of integers)

3.2 Search

First, we define a search state as a code sequence, and the depth is the length of the code sequence. In the actual implementation, a search state does not contain the whole sequence, but only the last instruction in the sequence and the pointer to the previous search state. Therefore, we can always recover the whole code sequence while avoiding storing redundant data in memory. Depth is also part of the search state so that we can know the depth of a search state efficiently.

The basic idea is, to find all programs in depth d, append an instruction to each program produced in depth $d-1$. If a program passed the verification, it is output. The instructions to append are generated by calculate all permutations of the instructions and their parameters, which includes all memory locations. An pseudo-code is provided to clarify this procedure. (1)

The naive search algorithm is complete, but the search space is very large because it tries the permutations of instructions and their parameters Therefore, the naive search algorithm by itself cannot efficiently solve the problem. However, certain optimization techniques can enable us to find redundant states and remove them from search space. The details are covered in Section 4.

[1] In the actual implementation, the two changes are reverted back but their effects are preserved. The input stream is preserved but contains only elements in one loop. The reduced memory model is replaced with optimization techniques achieving the same goal. The approach to only synthesize the loop body is preserved.

Algorithm 1 Basic Search

```
1:  procedure BASICSEARCH(max_depth)
2:      initial_st = Build initial state
3:      stacks = Empty list for 0-max_depth
4:      Put initial_st into stacks[0]
5:      for i = 0 to max_depth do
6:          for each state in stacks[i] do
7:              new_st = Enumerate next possible states
8:              stacks[i+1].append(new_st)
9:          end for
10:     end for
11:     paths=Backtrack all states in stack[max_depth]
12:     output=path in paths passing verification
13:     return output
14: end procedure
```

3.3 Verification

The search algorithm cannot guarantee the correctness of the searched result. To prove a program is correct, we need to verify it. Our approach to verify a program is to leverage a Python library called Sympy[2]. Sympy is a Python library for symbolic computation, with which you can define symbolic variables. When arithmetic operations are performed on symbolic variables, the result is a symbolic expression. Because there are only addition and subtraction in the game, to verify a program, we simply send symbolic variables as input and check whether the output is symbolically equivalent to the desired output. This observation is further exploited to incorporate verification into the search process. The details are covered in Section 4.3.

4 Optimization Techniques

Based on the basic search algorithm in Section 3.2, we added some techniques to prune the search space so as to improve the search efficiency. All the techniques, except the one described in Section 4.3, are based on concrete I/O examples.

4.1 Avoid Redundant Sequences

By observing the states, we have found that there are many states that are redundant. By removing these states, the search space would be smaller. There are basically two types of such states.

4.1.1 Prune Invalid Outputs

The first type of redundant states is the states that contain invalid output elements. A state with a valid output o means that its output is a prefix subset of the target

output t, which means $o \bigcap t = o$ and o is a prefix of t. According to this fact, there comes the definition of an invalid output.

Definition 4.1. *An invalid output o is an output sequence where $o \bigcap t \neq o$ or o is not a prefix of the target output t.*

For a state that already contains an invalid output sequence, since it is impossible to change any content in the output sequence, there is no such an action that removes the invalid element in the output sequence to make the state valid. Thus, it is unnecessary to continue searching the forward states from an invalid state. So, any state with invalid output sequence should be pruned.

4.1.2 Avoid Invalid Operations

The second type of redundant states is the states that contains redundant read or write operations on some of memory locations. Consider a memory location l. A "*copyTo l*" instruction copies a number in the register to l. If such an instruction is followed by another "*copyTo l*" without using the element at l after the first copy, the first copy is useless and should be considered redundant. Similarly, a *copyFrom* can also be redundant if it is followed by another *copyFrom* without using the element copied at the first time. Here comes the definition of an invalid operation.

Definition 4.2. *An invalid operation is an operation which causes the result of previous operation(s) to be unused or reassigned.*

To identify invalid operations, a bit sequence is generated for the memory in each state. A memory location is marked *Written* and becomes invalid to write before next read. Each time a memory location is accessed, its corresponding bit will be checked and if the access is invalid, no operations will be performed.

4.2 Merge Equivalent States

First, we define the equivalence of two game states.

Definition 4.3. *Two game states are equivalent if they have the same input sequence, memory, output sequence and bit sequence.*

If there exists multiple code sequences that could reach the same game states with the same steps, these game states can be combined and only perform next-step search once. However, all the code sequences should be preserved. When reaching the final game states, the multiple code sequences should all be recovered. To perform this method, when searching the next possible states, we always start from the game state instead of the search state with instruction tuples, but the search states are always saved in the back pointers.

