UNIVERSITY OF CALIFORNIA
Los Angeles

Grammar Refinement
for Grammar-Based Test Input Generation

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Computer Science

by

Ricky Yu-Taai Fok

2024

ABSTRACT OF THE THESIS

Grammar Refinement
for Grammar-Based Test Input Generation

by

Ricky Yu-Taai Fok
Master of Science in Computer Science
University of California, Los Angeles, 2024
Professor Miryung Kim, Chair

Grammar-based fuzzing is an effective method of testing software that requires highly structured inputs. However, these fuzzers require a user-provided input grammar that often does not exist for niche and emerging domains.

Grammar inference algorithms fill this gap by inferring a grammar using a set of training example inputs that satisfy a black-box oracle. However, existing grammar inference algorithms have shown to require on the order of hours to days to infer an input grammar from a handful of examples. We observe that some software systems such as AWS CloudFormation (CF) and Ansible take inputs that conform to a schema built on top of common formats like JSON or YAML. Our key insight is that the grammars for these inputs are simply refinements of the parent grammars of the common formats.

We introduce GramRefine, a grammar inference algorithm that refines the given base grammars into more precise input grammars. By using the base grammar as a starting point, we significantly reduce the time needed to infer a grammar and obviate the need for language-specific heuristics, such as matching parenthesis or specific delimiter usages.

We conduct a comprehensive evaluation of GramRefine against the recent state-of-the-art black-box grammar inference algorithm TreeVada [4] on AWS CloudFormation templates, Ansible playbooks, and four sets of MLIR dialects, using a generic JSON grammar and generic MLIR grammar as the base grammars respectively. GramRefine is on average 654× faster and achieves 194× greater branch coverage than TreeVada. GramRefine's refined grammars creates qualitatively better examples than the base grammars—With more work to improve input validity (i.e oracle satisfaction), this could eventually translate to better coverage. GramRefine has the significant potential to improve input coverage and fault detection, while altering the burden to write a complex grammar by hand.

The thesis of Ricky Yu-Taai Fok is approved.

Jens Palsberg

Achuta Kadambi

Miryung Kim, Committee Chair

University of California, Los Angeles

2024

To my father . . .

who showed me the wonders of technology

and justified my education above all else.

And to my beautiful wife and son that gave me the strength

to move forward.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This masters thesis is based on the early findings and works of a conference paper being prepared for submission. Chapters One, Two, Three, and Four are altered versions of text from the unpublished paper. Chapter Five details current implementations and algorithm changes, from the draft under preparation.

The unpublished paper is being advised by Miryung Kim, and is co-authored by myself and Ben Limpanukorn. Ben collaborated on most parts of the paper, including the implementation of the clustering algorithm, GramRefine's runnable scripts, evaluation results, and a large portion of the writing in the unpublished draft that has been used Chapters One, Two, Three, and Four. Independent contribution lies in the lexer rule synthesis implementation, which is based on sample code from Hong Jin Kang, initial implementations, and demos for grammar refinement, and writing and modifying the parts of the unpublished draft to discuss the earlier findings.

Furthermore, I would first like to express my deepest appreciation to my advisor, Miryung Kim, for taking a chance in me as well as her valuable guidance throughout this paper and my time here at UCLA. I would also like to extend my wholehearted gratitude to Ben Limpanukorn for his unwavering patience and extensive contribution to this collaborative work. I'd like to acknowledge my committee advisors, Jens Palsberg and Achuta Kadambi, whom, although may not know, have had a large impact in my educational journey. Many thanks to all the students and post doctorate fellows in the Software Engineering Analytics Laboratory for their continuous support. And last, but most definitely not least, I would like to thank my friends and family for giving me their support during such a difficult time in my life.

CHAPTER 1

Introduction

Fuzzing has demonstrated significant potential in its ability to detect software faults that are overlooked by human-written test cases. However, many software applications like compilers and infrastructure-as-code (IaC) tools expect highly structured inputs that traditional fuzzers without domain knowledge struggle to generate. Grammar-based fuzzers, such as Nautilus [5] and Grammarinator [7], constrain generation with a user-provided context-free grammar. This method ensures that the generated test inputs conform to the syntactic rules of the target application expressed in a context-free grammar, increasing the likelihood of discovering deeper and more subtle bugs.

A drawback of grammar-based fuzzing is the need for the user to provide a grammar. For common languages such as JSON, Javascript, or C, these grammars may already exist, but many niche domains extend these base grammar formats, however they do not provide a refined grammar besides what is implicitly defined by a working implementation of an input validator. Modern approaches towards grammar-based fuzzing benefit from automated grammar induction/inference, since writing a context-free grammar is difficult and time consuming. Treevada [4], Arvada [8] and Glade [6] are examples of such efforts where example inputs are used induce a grammar that is then used to generate syntactically valid inputs.

One particular instance of this issue is in AWS CloudFormation (CF). CF files use templates defined in JSON to setup and model AWS resources. These templates follow a specification that constrains the structure of the JSON file. For example, cloud resources defined in a CF template must always be contain a `"Type"` key with a value that corresponds to a

valid resource type identifier such as `"AWS::S3::Bucket"`. To generate valid CF templates with grammar-based fuzzing, the grammar should encode both JSON syntax and CloudFormation's specific semantic constraints.

Similarly, the Multi-Level Intermediate Representation (MLIR) is an extensible compiler infrastructure that poses a similar, but more difficult challenge for test input generation [1]. In a previous work, an effort was made to port a general MLIR grammar from Lark [14] to ANTLR [9]. Although successful, extending this grammar for each of the 60 plus dialects (as of 2020) is impractical since MLIR is an inherently extensible infrastructure where compiler engineers define they own custom IR dialects [3]. These dialects are intermediate representations (IR)s based on MLIR's base IR, with unique sets of semantics and functionalities, such as the "`func.func`" operation which represents a function definition. For instance, the Circuit IR Compilers and Tools (CIRCT) project is an extensible compiler for heterogeneous compilation that defines 26 new MLIR dialects with a total of 145 new operations [3]. The generic MLIR grammar is therefore incapable of fuzzing CIRCT-specific operations that have custom semantics.

We observe that the current grammar-based fuzzing techniques are unable to generate test inputs robustly for a project where the base grammar is refined to specific domain-specific input constraints. In this paper, we introduce a novel algorithm called GramRefine that refines a generic base grammar into a specialized grammar for grammar-based fuzzing. At a high level, GramRefine breaks down a provided set of input examples into parse trees. These trees are then compared to identify and cluster similar subtrees. These subtrees are then used to generate new, alternative production rules that are subsequently *appended* into the defined generic grammar.

