Input Algebras

Rahul Gopinath · Hamed Nemati · Andreas Zeller
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
Email: {rahul.gopinath, hamed.nemati, andreas.zeller}@cispa.saarland

***Abstract*—Grammar-based test generators are highly efficient in producing syntactically valid test inputs and give their users precise control over which test inputs should be generated. Adapting a grammar or a test generator towards a particular testing goal can be tedious, though. We introduce the concept of a *grammar transformer,* specializing a grammar towards inclusion or exclusion of specific *patterns:* "The phone number must not start with 011 or +1". To the best of our knowledge, ours is the first approach to allow for arbitrary Boolean combinations of patterns, giving testers unprecedented flexibility in creating targeted software tests. The resulting specialized grammars can be used with any grammar-based fuzzer for targeted test generation, but also as validators to check whether the given specialization is met or not, opening up additional usage scenarios. In our evaluation on real-world bugs, we show that specialized grammars are accurate both in producing and validating targeted inputs.**

***Index Terms*—testing, debugging, faults**

I. Introduction

Software test generators at the system level (commonly known as *fuzzers*) face the challenge of producing *valid* inputs that pass through syntactic checks to reach actual functionality. The problem is commonly addressed by having a *seed,* a set of known valid inputs which are then mutated to cover more behavior. For complex input languages, though, mutations still mostly exercise syntax error handling. We can obtain better performance by using a *language specification* such as a grammar to produce inputs. As such inputs are syntactically valid by construction, they reliably reach actual functionality beyond parsing. The JSON grammar in Fig. 1, for instance, produces only syntactically valid JSON inputs and thus quickly covers JSON parser functionality.

Writing accurate input grammars can be a significant effort [1], which is why recent research [2], [3] has started *extracting* such grammars automatically from existing programs. One less discussed advantage of language specifications, however, is how much *control* they grant their users over which inputs should be generated—actually, much more control than for seed-based fuzzers. For our JSON grammar, for instance, a user could go and assign *probabilities* to individual productions. If the input should contain, say, several `null` values, users can assign a high probability to `null` productions. Likewise, users can *customize* the grammar. Adding an alternative expansion `"'); DROP TABLE STUDENTS; --"` to the ⟨*string*⟩ rule will quickly populate the generated JSON input with SQL injections, for instance.

But there's a catch. Adding individual specific alternatives or adjusting probabilities is easy. But how about *contextual properties*—say, a SQL injection only in a particular context? How about *negation*—say, any kind of input *except* for specific elements? And how about their *combination*? In principle, such features *can* be expressed in a context-free language, and hence in a grammar—but these grammars would be nowhere as compact and maintainable as the example in Fig. 1.

```
⟨json⟩ ::= ⟨elt⟩
⟨elt⟩ ::= ⟨object⟩ | ⟨array⟩ | ⟨string⟩ | ⟨number⟩
  | 'true' | 'false' | 'null'
⟨object⟩ ::= '{' ⟨items⟩ '}' | '{}'
⟨items⟩ ::= ⟨item⟩ | ⟨item⟩ ',' ⟨items⟩
⟨item⟩ ::= ⟨string⟩ ':' ⟨elt⟩
⟨array⟩ ::= '[' ⟨elts⟩ ']' | '[]'
⟨elts⟩ ::= ⟨elt⟩ | ⟨elt⟩ ',' ⟨elts⟩
⟨string⟩ ::= '"' ⟨chars⟩ '"'
⟨chars⟩ ::= ⟨char⟩ ⟨chars⟩ | ϵ
⟨char⟩ ::= '[A-Za-z0-9]'
⟨number⟩ ::= ⟨digits⟩
⟨digits⟩ ::= ⟨digit⟩ ⟨digits⟩ | ⟨digit⟩
⟨digit⟩ ::= '[0-9]'
```

Fig. 1: JSON grammar (simplified)

In the past, the need for such control (also known as *taming*) has been addressed by *tweaking* grammar-based test generators—that is, adding special options (often ad hoc and domain-specific) that address these concerns within the test generator [4], [5], [6]. With the test generator being Turing-complete, there is no limitation to what such options can do. However, they also mean that one is tied to the particular tool and its specific features.

In this paper, we address the problem of controlling grammar-based test generators from the ground up. We introduce the concept of a *grammar transformer*—a tool that takes a (simple) context-free grammar and specializes it towards a specific goal. As a result, we obtain a *specialized* context-free grammar, which can then be used to *produce* specialized inputs with *any* context-free grammar-based test generator [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20]. Also, the grammar can be used with any context-free *parser* [21], [22], [23], [24] for *checking* existing inputs whether they meet the specialization properties.

To specify specializations, we introduce a *language* that

```
def jsoncheck(json):
    if no_key_is_empty_string(json):
        fail('one key must be empty')
    if any_key_has_null_value(json):
        fail('key value must not be null')
    process(json)
```

Fig. 2: `jsoncheck()` fails if any key value is null or if no key is empty.

allows us to specify which features should be part of the specialized grammar and which ones should not. At the base of our language, we have *evocative patterns* (or *patterns* for short)—that form *constraints* over the values of specific nonterminals. Such patterns take the form

$$\langle \textit{nonterminal} \rangle \textbf{ is } \texttt{value}$$

and are interpreted that there should be at least one instance of ⟨*nonterminal*⟩ with value `value`. For our JSON grammar, for instance, we can create specialized grammars

- where at least one ⟨*item*⟩ has a value of `-1`—that is, ⟨*item*⟩ **is** ⟨*string*⟩`: -1`. The resulting grammar produces JSON strings such as `{ "foo": -1, "bar": 33 }`.
- where at least one key should be named `"zip"`, followed by a ⟨*number*⟩—that is, ⟨*item*⟩ **is** `"zip": `⟨*number*⟩. The resulting grammar would produce JSON strings such as `{ "foo": "bar", "zip": 45 }`.
- where at least one array should contain a `null` value—that is, ⟨*elts*⟩ **is** `null`. The resulting grammar would produce JSON strings such as `{ "foo": "bar", "zip": true, "qux": [2, 1, null] }`.

For more complex specializations, these patterns can be combined into *Boolean formulas*. These *evocative expressions* allows us to precisely specialize the produced inputs to match specific conditions. As an example, consider the function `jsoncheck()` in Fig. 2. The `process()` function is only reached if no key has a `null` value and at least one key is the empty string. To avoid the first branch to failure ("one key must be empty"), we can use the evocative pattern

$$\langle \textit{item} \rangle \textbf{ is } \texttt{"":}\ \langle \textit{elt} \rangle \tag{1}$$

This ensures that at least one item will have an empty key. Avoiding the second branch to failure ("key value must not be null"), we have to specify the *absence* of a pattern. We do so by *negation*, expressing that *no* element matching the pattern should be generated:

$$\neg\big(\langle \textit{item} \rangle \textbf{ is } \langle \textit{string} \rangle \texttt{: null}\big) \tag{2}$$

To finally reach the *process()* function, both conditions must be met. We obtain this by creating a *conjunction* of Equation (1) and Equation (2):

$$\big(\langle \textit{item} \rangle \textbf{ is } \texttt{"":}\ \langle \textit{elt} \rangle\big) \wedge \neg\big(\langle \textit{item} \rangle \textbf{ is } \langle \textit{string} \rangle \texttt{: null}\big) \tag{3}$$

The resulting grammar will produce `null` values, but never as the value of an ⟨*item*⟩; and ensure that at least one key name is the empty string. Hence, its outputs will always reach the `process()` function in Fig. 2.

To the best of our knowledge, this is the first work where users of a test generator can specifically express the *absence* and *conjunction* of patterns in given contexts. Going beyond simple data languages like JSON, evocative patterns thus allow us to produce very targeted inputs for testing compilers and interpreters—say, a series of `while` loops that contain no assignments; or SQL queries containing inner joins, left joins, and right joins all in one.

This ability to express conjunctions and negations, all within the grammar, may be come as a surprise: As context-free grammars are not closed under conjunction or complement, there is no way to transform a grammar into its negation. However, we show that our *specialized* grammars *are* closed under conjunction, disjunction, and complement. Hence, Boolean combinations of patterns result in *algebras* of specialized grammars.

