Extending TEESY Into New Domains

Study in the Modularization and Extension of a Domain Specific Tool into new Domains

Luke Roosje
Student
Computer Science
North Carolina State University
Raleigh, North Carolina
leroosje@ncsu.edu

1 Introduction

Natural Language Processing is something that has become more prevalent in modern systems with each passing release. Things like Apple's Siri, Window's Cortana, FlashFill, and many others benefit from the advances that have been made in a computer's ability to understand the meaning in what we're saying. As these techniques get more advanced, the frontier of what can be done with them grows.

There was an idea that floated around the Computer Science research community decades ago called autocoding; the ability for a computer to generate a program to complete a specified task. This is a two fronted NLP problem; it requires the computer to understand the specification that it's given and to yield a functioning program within the bounds of a grammar. In recent years, there have been incredible strides towards this goal. Programs like the aforementioned FlashFill are being widely distributed and used.

However, some issues still exist, especially in the latter stage of program generation. Techniques like deep learning are prevalent in the best solutions, and it entails a large training stage, extreme rigidity, and domain specificity. To continue progressing towards the ultimate goal of creating a functioning auto-coder, these must be resolved.

TEESY is a program that, instead of neural networks, opts for a rule based derivation system. It was developed by Dr. Xipeng Shen and many of his students at NC State University[1]. The purpose of TEESY is to generate ASTMatcher expressions based off a given NL specification. In it's first stage, it does dependency parsing on a specification. Then, it uses a rule base developed prior to running to generate the appropriate program.

Given TEESY's less rigid structure, it should be able to be ported into new domains. Further, successfully doing so will prove that alternate methods are viable, and their benefits are worth pursuing. This will be achieved by identifying code specific to the original domain and adjusting it.

PL'18, January 01–03, 2018, New York, NY, USA
2018. ACM ISBN . . . $15.00
https://doi.org/

2 Overview

TEESY has five overall steps to generating a matcher statement. They execute sequentially and independently of one another.

The first step is to use StandfordCoreNLP to generate a dependency tree for a given NL specification. This portion of the program also does some preliminary adjustment to the specification; certain statements are replaced with tags to be recognized later. Once completed, this section of code outputs three text files; a list of dependencies within the specification, a list of tokens (the words of the specification), and a list of intentions, which are the parts-of-speech tags for each token. These three files form the basis for synthesis of a statement.

These three output files form a tree of nodes with relations. The second step is to refine this tree and make it more useful. First, the tree is reordered, moving around words that were in the specification to generate a more logical sequence. Secondly, useless nodes are pruned. This is an especially important step; NL Specifications are ripe with words that contribute only to the syntax of the sentence, and not the meaning. The resulting tree is saved in an object and TEESY moves onto the next step.

Next, TEESY takes in the rules that have been predefined by the programmers. Each word that remains in the tree is iteratively checked against the list of rules to see what the word could map to in the final output program. Each node holds potential matches in a list, and a greatest length scheme is used to make sure that all potential matches are included.

Next, TEESY takes these lists, and again uses predefined rules to narrow each word to a single mapping. This is done using type matching and heuristics.

Finally, any inconsistencies between words and typing are eliminated by using further rule based refinement. At the end of this step, the final program is generated, and the execution is complete.

The primary changes that I have made to TEESY are in various set definitions, certain functions that look for string

literals, and of course, the rule base. I have elected to maintain the general data structures and algorithms of TEESY for the change to new domains. Further, I believe that only changing the base objects and rules will work well in this object oriented environment, and will port well into the new domains. There have also been a few other changes to improve TEESY's ability to run and function fluidly. These changes will be documented and detailed in the following section.

3 General Changes

In its base form that has been provided, TEESY does successfully generate some programs. However, what struck me most was how exceedingly difficult it was to write a test case that succeeds, or even compiles. Thus, my changes were mostly targeted at allowing a larger number of specifications to run.