We assess GramRefine's effectiveness and efficiency by measuring the number of generated inputs using the refined grammar and the time required to infer a grammar and generate 1000 inputs, respectively. This evaluation is conducted across four sets of MLIR dialects (`arith`, `async`, `krnl`, and `onnx`) and AWS CloudFormation files. Our findings indicate that

GramRefine maintains perfect recall and achieves an average branch coverage per second that is 194 times greater than that of Treevada. Additionally, we conducted a qualitative analysis to compare the outputs using GramRefine's grammar and the base grammar for their semantic and syntactic quality. In this analysis, GramRefine demonstrated superior performance, generating more extensive and natural inputs.

The remainder of this paper is organized as follows. Chapter 2 introduces a motivating example for GramRefine, followed by Chapter 3 where we present the design and implementation of GramRefine. In Chapter 4, we provide the design of our experiments and the empirical evaluation results. Finally, we present the conclusions in Chapter 5. In future work, we aim to enhance the amount of valid generated inputs by addressing fundamental issues with our implementation.

CHAPTER 2

Motivation

While learning a context-free grammar can suit many purposes, its most common use is to generate new test inputs for automated testing. Prior works like Treevada [4], Arvada [8], and Glade [6] infer a context-free grammar from scratch under the problem setting where only a set of example inputs and a black-box validator are available. The goal of these algorithms is to learn the golden grammar, i.e. the grammar implicitly defined by the black-box validator that serves as an oracle for valid inputs.

Listing 2.1: Example CloudFormation file.

```
{
    "Resources": {
        "MyBucket": {
            "Type" : "AWS::S3::Bucket",
            ...
        },
        "MyInst": {
            "Type" : "AWS::EC2::Instance",
            ...
        }
    }
}
```

However, there also exists a set of input domains where the golden grammar is a specialization of a more general base grammar that is already known and widely available. For example, AWS CloudFormation (CF) templates [11] and Ansible Playbooks [10] are a subset of JSON. However, using the base grammar alone for grammar-based fuzzing is ineffective as the base grammar does not respect the domain-specific constraints, such as special domain syntax and initial declaration requirements, imposed by the downstream golden grammar. AWS CF template are used to define and provision AWS infrastructure. In Listing 2.1 we have a properly formatted CF template that requires each resource definition, such as `"MyBucket"` on line 2, to be nested within a `"Resources"` entry (line 2) and contain at minimum the `"Type"` key-value pair (line 4 or 8) whose value must be formatted as `"AWS::PID::RTYPE"` where `PID` is the product identifier and `RTYPE` is the resource type.

A general JSON grammar shown in Listing 2.3 can parse CF templates like the one shown in Listing 2.1, but this grammar is ineffective for grammar-based test generation, as the grammar does not encode the domain specific semantic constraints imposed by the CF template validator, `cfn-lint` [12]. `cfn-lint` evaluates AWS CloudFormation templates against the CloudFormation spec and checks for both best practices and potential issues before deployment [12]. When an issue or warning arises, as shown in a run of `cfn-lint` on an invalid CF template in Listing 2.2, `cfn-lint` will output an error/warning code alongside the source where the error/warning is coming from. For example, suppose a grammar-based fuzzer attempts to generate a CF template by following the general JSON grammar (Listing 2.3). In this case, it may start with the `object` rule on line 8 which only specifies that an object contains zero or more pairs. At this point, the fuzzer may choose to generate zero pairs, resulting in an empty object and a trivially invalid template. Even assuming that the fuzzer exercises the `pair` rule on line 6, it is still unlikely for the fuzzer to generate a key whose value is exactly `"Resources"` as required by the CF template specification.

Listing 2.2: `cfn-lint` output when attempting to validate an invalid CF template, SNSTopic.json.

```
// Output from "cfn-lint SNSTopic.json"
E1001 Missing top level template section Resources
SNSTopic.json:1:1

E1012 Ref SNSTopic not found as a resource or parameter
SNSTopic.json:29:7

E1012 Ref SNSTopic not found as a resource or parameter
SNSTopic.json:36:9

E1010 Invalid GetAtt SNSTopic.TopicName for resource QueueName
SNSTopic.json:40:21
```

Listing 2.3: Excerpt from the JSON base grammar. Some rules have been simplified for clarity.

```
// A 'key' is a string
key : STRING ;
// A 'value' is a string or an object
value : STRING | object ;
// A 'pair' is a key-value pair of an object
pair : key ':' value ;
// An object contains 0 or more pair entries
object : '{' pair* '}';
```

To obtain valid CF files from grammar-based fuzzers, the grammar must encode additional syntactic constraints specific to CloudFormation. Listing 2.4 shows an excerpt of a JSON grammar that has been tailored to represent a CF golden grammar. In this grammar,

specialized versions of the `key`, `value`, and `pair` rules are added. A fuzzer following the production rule for `pair_rlist` for example, will always generate a key with the concrete string "Resources" whose value is a set of `pair_resource`'s. Each `pair_resource` is then constrained to always contain at minimum a `pair_type` entry which corresponds to the `"Type":"AWS::PID::RTYPE"` entry.

Listing 2.4: Excerpt of a theroetical CF golden grammar. Some rules have been simplified for clarity.

```
1  key_rlist : '"Resources"' ;
2  value_rlist : '{' pair_resource* '}' ;
3  pair_rlist : key_rlist ':' value_rlist ;
4  pair_resource : key ':' value_resource ;
5  value_resource : '{' typePair ',' pair* '}' ;
6  key_type : '"Type"' ;
7  pair_type : key_type ':' typeSpec ;
8  value_type : 'AWS::' PID '::' RTYPE ;
9  PID : 'S3' | 'EC2' ;
10 RTYPE : 'Bucket' | 'Instance' ;
```

CHAPTER 3

Approach

At a high level, GramRefine infers a refined grammar from a base grammar and a set of training examples, using an oracle that returns true if a given input is accepted by the golden grammar. The golden grammar is implicitly defined by the oracle's implementation, as it does not exist explicitly for any of our baselines. We assume the base grammar to accept all the examples accepted by the oracle (100% recall) but may also accept examples that do not conform to the golden grammar (low precision).

