Synthesising Tree Traversal Programs from Examples

Report for Project 2

Benjamin Bichsel Marcel Mohler Gaurav Parthasarathy

ETH Zurich

{bichselb,mohlerm,gauravp}@student.ethz.ch

Abstract

We introduce a concept to efficiently synthesize tree traversals in a JavaScript AST. Instead of finding an explicit path from a start to a target node, we gather constraints about the context of the target node with respect to the context of the start node. We then search for the first node that satisfies these constraints. Furthermore, we provide a working implementation of this concept that is very efficient under some reasonable assumptions.

1. Introduction

Syntactic structure of programming language source code can be represented in the form of an Abstract Syntax Tree (AST). Move operations possible on this tree can be defined by a Domain Specific Language (DSL) and examples of possible operations are *moving to first child (DownFirst)*, *moving to parent node (up)*, etc. A chain of DSL operations starting in a start node and ending at a target node can be seen as a specific type of a tree traversal program.
Consider the simple JavaScript code snippet in Listing 1 and its corresponding AST in Figure 1.

```
function neg(a)  {
    return -a;
}
```

Listing 1: Example JavaScript program

Imagine we want to jump from the return statement (node with Id 5) to the function name (node with Id 2). Starting from node 5 and using the program $Up \rightarrow Up \rightarrow DownFirst$ reaches the target node 2.

We present an efficient synthesis procedure, which, given a set of start and target nodes in a JavaScript AST, finds a suitable program. Applying this program to the start nodes yields the corresponding target nodes. For previously unseen start nodes, we then use the synthesized program to predict appropriate target nodes.
In our approach, we do not use the DSL directly to find an explicit path from start to target node. Instead our synthesizer uses the DSL to capture the *context* of the two nodes. This means that we describe the target node by capturing constraints of it and its close relatives (parents, children etc). Our tree traversal program then takes a new start node and runs a backwards search to find the last node which satisfies these constraints.
In Section 2 we provide a detailed explanation of our approach as well as concrete semantics of the tree traversal programs we used. We continue by comparing our tree synthesis approach to other concepts as well as analyzing its limitations in Section 3. We also provide a detailed evaluation focusing on error rate, performance, strength and weaknesses in Section 4. Finally, we conclude with some final notes and mention possible further work in Sections 5 and 6.

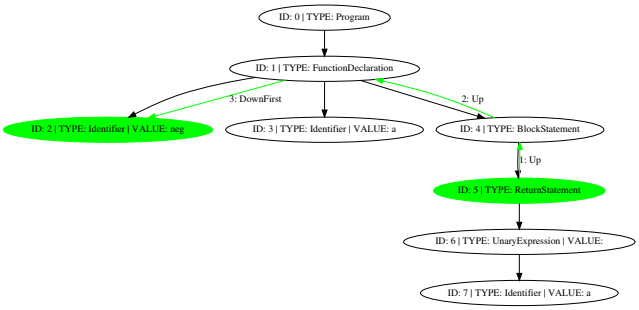


Figure 1: The AST derived derived from Listing 1

2. Our solution

2.1 Overview

Our main idea is to describe a possible solution to the program synthesis task as a formula f. Each formula consists of a number of so-called checks, which represent a property of the context of the nodes $start$ and $target$. We formalize the concepts below.
After constructing f, we use the theorem prover Z3 (De Moura and Bjørner 2008) to solve it. We then translate the solution obtained from Z3 to an actual tree traversal program.

2.2 Formalizing our approach

In the following, we explain how we build the formula that describes the possible solutions to our program synthesis task.
The formula consists of a part for each of the examples we consider.

$$\begin{aligned} f = {} & example_1(start_1, target_1) \wedge \\ & example_2(start_2, target_2) \wedge \\ & \dots \\ & example_{n\_examples}(start_{n\_examples}, target_{n\_examples}) \end{aligned}$$

For simplicity, here we consider only the case where all examples come from a single program. Thus, an example is simply a pair

$(start_i, target_i)$ with the intended meaning that our synthesized program should go to $target$ when starting from $start$. Note however that our approach also generalizes to the case of multiple programs.