4.3 Incorporate Verification into Search

The technique of merging equivalent states, described in Section 4.2, suffers from two shortcomings. The first one is when we discovered an equivalent state, it cannot be discarded, but must be saved and recovered in the final result, because two states might be equivalent only under the specific input. The recovery step is typically very slow as it calculates the Cartesian product of all code sequences. Based on the first shortcoming comes the second one, which is two search states can only be merged when their corresponding code sequence have the same depth. A search state with 5 instructions cannot be merged with a search state with 2 instructions, because the merged state will not have a well-defined depth. This limits the effectiveness of state merging and dictates a breadth-first search.

The key problem here is a lack of confidence, due to the incompleteness of concrete I/O examples. To avoid the above problem, we can directly pass the symbolic variables as the input, so that we can prune and merge states based on symbolic expressions, effectively incorporating verification into the search process. In this approach, when two search states produce equivalent game states, they are considered equivalent. This would enable us to search much more efficiently. First, equivalent states no longer need to be saved because now we can be absolutely sure it is redundant. Second, merging search states across different depth is possible, because the merge never happens and equivalent states are just discarded. This also enables us to perform depth-first search, discarding any search state already seen by maintaining a 'seen' set. The search ends whenever we find a program that produces the desired output.

The catch here is the verification must be efficient enough so as to not add too much overhead to the search process. This is guaranteed by the observation that there are only addition and subtraction in the game. Therefore, any symbolic expression that can appear in the game is a linear combination of the symbolic variables, which can be efficiently stored and checked for equality by only storing as a vector the coefficient of each symbolic variable and the constant. Without such observation, we cannot make such a guarantee, and two symbolic expressions must be converted to a canonical form before they can be compared for equality[1], which can be very slow.

This optimization, combined with previously mentioned ones, gives rise to a blazing fast synthesis algorithm for the first tier of the game, where branching is not required to solve the problem. In the most difficult level in tier 1, the Tetracontiplier, the algorithm can produce a correct program within 5 seconds while other approaches without this optimization would require several minutes.

5 Synthesis Programs that have Branches

Although the algorithm described in Section 4.3 solved all the problems in tier 1 perfectly, it cannot deal with branches. The biggest problem is Sympy cannot encode flow control into the symbolic expression. Ideas to deal with the problem are discussed in Section 9.1. In fact, branching introduces such a big challenge that none of the optimization techniques described in section 4 can be applied. The reasons are described in Section 5.2.

5.1 Guarantee the Completeness of Search

The original search algorithm, described in Section 3.2, can be modified to support searches. However, it is not so intuitive. The problem is we are performing iterative search, therefore, if the algorithm decides to add a jump instruction at depth 3 which jumps to the fifth instruction, the fourth instruction is not yet known, making the behavior of the code sequence ill-defined. The solution here is, instead of inserting jump instructions, we try to append a *jumpfrom* instruction, along with every other instructions, at each iteration. This instruction records the line number it jumps from, by backtracking the execution history and find all possible line numbers. We extended our instruction set by adding *jumpFrom* and *jumpIfZeroFrom*. When a code sequence is found, we can recover the original program by extracting all *jumpFrom* and *jumpIfZeroFrom* instructions to find where the original *jump* and *jumpIfZero* instructions should be inserted.

Such a search algorithm is complete, which means it will not miss a correct program. However this approach is very inefficient and breaks all optimization techniques. The generated programs also cannot be verified in the current implementation; the challenge and ideas are described in Section 9.1. However, with carefully picked I/O examples, manually preparing some steps and choosing a good instruction set, and using depth-first search so that we can kill the search program once a correct program is found, correct programs for problems in level $7-17$ in tier 2 can be found in a few minutes. The running time depends on how good the preparations are; a correct program is guaranteed to be outputted eventually.

5.2 Challenge of Branching

This technique dealing with branches fails some state pruning strategies. Since *jump* instructions may skip some states in a execution path, it is unsafe to prune all states with wrong outputs since they may be skipped in later iterations. For the same reason merging equivalent states is useless, because a *jumpFrom* instruction might need to be inserted inside the merged code sequence. In this situation, we need to try each code sequence separately, even if they are merged before, because their behavior might be different when the jump instruction is inserted, thus nullifying the effect of merging. The techniques described in Section 4.1.2 may still be applied, but we did not figure it out.