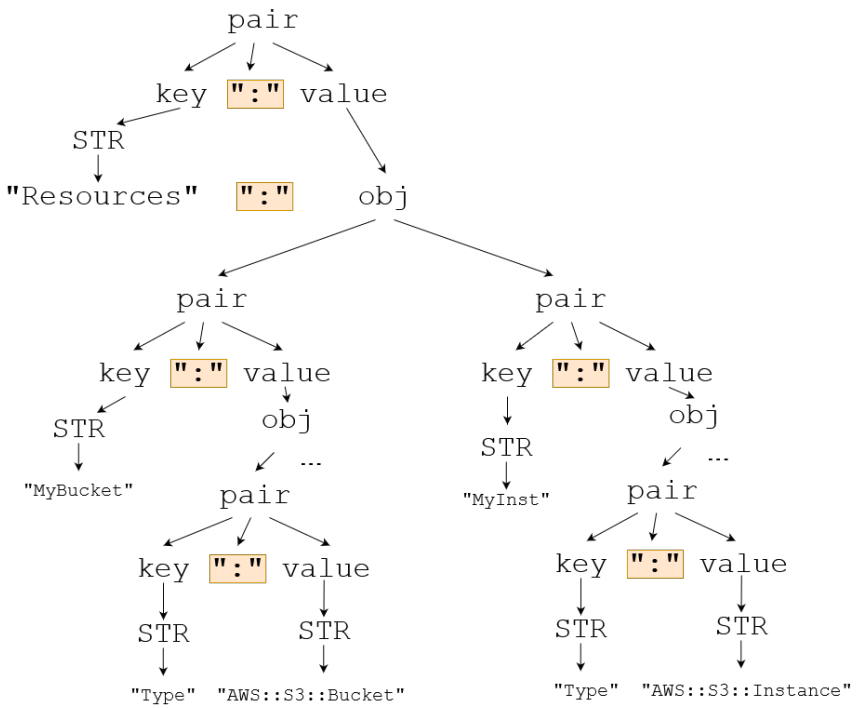

Figure 3.1: Grammarinator parse tree visualization for Listing 2.1.

GramRefine first processes each example from the training set, which include true examples in a user-specified domain, into parse trees using the respective base grammar. For instance, our training set for CloudFormation (CF) files comprises true examples sourced from the PIPR dataset, which will be discussed in more detail in the evaluation [13].

GramRefine then infers the refined grammar by creating parser rules that mirror the parse-trees of the training set. For each token position in a cluster, GramRefine infers a generalized lexer rule from the tokens that appear in that position of the cluster. For example, in a cluster where each subtree contains a terminal node representing mathematical operators, two nodes representing the operators + and - could be mutually interchangeable. Thus, GramRefine would infer a production rule such as "`MATH_OP : '+' | '-'`" which accepts both the + and - operators as the same token. The new, inferred rules are then appended to the original grammar.

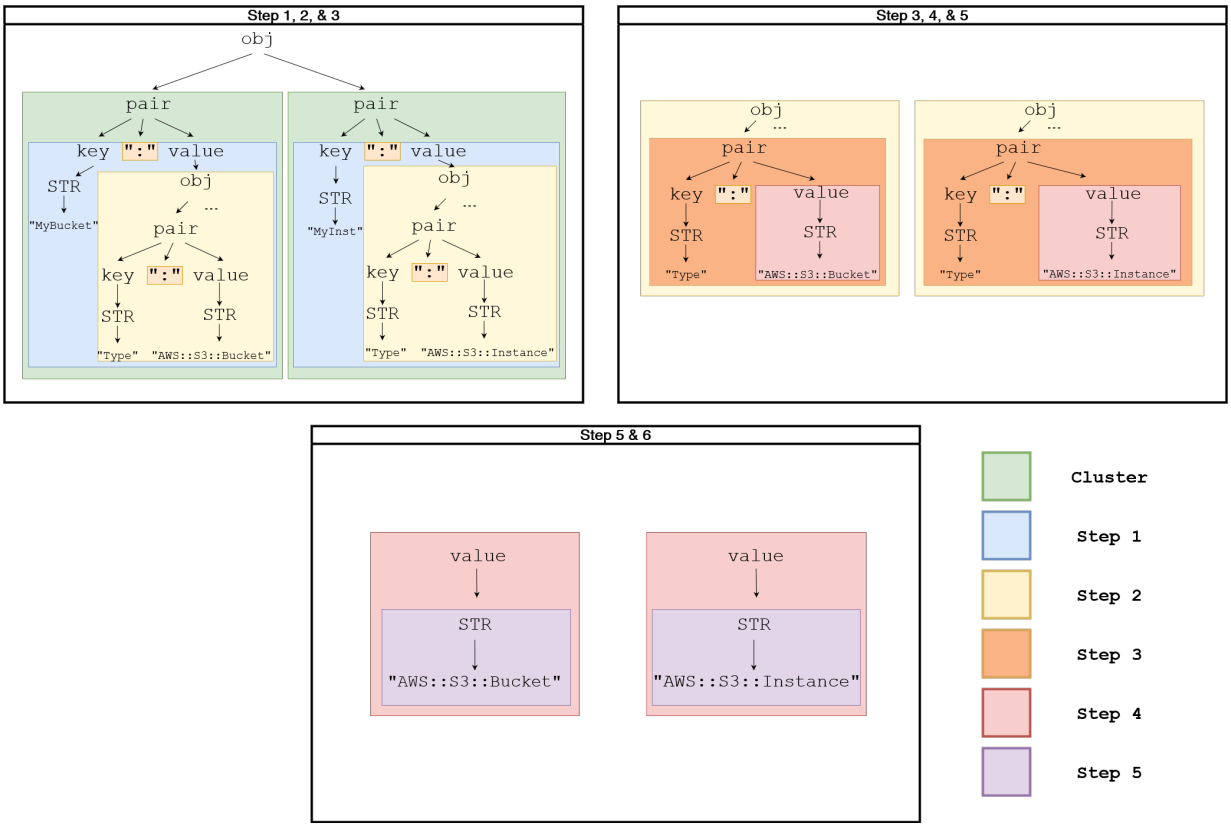

Figure 3.2: Parse tree steps for one recursive call to the clustering algorithm on the CF file defined in Listing 2.1.

3.0.1 Algorithmic Proof for Refinement

One of GramRefine's main implementation details is that it appends rules to a base grammar to generate a refined grammar. Because of this, when GramRefine adds rules to the base grammar, it will not cause the new, refined grammar to reject any previous inputs.

Suppose we add a specialized version of the `pair` rule, called `c_pair`, into a base grammar, and there was an input that was parsed before we added this rule. When we parse with the new, refined grammar and encounter a part of the input that would have previously matched the `pair` rule, the parser now has an additional rule to utilize. Therefore, the parser will take one of two paths:

1. It matches the input with the `pair` rule, since the rules of the base grammar still exist.

2. It matches the input with the newly defined `c_pair` rule.

3.0.2 Clustering subtrees

GramRefine clusters subtrees across the parse trees of the training set. Clustering, in this context, involves identifying and grouping subtrees that have identical interior structures and nodes. This process allows GramRefine to identify common subtrees that can be represented with the same rule definitions. In other words, GramRefine finds and groups subtrees that have exactly matching structures, thereby enabling the creation of generalized rules for those common subtrees. For example, the subtrees `A` → `B` → `"Text1"` and `A` → `B` → `"Text2"` have matching structures and will therefore be grouped together.

