Automated Ambiguity Detection in Layout-Sensitive Grammars

JIANGYI LIU[*], Tsinghua University, China
FENGMIN ZHU[*†], Tsinghua University, China and CISPA Helmholtz Center for Information Security, Germany
FEI HE[‡], Tsinghua University, China, Key Laboratory for Information System Security, Ministry of Education, China, and Beijing National Research Center for Information Science and Technology, China

Layout-sensitive grammars have been adopted in many modern programming languages. In a serious language design phase, the specified syntax—typically a grammar—must be unambiguous. Although checking ambiguity is undecidable for context-free grammars and (trivially also) layout-sensitive grammars, *ambiguity detection*, on the other hand, is possible and can benefit language designers from *exposing potential design flaws*.

In this paper, we tackle the ambiguity detection problem in layout-sensitive grammars. Inspired by a previous work on checking the *bounded ambiguity* of context-free grammars via *SAT solving*, we intensively extend their approach to support layout-sensitive grammars but via *SMT solving* to express the ordering and quantitative relations over line/column numbers. Our key novelty lies in a *reachability* condition, which takes the impact of layout constraints on ambiguity into careful account. With this condition in hand, we propose an equivalent ambiguity notion called *local ambiguity* for the convenience of SMT encoding. We translate local ambiguity into an SMT formula and developed a *bounded ambiguity checker* that *automatically* finds a *shortest* nonempty ambiguous sentence (if exists) for a user-input grammar. The *soundness* and *completeness* of our SMT encoding are mechanized in the Coq proof assistant. We conducted an evaluation on both grammar fragments and *full* grammars extracted from the language manuals of domain-specific languages like YAML as well as general-purpose languages like Python, which reveals the effectiveness of our approach.

CCS Concepts: • **Software and its engineering → Syntax**; *Parsers*; • **Theory of computation → Grammars and context-free languages**; **Constraint and logic programming**.

Additional Key Words and Phrases: layout-sensitive grammar, ambiguity, SMT, Coq

ACM Reference Format:
Jiangyi Liu, Fengmin Zhu, and Fei He. 2023. Automated Ambiguity Detection in Layout-Sensitive Grammars. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 262 (October 2023), 27 pages. https://doi.org/10.1145/3622838

[*]Both authors contributed equally to this work.
[†]Early revisions of this work were done when the author was in Tsinghua University.
[‡]Corresponding author.

262

Authors' addresses: Jiangyi Liu, School of Software, Tsinghua University, Beijing, 100084, China, liujiang19@mails.tsinghua.edu.cn; Fengmin Zhu, School of Software, Tsinghua University, Beijing, 100084, China and CISPA Helmholtz Center for Information Security, Saarbrücken, Saarland, 66123, Germany, fengmin.zhu@cispa.de; Fei He, School of Software, Tsinghua University, Beijing, 100084, China and Key Laboratory for Information System Security, Ministry of Education, Beijing, China and Beijing National Research Center for Information Science and Technology, Beijing, China, hefei@tsinghua.edu.cn.

2475-1421/2023/10-ART262
https://doi.org/10.1145/3622838

1 INTRODUCTION

Layout-sensitive (or *indentation-sensitive*) grammars were first proposed by Landin [1966]. Nowadays, they have been adopted in many programming languages, e.g., Python [Van Rossum and Drake 2011], Haskell [Marlow et al. 2010], F# [Syme et al. 2010], Yaml [Evans et al. 2014] and Markdown [Gruber 2012]. These grammars are also introduced as new features during language evolution, e.g., indentation rules now have been introduced in Scala 3[1]. Although this amazing feature gives rise to a stylized, structural, and elegant syntax, due to the presence of layout constraints, *indentations* and *whitespaces* intensively affect how a program should be parsed, and this brings more difficulties and challenges in the study of layout-sensitive grammars than *context-free grammars* (CFGs), where indentations and whitespaces do not matter.

To avoid those difficulties and challenges, some languages choose to use *limited* layout features so that their grammars are reducible to CFGs. For instance, Python's grammar can be viewed as context-free (and parsed by a PEG parser) once a preprocessor (implemented in the CPython interpreter) converts all whitespaces and tabs into special `INDENT`/`DEDENT` tokens. However, such a method does *not* work for *all* layout-sensitive languages, especially those with more complex layout features. One example is YAML, whose lexer manually deals with whitespaces and indentations in an ad-hoc way. Moreover, languages like Haskell cannot even be parsed with context-free PEG parsers. Therefore, we believe that layout-sensitive grammar is an area worth investigating.

One problem that has been discovered in this area is *layout-sensitive parsing* [Adams 2013; Amorim et al. 2018; Erdweg et al. 2013]. To apply these parsing techniques, one must, first of all, have the grammar formally specified. In a serious language design phase, this grammar should be *unambiguous*. Due to the presence of layout constraints, manually checking the ambiguity (especially for large grammars) is tedious and error-prone. Therefore, tool support for ambiguity checking is essential to reduce human labor and to expose (and even to avoid) human mistakes.

Apparently, the most ideal way is to prove the unambiguity. Unfortunately, this is impossible: telling if a CFG is ambiguous or not is already *undecidable* [Chomsky and Schützenberger 1963; Hopcroft et al. 2001] so it is even *harder* for layout-sensitive grammars, which are more expressive than CFGs. However, for CFGs, an alternative method does exist for ensuring unambiguity: proving this CFG belongs to a known *unambiguous fragment* such as $LL(k)$ [Aho et al. 1986] or $LR(k)$ [Knuth 1965]. This method is *non-universal* in the sense that if the CFG does not belong to any known unambiguous fragment, no conclusion can be drawn. When it comes to layout-sensitive grammars, unfortunately, what could be an unambiguous fragment has not yet been investigated so far (to the best of our knowledge).

On the other hand, the *bounded ambiguity checking* problem—determining whether a grammar contains an ambiguous sentence *within a given length*—is both *universal* and *solvable*. This problem gives ways to *ambiguity detection*, i.e., to find ambiguous sentences. Suppose a language designer uses an ambiguity detection tool to check their syntax specification, then any discovered ambiguous sentence will immediately expose potential design flaws. Fixing these issues helps avoid future bugs and vulnerabilities caused by ambiguous syntax.

Axelsson et al. [2008] have studied bounded ambiguity checking on CFGs. In this paper, we intensively extend their approach *for layout-sensitive grammars*. In their work, bounded ambiguity is encoded as a propositional formula and checked via SAT solving. Inspired by the use of constraint solving—and more importantly, to achieve automation—we find SMT (Satisfiability Modulo Theories) solving suitable for layout-sensitive grammars: layout constraints are expressed by ordering (e.g., $<$, $>$, $=$) and quantitative relations over the line and column numbers and these relations are encodable in integer difference logic, a theory that a majority of SMT solvers support.