6 Evaluation

To evaluate our approaches above, we mainly chose some real problems from the Human Resource Machine game. Our algorithm is implemented in Python and for symbolic executions, we used Sympy package. All the tests are done on the a 13-inch MacBook Pro with 2.6GHz Intel Core i5 CPU and 8GB memory running macOS 10.13 and Python 3.6.

6.1 Performance Analysis

This section mainly focus on the performance of the BFS and DFS algorithm. We selected the first 15 problems in the game and removed the main loop. Some problems (7, 9, 13, 14, 16, 17) require *jump* related instructions and others do not.

The Heuristic Search algorithm is based on breadth-first search and I/O examples since it is used to compare to Depth-First search. All the optimizations in BFS search is turned on. And as indicated above, the DFS search does not have any optimization. For each run, we use the oracle "size challenge" as the search depth limit so that the answers are guaranteed to satisfy the size challenge. The number in the table shows the "speed challenge". If our algorithms generate multiple solutions, we take the one that satisfies the challenges best.

Level Name	Oracle	Heuristic BFS	DFS Search
1-Mail Room	6	6	6
2-Busy Mail	25	25	25
3-Copy Floor	6	6	6
4-Scrambler	21	21	21
6-Rainy Summer	24	24	24
7-Zero Exter.	23	23	23
8-Tripler	24	24	24
9-Zero Pres.	25	25	25
10-Octoplier	36	36	36
11-Sub	40	40	40
12-Tetracont.	56	56	56
13-Equaliz.	27	-	30
14-Maximiz.	34	-	34
16-Abs.	36	-	36
17-Exclusive	28	-	34

The tables shows that the basic heuristic search algorithm could almost find the best solutions for each level that does not require *jump* related instructions. For some *jump* related problems, it could still work. However, when the *jump* related problems requires a larger number of instructions to solve, the BFS algorithm may fail in a reasonable time. However, the DFS algorithm could always find a solution.

For the problems that do not require *jump*, both of the algorithms could pass the speed challenge. Since there is no branch, all the instructions are in the same basic block. The number of required steps is related to the size challenge. Thus, both algorithms perform the same since search depth is fixed. However, for the problems that require *jump*, the *jump* instructions would change the execution order so that the execution speed might be harder to control. Even if the DFS algorithm finds correct solultions, they are not always optimal.

6.2 Efficiency

This section evaluates the performance of different pruning methods used in the BFS heuristic search. For this section, we use different metric to measure the performance. The first one is the time each algorithm takes for the same I/O examples and memory status. The second is the number of search nodes that the algorithm visits. A smaller visited node number would lead to a better efficiency. Since there are three optimization methods, we performed five tests, one with no optimization, one with all optimization and others with one single optimization technique.

The test in this section is based on the problem *8-Tripler*. To observe the difference results, the memory size is set to 3. All tests are started with the same I/O example.

Optimization Techniques	Execution Time	Visited Search Nodes
No Optimization	1.3509s	8826
Prune Invalid Output	0.4939s	3913
Prune Useless Copy	0.4347s	3608
Merge Same States	0.2935s	1375
All Optimization	0.0868s	538

The table shows that any optimization is significantly helpful in shrinking search space. Based on the number of visited nodes, "Merge Same States" only need to visit 15% nodes in total. Pruning invalid output and useless *Copy* can also prune half of the states. The combination of all optimization techniques makes it possible to only visit 6% of the total search nodes, which can be helpful when solving larger scale problems.

6.3 Symbolic Computation

This section evaluates the effect of the technique described in Section 4.3. Since symbolic calculation guarantees to find correct code sequences, rather than I/O examples, the performance should be good. However, the we need to evaluate if such symbolic execution would slow down the search process.

For this section, problem 12-Tetracont. is selected. The I/O example based search is based on the BFS algorithm with all optimization techniques. Since we are not sure if its answers are working, we need to collect all answers for validation, which is not measured here. For Symbolic based search, DFS search algorithm is applied. It will terminate once it finds a correct answer.

Example Type	Execution Time	Visited Search Nodes
Concrete	461.5589s	690494
Symbolic	5.4182s	11714

The table shows that for I/O examples based algorithm, it cannot solve large scale problems in a short time because it has to generate all possible paths for validation. However, the symbolic based DFS search would handle these problems in a much more reasonable time. However, the trade-off is that DFS search does not guarantee to find the most optimal way before it terminated. With all paths generated by BFS search, it can be easier to find the shortest one.

7 Other Approaches

This section introduces approaches we have tried but are either incomplete or prove to be ineffective or not working. The approaches and their shortcomings are described below.