Algorithm 1 Clustering

Require:

- *context* ← the parameterized context
- *recipient* ← the recipient parse tree

Ensure:

- *mutateLocation* ← a parse-tree node in the recipient test case that represents a valid mutate location

```
1: C ← {}
2: for all candidate ← walk(recipient) do
3:     for all node ∈ candidate do
4:         foundCluster ← false
5:         for all representativeNode ∈ C do
6:             if exactMatch(representativeNode, node) then
7:                 Add node to the cluster of representativeNode
8:                 foundCluster ← true
9:                 break
10:            end if
11:        end for
12:        if ¬foundCluster then
13:            Create a new cluster with node as its representative
14:        end if
15:    end for
16: end for
17: return C
```

Our clustering algorithm traverses every candidate node of all parsed trees and compares them with the clusters in C. A candidate node is defined as any node in any parse tree. We have designed C as a dictionary where the key is the root node representing a potential cluster, and the value is a set of subtrees that exactly match the subtree of the key. Given that each node within the cluster shares an identical structure, determining whether a new node belongs to the cluster requires an exact match comparison against only one of the trees within the cluster (the key values of C). If the candidate node's subtree does not match with any of the existing clusters, we generate a new cluster with the candidate node as the representative key value.

After identifying the clusters, GramRefine filters clusters based on two heuristics: cluster size and cluster tree height. The cluster size determines the minimum number of sub trees within a cluster, which prevents GramRefine from synthesizing rules that are based on a small or singular instance of the subtree pattern. These one-off rules have no generalizable properties and do not capture the broader patterns in the dataset. The cluster tree height determines the minimum height for the subtrees within the cluster, which helps to eliminate generic and ineffectual rules. Clusters with small heights do not contain sufficient information. For example, a subtree created for the JSON structure with a `key` parent node pointing to a single `STRING` node, as shown in two instances in Figure 3.2 (Step 1), would be too shallow and generic, rendering it inadequate without additional context. If we set the cluster size to be less than 2, such a subtree would be included, compromising the effectiveness of the clustering process. Figure 3.2's highlighted subtrees labeled `Cluster` represent a cluster found between the two child nodes `pair` within the `"Resources"` value object from Figure 3.1.

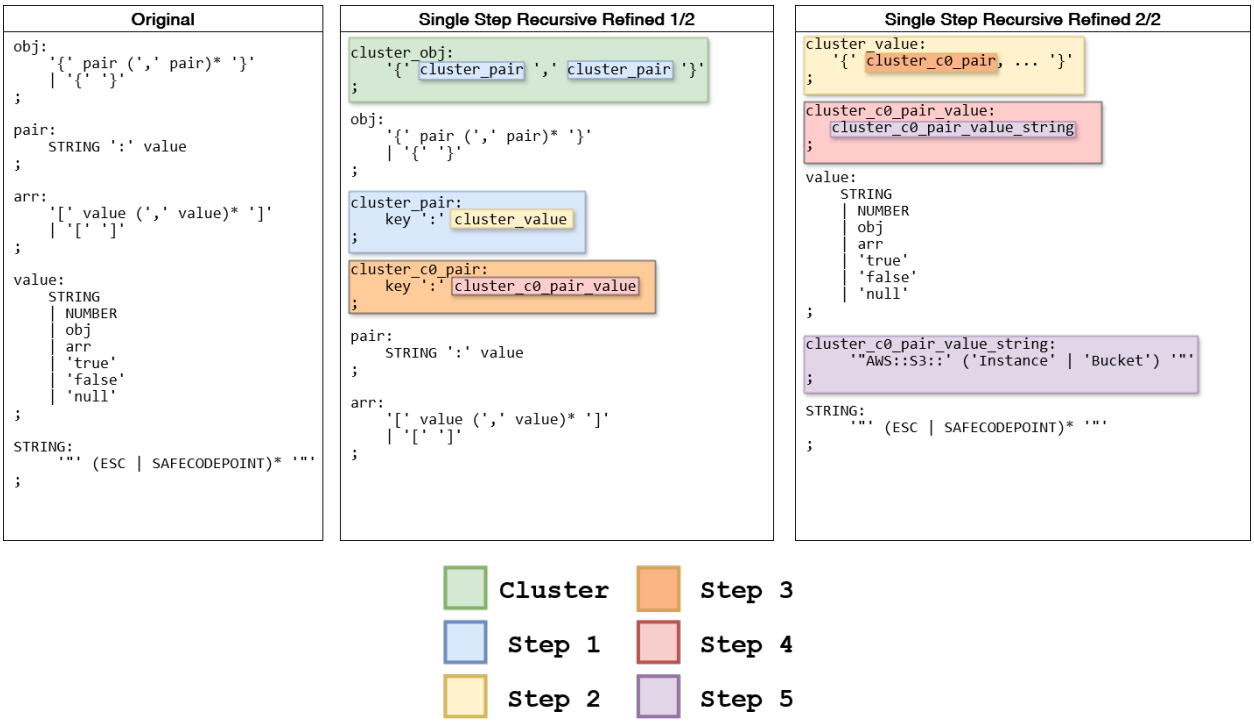

Figure 3.3: Refined grammar results for one recursive clustering algorithm call on the CF file defined in Listing 2.1

3.0.3 Inferring rules for clusters

GramRefine then uses a recursive algorithm to generate rules that create a single, dependant chain of dependencies. This algorithm is applied to each cluster and subsequently appended to the original grammar.

Algorithm 2 Rule Generation

```
1: procedure SYNTHESIZERULE(cluster_data, clusters)
2:     node ← firstTree(clusters)
3:     rule_name ← createRuleName(node.name, clusterID)
4:     rule_expression ← ""
5:     for all child ∈ getChildren(node) do
6:         if isLiteral(child) or reachedDepthLimit(child) then
7:             rule_expression ← child
8:             continue
9:         end if
10:        if isUnlexerRule(child) then
11:            new_element_name ← synthesizeLexerRule(child, ST, DT)
12:        else
13:            new_element_name ← synthesizeRule(child)
14:        end if
15:        rule_expression ← new_element_name
16:    end for
17:    if isEmpty(rule_expression) then
18:        rule_expression ← ""
19:    end if
20:    rule ← finalizeRule(rule_expression)
21:    insertRule(node.name, rule)
22:    return rule_name
23: end procedure
```

For every iteration, GramRefine loops through the current node's children (line 5 in Algorithm 2) and updates the current rule expression based on the types of nodes encountered by the generation algorithm (line 6-15 in Algorithm 2). Literals are added directly to the `rule_expression` variable, as seen at line 6 in Algorithm 2. For example, all the literals in Figure 3.2 highlighted as the head nodes in steps 2, 3, and 4 correspond to their color-matching code snippets in Figure 3.3.

Each step in the rule generation process relies on the next rule generation call in the call stack. In other words, each rule is built incrementally, with each call within the call stack contributing to the construction of the final rule expressions.

When encountering leaf nodes (lexer rules), GramRefine calls its lexer rule synthesis

algorithm. Lexer rules define the lexical structure of the language, specifying patterns for tokens such as keywords, identifiers, and symbols. For example, Step 5 in Figure 3.2 shows the lexer rules in the example parse tree. The lexer rule synthesis algorithm generates a rule that represents the possible combinations of these tokens throughout the subtrees.