The first change is that line 52 of synthesizer3-2 is commented out, because it checks for a node with 'hasname'. Next, line 459 is removed because it references a node that is sometimes undefined. Line 655 is commented out because it references a field that is occasionally null. Finally, on like 708, there is an else clause that checks for inconsistent node types.

Each of these snippets causes compilation errors, and halts generation of a program. Removing them, in practice, does not harm the generation of good programs and also enables TEESY to give detailed output for specifications it cannot handle. This is vital for testing and extending into new domains.

4 Modularization

The first step of modularizing TEESY is understanding it; this was one of the most difficult parts of this work. It is an extremely dense program, and is unfriendly to those unfamiliar with it. The supplementary paper helps, but does not give a comprehensive look into its core. Rather, it gives a strong overview. Of course, with time and effort, I believe I now have an accurate understanding of TEESY and its domain specific components. In this section, I will detail what I moved around in certain files, and give justification for it. I will be directly referencing line numbers of the synthesizer file. While they may not be precise, it should be easy to locate these functions if you want to look at them yourself using the descriptions I provide.

4.1 Rule sets - Narrowing/Node/Traversal Functions

The most important part of TEESY to change, by far, are the rule sets that govern transformations. These pre-made rules, in their original form, and derived from the API documentation for ASTMatcher. Each possible function is defined in these files, so obviously each new function in the new domain will need to be defined here.

I do not believe that these files need to be entirely replaced, however. Some functions, such as hasType, could possibly be useful in new domains. They may need to be retooled in output/matching/etc., but, at worst, leaving these other API definitions should not harm work in other domains.

Thirdly, I will maintain the three classifications of APIs (node, traversal, narrowing). These classifications have nice justification that seems domain-agnostic, so leaving them in offers a predefined structure to define new rules in. Also, if these classifications were removed, TEESY would be gutted; at which point, using TEESY as a frame becomes an afterthought.

4.2 Defined Sets of Strings

element dict, defined line 29: This set of strings makes note of mappings that were done of certain tokens in NLParsing.py. When they eventually need to be processed, this set is looked at to grab the proper information.

special rule keywords, defined line 35: This set is defined and then referenced later in code exactly once, when it is iterated through to set a flag within a node object. The code that uses this flag later is also modularized, and will be described in the next subsection.

ast group string, defined line 58: This is a massive set of strings that are critical for type matching later in TEESY. It must be redefined and changed to better suit new domains, so that the final refinements can take place.

4.3 Functions

Sensitive word parse, called line 41: This function looks for specific "sensitive words" that could affect the derivation of a correct program. In its base form, these two words are cxx and objc; two strings that occur extremely often as substrings of tokens. If either of these are found in any token, then they are appended to a sensitive word list, which is used in later functions. In new domains, new sensitive words could appear, and must be addressed.

Find special mod, called line 217: This function is incredibly useful for modularization, because it offers the ability to strictly match tokens to final APIs. This is completed by searching for literal matches for tokens, and then mapping them directly onto final APIs. Obviously, overusing this function for mapping tokens defeats the point of using rule-based generation or even investing in such a project, but for very precise language, this could be useful.

Keyword matching, called line 263: This function parses the tree, node by node, to look for possible matches for each token. It is done in a fairly simple way; if the token exists in a specification for a match, the match is added to the tree. This part is domain-agnostic, because the specifications are encapsulated in the three text files mentioned earlier. However, the latter parts of this function must be changed. They search the just created matcher list for certain words, including the sensitive words mentioned previously, and removes

any matches that contain them. This will need to be adjusted, probably in a way that corresponds to the sensitive word parse.

Special rules, called line 408: Similar to the previous functions, special rules parses through the tree to try and find nodes that are marked. These are the same special rules defined in the set mentioned above. Interestingly, in its given form, TEESY only handles and adjusts only one of the two special rules it defines. In this case, TEESY searches for nodes that could be a declaration. If found, special transformations occur. First, the final mapping of the token to API is set. Then, specific to TEESY's original domain, any possible matchers of both the parent and children nodes not referring to a declaration are removed. Because literal strings are being searched for, and there are very domain-specific rules in play here, this function will most certainly need to be changed.