2.2.1 Checks

For each example, we generate n_checks checks, and the $target$ must satisfy all of them.

$$\begin{aligned} example_i(start, target) = \\ & check_1(start, target) \wedge \\ & check_2(start, target) \wedge \\ & \dots \\ & check_{n\_checks}(start, target) \wedge \\ & no\ false\ target\ between\ target\ and\ start \end{aligned}$$

Moreover, we prohibit $false\ targets$. This means that no node between $target$ and $start$ should satisfy all checks. In other words, each node between $target$ and $start$ has to fail at least one check. This enables us to formulate the synthesized program as follows:

> Given start node $start$, we go back node by node in pre-order until we find a node that satisfies all checks. We return this node as $target$.

2.2.2 A single check

The i-th check derives a value $value_{i,start}$ from the context of $start$ and a value $value_{i,target}$ from the context of $target$. The check is satisfied if $value_{i,start} = value_{i,target}$.

$$\begin{aligned} check_i = & steps_{i,start}(start, x_{i,n\_steps}) \wedge \\ & read_{i,start}(x_{i,n\_steps}, value_{i,start}) \wedge \\ & steps_{i,target}(target, y_{i,n\_steps}) \wedge \\ & read_{i,target}(y_{i,n\_steps}, value_{i,target}) \wedge \\ & value_{i,start} = value_{i,target} \end{aligned}$$

More precisely, to derive $value_{i,start}$, the i-th check does some steps starting from the $start$ node and ending in the node x_{i,n_steps}. It then reads a particular value $value_{i,start}$ from the node x_{i,n_steps}. Deriving $value_{i,target}$ works analogously.

2.2.3 Steps along a path

Even more precisely, the i-th check performs n_steps steps starting from $start$. This leads to a path $(start, x_{i,1}, x_{i,2}, \cdots, x_{i,n\_steps})$.

$$\begin{aligned} steps_{i,start}(start, x_{i,n\_steps}) = \\ & start = x_{i,0} \wedge \\ & step_{i,1,start}(x_{i,0}, x_{i,1}) \wedge \\ & step_{i,2,start}(x_{i,1}, x_{i,2}) \wedge \\ & \dots \\ & step_{i,n\_steps,start}(x_{i,n\_steps-1}, x_{i,n\_steps}) \end{aligned}$$

The j-th step of the i-th check from node $from$ to node to is controlled by a control-variable $step\_control_{i,j,start}$. Each value of $step\_control_{i,j,start}$ stands for a specific instruction in the DSL defined in Table 1. To capture each instruction, we precompute an arrays that describes the effect of that instruction on each node. Depending on the value of $step\_control_{i,j,start}$, the formula looks up the next value to in the respective array at location $from$.

$$\begin{aligned} step_{i,start}(from, to) = \\ & \Big(step\_control_{i,j,start} = \text{“}Nop\text{”} \implies \\ & \qquad to = nop[from]\Big) \wedge \\ & \Big(step\_control_{i,j,start} = \text{“}Up\text{”} \implies \\ & \qquad to = up[from]\Big) \wedge \\ & \dots \\ & \Big(step\_control_{i,j,start} = \text{“}DownLast\text{”} \implies \\ & \qquad to = downlast[from]\Big) \end{aligned}$$

The arrays contain values $v \in \{-1, \dots, total\ number\ of\ nodes\}$. For example, $up[12] = 5$ means that going up from node number 12 will lead to node number 5. On the other hand, $left[12] = -1$ means that node number 12 does not have a left neighbour in the tree. In order to enforce that we cannot go left in this case, we add a constraint for variable to: $to \geq 0$. All these arrays are listed in Table 1.

The steps starting from the $target$ node are enforced similarly,

Instruction name	Short description
Nop	Do not move, stay at the current node.
Up	Move up one node.
Left	Move to your left neighbour.
Right	Move to your right neighbour.
DownFirst	Move down to your first (left-most) child.
DownLast	Move down to your last (right-most) child.

Table 1: List of all instructions in our DSL that describe the steps our program can take

by separate controls $step\_control_{i,j,target}$. Thus, the steps from $start$ and $target$ are independent.