7.1 Synthesis with Rosette

Our first attempt to perform the program synthesis is to use Rosette. Given the definition of each instruction and the final verification constraints, Rosette can automatically synthesize the program with the given instruction set that fits the verification constraints. However, certain usability issues prevented us from leveraging this tool effectively. The problem is all of our instructions inherently requires a parameter, besides the game state. For example, the add instruction takes an address as the parameter. We want Rosette to synthesize the add instruction as well as the parameter, but not the game status before it. However, we cannot find out how to tell Rosette to distinguish the inherent parameter from the game state. In all the examples we can find, the functions to choose from all take no parameter other than the

previous state of the symbolic variable. If we do not include the inherent parameter in the constraint, Rosette would fail. As a result, we have to give up Rosette.

7.2 Three-address Code

Although each arithmetic instruction in the game takes only 1 parameter, they actually takes 2 additional implicit parameters. These instructions assume there is something in the register, which must come from somewhere (implicit second source). The result must be used somewhere else, otherwise it will just be overwritten and the instruction would be a no-op (implicit destination). With these observations, we can transform the original instruction set into three-address code, where each parameter is a memory location. In this abstraction we no longer need the notion of register.

This abstraction nicely fits in the memory-to-memory model[3], but the I/O instructions and constants needs to be taken care of. This abstraction is intended to be used in rewriting-based synthesis, which is discussed in Section 9.2. Since this abstraction has the potential to avoid redundancy in the first place, a search based on this abstraction is described in Section 7.3, which does not give a good result.

7.3 Search based on Three-address Code

Since redundant *copyTo* and *copyFrom* can make the path really deep, we tried to make use of the new abstraction described in Section 7.2 to avoid them in the first place, since this abstraction has no notion of register. By eliminating the operations between memory and register, the depth of the search would shrink, so that the search could terminate in fewer steps, but the branching factor of the search tree will also grow dramatically.

The original search algorithm is modified to fit the three-address code abstraction. Since one instruction in three-address code form can map to two or three game instructions, the challenge is to control the total number of the instructions after the search in each step. Intuitively, we need to evaluate the current instruction length in each state, which requires a evaluation of the previous instructions. Our strategy is to create n stacks, where n is the oracle instruction length in the game. For each search state, we evaluate the length t of game instructions until this state. Then, we put this new state into the stack indexed by t.

However, this might generate redundant instructions. Here is an example.

$$add : 0, 0, 1 \quad (1)$$

$$add : 1, 0, 0 \quad (2)$$

Generally, an *add* or *sub* instruction would be mapped to a game instruction of length 3 and *inbox* and *outbox* can be mapped to a length of 2. However, instruction (1) and (2) actually have some overlaps. Instruction (1) copies the register value to memory location 1 and instruction (2) copies the value in memory location 1 to the register. However, these two steps are not necessary. These two three-address code instructions should be mapped to four game instructions. This can be solved by evaluating the memory address of two neighbouring instructions.

However, this can be challenging since we need to go over each past search state in order to evaluate the length of the instructions, making the optimization of merging equivalent states described in Section 4.2 useless.

7.4 Optimization using LLVM

Since we do not have enough time to implement the idea described in Section 9.2, we tried to use the built-in optimizer of LLVM to see if it would work. The result is very bad, as the instruction set of LLVM is much more abstract than the one in the game. Therefore, for the level the Tetracontiplier, where the player is asked to multiply an element by 40, LLVM will directly use a multiplication instruction to achieve the result.

7.5 Prune with multiple I/O examples

We also tried to use multiple I/O examples to see if it can further prune the search space. However, this techniques seems pretty useless. The reason is, first, I/O examples can only prune away code sequences that ends in an outbox instruction, yet outbox instructions is only a very small part of all possible instruction in that step. Therefore even with perfect pruning, it would not reduce the search space dramatically. Second, for most cases, even with only one I/O example, the output does not contain any wrong programs. Therefore one pair of I/O example is enough.

8 Related Work

8.1 Rosette

Rossette[8] is a Racket library for syntax-guided program synthesis. The idea to describe the search space with syntax, and use symbolic variables to verify a program is derived from Rosette. Although look similar, the way Rosette verifies a program is very different from our approach to use Sympy. Rosette encodes the symbolic execution as an SMT formula and uses an SMT solver to verify the program, while we make use of Python's duck

typing to actually execute the program with symbolic inputs and check the symbolic form of the output.

8.2 Synthesis with I/O Examples

Many ideas of our project comes from [9], including pruning with I/O examples and merging equivalent states. Their work used a more abstract language to limit the breadth of the search tree and used encoding to speed up the search process, but it is unknown how we can adapt these techniques to fit our problem.