Besides controlling test generation, our evocative expressions (which we call *ewoks*) can be used in various ways. For example:

1) When used in *debugging,* the evocative patterns can quickly generate test inputs that contain a given failure-inducing input, validating fixes. This works particularly well for *abstract failure-inducing inputs* —strings in which specific parts are generalized to nonterminals, abstracting over concrete values [25]—, which translate immediately into evocative patterns.
2) When used in conjunction with *abstract failure-inducing inputs* which represent specific program behaviors, our evocative expressions can be used to generate inputs that represent complex program behavior, that can be checked for conformance. For example, one may mine *abstract inputs* that represent the coverage of specific portions of the program. The evocative patterns corresponding to such *abstract inputs* may be used to produce complex evocative expressions that correspond to inputs that (1) cover various parts of the program, (2) does not cover specific parts of the program, or (3) any combination where some parts are guaranteed not to be covered, while other parts are guaranteed to be covered. That is, with evocative expressions, one can build complex *oracles* out of simpler oracles.
3) When used with a *parser,* they can check whether a given input meets a specified evocative expression—for instance, to identify inputs that trigger a failure or vulnerability.
4) Evocative expressions can serve as *query languages* for searching and capturing specific patterns in structured data. Any of the above patterns could be used in a query.
5) Going further, evocative expressions might serve as *alternatives for regular expressions*—especially as they allow to express negation. Given their verbosity, structure, and named subexpressions, evocative expressions may be easier to read, write, and maintain than regular expressions.
6) Evocative expressions can serve as *configuration language* to adapt generic configurations towards specific goals.

Features in software product lines, customizations of user interfaces, and permissions in access control lists could be combined in arbitrary ways.

This work opens the door towards giving developers simple, yet powerful controls over what fuzzers produce, without requiring them to modify fuzzer sources or rewrite grammars and other specs from scratch. Beyond fuzzing, usages of specialized grammars also include 1) generating targeted failure-inducing inputs for validating fixes; 2) *parsing inputs,* checking whether they contains a particular pattern (or combination thereof); 3) *querying* patterns in data (possibly even replacing regular expressions); 4) *configuring* and customizing systems towards specific goals.

After introducing basic definitions (Section II), the remainder of this paper is organized along its contributions:

Abstract patterns. Section III introduces *abstract patterns,* which form the base for grammar specializations.

Specializing grammars. In Section IV, we introduce the algorithm for *specializing grammars* to contain at least one instance of the pattern on any of the generated inputs.

Specialization algebras. Section V shows how to combine specializations into *algebras* using Boolean operators. We provide (partially mechanized) proofs that grammar specializations are closed under disjunction, conjunction and negation.

Implementation and usage. Section VI introduces *Evogram,* a grammar transformer prototype that implements these techniques, as well as *usage scenarios* for testing, debugging, and adaptation.

Evaluation. In Section VII, we show that the specialized grammars accurately *produce* and *recognize* inputs that satisfy the given patterns.

Related work. Comparing to related work (Section VIII), ours is the first approach that allows test generators to express absence and conjunction of patterns in a given context. In contrast to the closest related work [25], Evogram can *produce* and *recognize* inputs that contain failure-inducing patterns in arbitrary contexts.

As detailed in Section IX, Evogram and all experiments are available for reuse and replication. We close with conclusion and future work.

II. Definitions

The following definitions are based on [25].

Alphabet. The *alphabet* of the input language of a program is the set of all symbols accepted by the program.

Input. A contiguous sequence of symbols from the alphabet that is passed to a given program.

Terminal. A sequence of symbols from the alphabet. These form the leaves of the derivation tree.

Nonterminal. A symbol outside the alphabet whose expansion is defined in the grammar.

Rule. A finite sequence of *terms* (two *types* of terms: terminals and nonterminals) that describe an expansion of a given nonterminal.

Definition. A set of rules that describe the expansion of a nonterminal or how the nonterminal is *matched.*

Context-Free Grammar. The context-free grammar is composed of a set of nonterminals and corresponding definitions that define the structure of the nonterminal.

Derivation. A terminal *derives* a string if the string contains only the symbols in the terminal. A nonterminal *derives* a string if the corresponding definition *derives* the string. A definition *derives* the string if one of the rules in the definition *derives* the string. A rule *derives* a string if the sequence of terms that make up the rule can derive the string, deriving one substring after another contiguously (also called *parsing*). *Generation* is defined complementarily [25].

Derivation Tree. An ordered tree that describes how an input string is *derived* by the given start symbol. An abstract derivation tree has some nodes marked as abstract, and abstract nodes may not have child nodes.

Compatible Node. A node is *compatible* to another if both have the same nonterminal. A tree is *compatible* to another if both have compatible root nodes.

Reachable Nonterminal. A nonterminal a is *reachable* from another nonterminal b if a is reachable from any of the rules in the definition of b. A nonterminal is reachable from a rule if 1) that nonterminal is present in the rule or 2) that nonterminal is reachable from any of the nonterminals in the rule. For example, $\langle$*array*$\rangle$ is reachable from $\langle$*json*$\rangle$ $\langle$*elt*$\rangle$, $\langle$*object*$\rangle$, $\langle$*items*$\rangle$, $\langle$*item*$\rangle$, $\langle$*elts*$\rangle$ and also itself.

Subtree. For any node, a *subtree* is a tree rooted in any nonterminal reachable from that node. The *characteristic node* of that subtree is its top most node and its nonterminal is the *characteristic nonterminal.*

We also introduce the following new definitions, generalizing beyond failure-inducing inputs to *evocative* inputs—that is, inputs that trigger a specific program behavior.

Evocative Input. An input string is *evocative* or failure-inducing, if on execution of the program with the string as input, the program fails in a particular fashion or displays an expected behavior.

Evocative Fragment. A fragment of an evocative input string is *evocative* or failure-inducing if some property of the fragment is the cause of the behavior or failure observed. That is, we can transplant the subtree corresponding to that fragment to a compatible node in another derivation tree, and the corresponding string reproduces the same behavior.

III. Abstract Patterns

Our end goal is to generate specialized grammars that obey given constraints. Our ongoing example will be the code constraints from Fig. 2, requiring 1) that at least one JSON object has an empty string as key, and 2) no JSON object has `null` as a key value. In our pattern language, we can produce

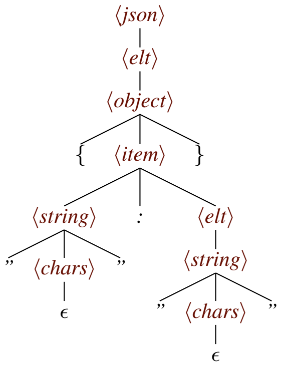


Fig. 3: Derivation tree for {"":""}

the following evocative expression from which a grammar that obeys the given constraints can be produced.

$$\langle json_{E\wedge\overline{N}}\rangle \tag{4}$$
$$\textbf{where } \langle item_E\rangle \textbf{ is } \texttt{""}\texttt{:}\langle elt\rangle \tag{5}$$
$$\langle item_N\rangle \textbf{ is } \langle string\rangle\texttt{:null} \tag{6}$$

This expression is equivalent to the compound expression in Equation (3). Here, $\langle item_E\rangle$ (5) represents the constraint of **E**mpty key, and $\langle item_N\rangle$ (6) represents the constraint of **N**ull value. The expression $\langle json_{E\wedge\overline{N}}\rangle$ represents the specialized grammar with the combined constraint of *at least one empty key*, and *no* `null` *values* in any generated input.[1] In the rest of the paper, this forms the running example, and we see how each component of this expression is derived.

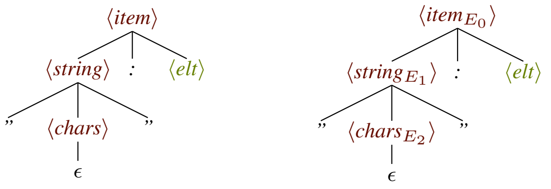
How do we relate these expressions to grammars? The constraints are predicates on the derivation tree. That is $\langle item_E\rangle$ **is** `"":`$\langle elt\rangle$ represents the constraint that a derivation tree produced should contain an $\langle item\rangle$ node with two children. The first should be an empty string, and the second could be any $\langle elt\rangle$.