2.2.4 Reading the value

Finally, reading the value from the last node x_{i,n_steps} is controlled by a control-variable $prop\_control_i$. It can assume one of the following values:

- *Type*: Read the type of node x_{i,n_steps}.
- *Value*: Read the value of node x_{i,n_steps}.
- *Id*: Read the id of node x_{i,n_steps}.

Again, reading the actual value is implemented by a lookup in the corresponding array.

$$\begin{aligned} read_{i,start}(x_{i,n\_steps}, value_{i,start}) = \\ & \Big(prop\_control_i = \text{“}Type\text{”} \implies \\ & \qquad value_{i,start} = types[x_{i,n\_steps}]\Big) \wedge \\ & \Big(prop\_control_i = \text{“}Value\text{”} \implies \\ & \qquad value_{i,start} = values[x_{i,n\_steps}]\Big) \wedge \\ & \Big(prop\_control_i = \text{“}Id\text{”} \implies \\ & \qquad value_{i,start} = x_{i,n\_steps}\Big) \end{aligned}$$

Reading the value for the $target$ works analogously. Note that $prop\_control_i$ only depends on the check number i, not on whether or not we are reading for the $start$ or $target$ node. This reflects that we enforce the check to read the same kind of value at $start$ and $target$.
Also note that we omitted a detail in our description: We also allow the $start$ node to read a constant value $const_i$. This reflects the fact that we can enforce the $target$ node to have a specific constant value in its context. Of course, in this case we do not require the $start$ and $target$ node to both read this constant. Instead, $start$ reads the constant $const_i$ (which does not even depend on the path it takes), and $target$ performs its steps as usual, and then reads one of *Type* or *Value* of y_{i,n_steps}.

2.3 Examples

In this section, we explain intuitively how our approach works for some examples.

2.3.1 Previous node with the same *Value*

Consider the JavaScript code in Listing 2, which contains multiple occurrences of the identifier c.

```
var keyword =
  (c === 'this') || (c === 'else') ||
  (c === 'case') || (c === 'void');
```

Listing 2: JavaScript Code with multiple occurrences of c

Say we want to learn a program that goes from a current identifier c to the last occurrence of the same identifier c. Assume that we are given multiple examples $(start, target)$, where $start$ is the id of the node of identifier c, and $target$ is the id of the last node of identifier c before $start$. Our formula allows various solutions to this task.
We can learn the program **"go to the previous node with the same *Value*"**. More formally, this program performs only one check, and does not require any steps. In fact, it is enough to read the *Value* of the $start$ node and the $target$ node. This means $n\_checks = 1$, $n\_steps = 0$ and $prop\_control_1 =$ "$Value$".
Note that even if we search for a program with multiple steps, we can always take less actual steps by using $step\_control_{i,j,\cdot} =$ "Nop". Also, even if we search for a solution with multiple checks, we can just perform the same check multiple times.
Alternatively, we can learn the program **"go to the previous node with the *Value* c"**. Formally, this program performs again just one check and no steps. However, this time, it reads a constant at the $start$ node (namely c), and read the *Value* at the $target$ node. This means $n\_checks = 1$, $n\_steps = 0$, $prop\_control_1 =$ "$Value$" and the $start$ node reads the constant value c (we did not formalize this here).
Note in particular that we do not need a step that goes to the previous node with the same *Value* explicitly. This is handled implicitly by searching for the last node that satisfies all constraints.

2.3.2 Previous identifier with the same name

But what if we actually want to move to the last *identifier* with the same name? Consider Listing 3, which is similar to Listing 2, but now contains c not only as an identifier, but also as a string.

```
var keyword =
  (c === 'c'    ) || (c === 'else') ||
  (c === 'case') || (c === 'void');
```

Listing 3: JavaScript Code with multiple occurrences of c

Figure 3 shows the part of the AST that stands for (c === 'c'). It illustrates that there is a node between the two identifiers c that also has the *Value* c.