5 New Domains

For this work, I have chosen two new domains to extend TEESY into. These two domains share enough similarities to ASTMatcher that TEESY should be able to handle them, but also are different enough that extending TEESY into them is worthwhile.

5.1 Java.Math[2]

Firstly, I decided to try to add the ability to generate Java.Math functions. This class in Java is an excellent choice due to its large documentation, the way that functions can be nested, and accessibility. Further, NL regarding mathematics is inherently more concise than general NL, so the task should, in theory, be more direct. This includes a larger usage of the find_special_mod function that was detailed in the previous section; certain math terms should map exclusively to their respective functions, and using that function is an easy way to force this.

This API is not without its difficulties. Not only do the functions within the API have to be defined in the rule set, but so do more general java objects. Matches for things like data types already exist in TEESY, so I recycled these. Another tricky part of this API is the ordering of the parse tree that TEESY generates. Unlike general semantics, mathematical functions are extremely dependent on their ordering to be correct. Finally, the largest difficulty is the type of code that needs to be produced; ASTMatcher statements describe structures in code. Java.Math is actual code. Thus, in the future, some syntax and things like punctuation may need to be changed.

It is also worth noting that this API does not include basic math functions, like multiplication or subtraction. While it is entirely possible to use these in conjunction with .Math functions in actual java code, they are separate from the class itself, and thus, excluded from my work.

5.2 Spark RDD[3]

Secondly, I believe TEESY will extend nicely into Spark's RDD APIs. Programs in this API have very nice sequencing and nesting structures, similar to that of ASTMatcher. This also offers a nice contrast to Java.Math, where code structure is one of the larger difficulties.

However, this API also has its own unique set of challenges. The largest, by far, will be lambda functions. TEESY has some capability to recognize lambda functions, but not to generate them. It may be possible to retool some rules and dependencies to notice certain functions preceding them, and work accordingly. Another large issue will be the ordering of output. In this domain, each function in a sequence takes the previous' output as its input, performs an operation on it, and then gives the next function its own output. Thus, any difference in ordering will dramatically change the correctness of a program.

6 Extending TEESY

Extending TEESY into the two new domains was a tricky process. Once modularized, I had to decide what needed to be changed so proper transformations could be made. The changes for each domain were different. Some modularized areas were changed heavily, and others were left alone. Each will be detailed as so.

6.1 Java.Math

Java.Math was very interesting to extend TEESY into. The changes were extremely localized, and many functions were able to be left alone or ignored. Of the two domains, this was far more friendly for TEESY's framework.

6.1.1 Large Changes

Narrowing Functions: The Narrowing Functions text file has been largely cleaned for the transition to Java.Math. The original purpose of these were to give other matchers more traits so that the correct match could be generated more effectively. I decided that this would still be useful in this domain, but many functions had to be removed because they were not relevant, and could harm the results.

After removing APIs that could be harmful, 39 Narrowing Functions were leftover. Each pertained to primitive typing of variables, expression equivalences, declarations, and node equivalences. As mentioned previously, these could be very useful for distinguishing between similar functions in the Math API. They also covered some of the implicit Java APIs that .Math uses, like primitive typing. Finally, I do not believe it is necessary to add any APIs to this file, as no Java.Math API fits the classification.

Node Functions: This file had the largest amount of changes. Each node references individual functions or types of functions, and thus, this is the most appropriate place to start adding the APIs of .Math. Similar to Narrowing, I left

some functions in this file that could be useful for the generation of .Math programs, but removed the vast majority. 35 of the original functions still remain in Node Functions. These functions all concerned variables, typing, declarations and basic code structures like if and for statements. The last of these could probably be removed, but leaving them in has not harmed any testing. Further, they could be useful for future work. This will be discussed further in the results section.