[1] https://docs.scala-lang.org/scala3/reference/other-new-features/indentation.html

To apply SMT solving, the central technical problem is to encode bounded ambiguity as an SMT formula. Because directly encoding the standard notion of ambiguity—"there exist two (or more) different parse trees"—is hard, Axelsson et al. [2008] instead consider an alternative condition that is easier to encode: "there exist two subtrees that differ in a node on level 1". We tried to apply this condition to layout-sensitive grammars, but found it *unsound*—insufficient to prove ambiguity. This is due to the neglect of the required fulfillment of layout constraints. In other words, their approach does not directly apply to layout-sensitive grammars; we need to extend it intensively. Our solution is to add a *reachability* condition that takes the impact of layout constraints on parsing and ambiguity into careful account. With the reachability condition in hand, we propose *local ambiguity*, a definition that is *logically equivalent* to the standard ambiguity while being convenient for SMT encoding.

Through a translation of local ambiguity, we obtain an SMT encoding for bounded ambiguity. This encoding is *sound* and *complete* on *acyclic* grammars. Being acyclic is *not* a strong assumption: well-designed grammars should all have this nice property. Based on this encoding, we have developed a *bounded ambiguity checker* that accepts a user-specified grammar as input (with a configurable bound); it either generates the *shortest* nonempty ambiguous sentence or proves the bounded unambiguity. When an ambiguous sentence is produced, our checker also yields all its parse trees, which eases the process of understanding the cause of ambiguity and resolving it. We believe this feature is helpful for language designers to detect and enhance potential design flaws in an early stage of language development.

Our definition of local ambiguity, our encoding of bounded ambiguity and all related lemmas and theorems have been formulated and proved in the Coq proof assistant. The main theorems we have shown are: (1) local ambiguity is logically equivalent to the standard notion of ambiguity (Theorem 4.5); (2) our encoding is sound and complete for any acyclic layout-sensitive grammar (Theorems 5.9 and 5.10). In our formulation, a layout-sensitive grammar is represented by a *layout-sensitive binary normal form* (Definition 3.1) that allows not just layout constraints but any decidable predicate. This reveals that all of our theoretical results apply not just to layout-sensitive grammars—our focus of this paper—but any "CFG + decidable predicates", e.g., *data-dependency grammars* [Jim et al. 2010]. As a side product, if the predicates are furthermore encodable in SMT theories, our bounded ambiguity checker can be extended (in a rather straightforward way) to benefit the audiences of data-dependency grammars.

We conducted an evaluation on grammars extracted from five popular programming languages: Python, SASS, YAML, F# and Haskell. Our dataset consists of 25 grammars: 5 seed grammars (one for each language) and 20 variants (four for each seed grammar). For Python and SASS, we collected their full grammars. For the other three languages, we collected grammars fragments that involve interesting layout features. On this dataset, our bounded ambiguity checker generated ambiguous sentences with lengths varying from 2 to 11. Manually inspecting the corresponding parse trees, we found it not hard to understand the cause of ambiguity and resolve it by adding more layout constraints to the input grammar.

Contributions. To sum up, this paper mainly makes the following technical contributions:

- We propose *local ambiguity*, an equivalent notion for ambiguity that is SMT-friendly (§4).
- We present an *SMT encoding* for checking bounded ambiguity of layout-sensitive grammars, with which an automated bounded ambiguity checker was developed (§5).
- We formally proved the *soundness* and *completeness* of the SMT encoding on acyclic grammars in the Coq proof assistant (§6).
- We conducted an evaluation on real-world grammars and their variants, which reveals the effectiveness of our approach (§7).

We start in §2 with an overview of our approach by example, provide background knowledge in §3, discuss related work in §8, and finally conclude in §9. See the end of the paper for a link to our open-source artifact that contains the proofs, the tool, the evaluation dataset and the results.

2 A MOTIVATING EXAMPLE

In this section, we present a concrete scene to showcase how our bounded ambiguity checker can detect ambiguity, and how a generated ambiguous sentence with its parse trees can help a user understand the cause of ambiguity so that they will be able to further resolve the ambiguity.

Imagine a user is now designing an imperative language where a block is defined as a list of statements that are aligned with each other, where each statement (stmt) is either an empty statement (nop) or a do-block that recursively takes a block as its body. The following *extended Backus-Naur form* (EBNF) expresses the intended grammar G_{block} (block is the start symbol):

$$\text{block} \longrightarrow \|\text{stmt}\|^{+}$$
$$\text{stmt} \longrightarrow \texttt{nop} \mid \texttt{do}\ \text{block}$$

As a convention, terminal symbols are in `typewriter` font and nonterminal symbols are in sans serif font. To express the alignment of a sequence of elements, we lift the standard Kleene plus operator "A^{+}" into "$\|A\|^{+}$": it represents a sequence of elements (parsed from the nonterminal A) that are aligned with each other, i.e., the column numbers of their first tokens are identical.

Due to the presence of the alignment layout constraint, this grammar is *layout-sensitive*. This alignment constraint is indeed necessary because it marks the border of the do-block body. For instance, the sentence w_1:

```
do nop
   nop
```

is parsed into a block of a single do-statement whose body contains two nop statements; while the sentence w_2:

```
do nop
nop
```

is parsed into a block of two statements, where the first element is a do-statement whose body contains a single nop statement, and the second element is a nop statement. Without this alignment layout constraint, both w_1 and w_2 will be ambiguous[2]: the last nop statement can belong to either the do-block (as in w_1) or the top-level block (as in w_2).

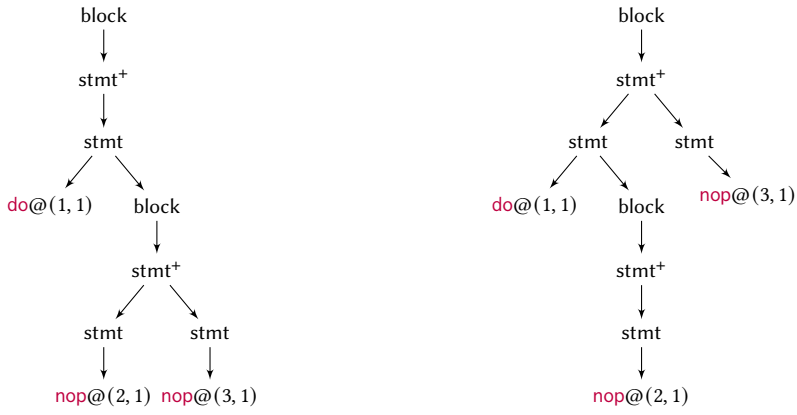
Now that this grammar looks perfect to distinguish between sentences like w_1 and w_2, can we conclude that it is unambiguous?

Detect ambiguity. To detect any potential ambiguity in G_{block}, the user feeds this grammar to our bounded ambiguity checker. The checker internally generates SMT formulae for checking the bounded ambiguity. It does not find any ambiguous sentences with lengths 1 and 2, but it produces the following ambiguous sentence w_3 with length 3:

```
1 do
2 nop
3 nop
```

In other words, G_{block} is indeed *ambiguous*.

[2]One may identify this ambiguity in our bounded ambiguity checker: input the layout-free version of G_{block} (i.e., by removing the alignment constraint); our checker can generate the ambiguous sentence do nop nop which exhibits the ambiguity described here.



(a) parse tree t_1 (b) parse tree t_2

Fig. 1. Parse trees of the ambiguous sentence w_3.