3.0.4 Lexer Rule Synthesis

Listing 3.1: Lexer rule synthesis example using different types CloudFormation type strings

```
# Clustered Tokens
["\"AWS::S3::Bucket\"", "\"AWS::S3::BucketPolicy\"",
    ↪ "\"AWS::EC2::Instance\"", "\"AWS::EC2::Instance\"",
    ↪ "\"AWS::EC2::Host\""]

# Token Splitting
["\"", "AWS", "::", "S3", "::", "Bucket", "\""],
["\"", "AWS", "::", "S3", "::", "BucketPolicy", "\""],
["\"", "AWS", "::", "EC2", "::", "Instance", "\""]
["\"", "AWS", "::", "EC2", "::", "Host", "\""]

# Grouping
## Group 1
["\"", "AWS", "::", "S3", "::", "Bucket", "\""],
["\"", "AWS", "::", "S3", "::", "BucketPolicy", "\""],
## Group 2
["\"", "AWS", "::", "EC2", "::", "Instance", "\""]
["\"", "AWS", "::", "EC2", "::", "Host", "\""]

#  Concatenation
```

```
19 ["\"AWS::S3::(Bucket|BucketPolicy)\"",
    ↪ "\"AWS::EC2::(Instance|Host)\""]
```

To represent these variations, GramRefine employs a custom template generator that returns a list of all possible cluster-matching strings. In other words, GramRefine generates multiple lists, each proposing a set of rules that fully represent the possibilities for the lexer rule. To convert these strings into lexer rules, GramRefine splits each string into identifiers and symbols, utilizing two metrics to group them: the similarity threshold (ST) and the difference threshold (DT), as seen in line 11 in Algorithm 2. The ST determines the number of matching substrings required for clustering, while the DT specifies the allowable number of unmatched tokens within a group.

Consider the outputs from the different steps for the lexer rule synthesis shown in Listing 3.1. The `"Clustered Tokens"` represent a set of mutually interchangeable, CF terminal nodes. When a token is split, we obtain an array of substrings that represents each candidate, as illustrated in the `"Token Splitting"` step. When one of these arrays (string arrays), such as those in lines 5, 6, 7, or 8 in Listing 3.1, has a *number of matching substrings greater than or equal to the ST value* compared to another string array, they are grouped together. In our recurring example, lines 5 and 6 are grouped into an array shown in lines 12 and 13, and lines 7 and 8 are grouped into an array shown in lines 15 and 16 when using an ST value of 2. If you theoretically set your ST value to 7, each of the string arrays would be placed in their own grouping because the 5th indexed items (`"Bucket"`, `"BucketPolicy"`, `"Instance"`, and `"Host"`) do not match.

In addition to the ST heuristic, we also use the DT heuristic to determine if the string arrays should be grouped based on the difference in the *total number of substrings between string arrays*. In our example, DT is set to zero, meaning that the string arrays in a grouping must have the same number of substrings. This results in the grouping shown in Listing 3.1. If you set DT to 1, allowing for one additional substring, a string array exactly like the one

in line 5 with an additional substring "`Temp`" after the substring "`Bucket`" would also be grouped into `Group 1`.

If there was another token that did not match our grouping, such as `"None"`, it would be placed in its own group. Consequently, instead of "`Grouping`" containing a single array, we would have a second array with only the token-split version of `"None"`, representing this non-group-conforming candidate.

3.0.5 Concertizing cluster rule's parents

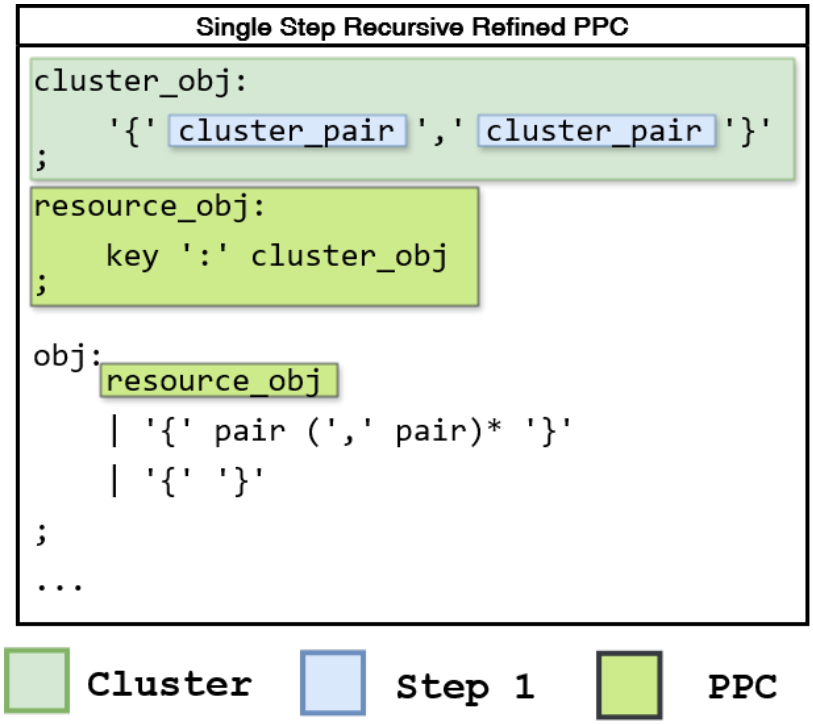

Figure 3.4: Parent Cluster Concretization (PPC) applied after Figure 3.3

Once the proposed rules are generated, GramRefine needs to synthesize additional rules for each of the parent nodes of the clusters, otherwise known as Parent Cluster Concretization (PPC). This process back-propagates until it has made a rule for the root node to ensure that the clusters appear at the correct depth during fuzzing. These parent rules are then appended into the grammar.

In our CF example in Listing 2.1, PPC generates the code labeled `PPC` in Figure 3.4. The first call of the PPC back-propagation generates `resource_obj`, referencing our head cluster rule, `cluster_obj`. Subsequently, `resource_obj` is appended as an alternative rule to the existing `obj` definition from the original base grammar. This process ensures that our cluster appears only with the additional parent context. In our example, this means that our cluster would appear as an object value for the `"Resources"` key value.

CHAPTER 4

Evaluation

We answer the following research questions in our study:

RQ1: How "correct" are the grammars refined by GramRefine in terms of precision and recall?

RQ2: How effective are the grammars refined by GramRefine for fuzzing in terms of branch coverage?

RQ3: How efficiently does GramRefine refine grammars?

4.0.1 Experiment Design

Once GramRefine produces a new, refined grammar, we utilize Grammarinator, a test generation tool that uses the provided grammar to generate 1000 test cases [7]. Note that the number of test cases generated for evaluation is arbitrary.