The Java.Math APIs were all added in this file. Currently, 14 major functions have been added. While there are more than this in Java.Math, focusing on these common functions allow for more precise optimizations and testing. These include trig statements, logarithmic statements, min/max and floor/ceiling statements, plus the cube root function. These statements have been assigned input/output types of MathStmt, RoundingStmt, or TrigStmt. Rounding includes floor and ceiling, Trig includes the 6 trig functions, and Math encompasses the remaining 6. Finally, the corresponding descriptions and POS entries have also been added. These are simply descriptions consistent with the format of other entries in the file.

Find Special Mod: This function was used in the original TEESY to directly map some tokens onto final matchers. As discussed earlier, if overused, this function could easily defeat the point of using a rule-based derivation scheme. However, for Java.Math, it was appropriate to load it up. The language of .Math is very precise; a specification that states "find the cosine of a number" is certainly looking for the cos function. Thus, this function conducts many direct mappings.

Of the 14 functions that were included, 8 of which use direct mappings within this function. Specifically, sine, cosine, tangent, arcsine, arccosine, arctangent, floor and ceiling were programmed in. These 8 were included because they lack ambiguity; I assume that anyone who uses the name of these functions in a specification is certainly looking for that particular function. They cannot be used in multiple ways, such as a word like "maximum," "minimum," etc. Other functions like cube root and logarithm were considered for this method, but excluded. Cube root is two words, which could be separated by some other word in a sentence, and the number of ways to describe which type of logarithm you want is too large for this literal mapping.

As mentioned earlier, overusing this function does defeat the purpose of TEESY as a whole. In this case, over 50% of the functions are actually mapped with it. This is a fairly large chunk, and must be considered. This will be discussed later in results.

Ast group string: In the later stages of TEESY, type matching is conducted on the matcher list for undecided nodes to finalize the matching. This set of strings is the complete list of typings that are searched on. Thus, many useless types need to be removed so that they do not disrupt the generation of a good program.

The list was not fully cleared for porting to Java.Math. Since many functions were left in Narrowing, Node and Traversal files, the corresponding types should be left as well. However, this ends up being a minority. The typing of TrigStmt, MathStmt, and RoundingStmt were also added so the added functions could be type matched.

6.1.2 Unchanged or Unnecessary Functions

Traversal Functions: This is, by far, the largest unchanged module. All of the original functions remain in the file, and there are no additions. These functions are described as nodes that assist in the traversal of the dependency tree that is created to represent the specification. Similar to Narrowing Functions, they add traits to the other nodes so that the generation is more specific.

I did not change this file because it relies on the relationships of other nodes to actually be useful. Node and Narrowing functions are cleaned and limited to relevant functions, so any relationships found between them by using this file should still be useful to synthesization. Further, keeping these functions should, at worst, cause no change to the final program. Thus, changing them is unnecessary.

Element Dict, Sensitive Word Parse: Java.Math does not have any special words that are within many tokens. Thus, no special words need to be defined. The current definition of Element Dict is for "cxx" and "objc". Neither of these are present, so they can be entirely removed. Because these are removed, Special Word Parse has no purpose, and can also be removed.

Keyword Matching Part of keyword matching is critical to TEESY. The first loop, for all unmatched nodes, searches for possible matchers and adds potential matches to a nodes list. This part of keyword matching remains untouched.

The second section of this function is very domain specific. It is entirely focused around searching nodes in the tree for certain string literals, like those defined in Element Dict, and obviously will never find them. It shares the same weakness as Sensitive Word Parse in this domain; there are no major words that show up in many tokens, and thus this part of the function is unnecessary. It is removed from the code because of this.

Special Rule Keywords and Special Rules: This set and function are used to look for other special keywords and adjust the nodes on the tree accordingly. Unlike other functions that just remove or focus the set of potential matchers, this one directly maps a given node to a certain output. It also adjusts the potential matcher lists of both its parent and children nodes to be syntactically consistent with that final matcher. Java.Math does not have any cases in which this could be useful, so this function is removed from the program.

