Detecting Ambiguity in Programming Language Grammars

Naveneetha Vasudevan and Laurence Tratt

Software Development Team, King's College London
http://soft-dev.org/
{naveneetha@yahoo.com,laurie@tratt.net}

Abstract. Ambiguous Context Free Grammars (CFGs) are problematic for programming languages, as they allow inputs to be parsed in more than one way. In this paper, we introduce a simple non-deterministic search-based approach to ambiguity detection which non-exhaustively explores a grammar in breadth for ambiguity. We also introduce two new techniques for generating random grammars – Boltzmann sampling and grammar mutation – allowing us to test ambiguity detection tools on much larger corpuses than previously possible. Our experiments show that our breadth-based approach to ambiguity detection performs as well as, and generally better, than extant tools.

1 Introduction

Context Free Grammars (CFGs) are widely used for describing formal languages, including Programming Languages (PLs). The full class of CFGs (grammars from now on) includes ambiguous grammars—those which can parse inputs in more than one way. Needless to say, ambiguous grammars are highly undesirable. If an input can be parsed in more than one way, which one of those parses should be taken? We would not enjoy using a compiler if it were to continually ask us to choose which parse we want. Unfortunately, we know that, in general, it is undecidable as to whether a given grammar is ambiguous or not [7]. While there are various parsing approaches which allow a user to manually disambiguate amongst multiple parses, one can not in general know if all possible points of ambiguity have been covered. Perhaps because of this, most tools use parsing algorithms such as LL and LR, which limit themselves to parsing only a subset of unambiguous grammars. This leads to other trade-offs: grammars have to be contorted to fit them within these subsets; and these subsets rule out the ability to compose grammars [14].

As a consequence, there has been a steady stream of work trying to detect ambiguity in arbitrary grammars, in order to bring most of the benefits of the full class of CFGs without the disadvantages. Exhaustive methods such as AMBER [12] systematically generate strings to uncover ambiguity, but for even medium-sized grammars, this quickly leads to infinite state spaces. Approximation techniques, on the other hand, sacrifice accuracy for termination. ACLA [5] transforms a language to an alternative whose accepted inputs are a

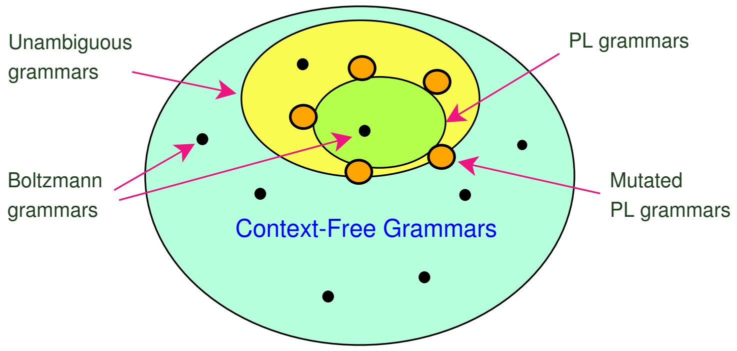

Fig. 1. An intuition about the relation between various classes of CFGs.

superset of the original; it never reports false negatives, but may report false positives. Hybrid approaches marry approximation techniques with exhaustive search. Basten's hybrid approach [4] first applies a noncanonical unambiguity test to filter out provably unambiguous portions of a grammar before running AMBER on the result. However, such hybrid approaches still rely on an exhaustive search, although on a smaller state space. Bounded length approaches are in a sense a subset of exhaustive methods: they exhaustively explore a small, fixed part of the search space. CFGAnalyzer [1] uses a SAT solver to explore strings of bounded length. Cheung and Uzgalis' method [8] deterministically expands rules from the start terminal until a fixed bound is reached.

Whereas previous ambiguity detection approaches are deterministic and explore a grammar in 'depth', our hypothesis is that approaches which explore a grammar in 'breadth' have a greater chance of discovering ambiguity. By depth we mean that a subset of the grammar is explored in (possibly exhaustive) detail; by breadth that a large portion of the grammar is explored but not exhaustively so. In other words, we suspect that a scatter-gun approach to detecting ambiguity will be more successful than a focused beam.

To that end, we have created a tool *SinBAD* which houses a number of ambiguity detection approaches. This paper details one of SinBAD's non-deterministic ambiguity detection algorithms which is intended to explore a grammar in breadth rather than depth. The algorithm is extremely simple, with its core explained in less than a page. Despite the simplicity of the algorithm, experimental results show that it performs at least as well as, and generally better than, more complex deterministic approaches. Furthermore, good results are found more quickly than by previous approaches.

Understanding the relation between grammars, and its various subsets is key to understanding the motivation for, and the results of, our work. Figure 1 is our attempt to give an intuition about these relations. Since all the sets involved are infinite, this diagram is necessarily an approximation, but is hopefully helpful. The set of unambiguous grammars is a strict subset of the grammars. Virtually all PL grammars reside within this unambiguous subset. Our underlying hypothesis is that PL grammars often stretch to the very edge of the class of unambiguous grammars. Stated differently, we suspect that PL grammars are often only a small step away from being ambiguous.

This paper also provides new techniques for evaluating the effectiveness of ambiguity detection tools. We believe that evaluating such tools requires much larger input corpuses than previously used: ours contains over 20,000 grammars of various types. In order to generate such a large corpus, we cannot rely on hand-written grammars. We therefore provide two classes of random grammars. The first is generated using Boltzmann sampling, an approach which provides some statistical guarantees about the randomness of the resulting generators. The second class is generated by mutating existing PL grammars. This latter category is particularly interesting as we, like most others working in this field, are particularly interested in the ambiguity of PL-like grammars. There is an inevitable problem with this: most PLs are written for approaches such as LR parsing that accept only unambiguous grammars. Basten hand-modified 20 PL grammars to be ambiguous [2] which we reuse in our suite for comparison purposes. However, one can easily, and inadvertently, create a solution which works well for such a small corpus but little beyond it. By generating a huge number of possibly ambiguous PL-like grammars, we can explore a much wider set of possibilities than is practical by hand.

To summarise, our work has two hypotheses:

H1 Covering a grammar in breadth is more likely to uncover ambiguity than covering it in depth.

H2 PL grammars are only a small step away from being ambiguous.