In our evaluation of GramRefine, we select 4 sets of MLIR dialects from 2 unique projects [2] [1], as well as a set of AWS CloudFormation files [11] from the PIPR dataset [13].

We build a corpus of test cases for each input domain. For the `arith` and `async` dialects, the files were taken from unit tests within the LLVM project, and the `krnl` and `onnx` dialects were sourced from the ONNX-MLIR project [1] [2]. Example test cases for each MLIR dialect are provided in the Appendix. The CloudFormation templates were sourced from a dataset of IaC (infrastructure as code) programs called PIPR [13], more specifically the IaC Analysis

Table 4.1: Benchmark subjects

Domain	Description / Oracle (Subject Program)	Train Inputs	Test Inputs
arith	MLIR files pertaining to the arith dialect and other supporting dialects from llvm-project. Oracle: `mlir-opt --convert-arith-to-llvm`	30	737
async	MLIR files pertaining to the async dialect and other supporting dialects from llvm-project. Oracle: `mlir-opt --async-to-async-runtime --async-runtime-ref-counting --convert-async-to-llvm`	10	20
krnl	MLIR files pertaining to the krnl dialect and other supporting dialects from onnx-mlir. Oracle: `onnx-mlir-opt --convert-krnl-to-llvm --canonicalize`	7	15
onnx	MLIR files pertaining to the onnx dialect and other supporting dialects from onnx-mlir. Oracle: `onnx-mlir-opt --shape-inference --convert-onnx-to-krnl --canonicalize`	30	1,350
cf	CloudFormation templates. Oracle: `cfn-lint --non-zero-exit-code error`	30	527

project that converts IaC programs into CloudFormation templates [10]. Each dataset was split into a training dataset containing 30 inputs (or approximately 30% of the total inputs if the dataset has fewer than 100 total inputs) and a testing dataset comprising the remaining inputs. The training set size was determined by our hardware limitations. We found that a training set size of approximately 30 was the maximum size possible by measuring the peak memory usage and inference time of GramRefine and Treevada.

Table 4.1 presents each subject program along with their respective numbers of training and testing inputs, as well as their descriptions. The table also includes the oracle function call used to verify the generated input files for each subject program.

Each MLIR project provides a utility executable named `<project>-opt` which is used

to independently invoke and test one or more compilers using the generated fuzzer and represents the oracle (subject program) for the input domains. To fuzz each set of dialects, we invoke the `<project>-opt` executable with a pipeline of selected compiler passes on each test case generated by sampling from the inferred grammars.

Each MLIR input domain utilizes the same executable to represent their oracle, distinguished by different combinations of flags used with the executable. The `async-runtime-ref-counting` flag is employed for automatic reference counting in `async` runtime operations. The `convert-arith-to-llvm` flag converts the `arith` dialect to the LLVM dialect. The `async-to-async-runtime` flag lowers all high-level `async` operations to explicit `async.runtime` and `async.coro` operations. The `shape-inference` flag returns the `onnx` shape inference. Finally, the `canonicalize` flag converts operations to their canonical form.

Conversely, CloudFormation templates are tested using a custom linter named `cfn-lint` [12]. The `--non-zero-exit-code error` flag is included to ensure that the linter only returns error codes, excluding optimization and warning codes.

Since there are currently no other grammar refinement algorithms, we compare GramRefine against Treevada, the current state-of-the-art grammar inference tool at the time of writing this paper, performs the best in evaluations [4]. Treevada learns context-free grammars from a set of positive (valid) examples and an oracle.

4.0.2 Evaluation Metrics

To evaluate and compare the qualitative potential of GramRefine, we use the following measures:

- **Precision** is the proportion of valid generated inputs over the total amount of generated inputs. A valid input is an input that is accepted by the oracle. For instance, running the oracle on example 1 in Listing 4.1 would return a negative result since the example is an invalid CF file.

- **Recall** is the proportion of inputs in the test set that are accepted by parsing with the inferred grammar over the total size of the test set. For instance, GramRefine's refined grammar would accept an MLIR input from the testing set shown in Listing 4.2.

- **Coverage** measures the branch code coverage of the target program. For MLIR programs, we use LLVM's SanitizerCoverage tool [1]‘.

- **Time** is number of seconds taken to infer the grammar and generate 1000 inputs using Grammarinator.

Listing 4.1: Two examples of inputs generated using a GramRefine refined JSON grammar for CF templates

```
// Example 1
{ "":"D\n",
    "": {
        "":0.6E-7,
            "\b": {
                "BuildSpec":"reddit-compiler-api-api-apiGW",
                "Default":"Public"
            }
        },
    "+":true }
// Example 2
{ "":{},
    "":{
        "]":true,
        "": {
            "DeploymentOption":"DB_NAME",
            "Description":"-1.888154589708815e+289"
        }
    },
    "":false}
```

Listing 4.2: MLIR example input from testing set

```
"builtin.module"() ({
  "func.func"() <{function_type = (index) -> (index, index, index),
     ↪ sym_name = "static_basis"}> ({
  ^bb0(%arg0: index):
    %0 = "arith.constant"() <{value = 16 : index}> : () -> index
    %1 = "arith.constant"() <{value = 224 : index}> : () -> index
    %2 = "arith.constant"() <{value = 224 : index}> : () -> index
    %3:3 = "affine.delinearize_index"(%arg0, %0, %1, %2) : (index,
       ↪ index, index, index) -> (index, index, index)
    "func.return"(%3#0, %3#1, %3#2) : (index, index, index) -> ()
  }) : () -> ()
}) : () -> ()
```

4.0.3 RQ1: Precision & Recall

Table 4.2 shows the performance of Treevada's output grammar and GramRefine's output grammar. GramRefine retains perfect recall across all input domains, because it refines the base grammar, only adding alternatives to the existing rules. In contrast, Treevada learns context-free grammars from scratch from a set of positive (valid) examples [8].

GramRefine was unable to generate any valid inputs for `arith`, `krnl`, and `cf`, and had 98% fewer valid inputs for `async` and 99.7% less for `onnx` compared to Treevada. This indicates that GramRefine struggles to produce grammars that generate valid inputs when used to generate inputs for fuzzing. This occurs because the PPC only concretizes the parent nodes, increasing the likelihood that Grammarinator will select a base rule rather than a newly added rule from the refinement process. Although Treevada exhibits higher precision than GramRefine, a detailed inspection of the results reveal that its generated inputs are largely reproductions of the seed/training inputs that lack diversity and novelty.

	Treevada Grammar				GramRefine Grammar			
	Recall	Precision	Time (s)	Coverage	Recall	Precision	Time (s)	Coverage
arith	0.000	0.608	34,528	762	1.000	0.000	43	286
async	0.150	0.951	14,319	453	1.000	0.017	34	182
krnl	0.067	0.755	5,288	9	1.000	0.000	7	2
onnx	1.000	0.977	17,848	168	1.000	0.003	26	8
cf	N/A	N/A	N/A	N/A	1.000	0.000	792	N/A

Table 4.2: CF files have no coverage and were incompatible with Treevada. GramRefine achieves greater recall than Treevada.