8.3 Superoptimization

Superoptimizers[4] has been around since the 80's. Our project can also be viewed as a superoptimizer, since it tries to find an optimal, loop-free program with the desired behavior. STOKE[7], the recently introduced stochastic superoptimizer for loop-free x86 program sequences, is an interesting approach as our future work.

9 Future Work

9.1 Incorporate Verification into Search for Branches

As introduced in Section 4.3, incorporating verification into the search process can dramatically improve the synthesis efficiency. However, this technique cannot be adapted to programs with branches straightforwardly, since Sympy cannot encode flow control into the symbolic expression. To circumvent this problem, we can try to describe the symbolic expression as a piecewise linear function. When comparing two piecewise linear functions for equality, we need to perform the following steps:

1. Region detection. Two equivalent piecewise functions might look different because they describe the regions in different ways. To deal with the problem, we either merge equivalent regions in each function, or find all regions defined by both functions. This can be difficult when the number of variables are large.

2. Compare equivalence of symbolic expressions within each region. This should be very straightforward as described in Section 4.3. Because linear functions are always continuous within a region, the end points do not require special treatments. The only catch is lone points, where we must substitute all symbolic variables with their values defined at this point.

How to build the piecewise functions incrementally is still a problem. The desired output must also be expressed as a piecewise function, but it should not cause any problem. It might even have some benefits, because with regions explicitly expressed, we can try to search for symbolic expressions that match the boundary conditions of each region, then search for symbolic expressions that match the function body within each region, and link them together.

9.2 Rewriting-based Synthesis

The search-based approach used in this project has the advantage of guaranteeing the result is optimal, if a result is found. However it has a noticeable shortcoming that the search process does not take the nature of the program into consideration. This approach generally will probe a vast search search space, searching for only one or two correct answers. For problems like the Tetracontiplier, this approach works quite well as it can discover a solution that is hard to conceive by human players. However, for other problems in the game, the player can easily come up with an algorithm. The tricky part is to implement the algorithm with the weird instruction set under the size constraint.

Given the above observation, a different approach to our project would be start with a user-provided correct program, and try to rewrite or optimize the correct program to reduce its size. The three-address code form, introduced in Section 7.2, was originally intended for this purpose, as most existing optimization techniques are based on three-address code. The I/O instructions require careful modeling, as they may introduce side effects hidden from the optimizer.

Traditional optimization technique typically do not guarantee an optimal result, though it may be enough to beat the game. If we do want to find the optimal result, we can try to adapt the technique introduced in [5], which means applying a database of rewriting rules to rewrite the correct program so as to reduce its size. STOKE[7] is also an interesting project which we can adapt some techniques from. This approach would also have the potential to deal with loops, which is nearly impossible in the search-based approach.

10 Conclusion

In our project, we tried a wide variety of approaches, and we found a very powerful synthesis algorithm for programs without branches or loops. We also implemented a complete but not very efficient algorithm for searching programs that have branches but no loops. We also made a clear plan for future works, by pointing out the key problems and possible solutions.

References

[1] Gotchas and pitfalls - sympy 1.1.1 documentation. http://docs.sympy.org/latest/gotchas.htmldouble-equals-signs. Accessed: 2017-12-08.

[2] Sympy. http://www.sympy.org/en/index.html. Accessed: 2017-12-08.

[3] HATHHORN, C. Engineering a compiler, second edition by keith d. cooper and linda torczon. *SIGSOFT Softw. Eng. Notes 37*, 1 (Jan. 2012), 335.

[4] MASSALIN, H. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems* (Los Alamitos, CA, USA, 1987), ASPLOS II, IEEE Computer Society Press, pp. 122–126.

[5] PANCHEKHA, P., SANCHEZ-STERN, A., WILCOX, J. R., AND TATLOCK, Z. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2015), PLDI '15, ACM, pp. 1–11.

[6] PANCHEKHA, P., AND TORLAK, E. Automated reasoning for web page layout. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2016), OOPSLA 2016, ACM, pp. 181–194.

[7] SCHKUFZA, E., SHARMA, R., AND AIKEN, A. Stochastic program optimization. *Commun. ACM 59*, 2 (Jan. 2016), 114–122.

[8] TORLAK, E., AND BODIK, R. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (New York, NY, USA, 2013), Onward! 2013, ACM, pp. 135–152.

[9] WANG, C., CHEUNG, A., AND BODIK, R. Synthesizing highly expressive sql queries from input-output examples. *SIGPLAN Not. 52*, 6 (June 2017), 452–466.