The contributions of this paper are as follows. First, we show a new search-based approach to ambiguity detection, which is simpler than previous approaches. Second, we provide new means of evaluating the effectiveness of ambiguity tools by providing the ability to produce large quantities of grammars using Boltzmann sampling and grammar mutation. Third, we provide the first large-scale evaluation of such tools. In so doing, we show that our simple search-based approach performs at least as well as, and generally better than, existing tools. The basic idea of our search-based approach was first presented in a workshop paper [15]. This paper extends that with Boltzmann sampled grammars, grammar mutation, and a significantly larger experiment. SinBAD, the grammar generators, the grammar corpus we used, and the results obtained can be downloaded from our experimental suite:

`http://figshare.com/articles/cfg_amb_experiment/774614`

The structure of this paper is as follows. In Section 3 we describe our search-based approach to ambiguity detection. Sections 4 and 5 describe our algorithms for generating Boltzmann and mutated grammars respectively. In Section 6 we set out the methodology for our experiment, which is split into 3 sub-experiments. In section 10 we consider our hypotheses in the light of our experimental results.

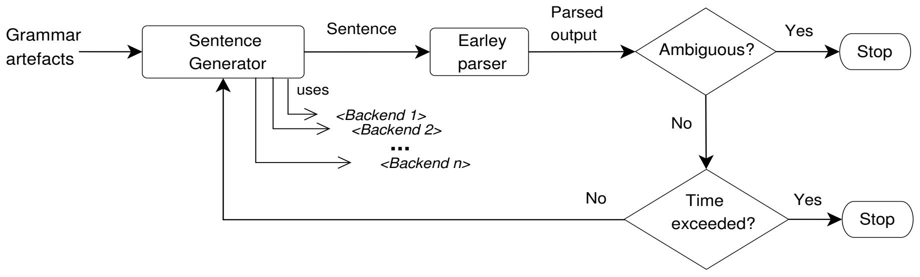

Fig. 2. SinBAD architecture.

2 Definitions

Before presenting our algorithms and descriptions, we first introduce some brief definitions (mostly standard) and notations.

A grammar is a tuple $G = \langle N,T,P,S \rangle$ where N is the set of non-terminals, T is the set of terminals, P is the set of production rules over $N \times (N \cup T)^*$ and S is the start non-terminal of the grammar. A production rule $A\colon \alpha$ is denoted as $P[A]$ where $A \in N$, and α is a sequence of strings drawn from $(N \cup T)^*$. N_ϵ denotes set containing non-terminals that have at least one empty alternative. For a rule $P[A]$, $P[A]_{alt}$ denotes a single alternative and $\Sigma P[A]_{alt}$ all its alternatives. We define the size of a grammar as $size(G) = |N|$. We define a sentence of a grammar as a string over T^*. A sentence is ambiguous if it can be parsed in more than one way. A grammar is ambiguous if there exists a sentence which is both accepted and ambiguous. We define 'symbol' to mean either a terminal or a non-terminal.

For a list ℓ, let $\ell[i]$ denote the element at position i, and $\ell[i{:}j]$ the items from positions i (inclusive) to j (exclusive). Let $insert(\ell, i, \alpha)$ denote insertion of α into ℓ at position i and $delete(\ell, i)$ denote deletion of element from ℓ from position i. Let $append(\ell, \alpha)$ denote appending element α to list ℓ. For a dictionary $\mathcal{D}$ containing key-value pairs, $\mathcal{D}[x \mapsto a]$ denotes an update to key x with value a, $\mathcal{D}[x]$ denotes a lookup of key x, $\mathcal{D}_{values}$ denotes its list of values. Let $\mathcal{R}(\ell,n)$ denote a list of n items chosen randomly from the list ℓ. Further, let $\mathcal{R}[m..n]$ denote a number chosen randomly between m (inclusive) and n (inclusive).

3 Search-based ambiguity detection

Search-based techniques seek to find 'good enough' solutions for problems that have no feasible algorithmic solution and whose search space is too big to exhaustively scan. Such techniques have been applied to a wide range of problems including software itself (see e.g. [9]). Search-based techniques are either random or guided by a fitness function.

In order to apply search-based techniques to ambiguity detection, we created *SinBAD*, a simple tool with pluggable backends. Figure 2 shows SinBAD's architecture. Given a grammar and a lexer, the *Sentence Generator* component generates random sentences using a given *backend*. A backend, in essence, is an algorithm that governs how sentences are generated. For instance, a backend can use a unique scoring mechanism to favour an alternative when expanding a non-terminal, or one that can generate sentences of bounded length. The generated sentence is then fed to a parser to check for ambiguity (we use ACCENT [11], a fast Earley parser for this). The search stops when an ambiguity is found or when a time limit is exceeded. In this paper we consider the most successful SinBAD backend we have created so far: *dynamic1*[1].

3.1 The *dynamic1* backend

Given a grammar, the *dynamic1* backend shown in Algorithm 1 non-deterministically creates a valid sentence which can then be used to test for ambiguity. In essence, the algorithm continually picks random alternatives to follow for sentence generation, recursing into the grammar. However, doing this naively leads to frequent non-termination [15]. Therefore, the backend is parameterised by a user-configurable integer D[2]. Once the algorithm has recursed beyond depth D, it favours alternatives which immediately terminate (i.e. rules that contain no non-terminals). When this is not possible – some rule's alternatives all contain non-terminals – the favouritism then chooses whichever rules have been least visited. In this way, the generator tends not only to terminate in reasonable time, but also to explore a grammar's rules semi-uniformly.

The function START is initialised with a user-defined grammar G and threshold depth D. The current depth d is set to zero. We initiate sentence generation by deriving the start symbol S of the grammar. We keep a note of when we have entered a rule and when we have exited. To derive a non-terminal, we randomly select one of its alternatives (line 11). When the depth of the recursion exceeds a certain threshold depth, we start favouring alternatives (lines 8 and 9).

The FAVOUR-ALTERNATIVE function is called when the algorithm wishes to try and terminate. Given a rule, the function generates a score for each alternative and the one with the lowest score is selected. In the event of a tie, one of the lowest scoring alternatives is arbitrarily selected. Terminals are scored as 0. Non-terminals are scored as a ratio of the number of derivations that haven't been fully derived yet to the total number of derivations (line 30).

dynamic1's simplicity means that our experimental corpus has uncovered a handful of cases (1.4% of grammars in the corpus) where it doesn't terminate. This is due, in an unintended irony, to the one deterministic part of *dynamic1*: the favouring of alternatives. When FAVOUR-ALTERNATIVE is called, it scores rules, selects those with the equal lowest score, and then non-deterministically picks amongst them. If one alternative always has the lowest score, then it will be picked every time. Consider the rules `P: Q` and `Q: P | R S`. If, at a given point of time, the scores for the first and second alternatives

[1] For a discussion of some other backends, see [15].