4.0.4 RQ2: Refinement Effectiveness

To assess the effectiveness of the refined grammar, we compare the increase in branch coverage achieved by the generated inputs and the training set between GramRefine and Treevada. The increase in branch coverage is computed by measuring the coverage achieved by the 1000 inputs generated with the grammar and subtracting any branches already covered by the training set. For the `arith`, `async`, `krnl`, and `onnx` subject programs, Treevada's coverage was `2.6`×, `2.5`×, `4.5`×, and `21`× faster than GramRefine respectively. Since most of the generated inputs from GramRefine are invalid, GramRefine is unable to exercise deeper code within the subject program, resulting in lower branch coverage.

Additionally, we conduct a qualitative analysis of the inputs generated by GramRefine versus those generated by the base grammars. Our qualitative analysis indicates that GramRefine's refined grammar produces more coherent and natural inputs than the base grammar. For example, the base inputs in Listing 4.3 for CF are often brief and lack naturalness compared to the GramRefine inputs in Listing 4.1.

For instance, consider examples 1 & 2 in Listing 4.3 that is produced using the base grammar: one has a single key-value entry, while the other contains a single boolean value; neither of which conform to the constraints of a CF template. In contrast, example 1 in Listing 4.1 produced by using the grammar refined by GramRefine conforms much closely

to a CF template in that they are objects that contain key-value pairs such as `"BuildSpec"` and `"reddit-compiler-api-api-apiGW"` at the correct level of nesting.

Listing 4.3: Four examples of inputs generated using the base JSON grammar.

```
// Example 1
{"a":null}
// Example 2
true
// Example 3
[{"b":-8.424e71},null,{"":[0]},[]]
// Example 4
[]
```

4.0.5 RQ3: Refinement Efficiency

In terms of execution time, GramRefine's outperforms Treevada in all subject programs, excluding `cf`. We could not include an evaluation of TreeVada on `cf` because its execution time exceeded our limit of 200 hours. We were able to measure the execution time for `cf` in GramRefine. GramRefine's execution time for the `cf` program is significantly more that the MLIR dialect programs, because the `cfn-lint` oracle requires 1-2 seconds to validate each file.

`arith`'s execution time with Trevada is `34,528` seconds and `43` seconds with GramRefine, which is a `803`× increase in performance. For `async`, `krnl`, and `onnx`, GramRefine outperformed Treevada's execution time by `421.1`×, `755.4`×, and `686.5`× respectively. Averaging across all subject programs, GramRefine's execution time outperforms Treevada by `654.4`×. Treevada calls the oracle to test every rule it generates, which take up a significant portion of execution time (i.e `cf` oracle `cfn-lint`). In contrast, GramRefine does not need to call the oracle when inferring its grammar.

Additionally, we observe the amortized coverage of Treevada and GramRefine, calculated as coverage divided by time as shown in Table 4.4, and presented as a bar graph in Figure 4.1. We used a log scale in the Figure 4.1 to make the smaller values more visible. We evaluated both coverage and time metrics separately in RQ1 and RQ2. Although Treevada significantly outperformed GramRefine in coverage for all subject programs (excluding `cf`), GramRefine's substantial performance increase in time results in amortized coverage values that are significantly higher for GramRefine. Specifically, GramRefine is `303.5`×, `169.4`×, `134`×, and `33.6`× faster than Treevada in terms of amortized coverage for the `arith`, `async`, `krnl`, and `onnx` subject programs, respectively.

Table 4.3: Amortized Coverage

Category	**Treevada**	**GramRefine**
arith	0.022	6.677
async	0.032	5.422
krnl	0.002	0.268
onnx	0.009	0.302

Table 4.4: Amortized coverage is calculated as the subject program's branch coverage divided by time in seconds (branches covered per second). GramRefine, on average, is `160.1`× better in amortized coverage than Treevada.

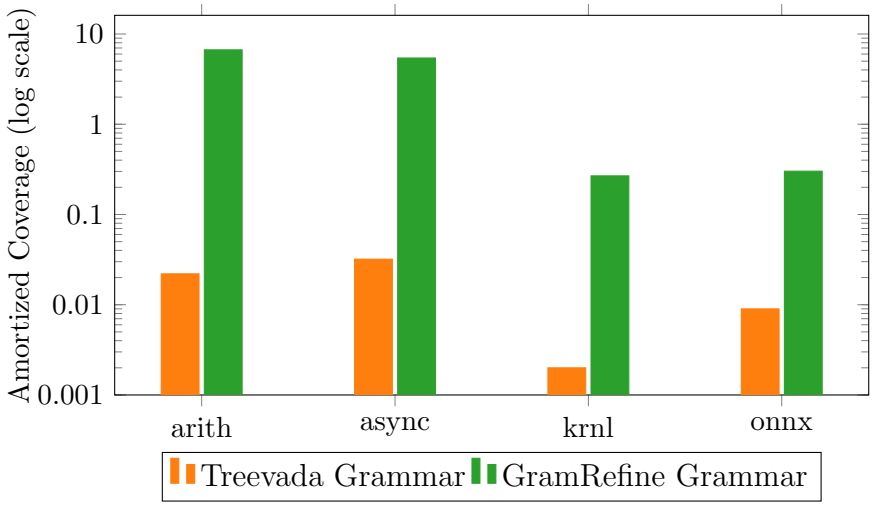

Figure 4.1: GramRefine achieves significantly higher results than Treevada. On average, GramRefine performs `645.4` × faster than Treevada.

CHAPTER 5

Conclusion

5.0.1 Future Work

The evaluation of our current research has highlighted several areas requiring significant improvement. The results, while insightful, indicate that all grammar inference algorithms struggle to produce a competitive number of valid inputs compared to other tools, including GramRefine. GramRefine encounters this issue primarily because the PPC process does not take into account the surrounding context, such as sibling and cousin nodes. The novel idea of grammar refinement that GramRefine presents is a step towards solving this problem. Consequently, our findings necessitate significant revisions not only in our rule synthesis methodology but also in other aspects of our novel algorithm, including the clustering algorithm.

Clustering Algorithm Since our current algorithm is restricted to exact-matching subtrees, the refined grammar captures only a narrow scope of the special rules. For example, if we modify Figure 3.1 so that the right pair does not exactly match the left pair, GramRefine would be unable to cluster these subtrees and consequently, it would never generate the refined rules shown in Figure 3.3.