Understand the cause of ambiguity. To help the user understand why G_{block} is ambiguous, our bounded checker not only produces a *shortest* ambiguous sentence w_3 but also all of its parse trees, as depicted in Fig. 1, where the line and column numbers of tokens are annotated by "@($line, col$)".

Above all, both parse trees fulfill the alignment constraint: in tree t_1, the token do at position $(1, 1)$ of the first statement (in the top-level block) is aligned with the token nop at position $(3, 1)$ of the second statement; in tree t_2, the two nop tokens at positions $(2, 1)$ and $(3, 1)$ of the two statements inside the do-block are aligned.

The reason for the ambiguity is somewhat similar to what we have discussed above: it is insufficient to tell whether the second nop statement belongs to the do-block (the case of t_1) or the top-level block (the case of t_2), even with the presence of the alignment constraint.

Resolve the ambiguity. Since the ambiguous sentence w_3 prevents the alignment constraint from deciding the border of the do-block, a possible way to resolve this ambiguity is to reject w_3. To achieve this, one possibility is to apply the *offside* rule [Adams 2013; Landin 1966] to any do-block: if this block contains multiple lines, then the starting column of every subsequent line must be to the right of the starting column of the first line (formal definition will be introduced in Fig. 2 in the next section). In this way, w_3 will be rejected, but sentences that obey the offside rule will still be accepted, including w_1 and w_2. Another instance is:

```
do
  nop
  nop
```

where the two nop tokens (both inside the do-block) are placed to the right of the do token. One more example is:

```
do
  nop
nop
```

where the first nop token (only this one is inside the do-block) is placed to the right of the do token.

Adding the offside constraint, denoted by "$\cdot^{\triangleright}$", to the do-block clause in the original grammar G_{block}, we obtain an enhanced grammar G'_{block} that resolves this ambiguity:

$$\text{block} \longrightarrow \|\text{stmt}\|^{+}$$
$$\text{stmt} \longrightarrow \text{nop} \mid (\text{do block})^{\triangleright}$$

To see if another ambiguity exists, the user now inputs G'_{block} and sets a large bound, say 20. This time, our checker will no longer find any ambiguous sentences within this bound. Because our SMT encoding is *complete*, this shows that G'_{block} does not have any ambiguous sentences up to length 20 (in fact, it is unambiguous). To this end, the user gains more confidence.

Key takeaways. To conclude this section, we list some key takeaways:

- When inputting a user-specified grammar, our bounded ambiguity checker can *automatically* generate one *shortest* ambiguous sentence (if exists).
- If an ambiguous sentence is found, this exposes potential syntax design flaws and oversights, such as the user's lack of a proper offside rule in G_{block}.
- By inspecting the ambiguous sentence with its parse trees, it is usually straightforward to identify the cause of ambiguity and further enhance the grammar by adding more layout constraints (like the offside rule in our example) to resolve this ambiguity.
- The above workflow can be applied for *multiple rounds* until our checker no longer finds any ambiguous sentences up to a specified length, which increases the user's confidence in the designed grammar.

3 PRELIMINARIES

Before diving into the details of solving bounded ambiguity, this section provides the necessary preliminary definitions and notations.

In formal language theory, a grammar G is a quadruple (N, Σ, P, S), where N is a finite (nonempty) set of *nonterminals*, Σ is a finite set of *terminals* (or *tokens*), $P \subseteq N \times (N \cup \Sigma)^*$ is a finite set of *production rules*, and $S \in N$ is the *start symbol*.

In a CFG, a *sentence* (sometimes also called a *word*) is a token sequence. In a *layout-sensitive grammar*, a sentence is a sequence of *positioned tokens*. Each positioned token has the form $a@(line, col)$, where $a \in \Sigma$ gives the terminal, and $(line, col)$ gives the position (i.e., line and column number) in the source file. We denote an empty sentence by ε. For a nonempty sentence w, we denote its length by $|w|$. We write $w[i]$ to access the positioned token at index i, and $w[i].\text{term}$, $w[i].\text{line}$ and $w[i].\text{col}$ resp. the terminal, line number and column number.

Well-formedness. All the sentences we consider throughout the paper are *well-formed*: the positions of the tokens are in ascending order. Formally, for all indices $0 \le i < j < |w|$:

$$(w[i].\text{line} < w[j].\text{line}) \vee (w[i].\text{line} = w[j].\text{line} \wedge w[i].\text{col} < w[j].\text{col}).$$

Note that ill-formed sentences do *not* physically exist: one cannot have two tokens overlap (e.g., $[a@(1,1), b@(1,1)]$) or have a latter token precede a former one (e.g., $[a@(1,2), b@(1,1)]$).

Layout constraints. A unary (resp. binary) *layout constraint* φ is a logical predicate over one (resp. two) sentence(s), specifying positional restrictions on tokens of the sentences. As a convention, an empty sentence trivially satisfies an arbitrary constraint: $\varphi(\varepsilon) = \text{true}$ (resp. $\varphi(\varepsilon, \cdot) = \varphi(\cdot, \varepsilon) = \text{true}$ for the binary case).

We support the following built-in layout constraints: alignment (binary), indentation (binary), offside (unary), offside-align (a variant of offside, unary), and single-line (unary). The first three are widely used in many practical grammars and have been well-studied in previous work [Amorim

$$\begin{aligned}
\mathsf{align}(w_1, w_2) &\triangleq (w_1 \neq \varepsilon \wedge w_2 \neq \varepsilon) \implies w_1[0].\mathrm{col} = w_2[0].\mathrm{col} \\
\mathsf{indent}(w_1, w_2) &\triangleq (w_1 \neq \varepsilon \wedge w_2 \neq \varepsilon) \implies w_2[0].\mathrm{col} > w_1[0].\mathrm{col} \wedge w_2[0].\mathrm{line} = w_1[-1].\mathrm{line} + 1 \\
\mathsf{offside}(w) &\triangleq w \neq \varepsilon \implies \forall t \in w : t.\mathrm{line} > w[0].\mathrm{line} \implies t.\mathrm{col} > w[0].\mathrm{col} \\
\mathsf{offside\text{-}align}(w) &\triangleq w \neq \varepsilon \implies \forall t \in w : t.\mathrm{line} > w[0].\mathrm{line} \implies t.\mathrm{col} \geq w[0].\mathrm{col} \\
\mathsf{single}(w) &\triangleq w \neq \varepsilon \implies \forall t \in w : t.\mathrm{line} = w[0].\mathrm{line}
\end{aligned}$$

Fig. 2. Logical predicates of built-in layout constraints.

et al. 2018]. Offside-align and single-line rules are what we find useful in our evaluation (§7). Their logical predicates are presented in Fig. 2, where $w[-1]$ denotes the last positioned token of w.

Alignment constrains the first token of the two sentences to be aligned at the column. Indentation requires the second part to have its first token to the right (at column) of the first part, and a new line exists in between. The offside rule was first proposed by Landin [1966] and later revised by Adams [2013]. It restricts that in the parsed sentence, any subsequent lines must start from a column that is further to the right of the start token of the first line. We also consider the offside-align rule, a variant of the above, where subsequent lines can start from the same column as that of the first line. The single-line rule restricts the parsed sentence to be one-line.