[2] Setting D to ∞ provides equivalent behaviour to the naive non-terminating approach.

Algorithm 1 The *dynamic1* algorithm

```
function START($G$, $D$)
    $Sen \leftarrow \varnothing$
    GENERATE($P[S]$, $G$, $Sen$, $d = 0$, $D$)
    return $Sen$
end function

function GENERATE($P[A]$, $G$, $Sen$, $d$, $D$)
    $P[A].entered \leftarrow P[A].entered + 1$
    if $d \geq D$ then
        $P[A]_{alt} \leftarrow$ FAVOUR-ALTERNATIVE($P[A]$, $G$)
    else
        $P[A]_{alt} \leftarrow \mathcal{R}(\Sigma P[A]_{alt}, 1)$
    end if
    for $Sym \in P[A]_{alt}$ do
        if $Sym \in N$ then
            $Sen \leftarrow Sen +$ GENERATE($P[Sym]$, $G$, $Sen$, $d + 1$, $D$)
        else
            $Sen \leftarrow Sen + Sym$
        end if
    end for
    $P[A].exited \leftarrow P[A].exited + 1$
    $d \leftarrow d - 1$
end function

function FAVOUR-ALTERNATIVE($P[A], G$)
    $scores \leftarrow \{\ \}$
    $scores \leftarrow \{scores[alt \mapsto 0] \mid alt \in \Sigma P[A]_{alt}\}$
    for $P[A]_{alt} \in \Sigma P[A]_{alt}$ do
        for $Sym \in P[A]_{alt}$ do
            if $Sym \in N$ then
                if $P[Sym].entered > 0$ then
                    $score_{alt} \leftarrow score_{alt} + (1 - (P[Sym].exited / P[Sym].entered))$
                end if
            end if
        end for
        $scores \leftarrow scores[P[A]_{alt} \mapsto score_{alt}]$
    end for
    $alts_{min} \leftarrow \{alt \in \Sigma P[A]_{alt} \mid scores[alt] = min(scores_{values})\}$
    return $\mathcal{R}(alts_{min}, 1)$
end function
```

of rule `Q` are <1 and >1 respectively, then the alternative favouring will always select the first alternative, as it has the lowest score. We briefly outline a possible solution for this in Section 12.

```
Cfg = Cfg Rule ... Rule
Rule = SingleAlt Alt | RuleAlts1 Rule Alt
Alt = EmptyAltSyms | SingleAltSyms1 Symbol | AltSyms1 Alt Symbol
Symbol = NonTerm NonTerm | Term Term
NonTerm = NonTerm1 | NonTerm2 | ... | NonTermN
Term = Term1 | Term2 | ... | TermN
```

Fig. 3. Tree specification for generating grammars.

4 Boltzmann sampled grammars

Boltzmann sampling is a framework for random generation of combinatorial structures (see [6] for further details). The basic idea is to give the sampler a class specification of a combinatorial structure and a value to control the size of the generated objects. For a given class C, and size n, the sampler provides approximate-size uniform random generation—objects are generated with approximate size n$\pm\epsilon$, where ϵ is a fixed tolerance, but objects of the same size occur with equal probability. This allows the sampler to generate large objects in linear time. In this section we provide the first Boltzmann sampler for grammars.

4.1 Class specification

A Boltzmann sampler class specification is a grammar containing a set of productions. A production is of the form: `A:` ⟨`rhs`⟩, where `A` is the name of the class being defined and ⟨`rhs`⟩ is a set of definitions. A definition is of the form `DefX Y`, where *DefX* denotes a constructor and Y is either a reference to a definition (if a definition `Y` exists) or a literal otherwise.

Since, as far as we are aware, this is the first time that Boltzmann sampling has been used to generate grammars, we were forced to create a class specification ourselves. Determining a good class specification is arguably the hardest part of Boltzmann sampling, and is complicated by the fact that grammars do not have a single, obvious specification. Furthermore, since grammars are unbounded in size, we necessarily have to restrict the size of the those generated to make using them practical. This immediately leads us to a difficult question: what style of grammars do we want? In reality, we are most interested in grammars which somewhat resemble PL grammars. Generating grammars with 2 rules containing 100 alternatives each may tell us something about grammars in general – though getting enough coverage to say something useful may be much harder – but little about programming languages. We have therefore crafted our use of Boltzmann sampling to lead to grammars which roughly resemble real PLs. In order to do this, we are forced to apply post-filters to restrict the grammars generated to those we are most interested in, as we shall soon see.

Our class specification is shown in Figure 3. Using [10] as a guiding principle, our specification is designed to give us control over three things: the number of empty alternatives, the number of alternatives per rule, and the number of symbols per alternative. `Cfg` denotes a context-free grammar, `Rule` a production rule, `Alt` a production alternative, and `Symbol` denotes either a non-terminal (a `NonTerm`) or a terminal (a `Term`) symbol. A CFG consists of 1 or more production rules (hence the references to multiple Rule definitions). `Rule` has two outcomes: it can either be called recursively to build a list of alternatives; or just build a list with single alternative. `Alt` has three choices: it can either be called recursively to build a sequence of symbols; or just build a sequence with one symbol (middle choice); or an empty string (`EmptyAltSyms`). The specification enforces equal numbers of `NonTerms` and `Terms` in a grammar, the 1:1 ratio seeming to us a reasonable heuristic based on our observations of real grammars.

While we do not claim that our specification is perfect, it is the result of considerable experimentation and the resulting grammars are close to those we might expect to see for PLs. Minor variations to the specification can lead to significantly differing "styles" of grammars being generated. For instance: replacing `SingleAlt Alt` by `EmptyAlt` would cause a much higher percentage of empty alternatives to be generated.

4.2 Precision

A Boltzmann sampler is parameterised by two values that control the size of the generated objects: singular precision and value precision. To get an efficient sampler, these two values need to be set as low as possible [6]. However, the lower these values are, the greater the likelihood of large objects being generated. This is a problem for us, as "large" means rules would have more alternatives and symbols per alternative than we desire. The challenge, then, is to find values that generate large numbers of relevant grammars in reasonable time. We settled on values of $1.0e^{-7}$ and $1.0\text{-}e^{-4}$ for the singular and value precisions respectively.

4.3 Grammar generation and filtering

Our Boltzmann class specification gets us in the rough neighbourhood of PL grammars, but some obvious differences remain. We also struggled to generate grammars of all sizes that we wished for.