This means that GramRefine's generated inputs often lack critical and descriptive information, resulting in grammars that do not accurately represent our dataset. For instance, in our CF example in parse tree form (Figure 3.1), the absence of the cluster identified in Figure 3.2 would drastically reduce the likelihood of GramRefine generating inputs similar to the cluster. Although the base grammar can theoretically generate the cluster, as it

still conforms to the base grammar, the probability of this occurring is near zero. Further research is necessary on how to better cluster our parse trees.

Rule Synthesis In general, GramRefine's main limitation stems from the number of valid inputs produced, as this directly correlates with branch coverage, as stated in the evaluation of RQ2. One potential solution is to modify the PPC algorithm to include additional surrounding context. However, this approach makes the rules so concrete that GramRefine would merely replicate the training set. This is a similar issue that Treevada faces as well. The goal for all grammar inference algorithms, including GramRefine, is to produce a grammar capable of generating new inputs that are novel, but also valid, meaning they are accepted by the oracle.

Another potential solution is to remove the base grammar rules. These rules currently prevent GramRefine from exercising deeper code in the subject program. Although removing the original rules might improve the precision of the refined grammar, it would sacrifice GramRefine's recall, reducing the diversity of the inputs generated. One alternative to removing the base grammar is to utilize Grammarinator's option to associate weights with each production rule. This allows us to bias generation towards the refined rules over the base rules. However, a refined alternative does not exist for every base rule, as our current refinement algorithm only concretizes some parent paths and sub-trees from the training set.

Our unpublished paper aims to investigate these ideas further and evaluate this new implementation against both GramRefine's current implementation and Treevada.

Evaluation Due to time constraints, we were unable to measure additional details regarding coverage. Although we measured the number of branches covered that were not already covered by the dataset, as mentioned in RQ2, we did not determine the specific branches covered. This limitation prevents us from examining the branches that GramRefine covered but Treevada did not. We will be addressing this gap in our forthcoming unpublished paper, where we will provide a comprehensive evaluation of the specific branches covered by GramRefine compared to Treevada.

Conclusion & Summary We present GramRefine, a novel grammar refinement algorithm that removes the need for fully-defined grammars in niche domains. GramRefine and runs significantly faster than SOTA solutions such as Treevada. While these results are promising, further research into GramRefine's clustering algorithm and rule synthesis methodology is necessary to fully realize the potential of GramRefine's approach. Continued investigation into the algorithm's precision performance, optimizations, and various use cases remain crucial for demonstrating the effectiveness and impact of this approach.

CHAPTER 6

Appendix

Figure 6.1 are example test cases for every MLIR dialect used in GramRefine's evaluation. This includes `arith`,`async`, `krnl`, and `onnx`

arith

```
"builtin.module"() ({
    "func.func"() <{function_type = (i64, i64) -> i64, sym_name = "test_addi"}> ({
    ^bb0(%arg0: i64, %arg1: i64):
      %0 = "arith.addi"(%arg0, %arg1) : (i64, i64) -> i64
      "func.return"(%0) : (i64) -> ()
    }) : () -> ()
  }) : () -> ()j:
```

async

```
"builtin.module"() ({
   "func.func"() <{function_type = () -> (), sym_name = "main"}> ({
     %0:2 = "async.execute"() <{operandSegmentSizes = array<i32: 0, 0>}> ({
       %8 = "arith.constant"() <{value = 1.234560e+02 : f32}> : () -> f32
       "async.yield"(%8) : (f32) -> ()
     }) : () -> (!async.token, !async.value<f32>)
```

krnl

```
"builtin.module"() ({
   "func.func"() <{function_type = () -> memref<10xi64>, sym_name =
 "test_constants_to_file_return_value"}> ({
     %0 = "krnl.global"() {alignment = 4096 : i64, name = "constant",
       shape = [10], value = dense<[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]>
       : tensor<10xi64>} : () -> memref<10xi64>
     "func.return"(%0) : (memref<10xi64>) -> ()
```

onnx

```
""builtin.module"() ({
   "func.func"() <{function_type = (tensor<3xf32>) -> tensor<3xf32>, sym_name =
 "test_add_constant_2"}> ({   ^bb0(%arg0: tensor<3xf32>):
     %0 = "onnx.Constant"() {value = dense<[0.000000e+00, 1.000000e+00, 2.000000e+00]>
 : tensor<3xf32>} : () -> tensor<3xf32>
     %1 = "onnx.Add"(%arg0, %0) : (tensor<3xf32>, tensor<3xf32>) -> tensor<3xf32>
     "onnx.Return"(%1) : (tensor<3xf32>) -> ()
   }) : () -> ()
 }) : () -> ()
```

Figure 6.1: Example files of MLIR dialects input domains

REFERENCES

[1] GitHub - llvm/llvm-project. `https://github.com/llvm/llvm-project`. [Accessed 06-06-2024].

[2] GitHub - onnx/onnx-mlir: Representation and Reference Lowering of ONNX Models in MLIR Compiler Infrastructure — github.com. `https://github.com/onnx/onnx-mlir`. [Accessed 06-06-2024].

[3] CIRCT: Circuit IR Compilers and Tools. `https://circt.llvm.org/`, 2023. Accessed: 2024-06-02.

[4] Arefin, M. R., Shetiya, S., Wang, Z., and Csallner, C. Fast deterministic black-box context-free grammar inference. In ICSE '24: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (New York, NY, USA, 2024), ICSE '24, Association for Computing Machinery.

[5] Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A.-R., and Teuchert, D. NAUTILUS: Fishing for deep bugs with grammars. In NDSS (2019).

[6] Bastani, O., Sharma, R., Aiken, A., and Liang, P. Synthesizing program input grammars, 2017.

[7] Hodován, R., Kiss, A., and Gyimóthy, T. Grammarinator: a grammar-based open source fuzzer. In Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (New York, NY, USA, 2018), A-TEST 2018, Association for Computing Machinery, p. 45–48.

[8] Kulkarni, N., Lemieux, C., and Sen, K. Learning highly recursive input grammars. In Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (2022), ASE '21, IEEE Press, p. 456–467.

[9] Limpanukorn, B., Wang, J., Kang, H. J., Zhou, E. Z., and Kim, M. Fuzzing mlir by synthesizing custom mutations, 2024.

[10] Qiao, F., Mohammadi, A., Cito, J., and Santolucito, M. Statically inferring usage bounds for infrastructure as code, 2024.

[11] Services, A. W. Aws cloudformation templates, 2024. Accessed: 2024-06-10.

[12] Services, A. W. cfn-lint: Cloudformation linter, 2024. Accessed: 2024-06-11.

[13] Sokolowski, D., Spielmann, D., and Salvaneschi, G. PIPr: A Dataset of Public Infrastructure as Code Programs, Nov. 2023.

[14] tbennun, kaushikcfd, amanda849, blaine fs, ro i, Berke-Ates, and joker eph. GitHub - spcl/pymlir — github.com. https://github.com/spcl/pymlir. [Accessed 04-06-2024].