A. Finding Abstract Patterns in Derivation Trees

To derive the constraint **E** that represents a particular failure condition, we start with an evocative input that induces the expected failure: `{"":""}`. This will have a derivation tree as given in Fig. 3. Are all parts of the input equally needed to satisfy the constraint? Indeed, we can easily see that the key value $\langle elt\rangle$ of value `'""'` can be replaced with any other $\langle elt\rangle$, and still induce the failure. That is, the following is an abstract representation of inputs that induces the same failure: `{"":`$\langle elt\rangle$`}`. We note that this is exactly the *abstract pattern* generated by DDSET [25].

That is, for any given derivation tree, the corresponding *abstract pattern* is the abstract string representation where the string representation of nodes marked as abstract are the *corresponding nonterminals* (called an abstraction). For example, `{`$\langle string\rangle$`:null}` is an abstract pattern, and contains $\langle string\rangle$ as an abstraction of the corresponding input fragment. When deriving an abstract pattern, a nonterminal also matches the corresponding abstraction.

Such abstract patterns concisely and precisely specify what parts of the input are important, and the abstract pattern is a recipe for *minimal* and *complete* evocative inputs. However, we are only interested in a smaller part of the corresponding abstract derivation tree. We are interested in the smallest subtree that when included in a larger input, induces the failure reliably.



(a) Abstract tree

(b) Specialized nodes

Fig. 4: Abstract trees for $\langle item_E\rangle$ **is** `"":`$\langle elt\rangle$

B. Abstract Patterns to Evocative Patterns

So, what part of the derivation tree actually caused the failure? On inspection, one can see that a much smaller part of the derivation tree under the node for $\langle item\rangle$ is sufficient to induce the failure. That is, one can replace an $\langle item\rangle$ node in any derivation tree with this node, and the resulting string will induce the failure. That is, the subtree given in Fig. 4a accurately captures the constraint. Such subtrees can be represented by their characteristic nonterminal, and the string representation of the rest of the tree, using nonterminal symbols where abstract nodes are present. We call these *evocative patterns*. For example, $\langle item_E\rangle$ **is** `"":`$\langle elt\rangle$ is the evocative pattern that represents the subtree given in Fig. 4a.

Given an abstract pattern, one can also automatically obtain the characteristic node by simply looking for the smallest subtree that contains all terminal symbols in the abstract pattern. We use such evocative patterns in our evaluation. However, the characteristic node obtained could likely be trimmed further to produce a smaller subtree that can still induce failures reliably, in a larger variety of contexts.

IV. Transforming Grammars

We now show how one can transform the underlying grammar such that any input generated from it would contain at least one instance of the evocative pattern. The underlying non-specialized grammar is called the *base grammar*, and the definitions, rules, and nonterminals in that grammar are called *base definitions*, *base rules* and *base nonterminals* respectively. Note that we require base grammars to be *non-ambiguous*. That is, any given string should have only a single derivation. Transforming the grammar means to specialize it, and this is accomplished by specializing each nonterminal, the corresponding definition, and each applicable rule in the definition. We describe specializations of nonterminals and their corresponding definitions next.

[1] While both evocative patterns have $\langle item\rangle$ as the characteristic nonterminal in this example, there is no such requirement. That is, $\langle json_{E\vee I}\rangle$ **where** $\langle number_I\rangle$ **is** `011`$\langle digits\rangle$ is allowed.

Specialized nonterminal. A specialized nonterminal is a base nonterminal name followed by a specialization suffix. For example, given a nonterminal $\langle item_N \rangle$, the $\langle item \rangle$ is the corresponding base nonterminal (also called the base representation of the specialized nonterminal) and the subscript `N` is the specialization representing the evocative expression. Note that a terminal cannot be a specialized. Hence, the base representation of a terminal is always itself. A term is same *kind* as another term if both have the same base representation.

The subscript `N` is constructed as follows: An evocative pattern is defined as

$$\langle nonterminal_{\ subscript} \rangle \ \textbf{is} \ \texttt{value}$$

where *subscript* refers to the particular constraint represented by *value*. We use a related notation for specialization. We use $\langle nonterminal_{subscript} \rangle$ to denote a specialized nonterminal where *subscript* is a Boolean expression of the constraints acting on the given nonterminal. Note that in terms of the context-free grammar, this is *just another identifier.* A context-free grammar containing such specialized nonterminals can be used exactly like any other context-free grammar. For us, however, it is a unique and expressive identifier that states the precise purpose.

Specialized definition. A specialized definition can have rules with specialized nonterminals or some of the expansion rules may be removed from the corresponding base grammar definition. The following is a specialized definition of $\langle object_E \rangle$, removing the empty object expansion, and includes specialized nonterminals$\langle items_E \rangle$ and $\langle items_{E_1} \rangle$.

$\langle object_E \rangle$ ::= '{' $\langle items_E \rangle$ '}' | '{' $\langle items_{E_1} \rangle$ '}'

Some of the specialized rules (such as above) may have the same base representation rule – $\langle object \rangle$ ::= '{' $\langle items \rangle$ '}' in the above example. Such a set of rules in a definition is called a *ruleset* for that base rule, and such rules have same *kind*.

A. Translating evocative patterns to pattern grammars

We translate the abstract derivation tree rooted at characteristic node to a grammar capable of producing instances of the evocative fragment, using our JSON grammar (Fig. 1) as running example. To do this, we specialize the non-abstract nodes from the abstract tree with the evocative pattern name and *unique suffixes* such that no two nonterminal symbols will reuse the same name and suffix. For example, given the abstract tree Fig. 4a, nodes are specialized in Fig. 4b. Next, with the characteristic nonterminal ($\langle item_{E_0} \rangle$ here) as the start symbol, this tree is collapsed into a grammar, children of each node forming a single rule definition for their nonterminal.

$\langle item_{E_0} \rangle$ ::= $\langle string_{E_1} \rangle$ ':' $\langle elt \rangle$
$\langle string_{E_1} \rangle$ ::= '"' $\langle chars_{E_2} \rangle$ '"'
$\langle chars_{E_2} \rangle$ ::= ϵ

Next, we will see how the characteristic symbol of pattern grammar ($\langle item_{E_0} \rangle$) is connected to the rest of the grammar.

B. Constructing a reaching grammar

1) Finding insertable positions: To be able to instantiate instances of the evocative fragment, one needs to be able to *reach* the characteristic nonterminal of the evocative pattern. For example, to reach $\langle item \rangle$ in the JSON grammar, one has to start from one of the following nonterminal symbols: $\langle json \rangle$, $\langle elt \rangle$, $\langle object \rangle$, $\langle items \rangle$, $\langle item \rangle$, $\langle array \rangle$, $\langle elts \rangle$.

For each rule in the grammar, we identify the nonterminal terms that can reach the nonterminal of the characteristic node. For JSON, the following (*) are the insertable positions.

$\langle json \rangle$ ::= $\langle *elt \rangle$
$\langle elt \rangle$ ::= $\langle *object \rangle$ | $\langle *array \rangle$
$\langle object \rangle$ ::= '{' $\langle *items \rangle$ '}'
$\langle items \rangle$ ::= $\langle *item \rangle$ | $\langle *item \rangle$ ',' $\langle *items \rangle$
$\langle item \rangle$ ::= $\langle string \rangle$ ':' $\langle *elt \rangle$
$\langle array \rangle$ ::= '[' $\langle *elts \rangle$ ']'
$\langle elts \rangle$ ::= $\langle *elt \rangle$ | $\langle *elt \rangle$ ',' $\langle *elts \rangle$

In the JSON grammar, $\langle string \rangle$ cannot reach $\langle item \rangle$. Hence, it is unmarked. None of the other nonterminals have any rules that can reach $\langle item \rangle$.