Binary normal form. It is well-known [Lange and Leiß 2009] that every CFG can be converted to an equivalent *binary normal form*. We extend this binary normal form to layout-sensitive grammars by allowing layout constraints. As a convention, we reserve A, B, C for nonterminals, a, b, c for tokens/terminals and φ for layout constraints.

Definition 3.1 (LS2NF). A layout-sensitive grammar is in *binary normal form*, called the *layout-sensitive binary normal form* (LS2NF), if for every production rule, its right-hand side (called a *clause*) is one of the following:

- an empty clause ε;
- an atomic clause a, where $a \in \Sigma$;
- an unary clause A^{φ}, where $A \in N$ and φ is a *unary layout constraint*;
- a binary clause $A\ \varphi\ B$, where $A, B \in N$ and φ is a *binary layout constraint*.

To avoid inconsistent layout constraints, the production rule set cannot contain the following rules simultaneously: (1) $A \to B^{\varphi}$ and $A \to B^{\varphi'}$ where $\varphi \neq \varphi'$; (2) $A \to B_1\ \varphi\ B_2$ and $A \to B_1\ \varphi'\ B_2$ where $\varphi \neq \varphi'$.

For convenience, we use notations in place of rule names for built-in layout constraints: we write

- $\alpha \parallel \beta$ for α align β,
- $\alpha \boxplus \beta$ for α indent β,
- $\alpha^{\triangleright}$ for $\alpha^{\mathsf{offside}}$, $\alpha^{\trianglerighteq}$ for $\alpha^{\mathsf{offside-align}}$, and
- $(\!|\alpha|\!)$ for α^{single}.

Throughout the paper, we study the ambiguity theory of layout-sensitive grammars in LS2NF, but use EBNF to express complex grammars for better readability. We desugar the standard EBNF operators "?" (optional), "+" (Kleene plus), and "$*$" (Kleene star) into the following sets of LS2NF production rules (where C is a fresh nonterminal symbol, and "$\rightsquigarrow$" means "desugars to"):

- $A \to B^{?} \rightsquigarrow \{A \to B, A \to \varepsilon\}$
- $A \to B^{+} \rightsquigarrow \{A \to B, A \to C, C \to A\ B\}$

- $A \to B^* \rightsquigarrow \{A \to \varepsilon, A \to C, C \to A\ B\}$

We also extend Kleene plus and Kleene star operators to alignment: $\|\cdot\|^+$ and $\|\cdot\|^*$. Their desugaring rules are as follows (where C is a fresh nonterminal symbol, and "$\rightsquigarrow$" means "desugars to"):

- $A \to \|B\|^+ \rightsquigarrow \{A \to B, A \to C, C \to A \parallel B\}$
- $A \to \|B\|^* \rightsquigarrow \{A \to \varepsilon, A \to C, C \to A \parallel B\}$

During the desugaring process, LS2NF rules are tagged with their corresponding locations in the EBNF. In later steps, parse trees generated by our approach (using LS2NF grammar) can be de-binarized with those attached tags. Therefore, any cognitive discrepancy between the language designer's view of the grammar and the internal representation can be avoided.

Parse trees. *Parse trees* are rooted trees (let $A \in N$ be the root) inductively defined as follows:

$$t ::= \mathbf{empty}(A) \mid \mathbf{leaf}(A, a@(i,j)) \mid \mathbf{unary}(A, t) \mid \mathbf{binary}(A, t_1, t_2).$$

Branches on the right side correspond to the derivation using the empty, atom, unary, and binary clauses. We write $\mathit{root}(t)$ and $\mathit{word}(t)$ to denote the root node and the represented sentence of a parse tree t. A parse tree is said *valid* if it follows the production rules and fulfills the layout constraints:

- $\mathbf{empty}(A)$ is valid if $A \to \varepsilon \in P$;
- $\mathbf{leaf}(A, a@(i,j))$ is valid if $A \to a \in P$;
- $\mathbf{unary}(A, t)$ is valid if $A \to \mathit{root}(t)^{\varphi} \in P$, $\varphi(\mathit{word}(t))$ is true and t is valid;
- $\mathbf{binary}(A, t_1, t_2)$ is valid if $A \to \mathit{root}(t_1)\ \varphi\ \mathit{root}(t_2) \in P$, $\varphi(\mathit{word}(t_1), \mathit{word}(t_2))$ is true and both t_1, t_2 are valid.

We say that a parse tree t *witnesses* a derivation from a nonterminal A to a sentence w, denoted as "$t \triangleright A \Rightarrow_* w$", if t is valid, $\mathit{root}(t) = A$ and $\mathit{word}(t) = w$. Using the above notions, the usual notion of derivation "$A \Rightarrow_* w$" is a short-hand for "$\exists t, t \triangleright A \Rightarrow_* w$".

Derivation ambiguity. We say the derivation $A \Rightarrow_* w$ is *ambiguous* if there exist at least two different parse trees that witness $A \Rightarrow_* w$. In this way, the bounded ambiguity problem can be restated as follows: for a given bound k, is there a sentence w such that $|w| \le k$ and the derivation $S \Rightarrow_* w$ is ambiguous?

4 LOCAL AMBIGUITY

To generate ambiguous sentences via SMT solving, the key technical problem is to construct an SMT formula $\Phi(k)$ such that it is *satisfiable* if and only if there exists an ambiguous sentence of length k, and that every satisfying model corresponds to such an ambiguous sentence. The most direct approach is to encode derivation ambiguity—"there exist two *different* parse trees that both witness the derivation"—but how to encode the existence of those trees is challenging. Generally speaking, the node pair that shows the difference can appear at an *arbitrary level* (or depth). Enumerating every possibility is sophisticated and inefficient.

An *indirect* but *more efficient* approach could be, use an *alternative definition* that is logically equivalent to derivation ambiguity while being *conveniently encodable* in SMT theories. In this section, we will propose such a definition which we call *local ambiguity*. This definition will eventually allow us to construct a *sound* and *complete* SMT encoding for the bounded ambiguity checking problem (will be discussed in §5).

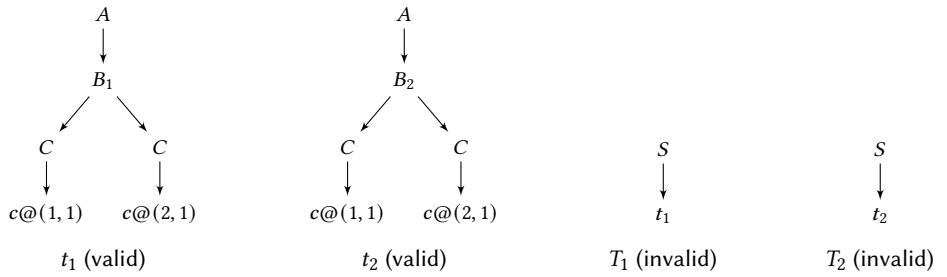


Fig. 3. A counterexample where bAMB is unsound.