6.2 Spark RDD

Spark's RDD API had the same base changes as Java.Math, but there were some differences, due to its different nature. I will not be going into as much detail regarding similar changes.

6.2.1 Large Changes

Narrowing and Node Functions: For the same reasons as Java.Math, this large list of APIs had to be cleaned to be useful in this new domain. The same general functions could be kept, but new functions had to be added.

In Node Functions, I added 11 new functions. I focused on these 11 so that more thorough testing could be conducted. Also, because of the nature of RDD, they were far more disparate. The functions are as follows: textFile, map, filter, forEach, println, range, sum, max, sortBy, first, count. Each has a very unique function, but there are similarities in the number of arguments they take, the way they transform their inputs, and their syntax. Thus, I have subdivided them with different types, fileStmt, objectAssignment, classifierStmt, quantifierStmt, and actionStmt. fileStmts exclusively deal with loading text files, so only textFile is included in it. I gave it its own type because of how common it is. objectAssignment includes map and range, two functions that directly work with bringing in large numbers of data. classifierStmt includes filter and sortby; two functions that change entries in a set according to a conditional. quantifierStmts were meant to include functions that assign a certain set of elements to be operated on. The only one that fits that is forEach, which brings every element before it to be worked on. Lastly, actionStmt encompasses the remaining functions. Each takes some input and has an output unique to the function itself.

Ast Group String: Again, this set had to be purged of a lot of unnecessary types. All the types mentioned above had to be added.

special rule keywords and Special Rules: This change is the largest departure from Java.Math, and from TEESY's base code as well. This function seemed to be an excellent choice in which to adjust the tree and matches based on keywords, of which there are plenty in RDD. Many nodes have extra syntax, or require lambda functions to be declared within them. The only issue is that due to the way specifications are tokenized, there is little room to adjust nodes based on other nodes in this domain. Thus, the function was changed dramatically.

The largest change is that there are no longer father and son node manipulations in this function. Instead, only specific nodes are changed. In a sense, I am appending extra syntax to the end of certain nodes so that the final output more closely resembles actual Scala code for Spark RDD. For example, "textFile" is changed to "textFile(sometextfilehere)". This is achieved using the same general flow as the original function; tokens that possibly use the special nodes are flagged and then a series of if statements determine which special rule it is a part of. Once determined, the final matcher name is changed to be that match with extra syntax. This is not logically equivalent to appending, but is practically equivalent. Only nodes that have already been assigned a certain match should be able to enter sections of code that adjust them. Thus, the output will be the same in any case.

6.2.2 Unchanged or Unnecessary Functions

Set Root Name: This is a function that was not mentioned in the modularization section because it existed as a single line in base TEESY. To follow the conventions of this work, I surrounding it in a function and modularized it. Its original purpose was to set the final match of the graph's root to "root". The root was always the first token in the tree, and because of the variety of specifications for RDD programs, this was detrimental and had to be removed.

Traversal Functions: For the same reasons as in Java.Math, these functions remain untouched.

Keyword Matching: The critical part of this function remains, but the rest is removed due to its domain specificity.

Find Special Mod: While this function was incredibly useful for Java.Math's direct mappings, it is unnecessary in this domain. Specifications for RDD programs can be very ambiguous, and there are many ways to describe any of the functions that have been included. For example, "sortBy" could be described by "sort", "order", "rearrange", etc. Further, each of those words are not precise; it is uncertain that their use implies that sortBy is intended.

Element Dict and Sensitive Word Parse: For the same reasons as in Java.Math, these functions remain untouched.

7 Results

The results of the transformations were far from perfect, but there are many successes worth noting. They will be described in detail for each domain. While great strides were made towards accurate generation of programs, there is still work to be done. The difficulties will be documented in the next section.

7.1 Java.Math

The transformation into Java.Math was tested on 10 separate statements. Many of these are similar, and the results will be summarized as such.