The sampler struggled to generate grammars when we restricted the number of symbols per alternative to 5, so we relaxed this criterion. Approximately 10–15% of alternatives from each grammar generated by the sampler have more than 5 symbols per alternative.

Similarly, the sampler tends to generate a much larger number of empty alternatives than are typical of PL grammars. Using Basten's PL grammar corpus as an example, the proportion of empty alternatives varied between 4% (Java) to 12% (Pascal). We therefore wrote a filter to remove all grammars that had a proportion of empty alternatives above 5%. Such filters are needed if one wishes to generate PL-like grammars.

Because the sampler is unaware of the precise semantics of grammars, it can and does produce grammars which are non-sensical or trivially ambiguous. We filter out all grammars which contain non-terminating cycles of the form `A: B` and `B: A` as they consume no input and generate the empty language. We also filter out grammars which contain alternatives with the same sequence of symbols (e.g. `A: X | X | ...`) which are trivially ambiguous.

We wanted to generate grammars of size ranging from 10 to 50 inclusive. However, the sampler was unable to generate any grammars for sizes 16, 20, 25, 26, 29, 32, 40, 42, and 49. This can be solved by making the precision greater than 0.005, but this causes other issues (see Section 4.2), so we did not do so.

5 Mutated grammars

Random grammar generators have one major problem from our perspective: even if they produce grammars in the general style of those used by PLs, it can be reasonably argued that they are never close enough. Of course, exactly what *is* close enough is impossible to pinpoint: it seems unlikely that any metric, or set of metrics, can reliably classify PL vs. non-PL grammars. Instead, we have little choice but to fall back on the intuitive notion that “we know one when we see one.” This means that past work has struggled to understand how ambiguity affects PL-like grammars: we simply can’t get hold of enough of them to perform adequate studies. The best attempt of which we are aware is the work of Basten, who took 20 unambiguous PL grammars and manually altered them to introduce ambiguity [3]. Manually altering grammars is tedious, hard to scale, and always open to the possibilities of unintentional human bias.

We have therefore devised a simple way of generating arbitrary numbers of ‘PL-like’ grammars with possible ambiguity. Our approach to grammar mutation bears no relation to grammar evolution or grammar recovery. Instead, our basic tactic is inspired by Basten’s manual modifications: we take in a real (unambiguous) grammar for a PL and perform a single random alteration to a single rule. Although there are numerous possible mutations, we restrict ourselves to the following four, each of which is applied to a single rule:

Add empty alternative This is only possible if a rule does not already have an empty alternative.

Mutate symbol Randomly select a symbol from an alternative and change it. A non-terminal can be replaced by a terminal and vice versa.

Add symbol Randomly pick an alternative and add a symbol at a random place within it.

Delete symbol Randomly delete a symbol from an alternative. Only non-empty alternatives are considered.

Our mutated grammars are therefore identical to a real PL grammar, with only a single change. This is the best way that we can imagine of solving the “we know it when we see it” problem. As we will see later, these simple mutations introduce a surprising number of ambiguities. The full algorithm is presented in Appendix A.

6 Experiment methodology

The objective of our experiment is to understand how well search-based approaches perform in detecting ambiguities. Since ambiguity is inherently undecidable, it is impossible to evaluate such a tool in an absolute sense. Instead, we evaluated our tool against three others: ACLA, AMBER, and AmbiDexter [3]. Each tool takes a different approach: ACLA uses an approximation technique; AMBER uses an exhaustive search; AmbiDexter uses a hybrid approach; and SinBAD uses a random search-based approach.

All the tools except ACLA have run-time options which adjust the way they operate and thus affect which ambiguities they find. We believe the fairest comparison is between the tools at their best and that we need to use the "best" run-time option values possible. However, discovering what the best options are by trying all possibilities on our full set of grammars is prohibitively expensive. Instead, we first run a "mini" experiment on a small set of grammars to determine good tool options. We do not claim that the option values discovered necessarily allow each tool to operate at its maximum potential; rather, we believe that they allow the tool to operate close enough to its maximum potential to make a meaningful comparison.

Using the run-time options determined by the mini experiment, we then run the "main" experiment on a larger set of grammars (about 7 times bigger) with each tool. Finally, we check that the proportion of grammars discovered as ambiguous scales up, by running a "validation" experiment using only *dynamic1* on a larger set of grammars again (about 5 times bigger than the main experiment).

Since grammars can specify infinite languages, grammar ambiguity tools can run forever. We are therefore also interested in how long it takes each tool to give quality results. For the mini and main experiments, we therefore run each tool for 10, 30, 60, and 120 seconds, enforcing the limit with the `timeout` tool. For AMBER the parser generation time is not included in the limit, whereas for SinBAD it is (as we were unable to break the two apart). Since this time is rather small (0.4s), we believe it does not unduly colour the results.

We evaluated the various tools on three different sets of grammars: Boltzmann sampled, altered PL grammars, and mutated grammars. Boltzmann sampled grammars were described in Section 4. Basten's altered PL grammars are taken from [4], where Pascal, SQL, Java, and C grammars were manually modified to produce 5 ambiguous variations of each. The mutated grammars were described in Section 5. Table 1 shows the size of the grammar sets used in each experiment. For the Boltzmann sampled grammars, each size (10-50) is represented equally (i.e. for the main experiment, 50 grammars of each size are used). Similarly, for the mutated grammars, each mutation category (add empty alternative, mutate symbol, add symbol, and delete symbol) is equally represented (e.g. for the main experiment, 500 grammars from each category are used). Note that we are not worried about differences in the ambiguous fragments identified: we care only whether a tool uncovers ambiguity in a grammar or not.

All experiments were performed on a cluster of identical Intel i7-2600 CPU 3.4GHz machines with 8GiB memory. For the mini experiment, where perfect

	Mini	Main	Validation
Boltzmann	384	1600	9600
Altered PL	20	20	20
Mutated	160	2000	11200
Total	564	3620	20820

Table 1. The number of grammars used in the various experiments.

Tool	Option	Values
AMBER	Search by length	5, 10, 15, 20, 25, 50, 100
	Search by example	10^{10}, 10^{20}, 10^{30}
	Ellipsis	Yes / No
AmbiDexter	From 0 to N	5, 10, 15, 20, 25, 50, 100
	From N to ∞	0
	Filter	None, LR0, SLR1, LALR1, LR1
dynamic1	Depth	5, 6, 7, ..., 30

Table 2. Options tried in the mini experiment.

precision was not necessary, we used 8 cores (4 real and 4 hyperthreading) per machine. For the main and validation experiments, where precision is important, we disabled hyperthreading and restricted ourselves to utilising 3 cores per machine. We used `parallel`[3] to parallelise our experiment. The experiments took around 3400 core-hours in total, broken down into: 600 hours for the mini experiment; 2000 for the main experiment; and 800 for the validation experiment.