4.1 Motivation: A failing attempt

Our first attempt is to apply Axelsson et al. [2008]'s *bAMB* condition[3] that applies to *reduced*[4] CFGs. The bAMB condition is stated as: "there exists a nonterminal A and a sentence w such that w has at least two different (valid) parse trees rooted at A and they differ in a node on level 1". Since parse trees are visual representations of derivations, such two trees correspond to two distinct derivations of w: using either two distinct production rules for A, or one rule in two distinct ways. In terms of logical constraints, bAMB is *weaker* than derivation ambiguity: (1) A is *existentially* quantified, thus can be any nonterminal; (2) the two parse trees differ in a node just below (i.e., a child of) their root A (a *determined* and *shallow* location). In this way, bAMB is conveniently encodable in SMT theories.

In order to use bAMB as an alternative definition, we must show that "bAMB $\Leftrightarrow$ derivation ambiguity". The $\Leftarrow$-part is trivial as we have seen that bAMB is weaker. The $\Rightarrow$-part, however, does *not* always hold for layout-sensitive grammars, meaning the bAMB condition is *unsound*. To see this, we construct the following counterexample—an LS2NF (S is the start symbol) that fulfills bAMB but is indeed unambiguous:

$$S \to (\!|A|\!) \qquad A \to B_1 \qquad A \to B_2 \qquad B_1 \to C \parallel C \qquad B_2 \to C\ C \qquad C \to c$$

The bAMB condition holds because the sentence $w^* \triangleq [c@(1,1), c@(2,1)]$ (note the two tokens are aligned) has two different (valid) parse trees rooted at A, i.e., t_1 and t_2 depicted in Fig. 3, that differ in a node on level 1 (i.e., B_1 v.s. B_2). While w^* is ambiguous on the *subgrammar* started with A, it *cannot* show the *original grammar* is ambiguous due to the *violation* of the single-line constraint required by $S \to (\!|A|\!)$. In detail, there is no way to extend t_1 and t_2 into valid parse trees rooted at S: the only possibilities are T_1 and T_2 in Fig. 3, but both are *invalid*. This makes a big difference with CFGs: when given a production rule $S \to A$, any valid parse tree of A gives rise to a valid parse tree of S. Although the bAMB condition holds, this grammar is indeed *unambiguous*: only $A \to B_2$ is applicable; every sentence produced by $A \to B_1$ contains at least two lines (due to the alignment constraint in $B_1 \to C \parallel C$), which violates the single-line constraint.

We learn from this counterexample that the following property is crucial to making the $\Rightarrow$-part provable: any valid parse tree t shall be extended into a larger valid parse tree T having t as its subtree. We formulate the above property as a relation over a pair of a nonterminal and a sentence, which we call a *signature*:

[3]In *their* paper, "bAMB" stands for "bounded ambiguity". We stick to the abbreviation "bAMB" here because the term "bounded ambiguity" has a different meaning in *this* paper.
[4]A CFG is said to be reduced if all nonterminal symbols are reachable from the start symbol.

Definition 4.1. (Sub-derivation) Let (A, w) and (B, v) be two signatures. We say (B, v) is a *sub-derivation* of (A, w), if for every parse tree $t \triangleright B \Rightarrow_* v$, there exists a parse tree T such that $T \triangleright A \Rightarrow_* w$ and t is a subtree of T.

Assuming (B, v) is a sub-derivation of (A, w), the idea of proving the $\Rightarrow$-part is as follows: Given $t_1 \neq t_2 \triangleright A \Rightarrow_* w_A$, if some premise indicates that (A, w_A) is a sub-derivation of (S, w_S) for some w_S, then we can conclude that $S \Rightarrow_* w_S$ is ambiguous by extending t_1 (resp. t_2) into T_1 (resp. T_2), where $T_1 \neq T_2 \triangleright S \Rightarrow_* w_S$. To execute this idea, we first propose *reachability* as the missing premise that implies sub-derivation (§4.2), and then define *local ambiguity* to be essentially "bAMB ∧ reachability", so that "local ambiguity ⇔ derivation ambiguity" becomes provable (§4.3).

4.2 Reachability: The Missing Piece

The definition of reachability should encode the necessary conditions that establish a sub-derivation relation. To make (B, v) a sub-derivation of (A, w), we must prove $A \Rightarrow_* w$, which can be split into two main steps: $A \Rightarrow_* s_1 B s_2 \Rightarrow_* w_1 v w_2 = w$. Given that $B \Rightarrow_* v$, it suffices to find the sentential forms $s_1 \Rightarrow_* w_1$ and $s_2 \Rightarrow_* w_2$ such that $A \Rightarrow_* s_1 B s_2$. Recall that in CFGs, $A \Rightarrow_* s_1 B s_2$ means "B is reachable from A". This motivates us to extend the reachability relation on CFGs to layout-sensitive grammars—taking sentences and the layout constraints they should fulfill into account.

Definition 4.2. (Reachability) The *reachability* relation $(A, w) \rightarrow_* (B, v)$ is the reflexive transitive closure of a one-step reachability relation $\rightarrow_1$ inductively defined as follows:

(1) If $A \rightarrow B^{\varphi} \in P$ and $\varphi(w)$, then $(A, w) \rightarrow_1 (B, w)$.
(2) If $A \rightarrow B\ \varphi\ B' \in P$, $\varphi(w_1, w_2)$ and $B' \Rightarrow_* w_2$, then $(A, w_1 w_2) \rightarrow_1 (B, w_1)$.
(3) If $A \rightarrow B'\ \varphi\ B \in P$, $\varphi(w_1, w_2)$ and $B' \Rightarrow_* w_1$, then $(A, w_1 w_2) \rightarrow_1 (B, w_2)$.

The three rules for $\rightarrow_1$ enumerate all the possibilities we can derive a nonterminal B from A in one derivation step, via the production rule $A \rightarrow B^{\varphi}$, $A \rightarrow B\ \varphi\ B'$ and $A \rightarrow B'\ \varphi\ B$, respectively. This definition is strong enough to imply the sub-derivation relation, as stated in the following lemma:

Lemma 4.3. *If $B \Rightarrow_* v$, then $(A, w) \rightarrow_* (B, v)$ iff (B, v) is a sub-derivation of (A, w).*

Proof Sketch. For the $\Rightarrow$-part: induction on the reachable relation (in this part, the hypothesis $B \Rightarrow_* v$ is useless). For the $\Leftarrow$-part: we have $T \triangleright A \Rightarrow_* w$ for every $t \triangleright B \Rightarrow_* v$; induction on T and check in every case $(A, w) \rightarrow_* (B, v)$. □

In the counterexample we explained in §4.1, (S, w^*) is not reachable to (A, w^*): although the nonterminal symbol S is reachable to A via the production rule $S \rightarrow \langle\!|A|\!\rangle$, the single-line constraint $\mathsf{single}(w^*)$ violates. Thus, the lemma above does not apply. Furthermore, no sentence w fulfills $(S, w) \rightarrow_* (A, w^*)$ due to the violation of the single-line constraint. On the other hand, let $w' \triangleq [c@(1,1), c@(1,2)]$, we have $(S, w') \rightarrow_* (A, w')$ because the single-line constraint $\mathsf{single}(w^*)$ now fulfills. By the lemma above, a parse tree that witnesses $A \Rightarrow_* w'$ can be extended to a full parse tree that witnesses $S \Rightarrow_* w'$.