Find the logarithm of 10, Find the logarithm of 4, Find the natural logarithm of 8: These three statements are intentionally similar to test the sensitivity of transformations to either log10 or log. Eachś intended program is fairly obvious, and the results were reasonably close. The first statement transformed fairly well, but this was due to an unintented cause; the logarithm and 10 node were linked

by inner methods of TEESY, and each was mapped to log10, likely because TEESY saw both log and 10. The third statement was successfully mapped to log. The second statement was output with some ambiguity; the logarithm token was mapped to be part of either log or log10. This was a result of the general change made to TEESY, which affected the final type matching reduction of lists. It should be noted that logarithm was not directly mapped within Find Special Mod, so these results are significant; while not perfect, it shows great promise.

Find the arcsine of a number, Find the cosine: These two statements both had their mappings determined in Find Special Mod. As such, arcsine and cosine both map perfectly onto their respective APIs. The issue is with the token "number," which outputs with a potential match on any of the functions that have been added. Because these worked, it is reasonable to assume that all the trig functions will work as well, since they all follow the same flow.

Find the absolute value of some int, Find the closest integer to some number, Find the cube root of a number: These assorted specifications had varied success. The first statement had one perfect mapping, of the token "absolute" to abs. The tokens "value" and "int" both have extreme ambiguity, which could be resolved with future work. The second statement, which I hoped would map to either floor or ceiling, fails. No close matching is found. Beyond that, some of the tokens are pruned by TEESY early. The third statement acts strangely, but not badly; the tokens "root", "cube" and "number" are all linked by TEESY and all three are mapped correctly to cbrt. The challenging part is that taking three successive cube roots will obviously be incorrect.

Find the floor of the cosine of a number, Find the maximum of the cosine of one number and the ceiling of another: These specifications were included to test the rigor of ordering and mapping in large statements. The first statement benefits from the direct mapping of Find Special Mod; floor and cosine both correctly map to their functions. However, number ends with huge ambiguity. The second statement fails far more; cosine and ceiling are both directly mapped, and end with the correct tokens. Every other token, however, does not. Each suffers from large amounts of ambiguity. However, the correct match is within the list for each.

7.2 Spark RDD

The results for Spark RDD are less impressive than those of Math, but there are still some strong points worth noting. Among the 8 test cases, no programs were completely generated, but many correct matchings were made.

Return the number of lines in a text file, sum the lines of a textfile, Print the max of a text file: These three specifications were very successful. In the first two, the adjustment to textFile in Special Rules was made. The more impressive part of these results is that their unique portions were all correctly mapped. "sum" was correctly matched to sum. "Print" and "max" were both correctly matched to println and max respectively. Even more impressively, "the number of" was successfully matched to count. These three specifications are among those that could be perfectly generated with further work. One issue with the first specification is that return was mapped to a large number of possible matchers, when in a perfect generation, it would have been pruned.

return the first element of a range, map all objects in a range to them plus one: These two statements were both reasonable successful. The first statement had first and range both correctly mapped, and range had its Special Rule transformation. However, it had two extra tokens in its final match, return and element. This would certainly break a statement in a real setting. However, it is worth noting that both were ambiguously mapped, meaning TEESY was closer than it could have been to correctly generating the program. The second statement is trickier, as I intentionally included a specification for a lambda function, which should be pruned. It had two correct mappings as well, "map" to map and "range" to range. However, the extra portion was not pruned, and actually mapped incorrectly. "Them" mapped to filter. Intuitively, it is not too far off. It would still cause a generated program to fail.

sort the text file then return the sum, sort a range of numbers, filter all elements of a text file and print those greater than 5: These three functions were an extremely weird case, and certainly outliers. For some reason, although their corresponding token, intention and dependency files are intact, only the first node is ever considered and output by TEESY. The reason for this is entirely unknown. Still, "sort" in the first to statements correctly matches to sortby, and "filter" correctly matches to filter.

7.3 Summary

Overall, the synthesization went better than I expected. There were multiple correct mappings among the purely rule based generations, and even more when things like the direct mappings are considered. There is only one instance where a token is matched to an incorrect match. Of those that were not assigned a final match, the correct match was among their set.