Our experimental setup is fully repeatable and is available through our downloadable experimental suite.

7 Mini experiment

In the mini experiment, we wish to uncover what reasonable values for various options are. ACLA has no options, so does not to be considered further. The options and their values tried for the other tools are outlined in Table 2.

AMBER can search either by length (sentences up to a fixed length) or by example (search limited by number of sentences with no restriction on sentence length). The 'ellipsis' option causes non-terminals to be treated as tokens, which increases the chances of finding long ambiguous fragments. We found that in most cases, turning on the 'ellipsis' option led to better results: 22 with it set vs. 18 without. Only for the 'add empty alternative' variant of mutated grammars did the ellipsis option perform worse.

[3] http://www.gnu.org/software/parallel

Grammar set	ACLA	AMBER[a]	AmbiDexter[b]	*dynamic1*[c]
Boltzmann	n/a	ell+N=10^{10}	ik+unf	D=11
Altered PL	n/a	ell+len=10	k=15+LR0	D=9
Mutated	n/a	len=15	k=15+SLR1	D=17

[a] ell, len, N $\triangleq$ AMBER options *ellipsis*, *length* and *examples* respectively.
[b] ik, k, unf $\triangleq$ AmbiDexter options incremental length, maximum length of sentences to check, unfiltered version of a grammar respectively.
[c] D $\triangleq$ Threshold depth for *dynamic1*.

Table 3. Best performing options for each tool.

AmbiDexter has two modes of sentence generation: searching for sentences up to length N, or searching for sentences from a starting length N to ∞. AmbiDexter also supports filters that can identify and remove provably unambiguous subsets of a grammar. These filters are of varying power: LR0 (low) to LR1 (high). The more powerful a filter is, the greater the portion of a grammar it can filter out, but the longer it takes to do so. We evaluated the tool with both unfiltered and filtered versions of a grammar. Generating a filtered version of a grammar is included in the time limit.

SinBAD's *dynamic1* backend requires a depth option D to determine when it should attempt to unwind recursion. We evaluated D for values from 5 to 30. For lower values of D, *dynamic1* starts favouring alternatives much earlier, and therefore sentences are short and quick to generate. For higher values of D, *dynamic1* generates longer sentences. In the cases where *dynamic1*'s sentence generator did not terminate, we re-ran it (in such cases, the normal time limit still applied, preventing infinite re-runs).

The values we chose for the mini experiment were based on our experience of using the tools in question, and our need to choose a reasonable subset of options in order to have a tractable experiment. To check that the values we chose were not biased against the tools, we performed a brief sanity check on each of the 'best' values found, checking several of its near neighbours. Only with AMBER was there a measurable difference (when searching by example). Using a value of 10^{10} with the 'search by example' option, 238 Boltzmann sampled grammars were found to be ambiguous; with a value of 10^8, 240 were found to be ambiguous. For the mutated grammars, 10^{10} found 6 ambiguities whereas 10^7 found 7 ambiguities. In both cases, the differences are sufficiently small to make us comfortable with sticking with the original values.

Table 3 lists the best performing options for each tool. All the data involved are available from our downloadable experimental suite.

8 Main experiment

The main experiment is the largest cross ambiguity detection tool experiment to date. All the data involved are available from our downloadable experimental suite.

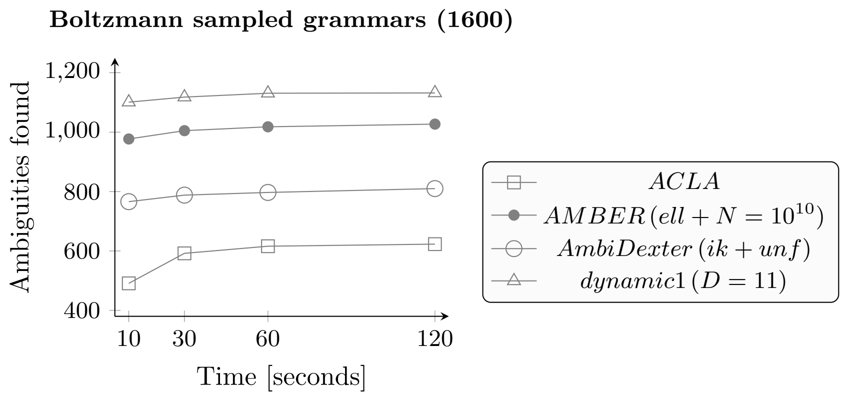

Fig. 4. Number of ambiguities found for Boltzmann sampled grammars.

Figures 4, 5, and 6 show the results of our experiments for each grammar set, for each time limit. In analysing some of the results from the main experiment, we had to perform additional experiments. In most cases, we used grammars from the main experiment. In only one case, for collecting data for sentence and ambiguous fragment length, have we used grammars from the mini experiment.

Our results from the main experiment indicated that three of our grammar sets were highly ambiguous: Boltzmann sampled (70%), the 'add empty alternative' mutated grammars (60%), and 'delete symbol' mutated grammars (45%). Manual observation of ambiguous grammars led to two observations:

Cyclic ambiguity Rules that contain cycles of the form (`A: A | ...`) or (`A: B | ...; B: A | ...`) contribute to cyclic ambiguity [13]. We manually calculated the percentage of cyclically ambiguous grammars to be: 22% (Boltzmann sampled), 0% (Altered PL), and 0.009% (Mutated). This appears to be by far the most common type of ambiguity we encounter.

Multiple ambiguity A grammar has multiple ambiguity if it has more than one ambiguous subset. 36% of Boltzmann grammars contained 2.5 ambiguities per grammar. For mutated grammars the figures are: 37% and 2.8 for 'add empty alternative'; 13% and 3 for 'mutate symbol'; 4% and 2.7 for 'add symbol'; and 23% and 2.6 for 'delete symbol'.

In the rest of this section, we explore what the results mean for each tool (in alphabetical order).