4.3 Local Ambiguity: Foundation of SMT Encoding

With the reachability relation defined, *local ambiguity* is essentially "bAMB ∧ reachability". In bAMB, "two parse trees *differ on a node at level 1*" is formally expressed by "they are *dissimilar*"

(denoted by $\not\simeq$), via a *similarity* relation $\simeq$ inductively defined by the rules below:

$$\mathbf{empty}(A) \simeq \mathbf{empty}(A) \qquad \mathbf{leaf}(A, tk) \simeq \mathbf{leaf}(A, tk)$$

$$\frac{\mathit{root}(t_1) = \mathit{root}(t_2) \quad \mathit{word}(t_1) = \mathit{word}(t_2)}{\mathbf{unary}(A, t_1) \simeq \mathbf{unary}(A, t_2)}$$

$$\frac{\mathit{root}(t_{11}) = \mathit{root}(t_{21}) \quad \mathit{word}(t_{11}) = \mathit{word}(t_{21}) \quad \mathit{root}(t_{12}) = \mathit{root}(t_{22}) \quad \mathit{word}(t_{12}) = \mathit{word}(t_{22})}{\mathbf{binary}(A, t_{11}, t_{12}) \simeq \mathbf{binary}(A, t_{21}, t_{22})}$$

Definition 4.4 (Local ambiguity). A signature (A, w) is said to be *locally ambiguous* if there exists (H, h) such that $(A, w) \rightarrow_* (H, h)$, and there exist $t_1 \not\simeq t_2$ that both witness $H \Rightarrow_* h$.

Local ambiguity is convenient for SMT encoding: both the similarity and reachability relations are inductively defined, and all their side premises are encodable in SMT theories. Our encoding (which will be explained in §5) is essentially translating the local ambiguity definition into an SMT formula. The following theorem—"local ambiguity ⇔ derivation ambiguity"—provides guarantees towards a sound and complete SMT encoding:

Theorem 4.5. *(A, w) is locally ambiguous iff the derivation $A \Rightarrow_* w$ is ambiguous.*

Proof Sketch. For the $\Rightarrow$-part: Let (H, h) be a signature such that $(A, w) \rightarrow_* (H, h)$. Let $t_1 \not\simeq t_2$ be two parse trees that both witness $H \Rightarrow_* h$. By Lemma 4.3, there exist $t \triangleright A \Rightarrow_* w$ and t_1 being a subtree of t. By substituting t_2 for t_1 in t, we obtain a new parse tree $t' \triangleright A \Rightarrow_* w$. We have $t_1 \not\simeq t_2 \Rightarrow t_1 \neq t_2 \Rightarrow t \neq t'$. Therefore, t and t' witness the ambiguity of $A \Rightarrow_* w$.

For the $\Leftarrow$-part: Let $t_1 \neq t_2 \triangleright A \Rightarrow_* w$. It suffices to extract two subtrees, say t_1' from t_1 and t_2' from t_2, such that: (1) they are both valid; (2) they are not similar; and (3) the signature they both have, say (H, h), is reachable from (A, w) (i.e., $(A, w) \rightarrow_* (H, h)$). The subtrees t_1' and t_2' can be found via tree difference: compare node pairs between t_1 and t_2 in a depth-first order and return immediately when the nodes are different. By the property of such a tree difference algorithm, one can discharge the above conditions (1) – (3). □

Using the above theorem, we can show that the grammar mentioned in §4.1 is unambiguous: one cannot find any sentence w such that (S, w) is locally ambiguous because no sentence w fulfills $(S, w) \rightarrow_* (A, w^*)$. On the other side, if we loosen $S \rightarrow \langle\!|A|\!\rangle$ to $S \rightarrow A$ (i.e., erasing the single-line constraint) in this grammar, it becomes ambiguous because (S, w^*) is now locally ambiguous: the two dissimilar trees t_1 and t_2 of Fig. 3 both witness $A \Rightarrow_* w^*$, and the reachability relation $(S, w^*) \rightarrow_* (A, w^*)$ now holds. The two possible parses of w^* are viewed as T_1 and T_2 in Fig. 3.

Comparison. Our concept of local ambiguity strengthens Axelsson et al. [2008]'s bAMB in two dimensions. First, a reachability relation is defined for layout-sensitive grammars as a nontrivial extension of the usual reachability notion on CFGs. This relation "locally" encodes the necessary conditions for ensuring the equivalence theorem on each signature. Second, our equivalence theorem (Theorem 4.5) holds for any layout-sensitive grammar and obviously also for any CFG, which makes it more general than bAMB which only applies to reduced CFGs.

5 BOUNDED AMBIGUITY CHECKING

With the notion of local ambiguity introduced in the previous section, we are now ready to construct an encoding $\Phi(k)$—the existence of a k-length ambiguous sentence—via a translation of local ambiguity into an SMT formula. We will present this translation in a top-down manner (§5.1, §5.2) and show that it is *sound* and *complete* (§5.3). Relying on an SMT solver (e.g., Z3) as the backend, our bounded ambiguity checker is facilitated by a bounded loop that finds the smallest $k > 0$ such

that $\Phi(k)$ is satisfiable, whose satisfying model gives a *shortest* ambiguous sentence of the input grammar (§5.4).

5.1 Satisfying Model

Before presenting $\Phi(k)$, let us see which variables should be included in a satisfying model m of this formula. First, we encode the ambiguous sentence w^m of the model m—a k-length positioned token sequence—by three groups of variables $\mathcal{T}_i$, $\mathcal{L}_i$ and $\mathcal{C}_i$ (for $0 \le i < k$): they resp. encode the terminal, the line number, and the column number of each positioned token in the sequence.

Second, we introduce auxiliary propositional variables to state whether a derivation or reachability judgment holds, as required by the definition of local ambiguity. The variables are split into three groups (where $w^m_{x,\delta}$ denote the δ-length subsentence of w^m starting at index x):

(1) $\mathcal{D}^A_{x,\delta}$ for $A \in N, 0 < x + \delta < k$ states whether $A \Rightarrow_* w^m_{x,\delta}$;
(2) $\mathcal{R}^A_\varepsilon$ for $A \in N$ states whether $(S, w^m) \to_* (A, \varepsilon)$;
(3) $\mathcal{R}^A_{x,\delta}$ for $A \in N, 0 < x + \delta < k$ states whether $(S, w^m) \to_* (A, w^m_{x,\delta})$.

We use two groups of variables to encode the reachability judgments, $\mathcal{R}^A_\varepsilon$ for the empty sentence and $\mathcal{R}^A_{x,\delta}$ for nonempty subsentence $w^m_{x,\delta}$. For the derivation judgments, however, we do *not* need variables $\mathcal{D}^A_\varepsilon$ to encode $A \Rightarrow_* \varepsilon$ because nullability does *not depend* on a sentence—it can be *precomputed* from the production rules and we write $\mathit{null}(A)$ to mean A is nullable.