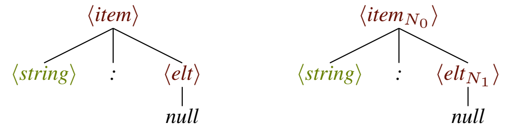
For the *reaching grammar*, we take each rule of a given definition, and identify the insertable positions in the rule. Next, for each insertable position, we produce a copy of the rule with the nonterminal at the identified position specialized with the evocative fragment name. The reaching definition corresponding to the new specialized nonterminal is the collection of such new rules. If the rule has no insertable positions, it cannot derive evocative fragments. Hence, such rules are discarded. If a nonterminal has no reaching rules, its definition is empty, and it is discarded from the *reaching grammar*. The *reaching grammar* is a grammar composed of reaching nonterminals and their corresponding *reaching definitions*. The definition for nonterminals in reachable grammar for `E` is as follows

$\langle json_E \rangle$ ::= $\langle elt_E \rangle$
$\langle elt_E \rangle$ ::= $\langle object_E \rangle$ | $\langle array_E \rangle$
$\langle object_E \rangle$ ::= '{' $\langle items_E \rangle$ '}'
$\langle items_E \rangle$ ::= $\langle item_E \rangle$ ',' $\langle items \rangle$ | $\langle item \rangle$ ',' $\langle items_E \rangle$
  | $\langle item_E \rangle$
$\langle item_E \rangle$ ::= $\langle string \rangle$ ':' $\langle elt_E \rangle$
$\langle array_E \rangle$ ::= '[' $\langle elts_E \rangle$ ']'
$\langle elts_E \rangle$ ::= $\langle elt_E \rangle$ ',' $\langle elts \rangle$ | $\langle elt \rangle$ ',' $\langle elts_E \rangle$ | $\langle elt_E \rangle$

C. Connecting pattern grammar and reaching grammar

For the final grammar, we merge the pattern grammar and the reaching grammar, and use the start symbol from the reaching grammar. The connection is made at the reaching characteristic nonterminal (here, $\langle item_E \rangle$). We merge the definition from the characteristic nonterminal of the pattern grammar (here, $\langle item_{E_0} \rangle$) to the reaching nonterminal of characteristic node. Note that $\langle string_{E_1} \rangle$ was previously defined and represents the empty string.

$\langle item_E \rangle$ ::= $\langle string \rangle$ ':' $\langle elt_E \rangle$ | $\langle string_{E_1} \rangle$ ':' $\langle elt \rangle$



(a) Abstract tree (b) Specialized nodes

Fig. 5: Abstract trees for $\langle \mathit{item}_N \rangle$ **is** $\langle \mathit{string} \rangle$`:null`

V. Algebra of Grammar Specializations

Next, we discuss how arbitrary Boolean expressions can be composed from other grammar specializations. We call these *evocative expressions*. Again, consider the expression introduced in Equation (4) that represents a specialized grammar producing inputs that contain at least one empty key, and does not contain a key value `null`. This will be our ongoing example for specializations.

We first look at negation of grammar specializations. In the grammar, we indicate negation by an over line. To illustrate the algebra, we use the second pattern—at least one instance of a `null` key value in the input. The abstract pattern from which the expression is derived is $\{\langle \mathit{string} \rangle$`:null`$\}$, and the evocative pattern is $\langle \mathit{item} \rangle$ **is** $\langle \mathit{string} \rangle$`:null`. The abstract tree is given in Fig. 5a, and its pattern tree in Fig. 5b. From this, the specialized grammar *contains all the above nonterminal definitions,* as well as

$\langle \mathit{json}_N \rangle$ ::= $\langle \mathit{elt}_N \rangle$
$\langle \mathit{elt}_N \rangle$ ::= $\langle \mathit{object}_N \rangle \mid \langle \mathit{array}_N \rangle$
$\langle \mathit{array}_N \rangle$ ::= '`[`' $\langle \mathit{elts}_N \rangle$ '`]`'
$\langle \mathit{object}_N \rangle$ ::= '`{`' $\langle \mathit{items}_N \rangle$ '`}`'
$\langle \mathit{elts}_N \rangle$ ::= $\langle \mathit{elt}_N \rangle$ '`,`' $\langle \mathit{elts} \rangle \mid \langle \mathit{elt} \rangle$ '`,`' $\langle \mathit{elts}_N \rangle \mid \langle \mathit{elt}_N \rangle$
$\langle \mathit{items}_N \rangle$ ::= $\langle \mathit{item}_N \rangle$ '`,`' $\langle \mathit{items} \rangle \mid \langle \mathit{item} \rangle$ '`,`' $\langle \mathit{items}_N \rangle$
 $\mid$ $\langle \mathit{item}\ N \rangle$
$\langle \mathit{item}_N \rangle$ ::= $\langle \mathit{string} \rangle$ '`:`' $\langle \mathit{elt}_N \rangle \mid \langle \mathit{string} \rangle$ '`:`' $\langle \mathit{elt}_{N_1} \rangle$
$\langle \mathit{elt}_{N_1} \rangle$ ::= '`null`'

Now we come to the actual definition of Boolean expressions. We start with the axioms for terms, where we have two basic requirements:

1) None of the operations can change the term *type* or *kind*.
2) In the case of two or more operands, operations are only defined with respect to the same *kind* of terms.

We use $\langle \mathit{nonterminal}_\top \rangle$ (short: $\langle \mathit{nonterminal} \rangle$ or $\top$ where unambiguous) to represent base nonterminal, and $\langle \mathit{nonterminal}_\perp \rangle$ or $\perp$ for nonterminals with empty definition. Rules with $\perp$ terms are removed recursively at the end of evaluation of the complete expression.

A. Negation of a single term

Negation of a term means that if a term derived a given substring, the negated term will not derive the same substring, and vice versa. Negation of a terminal is empty. A nonterminal with specialization is negated with respect to the specialization. That is, $\langle \mathit{array}_{\overline{E}} \rangle$ represents elements in $\langle \mathit{array} \rangle$ not derived by $\langle \mathit{array}_E \rangle$. Negation of a base nonterminal is $\perp$. Negation of an empty nonterminal is $\top$. That is, $\langle \mathit{item}_{\overline{\perp}} \rangle = \langle \mathit{item} \rangle$.

B. Conjunction of two terms

The term from conjunction of two terms will only derive (any and all) substrings derived by both the operands, and is defined only for terms of the same *kind*. The conjunction between two equal nonterminals is the same nonterminal, between $\top$ and a specialized nonterminal is the specialized nonterminal, between a nonterminal and its negation is $\perp$, and between $\perp$ and anything else is $\perp$. For example, conjunction of $\langle \mathit{array}_E \rangle$ and $\langle \mathit{array}_N \rangle$ is $\langle \mathit{array}_{E \wedge N} \rangle$ representing elements in $\langle \mathit{array} \rangle$ derived by both $\langle \mathit{array}_E \rangle$ and $\langle \mathit{array}_N \rangle$.

C. Disjunction of two terms

The term from disjunction of two terms will only derive (any and all) substrings derived by either operands, and is only defined for terms of the same *kind*. The disjunction between two equal nonterminals is the same nonterminal, between $\top$ and a specialized nonterminal is $\top$, between a nonterminal and its negation is $\top$, and between any nonterminal and $\perp$ is the first nonterminal. For example, disjunction of $\langle \mathit{array}_E \rangle$ and $\langle \mathit{array}_N \rangle$ is $\langle \mathit{array}_{E \vee N} \rangle$ representing elements in $\langle \mathit{array} \rangle$ derived by either $\langle \mathit{array}_E \rangle$ or $\langle \mathit{array}_N \rangle$.

D. Negation of a single rule

Negation of a rule produces as many rules as there are specialized nonterminals each with one specialized nonterminal negated, and the rules from negation of a specialized rule will not be able to derive a string derived by the operand, and vice versa. Say you have a rule $\langle \mathit{elt}_E \rangle$ `','` $\langle \mathit{elts}_E \rangle$. To negate the rule, we take each specialized nonterminal, and negate it one at a time. In this example, the result is two rules: $\langle \mathit{elt}_{\overline{E}} \rangle$ `','` $\langle \mathit{elts}_E \rangle$ and $\langle \mathit{elt}_E \rangle$ `','` $\langle \mathit{elts}\ _{\overline{E}} \rangle$.
Proof: $\lhd$ [2]A string derived by the operand could not have been derived by any of the new rules because each rule contains at least one nonterminal that will reject the corresponding derivation. $\rhd$ If any new rule derived a string, at least one nonterminal in operand will reject it[3].

E. Conjunction of two rules

Conjunction is defined only between rules of the same *kind*, and the result will derive (any and all) strings that could be derived by both the operands. To produce a conjunction of two rules, we line up both, and produce a *conjunction* of terms from each at corresponding positions. That is, given two rules $\langle \mathit{elt}_E \rangle$ `','` $\langle \mathit{elts}_E \rangle$ and $\langle \mathit{elt}_N \rangle$ `','` $\langle \mathit{elts}_N \rangle$, the conjunction is $\langle \mathit{elt}_{E \wedge N} \rangle$ `','` $\langle \mathit{elts}_{E \wedge N} \rangle$.
Proof: $\lhd$ Say the new rule derives a string. Pick an operand. Each nonterminal in the deriving rule could be replaced by one nonterminal in operand rule without affecting the derivation. $\rhd$ Say a string is derived by both operands. That string could be split into contiguous substrings such that each substring is derived by either a terminal or corresponding nonterminals from both operands. If two nonterminals derive the same string, then their conjunction also derives it.