8.1 ACLA

Given a grammar, ACLA will report it to be ambiguous, unambiguous, or possibly ambiguous (that is, it is unsure if the grammar is ambiguous). ACLA's approach to ambiguity detection is based on two linguistic properties: vertical and horizontal ambiguity. Vertical ambiguity means that during the parsing of

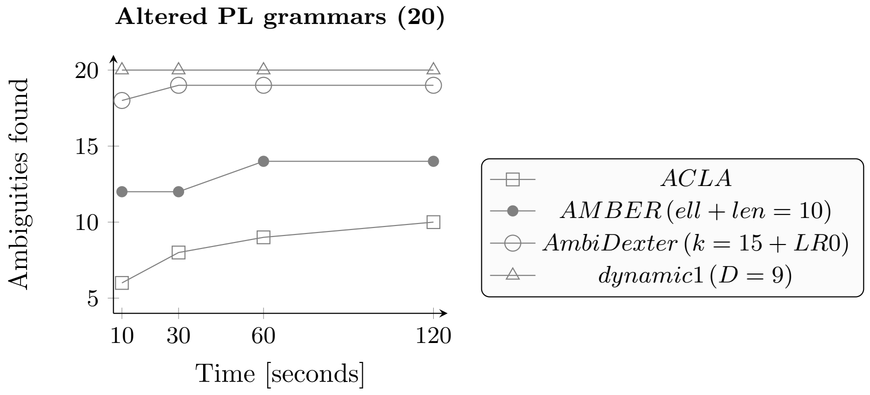

Fig. 5. Number of ambiguities found for altered PL grammars.

	ACLA		AMBER		AmbiDexter		*dynamic1*	
	Sen	Amb	Sen	Amb	Sen	Amb	Sen	Amb
Boltzmann	-	15	58	14	27	21	1554664	2671
Altered PL	-	11	10	9	15	15	281	88
Mutated	-	15	15	6	15	15	4392	502

Table 4. Maximum sentence ('Sen') and ambiguous fragment ('Amb') length detected by each tool using the options from Table 3. Note: we were unable to determine the sentence length for ACLA.

a string, there is a choice between the alternatives of a non-terminal. Horizontal ambiguity means that, when parsing a string according to a production alternative, there is a choice in how the string can be split.

For Boltzmann and mutated grammars, ACLA reported only one or two grammars to be unambiguous. On average across the grammar sets, ACLA was unsure whether 50–60% of the grammars were ambiguous or not. ACLA did not detect 12% of cyclically ambiguous grammars as being ambiguous. ACLA detects ambiguity in a grammar by iterating through each non-terminal, and checking its language for vertically or horizontally ambiguous strings. Although it is not clear what sort of string length it searches for, the length of ambiguous fragments that it detects, on average, ranges between 10 and 15 (see Table 4). In most cases, where the ambiguous subset is deeply nested, ACLA is unsure if the grammar is ambiguous. For most grammar sets, ACLA reaches a point of diminishing returns at 120s. Only in the case of mutated grammars, did our results (see Figure 6) seem to indicate that given additional time, ACLA might uncover further ambiguities. Running ACLA for an extended time limit of 240s only uncovered in 4 additional ambiguities being found.

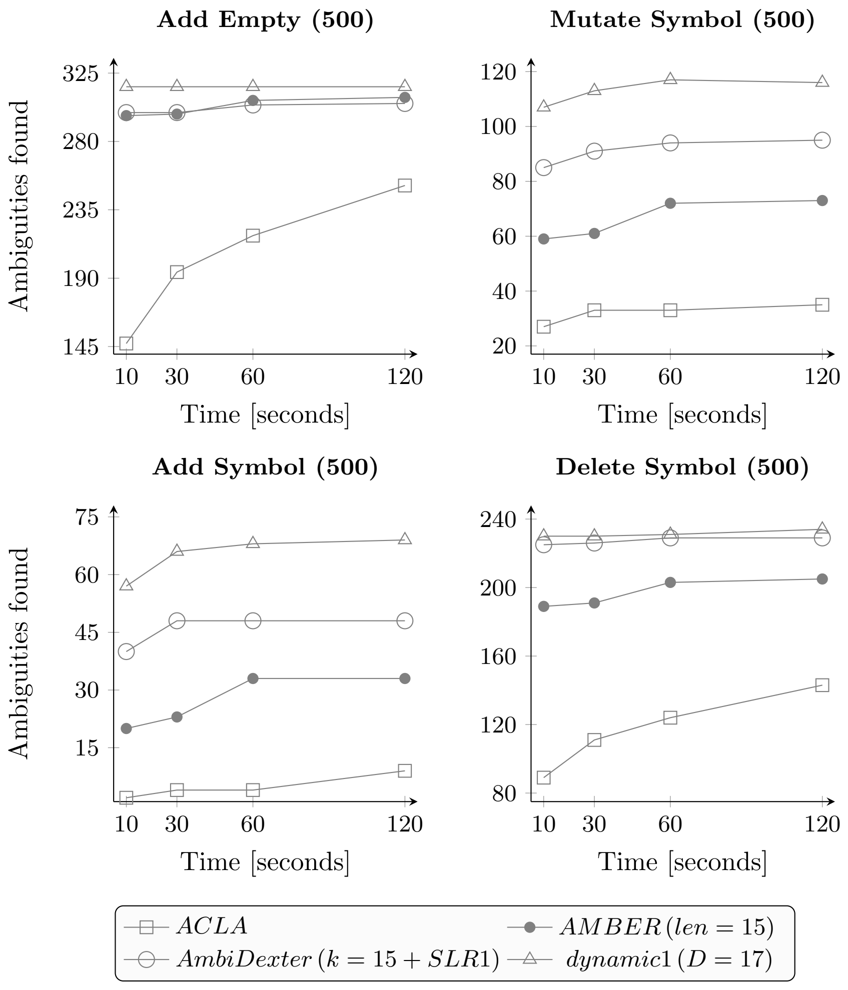

Fig. 6. Number of ambiguities found for mutated grammars.

8.2 AMBER

AMBER performs extremely well on the Boltzmann grammars, but less well on manually altered or mutated grammars. AMBER uses an exhaustive approach to ambiguity detection, whereby it systematically enumerates strings for a given grammar, and checks for ambiguity. There are two possible reasons why AMBER does well on Boltzmann grammars. First, these grammars contain multiple ambiguities, and a relatively high percentage of cyclically ambiguous grammars. AMBER was quick to detect these ambiguities. Second, the ambiguous subsets found are easily reachable, in the sense that they are referenced from very near the start of the grammar. For instance, in the case of Java.1, where the ambiguous subset originates from the rule `compilation_unit`, which is close to the start rule, and AMBER is quick to find it. In the case of Pascal.2, where the ambiguous subset originates from within an expression rule set (`term`) – that is, frequently referenced – AMBER is quick to find it. However, for some of the nested ambiguous subsets (as in Pascal.3, which contains a nested if-else ambiguous subset), AMBER struggles.