It is clear that some of the rule bases were not cleaned enough, due to the excess of extra matches that are present in the list of each token. Some of these are relevant, but others could certainly be removed.

Finally, it is clear that some of the general changes to TEESY affect the program generation. There are many tokens that still have a large list of possible matchers, which could be resolved by some of the code that I have chosen to exclude. However, there are still many concise generations.

8 General Difficulties and Potential Solutions

The largest issue with transferring TEESY to the new domain with the ambiguity with which many tokens resolved. Too many of the tokens did not have a final match, and this prevents many accurate programs from being generated. While this is certainly due to some general changes, I believe that an improved and more specific rule set would also help.

Secondly, in some specifications, many important tokens were pruned. This prevented TEESY from analyzing them. This may be fixable with extended rule sets as well as improved definitions of string sets. Further, changing the methodology of pruning branches may benefit these specifications.

In Spark RDD, ordering was a large issue. The output would certainly not work as an RDD program even among those that mapped correctly. This is certainly due to the way TEESY is set up to reorder the tree, which is not rigorous enough to handle the complexity of RDD specifications. Unlike ASTMatchers and Math, there is absolutely no good relationship between the ordering of keywords in the specification to the necessary ordering of APIs.

Another issue exclusive to Spark RDD is the inclusion of Lambda functions. I previously thought that these would be able to be handled using existing functions, but it turns out that TEESY completely lacks the capability to handle these as is. Further work could certainly fix this problem, but new functions must be generated and tested in order to fully include this property. Unfortunately, this also limits the domains that TEESY could cover in its current state.

Another recurring issue is that the TEESY's linking of tokens is very sensitive. On many occasions, tokens that could either be purged or are unrelated are considered together, and both mapped to the same API. This could cause a myriad of issues, including executing a function multiple times, or searching for too deep of a nesting. There are multiple ways this could be resolved. I believe the most effective solution would be to change the rules by which tokens are linked, and make them more strict. On the other hand, this could have the unintended consequence of leaving nodes that need to be linked separate. Alternatively, a filter for multiple successive generations of the same API could be implemented, in which all of them are merged into one. This has the consequence of possibly ruining statements that require stacking functions.

9 Future Work

There are plenty of good places to work on TEESY in the future. In this section, I will detail what I believe are the most prevalent.

Firstly, the output of TEESY is very non-descript. The final output is simply a mapping of tokens to matches, and nothing more. While a correct output would have them in the correct order with the correct matches, it still ignores the actual syntax and structure of the domain it is generating for. Implementing code syntax into the output of TEESY would make using it far easier, and much more intuitive because it would output a function that could be ran as is.

ASTMatcher is distributed using three classifications for APIs: Narrowing, Node and Traversal. These are well used in the original work, and extend well into the two domains I explored in this work. However, I believe that reformatting these may yield better results. Further, there may be other domains that certainly require different organizations.

Finally, changing TEESYs matching schemes may produce a better result in some cases. The longest/shortest match schemes are certainly effective, and work well when extended. However, I believe that changing these might be useful for future work.

10 Conclusion

Overall, the extension of TEESY into new domains was a success. The modularized and adjusted versions show great promise in their respective domains. While a domain like Java.Math can be extended into quite easily and effectively, there are larger issues in domains like Spark RDD that severely limit TEESY's capabilities.

Each issue that arose is a great oppurtunity for future work, and there are many places that can also be changed and improved. But, as it stands, TEESY can still be useful in many other domains. With more work and functionality, it can conquer an even larger set of domains. Clearly, the benefits of using rule based generation can be widespread.

References

[1] *TEESY: Natural Language-Based Training-Free Program Synthesizer for Code Analysis.* Xipeng Shen, Zifan Nan, Hui Guan, Chunhua Liao. PLDI19, June 22 through 28, 2019, Phoenix, Arizona, USA 2019.
[2] https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html
[3] https://spark.apache.org/docs/latest/rdd-programming-guide.html