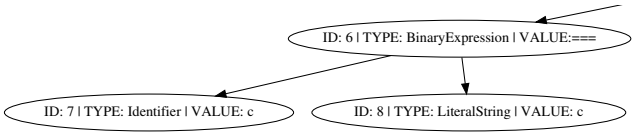


Figure 2: The part of the AST of Listing 3 that stands for (c === 'c')

If we want to skip the string 'c' and just move to the last identifier with the same name, we can reuse one of the checks suggested in Section 2.3.1. Then, we add an additional check that makes sure the $target$ node is an identifier. Thus, we could learn the program **"go to the previous node with the same *Value* and the same *Type*"**. This means $n\_checks = 2$, $n\_steps = 0$, $prop\_control_1 =$ "$Value$" and $prop\_control_2 =$ "$Type$".

3. Conditions and comparisons

In this section we start by analyzing the conditions under which our solution presented in the previous chapter performs well. We then compare our approach with a basic and a more involved approach. The comparison is intended to give a better understanding of our solution.
We define the *complete context* of a node n as the set of nodes that can be reached from n in at most n_steps steps using the DSL defined in Table 1. When we refer to the *context* of a node we describe some subset of the complete context of that node. It should be clear from the examples what subset of the complete context we mean.

3.1 Conditions for solution to work well

Condition 1: Structure. One of the main conditions that must hold for our solution to work is that the tree traversal problem must consist of a specific type of structure. To be more precise, there must be a connection between the context in the start nodes and the corresponding target nodes. Furthermore the connection must be the same for all the start nodes and corresponding target nodes. We illustrate this point with the following example.

```
... {name: 'world'} ... //target node t1 "name"
...
... {name: 'world'} ... //start node s1 "name"
```

Listing 4: Condition 1 Example 1

There are multiple ways to interpret the above example and there are different ways our approach could solve it. One interpretation is that one wants to jump from a field definition to the last field definition with the same name, where the same value is assigned. This could be solved by our approach because there is a connection between the start and target node context, namely that the *Values* of the start and target node match as well as the *Values* of their child nodes (which correspond to the value assigned to them). Assume now that the following example is also part of the same problem:

```
... {count: 10} ... //target node t2 "count"
...
... {count: 52} ... //start node s2 "count"
```

Listing 5: Condition 1 Example 2

Now the interpretation discussed earlier does not work to satisfy this example, but there still is a connection between start and target nodes that explains both this example and the example in Listing 4. This connection is that the *Values* of start node and target node must match as well as the *Types* of their children (for example, 10 has the same *Type* as 52). In this case the context of the start

node s_1 with the *Value* "name" is not identical to the context of the start node s_2 with the *Value* "count", but the connection between the contexts of s_1 and corresponding target node t_1 as well as the connection between the contexts s_2 and corresponding target node t_2 are identical. This emphasizes that our solution demands the same connection for all examples, but not the same context for all examples.
Finally consider this third example:

```
if(x > 0) { //target node t3
  ...
  x = 4; //start node s3
  ...
}
```

Listing 6: Condition 1 Example 3

Our discussed interpretation for the first two examples in Listings 4 and 5 cannot explain this new example. Actually there again is a connection between start and target node here but it is different than the possible connections in the first two examples, hence our solution will not be able to find an explanation for all three examples since the described condition is violated.
Condition 2: Ordering of nodes. An obvious condition that must hold for our solution to work at all is that all the target nodes in the training set as well as in the test set must be before the corresponding start nodes in the ordering of the nodes (where the ordering is given by the preorder of the nodes in the AST).
If in a specific case, for all the examples the target nodes are after their corresponding start nodes, then extending our solution would be simple. We could adjust the prediction to search in the opposite (forward instead of backward) direction.
But if some of the target nodes are after and some before the corresponding start nodes then it is not clear how to extend our solution to handle this case. The reason is that in the prediction phase we do not know which direction to search in. Searching in both directions leads to ambiguities since there may be two nodes in opposite directions with equal distance to the start node satisfying all the constraints.
Condition 3: Short distances between nodes. Conceptually, the distance between start and target node makes no difference. However, in practice, for our solution to work well the distance between start nodes and their corresponding target nodes should be small.
Recall that for a training example with start node s and corresponding target node e we add constraints which assert that no node in the ordering between s and e may be a valid target node. Adding these constraints is expensive and Z3 needs much more time to solve a formula with such additional constraints.
See Section 4 for a more quantitative discussion on scalability.
Also the prediction phase is affected by longer distances between start node and target node, because then the predictor needs to iterate through a lot more candidate nodes until it finds the target node. However, since prediction is usually very fast, this is not much of a problem in practice.
Condition 4: Not a huge training set. In practice, we noticed that for very large training sets Z3 needs a long time to find a solution for our constructed formula. This is not surprising since adding more examples to the training set leads to longer and often more complicated formulas. See section 4 for a more quantitative discussion on scalability.
To summarize the conditions we can see that if conditions 1 and 2 are violated then our solution will not even work in theory. Conditions 3 and 4 on the other hand are conditions which should hold for our solution to work well in practice.