[2] We use $\lhd$ and $\rhd$ to indicate the direction of the proof.

[3] The appendix for this paper contains a partial mechanization for proofs in HOL4 [26].

F. Disjunction of two rules

Disjunction is defined only between rules of the same *kind*, and the result will derive (any and all) strings that could be derived by one of the operands. To produce a disjunction of two rules, merge them or use each as alternatives in any resultant definition. One may merge two rules into one if the two rules differ only by a single specialized nonterminal which is replaced by a disjunction of operand specializations. Given two rules $\langle \mathit{elt}_E \rangle$ `','` $\langle \mathit{elts} \rangle$ and $\langle \mathit{elt}_N \rangle$ `','` $\langle \mathit{elts} \rangle$, the disjunction is $\langle \mathit{elt}_{E \vee N} \rangle$ `','` $\langle \mathit{elts} \rangle$. On the other hand, given $\langle \mathit{elt}_E \rangle$ `','` $\langle \mathit{elts}_E \rangle$ and $\langle \mathit{elt}_N \rangle$ `','` $\langle \mathit{elts}_N \rangle$, the disjunction is exactly the same as original two rules.
Proof: $\lhd$ Pick any operand. The merged rule differs from the operand rule in exactly one place, which is a superset of operand specialization, any string derived by the operand could also be derived by the new rule. $\rhd$ Any string that is derived by the new rule can be split into contiguous substrings such that each term in the new rule derives one substring. The new rule differs from operand rules in exactly one nonterminal, which is a disjunction of operand nonterminals.

G. Negation of a ruleset

The result of negation of a ruleset derives (any and all) strings that will not be derived by the negated ruleset. Given a ruleset with multiple rules, one has to only remember that they are alternatives. E.g. $S = R_1|R_2|R_3$. Hence, negation is based on Boolean algebra. That is, $\overline{R_1|R_2|R_3}$ is same as $\overline{R_1} \wedge \overline{R_2} \wedge \overline{R_3}$. Given that each rule negation results in multiple alternates, we apply distributive law. That is the new ruleset is as follows: $\{r_1 \wedge r_2 \wedge r_3 : r_1 \in \overline{R_1}, r_2 \in \overline{R_2}, r_3 \in \overline{R_3}\}$
Proof: $\lhd$ Any string derived by any of the new rules will be derived by each of the negations because it is a conjunction of one rule from negation of each rule. Since there cannot be a string that is derived from both a rule and its negation, there does not exist a derivation using the original ruleset. $\rhd$ Any string that is derived by the original ruleset will not be derived by the new because the new is composed of negations from each rule in the original.

H. Conjunction of two rulesets

Result of conjunction of two rulesets derives (any and all) strings derived by both of the operands. For conjunction between two rulesets, we take one rule from each, and compute the conjunction of both. The new rules will be all such conjunctions: $\{r_1 \wedge r_2 : r_1 \in S_1, r_2 \in S_2\}$
Proof: $\lhd$ Any string that is derived by one of the rules in the new ruleset will have been derived by at least one rule from both operand rulesets. $\rhd$ If both operand rulesets derived the same string, then there exist a rule in both that derived it, and the new ruleset contains conjunction of all such rules.

I. Disjunction of two rulesets

The result of disjunction of two rulesets derives (any and all) strings derived by any of the operands. Disjunction of two rulesets is a new ruleset with rules from both, merging rules that can be merged.
Proof: $\lhd$ Any string derived by the new ruleset will be derived by a rule in at least one of the operands. $\rhd$ Any string derived by a rule in a parent operand will be derived by new ruleset because the same rule is there in the result.

J. Negation of a definition

The result of a negation derives (any and all) strings that will not be derived by the operand. For negation of a definition, any base rule that is *not represented* by a ruleset in the non-negated definition is added directly to the negated definition. These correspond to the rules with empty terms that we discarded when the operand was produced. Then, negations from each rulesets are combined together and added to the negated definition. The negation of a definition is represented by the negation of its corresponding nonterminal.
Proof: $\lhd$ Say a string was derived by the operand. A rule that was not present in the original rulesets could not have derived the string as part of the original non-negated definition. Hence, non-represented rules could not derive a string that could be derived by the operand. Next, all other rulesets are negations of rulesets in the operand none of which can derive a string derived by the operand. $\rhd$ If a string was derived by the new definition, this string was either derived by one of the directly added rules, in which case, there is no corresponding rule in the operand, or by one of the negated rules, in which case, the operand could not have derived it anyway.

K. Conjunction of two definitions

Result of conjunction of two definitions derives (any and all) strings derived by both the operands. To produce a conjunction of two definitions, the rulesets of the same kind from each operand are paired up, and conjoined together, dropping those that do not have a pair in either operand. The conjunction of two definitions is represented by the conjunction of the specialization of their corresponding nonterminals.
Proof: $\lhd$ Any string derived by the result would be derived by one of the conjoined rulesets, which is a conjunction of rulesets from operands. $\rhd$ If a string was derived by both operands, a ruleset exists in both that derives the string. A conjunction of all such ruleset pairs exists in the new definition.

L. Disjunction of two definitions

The result of disjunction derives (any and all) strings derived by any of the operands. Disjunction of two definitions is again simply a combination of rulesets of the same kind from each operand, and is represented by the disjunction of the specialization of their corresponding nonterminals.
Proof: $\lhd$ Any string derived by the new definition was derived by one of the rulesets, which came from one of the operands. $\rhd$ Pick an operand, and say it derived a string. The ruleset from that operand is part of the new definition.

M. Negation of a specialized grammar

Negation of an arbitrary grammar specialization is produced by first negating its start symbol, and constructing the negations (and other expressions) for any specialized nonterminals that is needed from the definition of the start symbol recursively. Given below is the grammar for evocative expression $\langle \mathit{json}_{\overline{N}} \rangle$ **where** $\langle \mathit{item}_{N} \rangle$ **is** $\langle \mathit{string} \rangle$`:null`.

$$\langle \mathit{json}_{\overline{N}} \rangle ::= \langle \mathit{elt}_{\overline{N}} \rangle$$

$$\langle \mathit{elt}_{\overline{N}} \rangle ::= \text{‘}\texttt{false}\text{’} \mid \text{‘}\texttt{null}\text{’} \mid \text{‘}\texttt{true}\text{’} \\ \mid \langle \mathit{number} \rangle \mid \langle \mathit{string} \rangle \mid \langle \mathit{array}_{\overline{N}} \rangle \mid \langle \mathit{object}_{\overline{N}} \rangle$$

$$\langle \mathit{array}_{\overline{N}} \rangle ::= \text{‘}\texttt{[]}\text{’} \mid \text{‘}\texttt{[}\text{’}\ \langle \mathit{elts}_{\overline{N}} \rangle\ \text{‘}\texttt{]}\text{’}$$

$$\langle \mathit{object}_{\overline{N}} \rangle ::= \text{‘}\texttt{\{\}}\text{’} \mid \text{‘}\texttt{\{}\text{’}\ \langle \mathit{items}_{\overline{N}} \rangle\ \text{‘}\texttt{\}}\text{’}$$

$$\langle \mathit{elts}_{\overline{N}} \rangle ::= \langle \mathit{elt}_{\overline{N}} \rangle \mid \langle \mathit{elt}_{\overline{N}} \rangle\ \text{‘}\texttt{,}\text{’}\ \langle \mathit{elts}_{\overline{N}} \rangle$$

$$\langle \mathit{items}_{\overline{N}} \rangle ::= \langle \mathit{item}_{\overline{N}} \rangle \mid \langle \mathit{item}_{\overline{N}} \rangle\ \text{‘}\texttt{,}\text{’}\ \langle \mathit{items}_{\overline{N}} \rangle$$

$$\langle \mathit{item}_{\overline{N}} \rangle ::= \langle \mathit{string} \rangle\ \text{‘}\texttt{:}\text{’}\ \langle \mathit{elt}_{\overline{N} \wedge \overline{N_1}} \rangle$$

$$\langle \mathit{elt}_{\overline{N} \wedge \overline{N_1}} \rangle ::= \text{‘}\texttt{false}\text{’} \mid \text{‘}\texttt{true}\text{’} \\ \mid \langle \mathit{number} \rangle \mid \langle \mathit{string} \rangle \mid \langle \mathit{array}_{\overline{N}} \rangle \mid \langle \mathit{object}_{\overline{N}} \rangle$$