5.2 SMT Encoding

We now present the top-level encoding for $\Phi(k)$, where the auxiliary definitions $\Phi_D(k)$, $\Phi^\varepsilon_R(k)$, $\Phi^\not\varepsilon_R(k)$ and $\Phi_{\text{multi}}(H, x, \delta)$ will be introduced later:

$$\Phi(k) \triangleq \Phi_D(k) \wedge \Phi^\varepsilon_R(k) \wedge \Phi^\not\varepsilon_R(k) \wedge \bigvee_{H \in N} \left((\mathcal{R}^H_\varepsilon \wedge \Phi_{\text{multi}}(H, 0, 0)) \vee \bigvee_{\substack{0 < \delta \\ x + \delta \le k}} (\mathcal{R}^H_{x,\delta} \wedge \Phi_{\text{multi}}(H, x, \delta)) \right).$$

The first three conjuncts $\Phi_D(k)$, $\Phi^\varepsilon_R(k)$ and $\Phi^\not\varepsilon_R(k)$ resp. provide logical restrictions on the three groups of propositional variables $\mathcal{D}^A_{x,\delta}$, $\mathcal{R}^A_\varepsilon$ and $\mathcal{R}^A_{x,\delta}$, so that when $\Phi(k)$ is satisfiable, their truth values will indicate the validity of the derivation/reachability judgments as mentioned in §5.1.

The last conjunct encodes local ambiguity by Definition 4.4: there exists a nonterminal $H \in N$ and a subsentence $w^m_{x,\delta}$ such that:

(a) $(S, w^m) \to_* (H, w^m_{x,\delta})$, expressed by $\mathcal{R}^H_\varepsilon$ (if $w^m_{x,\delta} = \varepsilon$) or $\mathcal{R}^H_{x,\delta}$ (if $w^m_{x,\delta} \neq \varepsilon$); and
(b) there are two dissimilar parse trees that witness $H \Rightarrow_* w^m_{x,\delta}$, expressed by $\Phi_{\text{multi}}(H, x, \delta)$.

Well-foundedness. Realizing that derivation and reachability relations are recursively defined on nonterminals, the nonterminal set N should be *well-founded* so that *sound* encodings for $\Phi_D(k)$, $\Phi^\varepsilon_R(k)$, $\Phi^\not\varepsilon_R(k)$ and $\Phi_{\text{multi}}(H, x, \delta)$ can be constructed. To obtain a well-founded relation on N, we require the grammar to be *acyclic*, which intuitively means any cyclic derivation such as $A \Rightarrow_+ A$ is not allowed. This requirement does *not* weaken the practicality of our approach: a well-designed grammar should never be cyclic; cycles are very likely to cause ambiguity.

Formally, an LS2NF (N, Σ, P, S) is said *acyclic* if its *graph representation* is acyclic (in graph theory), where the graph representation is a directed graph $\langle N, E\rangle$ such that

$$(A, B) \in E \Leftrightarrow (\exists \varphi : A \to B^{\varphi} \in P) \vee (\exists \varphi, B' : A \to B\ \varphi\ B' \in P \wedge \mathit{null}(B')) \\ \vee (\exists \varphi, B' : A \to B'\ \varphi\ B \in P \wedge \mathit{null}(B')).$$

It is well-known that deciding the acyclicity of a directed graph is solvable in linear time [Tarjan 1972].

In an acyclic LS2NF, the edge set E of its graph representation forms a well-founded relation, called the *predecessor* relation, written $A \prec B$ for $(A, B) \in E$. The inverse (or transpose) relation E^{-1} is also well-founded, called the *successor* relation, written $A \succ B$ for $(B, A) \in E$ (or $(A, B) \in E^{-1}$).

Encoding derivation. The key observation is that the judgment $A \Rightarrow_* w$ $(w \neq \varepsilon)$ can be rewritten (being logically equivalent) in a "more verbose" disjunctive form, which is closer to an SMT formula:

$$(A \Rightarrow_* w) \Leftrightarrow \bigvee_{A \to a \in P} (|w| = 1 \wedge w_i.\mathrm{term} = a) \vee \bigvee_{A \to B^{\varphi} \in P} (B \Rightarrow_* w \wedge \varphi(w)) \\ \vee \bigvee_{\substack{A \to B_1\, \varphi\, B_2 \in P \\ w = w_1\, w_2}} (B_1 \Rightarrow_* w_1 \wedge B_2 \Rightarrow_* w_2 \wedge \varphi(w_1, w_2)), \tag{1}$$

where the three disjuncts enumerate all possible ways to achieve $A \Rightarrow_* w$ using a production rule:

- $A \to a$ if w has length 1 and a is the sole terminal of w;
- $A \to B^{\varphi}$ if B produces w and w satisfies φ;
- $A \to B_1\ \varphi\ B_2$ if B_1 produces a prefix of w, B_2 produces the rest (suffix), and the two satisfy φ. There are $(\delta + 1)$ ways of splitting w into two parts: the prefix length ranges from 0 to δ.

For every possible layout constraint φ used in the grammar, we assume there is an SMT formula that *consistently* encodes its semantics:

- for a unary constraint φ, $\Phi_{\varphi}(x, \delta) \Leftrightarrow \varphi(w^m_{x,\delta})$;
- for a binary constraint φ, $\Phi_{\varphi}(x, \delta', \delta) \Leftrightarrow \varphi(w^m_{x,\delta'}, w^m_{x+\delta',\delta-\delta'})$.

The encodings for the built-in layout constraints are direct translations of their definitions (Fig. 2).

The formula $\Phi_D(k)$ provides restrictions that every $\mathcal{D}^A_{x,\delta}$ is true if and only if $A \Rightarrow_* w^m_{x,\delta}$, and the latter is a direct translation of Eq. (1):

$$\Phi_D(k) \triangleq \forall A \in N, \delta > 0, x + \delta \leq k : \mathcal{D}^A_{x,\delta} \Leftrightarrow \\ \bigvee_{A \to a \in P} (\delta = 1 \wedge \mathcal{T}_x = a) \vee \bigvee_{A \to B^{\varphi} \in P} (\mathcal{D}^B_{x,\delta} \wedge \Phi_{\varphi}(x, \delta)) \vee \bigvee_{A \to B_1 \varphi B_2 \in P} \\ \left((\mathit{null}(B_1) \wedge \mathcal{D}^{B_2}_{x,\delta}) \vee (\mathit{null}(B_2) \wedge \mathcal{D}^{B_1}_{x,\delta}) \vee \bigvee_{0 < \delta' < \delta} (\mathcal{D}^{B_1}_{x,\delta'} \wedge \mathcal{D}^{B_2}_{x+\delta',\delta-\delta'} \wedge \Phi_{\varphi}(x, \delta', \delta)) \right).$$

The three disjuncts on the right-hand side of $\Leftrightarrow$ respectively correspond to the three disjuncts in Eq. (1). The last disjunct of Eq. (1) splits depending on if w_1 or w_2 is empty—this is necessary because $\mathcal{D}^{B_1}_{x,\delta'}$ is only defined for nonempty subsentences ($\delta' > 0$).