3.2 Comparison with naive approach

A naive approach to solve the tree traversal problem is to try to synthesize a straight-line program using some variation of the DSL described in (Raychev et al. 2016) which walks from a start node to the target node (a small example is given in Section 1). An obvious issue with this approach is that the existence of a program which solves the tree traversal problem is very strongly dependent on how expressive the DSL is. Consider the examples in Listing 4 and Listing 5. It is not clear how a DSL should look like to be able to solve these two examples together. One possibility is to introduce an instruction which does exactly what is demanded by this example (e.g. "go to the previous field of the same name"). But in general one does not know beforehand what tree traversals need to be solved, so this is certainly not optimal.
Our own approach does not have this particular issue since we are just interested in the connection between the contexts of start and target node. We do not need to find a program that can traverse explicitly from start to target. This allows us to use a very simple set of instructions to solve a large range of problems.
In fact it turned out that using a more expressive set of instructions negatively impacts the success rate of our approach because it expands the *complete context* of start and target node too much.

3.3 Comparison with branch approach

To overcome the issue with the naive approach described in the previous subsection branches could be added. One way of doing this which does not make the program space much larger is to add a single termination condition to each program. So we could, for instance, attempt synthesizing a straight-line program $P = instr_1; instr_2; ...; instr_n$ using some variation of the DSL described in (Raychev et al. 2016) with instructions $instr_i$ and a termination condition *cond* where the final program P_{branch} with branches would be defined as:

```
i := 0

while(cond does not hold && i <= n) {
  instr_i;
  i := i+1;
}
```

Listing 7: Branch approach

If after executing P_{branch} starting at some start node the condition *cond* does not hold then no valid target node is found.
For instance if P = *PrevNodeValue; PrevNodeValue; PrevNodeValue* and *cond* = *Type of current node's first child must be the same as the type of the start node's first child* then the resulting P_{branch} could solve the tree traversal problem given by the examples in Listing 4 and Listing 5 if the wanted node is found by going at most three times to the last node with the same *Value*.
This illustrates that there are certain parallels between our own approach and this branch approach. The branch approach synthesizes a program which traverses the tree until a condition holds. This is very similar to our approach since we essentially synthesize the straight-line program *P = PrevNode; PrevNode; ...* (where *PrevNode* moves to the previous node in the ordering) and the termination condition is synthesized using different start and target programs. The main difference is that in our approach we do not focus on the path from start to target (i.e. we do not focus on the straight-line program) but we rather focus on synthesizing the termination condition. Therefore the advantage in our approach is that it does not matter what lies between start and target as long as the contexts have a connection. The disadvantage is that since our straight-line program makes progress very slowly we need condition 3 (see above) to hold.

In summary, we can say that our approach does have a special form of branches.

3.4 More powerful branches

In the above approaches as well as our own approach we mainly deal with restricted branches (i.e. termination conditions). One could also think of approaches that try to synthesize programs with more "powerful" branches, for example, programs that contain subprograms such as

```
if(c1) {
  P1
} else {
  P2
}
```

where c_1, c_2 are conditions and P_1,P_2 are straight line programs. Such an approach may be able to solve a tree traversal problem that includes the examples in Listings 4, 5, 6 where conditions could be used to identify which problem needs to be solved. The issue with this approach is that the search space of possible programs is extremely large if one allows lots of branches with different conditions. One reason why we did not follow an approach that contains such branches is that we think it is only more powerful than our approach in situations where the tree traversal problem tries to solve multiple problems instead of a single one. In a way, such powerful branches actually make to expressiveness of the synthesized programs *too* powerful.
We believe that our approach yields a good trade-off between expressiveness and efficiency, which fits well with the kind of programs we want to generate.