N. Conjunction of two grammar specializations

The conjunction between two grammar specializations is produced by first conjoining their start symbol, and hence their corresponding definitions recursively. The following is the rest of the specialized nonterminals for the grammar specialization representing Equation (4).

$$\langle \mathit{json}_{E \wedge \overline{N}} \rangle ::= \langle \mathit{elt}_{E \wedge \overline{N}} \rangle$$

$$\langle \mathit{elt}_{E \wedge \overline{N}} \rangle ::= \langle \mathit{array}_{E \wedge \overline{N}} \rangle \mid \langle \mathit{object}_{E \wedge \overline{N}} \rangle$$

$$\langle \mathit{array}_{E \wedge \overline{N}} \rangle ::= \text{‘}\texttt{[}\text{’}\ \langle \mathit{elts}_{E \wedge \overline{N}} \rangle\ \text{‘}\texttt{]}\text{’}$$

$$\langle \mathit{object}_{E \wedge \overline{N}} \rangle ::= \text{‘}\texttt{\{}\text{’}\ \langle \mathit{items}_{E \wedge \overline{N}} \rangle\ \text{‘}\texttt{\}}\text{’}$$

$$\langle \mathit{elts}_{E \wedge \overline{N}} \rangle ::= \langle \mathit{elt}_{E \wedge \overline{N}} \rangle \mid \langle \mathit{elt}_{E \wedge \overline{N}} \rangle\ \text{‘}\texttt{,}\text{’}\ \langle \mathit{elts}_{\overline{N}} \rangle \\ \mid \langle \mathit{elt}_{\overline{N}} \rangle\ \text{‘}\texttt{,}\text{’}\ \langle \mathit{elts}_{E \wedge \overline{N}} \rangle$$

$$\langle \mathit{elt}_{\overline{N} \wedge \overline{N_1}} \rangle ::= \text{‘}\texttt{false}\text{’} \mid \text{‘}\texttt{true}\text{’} \\ \mid \langle \mathit{number} \rangle \mid \langle \mathit{string} \rangle \mid \langle \mathit{object}_{\overline{N}} \rangle \mid \langle \mathit{array}_{\overline{N}} \rangle$$

$$\langle \mathit{items}_{E \wedge \overline{N}} \rangle ::= \langle \mathit{item}_{E \wedge \overline{N}} \rangle \mid \langle \mathit{item}_{E \wedge \overline{N}} \rangle\ \text{‘}\texttt{,}\text{’}\ \langle \mathit{items}_{\overline{N}} \rangle \\ \mid \langle \mathit{item}_{\overline{N}} \rangle\ \text{‘}\texttt{,}\text{’}\ \langle \mathit{items}_{E \wedge \overline{N}} \rangle$$

$$\langle \mathit{item}_{E \wedge \overline{N}} \rangle ::= \langle \mathit{string}_{E_1} \rangle\ \text{‘}\texttt{:}\text{’}\ \langle \mathit{elt}_{\overline{N} \wedge \overline{N_1}} \rangle \\ \mid \langle \mathit{string} \rangle\ \text{‘}\texttt{:}\text{’}\ \langle \mathit{elt}_{E \wedge \overline{N} \wedge \overline{N_1}} \rangle$$

$$\langle \mathit{elt}_{E \wedge \overline{N} \wedge \overline{N_1}} \rangle ::= \langle \mathit{array}_{E \wedge \overline{N}} \rangle \mid \langle \mathit{object}_{E \wedge \overline{N}} \rangle$$

O. Disjunction of grammars

Disjunction of two grammars is built by disjunction of their start symbol, and corresponding definitions recursively.

P. Constructing definitions corresponding to nonterminals

To convert an evocative expression to grammar, we try to construct the start symbol of the new grammar. To do that, we need to apply the given operation to the corresponding definitions of the start symbols of the subject grammars. These in turn require further specialized nonterminals to be computed, which is done recursively until no more specialized nonterminals need to be constructed.

At each step, the nonterminal to be constructed is simplified into its canonical *DNF* form. We then reconstruct this term from our smallest specializations (i.e the specializations that correspond to a single pattern) that can be directly reconstructed from the evocative patterns.

We note that one can also collect all the Boolean expressions for the same nonterminal from operand grammars, and *solve* them to reconstruct the new term in terms of available terms. That is, the algebraic operations can be done on conforming grammars even if the smallest terms are not known, and the nonterminals are not labelled with pattern expressions.

VI. Implementation and Usage Scenarios

We have implemented Evogram as a standalone Jupyter[4] notebook with detailed steps and examples[5]. The grammars are accepted and produced in the Fuzzingbook [19] canonical format, and the evocative patterns in the DDSET format. Grammars can be converted from and to other popular formats, such as ANTLR.

We see the following usage scenarios for Evogram:

Precise control during fuzzing. Given a particular pattern or a set of patterns that is associated with some behavior of the program, one can use grammar specializations to specify that inputs generated satisfy arbitrary Boolean constraints, inducing certain failures while preventing others. The utility of such a grammar is that, one can precisely ask the fuzzer to concentrate on portions of code, or portions of input space one wants to explore further, avoiding the behaviors that one does not want to trigger.

Testing and validating fixes. The above control over test inputs works particularly well if a concrete failure-inducing input can be generalized towards an *abstract* failure-inducing input, as produced by DDSET [25]. When fed with such patterns, the Evogram produces a grammar whose produced strings all contain instantiated patterns, and thus variants of the original failure-inducing input in all sorts of contexts—which makes them helpful for validating fix correctness.

Validating inputs. Assuming that a certain vulnerability is known to be induced by a failure inducing pattern under specific contexts, one can encode the specific context under which the failure is induced, and allow only the negation of this pattern to proceed to the program, rejecting any string that may induce the failure. This provides the developers with a quick fix tool for identifying and quickly rejecting potentially malicious inputs such as SQL injections *without the knowledge of the internals* of the program.

Algebra of oracles. The *abstract behavior-inducing inputs* are mined from a given program and a given predicate using *DDSET*. Such *abstract inputs* correspond to specific program behaviors and are directly representable as evocative patterns. Given such evocative patterns, we can build evocative expressions that represent complex program behaviors and can be used to validate the program behavior for corresponding inputs.

Supercharged recognizers. Regular expressions (*regexes*) are typically used to match and extract parts of input by programmers. These inputs may often be encoded in a structured format such as JSON, XML, HTML, and SEXPR. However, the siren song of regular expressions is hard to ignore[6]. While the situation has started to change with projects such as Combi [27], SemGrep [28], and Coccinelle [29] that rely on the underlying grammar,

[4] https://jupyter.org

[5] https://zenodo.org/record/4456296

[6] https://stackoverflow.com/q/1732348/1420407

the constructs they use are limited (disjunction and at best conjunction), and the expressions need to be written by hand, which limits their usability. The evocative expressions can be 1) automatically mined from sample expressions and 2) combined to form arbitrarily complex matching specifications.

Further, evocative expressions can be extended to support constraints on each term such as `length` $<$ 10 or *variable* `a` is `defined`. The relational algebra of such constraints can be easily added to the underlying Boolean algebra.

Generating data structures. Algebraic data types (except GADTs) have a 1:1 mapping to context-free grammars, and property checkers such as *QuickCheck* rely on generation of data structures to verify properties. The specialized grammars from evocative expressions could precisely specify how and what properties these structures should contain.

Configuring and customizing systems. Most hierarchical structures can be represented as derivation trees from context-free grammars. Such structures include URLs, file systems, GUI navigation, object hierarchies, and more. All these structures have some concept of access control defining under what conditions certain elements can be reached, or certain actions can be performed. Such access control rules could be expressed via evocative patterns, specializing general access rules towards specific environments.

VII. Evaluation

We have evaluated Evogram and grammar specializations both in a *testing* context (producing inputs that contain specific evocative patterns) and in a *prevention* context (checking whether inputs contain a specific pattern). Specifically, we pose the following research questions:

RQ1 How effective are the specialized grammars produced by Evogram in *producing* expressions that contain (and do not contain) any evocative fragments?