8.3 AmbiDexter

AmbiDexter is effective for PL (altered and mutated) grammars, but is less effective for Boltzmann grammars. AmbiDexter does well on PL grammars for two reasons. First, PL grammars contain short ambiguous subsets (see Table 4) and AmbiDexter's exhaustive search, whereby it checks for short strings exhaustively, is quick to find it. Second, its filtering of unambiguous fragments was very effective on PL grammars. For mutated grammars, where SLR1 was the best performing filter, the percentage of rules filtered out were 60% (Pascal), 90% (SQL), and 24% (Java) and 20% (C), whereas for Boltzmann grammars, it was 19%. There was a noticeable difference in (SLR1) filtering time between mutated (1.3s) and Boltzmann grammars (0.7s). Since AmbiDexter uses an exhaustive approach, it struggles when the ambiguous subsets are long and deeply nested.

8.4 *dynamic1*

As Table 4 indicates, compared to other tools, *dynamic1* generates much longer sentences, and therefore, it does well, in detecting long and deeply nested ambiguous subsets. For lower values of the *dynamic1*'s depth option, sentences are short and quick to generate; higher values generate longer sentences.

Since *dynamic1* uses a non-deterministic approach, there can be significant variation in the sentence and ambiguous fragment length from run to run. The set of grammars discovered as ambiguous by *dynamic1* is sometimes different than other tools. For 111 of the Boltzmann and 2 of the mutated grammars that ACLA found ambiguous, *dynamic1* failed to do so; for AMBER, 4 Boltzmann and 2 mutated PL grammars; for AmbiDexter, 2 Boltzmann and 8 mutated grammars. Some of the grammars amongst these sets are common, but by no means all.

Of the 111 Boltzmann grammars for which ACLA detected ambiguity and *dynamic1* failed to detect any, 110 of them contained ambiguous subsets that were unreachable from the start rule. For instance, a grammar with rules `root: 'p'` and `A: 'q' | 'q'` contains an ambiguous subset that is unreachable from the start rule. Since ACLA's approach to ambiguity detection is by searching for ambiguous strings for each non-terminal, it can detect ambiguities that are unreachable from the start rule. We did not anticipate our Boltzmann sampler generating such non-sensical grammars, and recommend that future experiments filter them out. The remaining one grammar for ACLA, and the grammars for AMBER and AmbiDexter, contained common subsets, totalling 4 Boltzmann grammars. For 2 of them, *dynamic1* did not terminate, exited and re-ran (roughly 500 times for one of the grammars). For the remaining two grammars, one of them contained a short but deeply nested ambiguous subset, whereas for the other the ambiguous fragment was long and, for D=11 (the best performing option for Boltzmann grammar), *dynamic1* didn't generate sufficiently long sentences to uncover ambiguity.

For the mutated grammar set, *dynamic1* didn't detect ambiguity for a total of 9 grammars for which the other tools detected ambiguity. For 4 of these grammars, the ambiguous subsets were short but deeply nested. For 2 of these grammars, the ambiguous fragments were long, and D=17 (the best performing option for mutated grammars) did not generate sufficiently long sentences to uncover ambiguity. The remaining 3 grammars were cyclically ambiguous, and *dynamic1*'s sentence generator did not terminate for them.

9 Validation experiment

In order to ensure that the results of Figures 4, 5, 6 scale to larger sets of grammars, we used *dynamic1* to perform a validation experiment on a much larger set of grammars (see Table 1). The number of ambiguities found for 120 seconds were 70% (Boltzmann) and 63%, 21%, 13%, and 45% (for mutated types: add empty alternative, mutate symbol, add symbol and delete symbol respectively). The proportion of ambiguities found in our validation experiment is close to the number of ambiguities found in the main experiment (see Figures 4 and 6). All the data involved are available from our experimental downloadable suite.

10 Validating the hypotheses

In Section 1, we stated two hypotheses which informed our work. In this section, we revisit the hypotheses in the light of our results.

Hypothesis H1 postulates that "covering a grammar in breadth is more likely to uncover ambiguity than covering it in depth." *dynamic1*'s non-deterministic approach tends to naturally generate sentences which cover much larger portions of a grammar than previous approaches. It is therefore more successful at uncovering ambiguity against our grammar corpus than other tools. Although non-determinism clearly plays its part, we believe that *dynamic1*'s coverage is key and strongly validates hypothesis H1.

Hypothesis H2 postulates that "PL grammars are only a small step away from being ambiguous." The mutated grammars are our attempt to explore this hypothesis and as the validation experiment shows, just over a third of mutations to real PL grammars result in *dynamic1* detecting ambiguity. This proportion is a lower-bound: it is possible that there is further ambiguity in the mutated grammars that *dynamic1* (and, indeed, any other tool) does not discover. We consider this validation of hypothesis H2.

11 Threats to validity

The most obvious threat to the validity of our results are the grammars used.

In a previous experiment [15] we used a hand-written generator to create random grammars. In this paper we created a Boltzmann sampler to reduce the chances of bias in our hand-written generator. Interestingly, this made relatively little difference to the number of ambiguous grammars we found. However, it is impractical to generate completely arbitrary grammars, since they have no size limit. Our Boltzmann specification is therefore geared towards generating grammars which are "somewhat PL like". It is possible that it still produces overly biased grammars, particularly as we are forced to use filters to remove some grammars we consider irrelevant or unrepresentative. Our current Boltzmann sampler can create grammars which have subsets of rules which are not reachable from the start rule; ironically, these grammars penalise *dynamic1* relative to other ambiguity tools such as ACLA. However, we believe that, overall, it is more trustworthy than any previous random grammar generator.

The mutated grammars are also a potential threat to validity as we might have chosen unrepresentative grammars as a base. Since they come from an external source, we have some level of confidence in them.

The final threat to validity is our use of a mini experiment to determine a reasonable set of run-time options for the various tools used. It is possible that the grammars used in the mini experiment were unrepresentative of those used in the main experiment, though our measurements suggest this is unlikely. The percentage of ambiguous Boltzmann grammars were 67% (mini) and 70% (main). The percentage of ambiguous mutated grammars (add empty alternative, mutate symbol, add symbol, delete symbol) were mini (60%, 30%, 10%, and 42%) and main (63%, 22%, 13%, and 46%).