4. Evaluation

In this section, we describe the results of our approach on the provided test cases (see Table 2). For all the tests, we used a recommended number of 2 checks and 3 steps.

Test name	Short description
test1	Go to the previous declaration of a field with the same name.
test2	Same as test1, but the name of the field also occurs in other contexts.
test3	Go to the previous call of the same method.
test4 simple	Go to the previous occurrence of identifier *sp*.
test4 harder	Same as test4 simple.
test5 simple	Go to the previous occurrence of the same identifier.
test5 simple2	Same as test5 simple.
test5 harder	Same as test5 simple.

Table 2: Summary of the provided test cases

4.1 Success rate

In this section, we describe the success rate of our approach.
We ran the tests by performing the following steps:

1. Provide all examples from the `train` file as training examples and learn a program that satisfies all examples.
2. Evaluate if we get the expected results (as specified by the `expected` file) when testing with the `test` file.

When we run the tests exactly as they were provided to us, we pass all tests, meaning that we always generate a program that generalizes to the test data.

4.2 Effect of modifications in training data on success rate

However, when we permutate the training data, we sometimes get different results. This is because we use Z3 to solve the formula presented in Section 2.2, which is sensitive to how the formulas it contains are ordered.
In particular, when we evaluated 10 randomly selected permutations of each test, we got different results for "test 4 harder" (7 times) and "test 3" (1 time). All other tests always behaved as specified by the `expected` file.
To understand one alternative solution, we here explain a frequent solution to "test 4 harder". Consider the part of the program shown in Listing 8.

```
Sp.declaresType = function(name)  {
    this.scan();
    return hasOwn.call(this.types, name);
}
```

Listing 8: Part of the program of "test 4 harder"

The idea behind the examples is a program that goes from $this$ to the identifier Sp in $Sp.declaresType$. However, our approach is also able to find the following alternative explanation for the training data: **Go to the previous node of *Type* "Identifier" with a right neighbour of *Type* "Property"** (see Figure 3 to see what this means in the AST). Quite loosely speaking, the learned program is thus **"go to the previous occurrence of** $identifier.property$**"**, which also explains all the training examples.

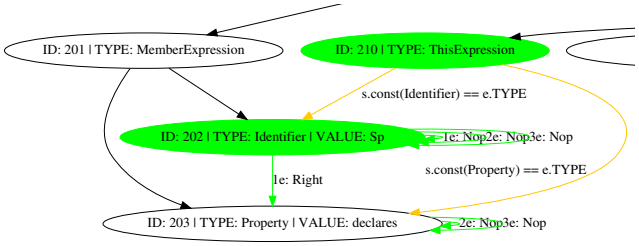


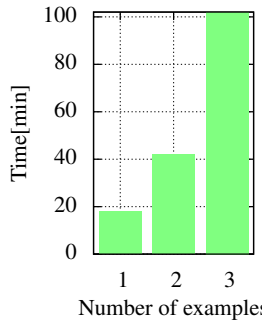
Figure 3: Graphic illustration of the alternative solution for "test 4 harder". The first check runs 3 Nops and then checks for the *Type* "Identifier". The second check runs "$Right \rightarrow Nop \rightarrow Nop$" and then checks for the *Type* "Property".

4.3 Performance

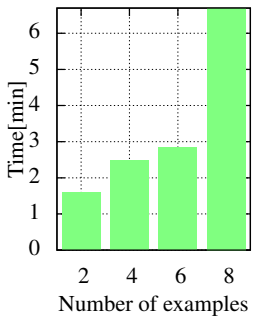
For the examples provided to us (see Table 2), our approach is very fast: it never takes more than 10 seconds.
This section investigates how well our approach scales up. The following factors influence the performance of our approach:

- Number of examples
- Size of the program
- Distance between $start$ and $target$ node