RQ2 How effective are the generated specialized grammars in *recognizing* expressions that contain (or do not contain) evocative fragments?

A. Evaluation Setup

For evaluation, we use the subjects from DDSET [25]. The first are programming language interpreters: *clojure*, *closure*, *rhino* and *lua*. Input languages for these subjects are more constrained than simple context-free languages, with the ability to declare and use variables and functions. When using a grammar-based fuzzer, the probability of randomly generating a *semantically* valid string is extremely low. Hence, for such language interpreters, we rely on evaluating the specialized grammars only as recognizers. For the UNIX utilities *find* and *grep*, only syntactic validity is required. Hence, we evaluate these utilities as both producers and recognizers.

- For the evaluation as **producers,** we generate inputs using two strategies: The first strategy is to insert evocative fragments into the grammar. The resulting grammar (we call this the *evocative grammar*, and the strategy the *evocative strategy*) is then used for input generation. The second strategy negates the previous grammar, and the resulting grammar (we call this the *non-evocative grammar* and the strategy the *non evocative strategy*) is used to produce inputs. The inputs thus generated are fed to the program, and the program is monitored for the expected behavior and its absence.
- For the evaluation as **recognizers,** we require a source of inputs that induces the failure, as well as a source of inputs that is guaranteed not induce the failure. Since we are using the subjects from DDSET, we use inputs generated during the pattern mining for this purpose.
  That is, we collect the semantically valid strings generated during minimization and abstraction, which were found to have induced a failure or otherwise. We then evaluate whether the specialized grammars produced by Evogram using the *evocative strategy* are capable of recognizing evocative inputs, and rejecting the non-evocative inputs. Similarly, we evaluate whether the specialized grammars produced by Evogram using *non-evocative strategy* are capable of rejecting any failure inducing input.

TABLE I: Percentage of inputs generated with *evocative* strategy that induces failure (F) and *non-evocative* strategy that did not induce failure ($\neg F$).

Bug	F	$\neg F$
find 07b941b1	100% of 100	96% of 100
find 93623752	100% of 100	97% of 100
find c8491c11	100% of 100	93% of 100
find dbcb10e9	100% of 100	89% of 100
grep 3c3bdace	100% of 100	100% of 96
grep 54d55bba	100% of 100	100% of 92
grep 9c45c193	100% of 1	100% of 94
Total	100% of 701	99% of 682

B. Evaluation Results and Discussion

Our evaluation results are as follows: The result of generating evocative inputs using the *evocative* and *non-evocative* strategies, tested against the program for reproducing the failure is given in Table I. The F column contains the percentage of inputs using *evocative strategy* that induced the failure when tested against the program. The $\neg F$ column contains the percentage of inputs using *non-evocative strategy* that *did not* induce the failure when tested against the program.

From Table I, we find that specialized grammars generated by Evogram can produce inputs that contain the given evocative fragment, and reproduce the expected failure with high accuracy (100%) when used as a test generator for UNIX utilities. Further, when used to produce inputs that do not contain the evocative fragment, Evogram succeeds in avoiding such failure inducing fragments in its inputs with an accuracy of 99.0%. The bug *grep 9c445c193* could not be abstracted. Hence, only a single input was produced during fuzzing, and only three (substrings) recognized.

TABLE II: Recognizing failure inducing inputs with F and non-failure inducing inputs with $\neg F$.

Bug	F	$\neg F$
find 07b941b1	89.7% of 213	100% of 7
find 93623752	99.5% of 193	100% of 7
find c8491c11	98.5% of 196	100% of 8
find dbcb10e9	100% of 374	100% of 6
grep 3c3bdace	94.6% of 203	100% of 28
grep 54d55bba	83.1% of 213	100% of 24
grep 9c45c193	33.3% of 3	100% of 11
clojure 2092	80.2% of 360	99.0% of 100
clojure 2345	99.2% of 393	96.1% of 52
clojure 2450	92.7% of 704	99.6% of 267
clojure 2473	93.7% of 695	98.3% of 58
clojure 2518	44.4% of 9	98.7% of 76
clojure 2521	24.3% of 847	99.0% of 111
closure 1978	94.8% of 1672	98.0% of 101
closure 2808	100% of 64	88.8% of 27
closure 2842	1.5% of 261	98.9% of 176
closure 2937	36.6% of 1293	91.4% of 35
closure 3178	95.1% of 719	98.0% of 99
closure 3379	57.6% of 646	80.7% of 26
lua 5.3.5 4	3.1% of 161	98.6% of 142
rhino 385	65.7% of 472	93.3% of 30
rhino 386	97.8% of 363	93.2% of 44
Total	73.2% of 10054	97.8% of 1435

Evogram grammars induce expected failures with 100% accuracy and prevent such failures with 99.0% accuracy.

The result of recognizing evocative inputs using the *evocative* strategy is given in F column of Table II. The result of recognizing non-evocative inputs using the *non-evocative* strategy is given the $\neg F$ column.

Looking at Evogram's performance as a recognizer, from Table II we find that grammars from Evogram are able to recognize the failure inducing inputs for most bugs. For a number of bugs, however (*lua 5.3.5 4*, *clojure 2521*, *closure 2937*, *clojure 2518*, *grep 9c45c193*), less than half of the failure-inducing inputs were predicted as such. This is due to the DDSET pattern specializing to the first failure-inducing bug, which then leads to an overspecialization in the resulting grammar. However, even with this caveat, the grammars produced by Evogram are able to recognize failure inducing inputs with 73.2% accuracy and non-failure inducing inputs with 97.8% accuracy.

Evogram grammars recognize failure inducing inputs with 73.2% and non-failure inducing inputs with 97.8% accuracy.

Evocative expressions represent specialized context-free grammars that guarantee production of the requisite pattern somewhere in the input string produced. However, having the particular pattern somewhere in the string is no guarantee that a given failure is induced. For example, the `null` check may be ignored under specific circumstances. Hence, it is important to validate how effective the grammars are, in converting the input patterns to actual failures. Our results from Table I using real world bugs show that the evocative expressions work well in generating inputs that contain the failure pattern, as well as in inducing the requisite failure. Similarly, the same table shows that we can generate inputs that *will not* contain the problematic patterns, thus freeing the fuzzer to explore rarer bugs. As our results from Table II show, evocative expressions shine in the role of recognizers of specific patterns in input, and can work in the role of a validator for inputs.

C. Threats to Validity

Our empirical evaluation is subject to the following threats to validity:

External Validity. The external validity (generalizability) of our results depends on the representativeness of our data set. Our subjects were a small set of language interpreters and two UNIX utilities. Further, only a few bugs were evaluated, and their patterns chosen for experiments. Hence, it is possible that our subjects are not representative of the real world. The main mitigation here is that these were actual bugs found in the wild, logged by real people. Further, the grammars are from popular and complex real world input and programming languages.

Internal Validity. The threat to internal validity (correctness of evaluation) of our results is mitigated by careful verification (including formal proofs) of our algorithms.

Construct Validity. To mitigate the threat to construct validity (the degree to which a test measures what it claims to be measure of), we measure how accurate our specialized grammars are as recognizers and generators, which are the main ways we expect practitioners to use our tool.

VIII. Related Work

A. Grammar-Based Testing

Already in 1970, Hanford suggested the use of *grammars* to systematically produce valid inputs [7]. Today's approaches [19] combine specification-based producers with symbolic constraint solving [8], reusing fragments from bug reports [11], search-based strategies [9], [10], grammar coverage [12], [13], annotations [14], search heuristics [15], power schedules [16], and local input generators [17]; search-based grammar testing is now also applied for security testing [18]. All these tools and approaches can immediately benefit from specialized grammars as produced by Evogram, targeting specific input subsets or the locations that process them.

B. Patterns and Pattern Languages

In Software Engineering, *patterns* have been frequently used to obtain specific search and configuration results. Besides the ubiquitous *regular expressions* and *wildcards,* a number of approaches allow to express *structural* properties.

Combi [27] shows how patterns based on a base context free grammar can be used for identifying and transforming structures in code. *Semgrep* [28] by *r2c* is another tool that allows programmers to specify patterns similar to evocative patterns. Note, though, that both are for *recognition*, which is a simpler problem than generation, as one can run the matching rules as as sequence of filters. We note that while the patterns used by Combi are similar to the evocative patterns, they are meant to be specified by hand only. Combi and Semgrep allow conjunction and disjunction of patterns, but algebras and negations are not supported.