12 Conclusions

In this paper, we introduced the concept of a search-based approach to CFG ambiguity detection with the SinBAD tool and its *dynamic1* backend. Using the largest grammar corpus to date, we showed that *dynamic1* can detect a larger number of ambiguities than previous approaches. The key to its success is its use of non-determinism, which has several surprising consequences. It frees us from having to design many complex heuristics. *dynamic1*'s only heuristic relates to the need to terminate the sentence generator. In turn, this allows *dynamic1* to explore a much larger portion of a grammar than by previous approaches and it increases the chances of detecting ambiguous fragments nested deep within a grammar. In essence, our results suggest that covering the breadth of a grammar's state space is more important than covering it in depth.

dynamic1's chief weakness is that its single deterministic point causes it not to terminate on some grammars. We suspect that a probabilistic approach which gives a lower chance to frequently derived alternatives – in other words, which makes it less likely, but not impossible, that they are picked – will make non-termination less likely whilst preserving *dynamic1's* general approach.

We also introduced two new ways of generating large grammar corpuses: Boltzmann sampling and grammar mutation. The grammars created using Boltzmann sampling were highly ambiguous and thus not entirely representative of PL grammars. The mutated grammars, on the other hand, are representative of PL grammars although how ambiguous they are depends on the mutation. Our results indicate that certain mutations tend to cause grammars to be highly ambiguous whereas others less so. Our experience suggest that for uses that require exploring wide class of grammars, one should use Boltzmann sampling whereas, uses that require exploring PL grammars, one should use grammar mutation.

Acknowledgements. We are extremely grateful to Alexis Darrasse (LIP6) for his advice in creating a Boltzmann sampler specification. We thank the Department of Informatics at King's College for giving extended access to computing facilities. Edd Barrett and Carl Friedrich Bolz gave insightful comments on drafts of this paper.

References

1. Axelsson, R., Heljanko, K., Lange, M.: Analyzing context-free grammars using an incremental sat solver. In: Proc. ICALP 2008. pp. 410–422 (2008)
2. Basten, H.J.S.: Ambiguity Detection Methods for Context-Free Grammars. Master's thesis, Universiteit van Amsterdam (Aug 2007)
3. Basten, H.J.S., van der Storm, T.: Ambidexter: Practical ambiguity detection. In: Proc. SCAM 2010. pp. 101–102 (2010)
4. Basten, H.J.S., Vinju, J.J.: Faster ambiguity detection by grammar filtering. In: Proc. LDTA. pp. 5:1–5:9 (2010)
5. Brabrand, C., Giegerich, R., Møller, A.: Analyzing ambiguity of context-free grammars. Science of Computer Programming 75(3), 176–191 (Mar 2010)
6. Canou, B., Darrasse, A.: Fast and sound random generation for automated testing and benchmarking in objective caml. In: Proc. Workshop on ML. pp. 61–70 (2009)
7. Cantor, D.G.: On the ambiguity problem of backus systems. Journal of the ACM 9(4), 477–479 (1962)
8. Cheung, B.S.N., Uzgalis, R.C.: Ambiguity in context-free grammars. In: Proc. SAC. pp. 272–276. ACM (1995)
9. Harman, M.: The current state and future of search based software engineering. In: FOSE. pp. 342–357 (2007)
10. Mougenot, A., Darrasse, A., Blanc, X., Soria, M.: Uniform random generation of huge metamodel instances. In: ECMDA-FA. pp. 130–145 (2009)
11. Schröer, F.W.: Accent, a compiler compiler for the entire class of context-free grammars. Tech. rep. (2000), `http://accent.compilertools.net/Accent.html`
12. Schröer, F.W.: Amber, an ambiguity checker for context-free grammars. Tech. rep. (2001), `http://accent.compilertools.net/Amber.html`
13. Tomita, M.: An efficient context-free parsing algorithm for natural languages. In: Proc. IJCAI. pp. 756–764 (1985)
14. Tratt, L.: Parsing: The solved problem that isn't. Hacker Monthly pp. 37–42 (Jun 2011)
15. Vasudevan, N., Tratt, L.: Search-based ambiguity detection in context-free grammars. In: Proc. ICCSW. pp. 142–148 (Sep 2012)

A Mutated grammar generation algorithm

Algorithm 2 shows how we generate a mutated version of a grammar. $\mu_{type} \in \{empty, mutate, add, delete\}$ indicates the type of mutation to be performed for a given grammar. The function MUTATE-GRAMMAR first creates a deep copy of the grammar. For the 'add empty alternative' mutation, we first identify non-terminals which do not already have an empty alternative (line 4), before randomly selecting one, and adding an empty alternative. For mutations of type 'add symbol' we randomly select a non-terminal, before randomly selecting one of its alternatives. From the selected alternative, we randomly pick a position and insert a randomly selected symbol from V (line 12). For mutation s of type 'mutate symbol' and 'delete symbol', we randomly select a non-terminal, before randomly selecting one of its non empty alternatives. To mutate a symbol, we randomly pick a position from the selected alternative and replace it with a randomly selected symbol from V (line 18). To delete a symbol, we randomly pick a position from the selected alternative, and delete it (line 20).

Algorithm 2 An algorithm to generate a mutated version of a grammar

```
 1: function MUTATE-GRAMMAR($G$, $\mu_{type}$)
 2:     $G_c \leftarrow copy(G)$                                ▷ $Gc = \langle N_c,T_c,P_c,S_c \rangle$
 3:     if $\mu_{type} = empty$ then
 4:         $N_\psi \leftarrow \{A \in N_c \mid A \notin N_\epsilon\}$
 5:         $A \leftarrow \mathcal{R}(N_\psi, 1)$
 6:         $\Sigma P[A]_{alt} \leftarrow append(\Sigma P[A]_{alt}, [\,])$
 7:     else
 8:         $A \leftarrow \mathcal{R}(N_c, 1)$
 9:         if $\mu_{type} = \{add\}$ then
10:             $alt \leftarrow \mathcal{R}(\Sigma P[A]_{alt}, 1)$
11:             $k \leftarrow \mathcal{R}[0,|alt|)$
12:             $alt \leftarrow insert(alt, k, \mathcal{R}(V, 1))$
13:         else if $\mu_{type} \in \{mutate, delete\}$ then
14:             $alts \leftarrow \{alt \in \Sigma P[A]_{alt} \mid |alt| > 0\}$
15:             $alt \leftarrow \mathcal{R}(alts, 1)$
16:             $k \leftarrow \mathcal{R}[0,|alt|\text{-}1)$
17:             if $\mu_{type} = \{mutate\}$ then
18:                 $alt[k] \leftarrow \mathcal{R}(V, 1)$
19:             else
20:                 $alt \leftarrow delete(alt, k)$
21:             end if
22:         end if
23:     end if
24:     return $G_C$
25: end function
```