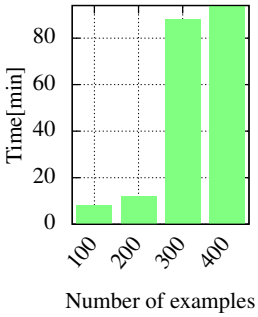
Figure 4 shows the effect of these parameters on our performance. Figure 4c shows that for up to 200 examples, our approach is very efficient. It also shows that the time does not only depend on the number of examples, but that other effects also make a difference. Figures 4a and 4b show the effect of large programs (leading to about 10000 nodes in the AST) on our performance. In particular, they illustrate that even in a large file, our approach still finds a solution within well below 10 minutes, as long as the distance between the $start$ and $target$ nodes is not too long. Only when the distance is very long, our approach takes prohibitively long, already for 3 examples. Table 3 summarizes these results.



(a) Large program, long distances



(b) Large program, short distances



(c) Short program, many examples

Figure 4: Performance of our approach in different scenarios

	Few examples	Many examples
Short paths, short program	Very fast	Acceptable
Short paths, long program	Acceptable	Infeasible
Long paths, long program	Infeasible	Infeasible

Table 3: Summary of the performance in different scenarios

4.3.1 Comparison to brute-force

In this section, we compare our approach of finding the solution using Z3 with a brute-force approach.
Each step can choose one out of 6 instructions from the DSL (see Table 1). Per check, we have n_steps steps for both $start$ and $target$ node. Then, each check can read one of *Id*, *Type* or *Value*. In total, we have n_checks. Thus, without optimizations, we have a total of

$$n\_programs = \left(\left(6^{n\_steps}\right)^2 * 3\right)^{n\_checks}$$

different programs that our approach might generate. Here, we ignore the option of our program to read a constant at the start node to get a lower bound on $n\_programs$.
Let us assume we use a processor running at $2GHz$, and that we can check for each of the $n\_programs$ in 10 cycles whether or not they work on the provided examples. Under these rather generous assumptions, consider Table 4 that shows some examples of how long it would take to search the whole search space. Table 4 shows that in particular for few checks, exhaustive search is a real alternative to using Z3. However, as soon as we want to increase

Settings	Approximate time for exhaustive search
$n\_checks = 1, n\_steps = 6$	30 seconds
$n\_checks = 2, n\_steps = 3$	2 minutes
$n\_checks = 2, n\_steps = 4$	1.5 days
$n\_checks = 3, n\_steps = 3$	160 days

Table 4: Some examples of how long it would take to search the whole search space

n_steps to 4 or n_checks to 3, exhaustive search takes too long. On the other hand, for Z3, we experimentally derived that using Z3 for example test1 (see Table 2), we finish in 3 seconds, even with $n\_steps = 5, n\_checks = 5$.

5. Conclusions

We presented an efficient solution to synthesize tree traversals in an Abstract Syntax Tree. We provided a detailed overview of the way these programs are synthesized and compared it with different approaches. Furthermore, the solution got evaluated and we show that it finds programs efficiently if certain reasonable assumptions are met.

6. Further work

The current implementation leaves room for further improvement. To deal with the inefficiencies arising if we have more than 200 examples, we could use a CEGIS-like approach. In particular, we could start off with very few examples, and check which examples the synthesized program cannot solve correctly. We would then only select one of these examples and add it to find a new solution. Currently, in case the synthesizer does not find a program that satisfies all constraints it will report that no program has been found and exit. One could use different values for n_checks and n_steps and restart the search with these parameters. It is possible that a larger search space will yield a result at the cost of additional runtime.
As mentioned in Section 4.2 a different order of inputs and therefore different constraints can result in different synthesized programs. One solution to reduce this instability is to run the synthetisation process several times with permuted inputs and then decide on the output of the majority of the runs (majority vote). This also imposes higher runtime overhead and it is not clear if this really provides better results.

References

L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2, 978-3-540-78799-0. URL `http://dl.acm.org/citation.cfm?id=1792734.1792766`.

V. Raychev, P. Bielik, M. Vechev, and A. Krause. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 761–774, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837671. URL `http://doi.acm.org/10.1145/2837614.2837671`.