Micro-grammars [30] is another general approach for partial parsing and semantic analysis.

Another related work is Coccinelle [29] that defines a pattern language for transformations of C programs such as the Linux kernel. While Coccinelle supports disjunction of patterns, conjunction and negation are not supported.

Conjunctive and Boolean grammars [31] are a generalization over context-free grammars, and specify grammars based on conjunction, (implicit) disjunction and negation of arbitrary context-free grammars. While one can reuse context-free grammar parsers (by applying one grammar after another), it is not clear whether efficient producers exist for such languages.

C. Generalizers

Generalizers are tools that turn a concrete input into a more general pattern with similar effect. DDSET [25] generates failure patterns from any given failure inducing input and the corresponding predicate. *SmartCheck* from Lee Pike [32] generalizes algebraic data structures. *Extrapolate* by Braquehais et al. [33], generalizes counter examples of functional test properties. Groce et al. [34] show one can identify canonical *minimal tests* with annotations for generalizations. In our context, the generalized patterns all serve as starting point for *evocative patterns*. None of these tools, would produce Boolean combinations of patterns.

IX. Conclusion and Future Work

With Evogram, we have introduced the concept of a *grammar transformer,* specializing a grammar towards Boolean combinations of pattern occurrences. Such transformations give testers unprecedented control over the test inputs to be produced. The transformations are proven to be correct; an evaluation shows the usefulness of transformed grammars for producing and checking patterns.

Our future work will focus on the following:

Applications. The lion's share of our future work will focus on further usage scenarios hinted at in Section VI, from parsing over pattern matching to configuration and adaptation.

Richer specifications. We have described how to insert at least one failure-inducing fragment into the input (or not). What if one wants a particular *number* of fragments (say, exactly one)? What if one wants *sequences* of fragments (one pattern should always *follow* another one)? Such constraints can still be represented in a context-free grammar, and we will be investigating specific transformations for them.

Constrained grammars. Some input properties cannot be easily expressed in a context-free grammar; *numerical properties* is a typical example. We want to attach (Turing-complete) constraints to individual patterns and then use transformation rules together with constraint solving to produce inputs that satisfy structural as well as other constraints.

Ambiguous grammars. Our algebra is restricted to non-ambiguous context-free base grammars. However, it may be extended to *all* context-free grammars by taking into account the fact that there could be multiple parse trees for an evocative pattern.

We provide a well explained Jupyter notbook[7] that details our technique as well as a partial mechanization of proofs (using HOL4) for verification[8]. The complete replication package is available as a virtual machine at

DOI:10.5281/zenodo.4456296

References

[1] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.83.445

[2] M. Höschele and A. Zeller, "Mining input grammars from dynamic taints," in *IEEE/ACM Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 720–725. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970321

[3] R. Gopinath, B. Mathis, and A. Zeller, "Mining input grammars from dynamic control flow," in *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020. [Online]. Available: https://publications.cispa.saarland/3101/

[4] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 197–208. [Online]. Available: https://doi.org/10.1145/2491956.2462173

[5] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing." New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2338965.2336763

[6] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with Zest," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 329–340. [Online]. Available: https://doi.org/10.1145/3339069

[7] K. V. Hanford, "Automatic generation of test cases," *IBM Systems Journal*, vol. 9, no. 4, pp. 242–257, Dec. 1970. [Online]. Available: https://doi.org/10.1147/sj.94.0242

[8] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 206–215. [Online]. Available: https://doi.org/10.1145/1375581.1375607

[9] F. M. Kifetew, R. Tiella, and P. Tonella, "Combining stochastic grammars and genetic programming for coverage testing at the system level," in *International Symposium on Search Based Software Engineering*, 2014, pp. 138–152. [Online]. Available: https://doi.org/10.1007/978-3-319-09940-8_10

[10] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 742–762, 2010. [Online]. Available: https://doi.org/10.1109/TSE.2009.52

[11] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments." in *USENIX Conference on Security Symposium*, 2012, pp. 445–458. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[12] R. Lämmel, "Grammar testing," in *Fundamental Approaches to Software Engineering (FASE)*, H. Hussmann, Ed., 2001, pp. 201–216. [Online]. Available: https://doi.org/10.1007/3-540-45314-8_15

[13] N. Havrikov and A. Zeller, "Systematically covering input structure," 2019, pp. 189–199. [Online]. Available: https://doi.org/10.1109/ASE.2019.00027

[7] https://nbviewer.jupyter.org/github/vrthra/Ewoks/blob/master/src/FAlgebra.ipynb

[8] https://github.com/vrthra/Ewoks/tree/master/mechanization

[14] F. M. Kifetew, R. Tiella, and P. Tonella, "Generating valid grammar-based test inputs by means of genetic programming and annotated grammars," *Empirical Software Engineering*, vol. 22, no. 2, pp. 928–961, 2017. [Online]. Available: https://doi.org/10.1007/s10664-015-9422-4

[15] R. Hodován, Á. Kiss, and T. Gyimóthy, "Grammarinator: a grammar-based open source fuzzer," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 45–48. [Online]. Available: https://doi.org/10.1145/3278186.3278193

[16] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, to appear; preprint at https://arxiv.org/abs/1811.09447.

[17] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with Zest," 2019, pp. 329–340. [Online]. Available: https://doi.org/10.1145/3293882.3330576

[18] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "NAUTILUS: Fishing for deep bugs with grammars." in *NDSS*, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[19] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "The fuzzing book," in *The Fuzzing Book*. Saarland University, 2019, retrieved 2019-09-09 16:42:54+02:00. [Online]. Available: https://www.fuzzingbook.org/

[20] R. Gopinath and A. Zeller, "Building fast fuzzers," 2019.

[21] J. Earley, "An efficient context-free parsing algorithm," *Communications of the ACM*, vol. 26, no. 1, pp. 57–61, 1983.

[22] M. Tomita, *Efficient parsing for natural language: a fast algorithm for practical systems*. Springer Science & Business Media, 2013, vol. 8.

[23] D. Grune and C. J. Jacobs, "Parsing techniques," *Monographs in Computer Science. Springer,*, p. 13, 2007.

[24] E. Scott and A. Johnstone, "Gll parsing," *Electronic Notes in Theoretical Computer Science*, vol. 253, no. 7, pp. 177–189, 2010.

[25] R. Gopinath, A. Kampmann, N. Havrikov, E. Soremekun, and A. Zeller, "Abstracting failure-inducing inputs," in *Proceedings of the 2020 International Symposium on Software Testing and Analysis*. ACM, 2020.

[26] "HOL4," https://hol-theorem-prover.org/. [Online]. Available: https://hol-theorem-prover.org/

[27] R. van Tonder and C. Le Goues, "Lightweight multi-language syntax transformation with parser parser combinators," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 363–378. [Online]. Available: https://doi.org/10.1145/3314221.3314589

[28] r2c, "Lightweight static analysis for many languages. find and block bug variants with rules that look like source code." https://semgrep.dev/, 2020, accessed: 2020-08-19.

[29] Y. Padioleau, J. L. Lawall, and G. Muller, "Understanding collateral evolution in Linux device drivers," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, 2006, pp. 59–71. [Online]. Available: https://doi.org/10.1145/1217935.1217942

[30] F. Brown, A. Nötzli, and D. Engler, "How to build static checking systems using orders of magnitude less code," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 143–157.

[31] A. Okhotin, "Conjunctive and Boolean grammars: The true general case of the context-free grammars," *Computer Science Review*, vol. 9, pp. 27 – 59, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S157401371300018X

[32] L. Pike, "SmartCheck: automatic and efficient counterexample reduction and generalization," in *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell*, 2014, pp. 53–64. [Online]. Available: https://doi.org/10.1145/2633357.2633365

[33] R. Braquehais and C. Runciman, "Extrapolate: generalizing counterexamples of functional test properties," in *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages*, 2017, pp. 1–11. [Online]. Available: https://doi.org/10.1145/3205368.3205371

[34] A. Groce, J. Holmes, and K. Kellar, "One test to rule them all," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 1–11. [Online]. Available: https://doi.org/10.1145/3092703.3092704