Lemma 5.1. *If $m \models \Phi_D(k)$, then for every $A \in N$, $\delta > 0$, $x + \delta \leq k$, $m(\mathcal{D}^A_{x,\delta}) = \mathit{true}$ iff $A \Rightarrow_* w^m_{x,\delta}$.*

Proof Sketch. By well-founded induction on δ and $\prec$. □

Example 5.2. Throughout this section, we showcase the concrete SMT encoding on the LS2NF for the EBNF G'_{block} defined in §2 (we introduce fresh auxiliary nonterminals r and s):

$$\begin{array}{ll} \mathsf{block} \to \mathsf{block} \parallel \mathsf{stmt} & \mathsf{block} \to \mathsf{stmt} \\ \mathsf{stmt} \to \mathsf{nop} & \mathsf{stmt} \to \mathsf{r}^{\triangleright} \\ \mathsf{r} \to \mathsf{s}\ \mathsf{block} & \mathsf{s} \to \mathsf{do} \end{array}$$

We set $k = 3$. The formula $\Phi_D(3)$ iterates over the nonterminals and all pairs of x and δ such that $\delta > 0 \wedge x + \delta \le k$. One of them is the following:

$$\begin{aligned} \mathcal{D}^{\mathsf{block}}_{0,3} \Leftrightarrow \bot \vee (\mathcal{D}^{\mathsf{stmt}}_{0,3} \wedge \top) \\ \vee [(\mathit{null}(\mathsf{block}) \wedge \mathcal{D}^{\mathsf{stmt}}_{0,3}) \vee (\mathit{null}(\mathsf{stmt}) \wedge \mathcal{D}^{\mathsf{block}}_{0,3}) \vee (\mathcal{D}^{\mathsf{block}}_{0,1} \wedge \mathcal{D}^{\mathsf{stmt}}_{1,2} \wedge \Phi_{\mathsf{align}}(0,1,3)) \\ \vee (\mathcal{D}^{\mathsf{block}}_{0,2} \wedge \mathcal{D}^{\mathsf{stmt}}_{2,1} \wedge \Phi_{\mathsf{align}}(0,2,3))]. \end{aligned}$$

Its right-hand side encodes all possible ways to achieve $\mathsf{block} \Rightarrow_* w^m_{0,3}$:

- the first disjunct is false ($\bot$) because $\delta = 3 \neq 1$;
- the second disjunct encodes the possibility of using "block $\to$ stmt": stmt should produce $w^m_{0,3}$ (encoded as $\mathcal{D}^{\mathsf{stmt}}_{0,3}$); no layout constraint required, i.e., true ($\top$).
- the last disjunct encodes the possibilities of using "block $\to$ block $\parallel$ stmt", further split into four sub-disjuncts:
  - $\delta = 0$: the prefix is empty, meaning block is nullable ($\mathit{null}(\mathsf{block})$); the suffix produces $w^m_{0,3}$ ($\mathcal{D}^{\mathsf{stmt}}_{0,3}$).
  - $\delta = 3$: the suffix is empty, which is the opposite to the above case.
  - $\delta = 1$: block produces $w^m_{0,1}$ ($\mathcal{D}^{\mathsf{block}}_{0,1}$) and stmt the rest $w^m_{1,2}$ ($\mathcal{D}^{\mathsf{stmt}}_{1,2}$); the two satisfy the alignment constraint ($\Phi_{\mathsf{align}}(0,1,3)$);
  - $\delta = 2$: similar to the above case.

Encoding reachability. The idea of rewriting a derivation judgment as a disjunctive form applies to the reachability judgment too:

$$\begin{aligned} (S,w) \to_* (B,w_B) \Leftrightarrow (B = S \wedge w_B = w) \vee \bigvee_{A \to B\varphi \in P} ((S,w) \to_* (A,w_B) \wedge \varphi(w_B)) \\ \vee \bigvee_{A \to B\varphi B' \in P} ((S,w) \to_* (A,w_B w') \wedge B' \Rightarrow_* w' \wedge \varphi(w_B,w')) \\ \vee \bigvee_{A \to B'\varphi B \in P} ((S,w) \to_* (A,w' w_B) \wedge B' \Rightarrow_* w' \wedge \varphi(w',w_B)). \end{aligned} \tag{2}$$

It encodes $\to_*$, the reflexive and transitive closure of $\to_1$, by explicitly stating reflexivity as the first disjunct, and then integrating transitivity into $\to_1$, yielding the last three disjuncts that respectively correspond to the three rules for $\to_1$ as defined in Definition 4.2. Depending on whether

w_B is empty or not, we obtain the encodings of $\Phi^{\varepsilon}_R(k)$ and $\Phi^{\not\varepsilon}_R(k)$ via a translation of Eq. (2):

$$\Phi^{\varepsilon}_R(k) \triangleq \forall B \in N : \mathcal{R}^B_{\varepsilon} \Leftrightarrow$$

$$(B = S \wedge k = 0) \vee \bigvee_{A \to B^{\varphi} \in P} \mathcal{R}^A_{\varepsilon} \vee \bigvee_{A \to B\varphi B' \in P} \left((\mathcal{R}^A_{\varepsilon} \wedge \mathit{null}(B')) \vee \bigvee_{\substack{\delta > 0 \\ x+\delta \le k}} (\mathcal{R}^A_{x,\delta} \wedge \mathcal{D}^{B'}_{x,\delta}) \right)$$

$$\vee \bigvee_{A \to B'\varphi B \in P} \left((\mathcal{R}^A_{\varepsilon} \wedge \mathit{null}(B')) \vee \bigvee_{\substack{\delta > 0 \\ x+\delta \le k}} (\mathcal{R}^A_{x,\delta} \wedge \mathcal{D}^{B'}_{x,\delta}) \right).$$

$$\Phi^{\not\varepsilon}_R(k) \triangleq \forall B \in N, \delta > 0, x + \delta \le k : \mathcal{R}^B_{x,\delta} \Leftrightarrow$$

$$(B = S \wedge x = 0 \wedge \delta = k) \vee \bigvee_{A \to B^{\varphi} \in P} (\mathcal{R}^A_{x,\delta} \wedge \Phi_{\varphi}(x, \delta))$$

$$\vee \bigvee_{\substack{A \to B\varphi B' \in P \\ x+\delta+\delta' \le k}} (\mathcal{R}^A_{x,\delta+\delta'} \wedge \mathit{ite}(\delta' = 0, \mathit{null}(B'), \mathcal{D}^{B'}_{x+\delta,\delta'}) \wedge \Phi_{\varphi}(x, \delta, \delta + \delta'))$$

$$\vee \bigvee_{\substack{A \to B'\varphi B \in P \\ \delta' \le x}} (\mathcal{R}^A_{x-\delta',\delta'+\delta} \wedge \mathit{ite}(\delta' = 0, \mathit{null}(B'), \mathcal{D}^{B'}_{x-\delta',\delta'}) \wedge \Phi_{\varphi}(x - \delta', \delta', \delta' + \delta)).$$

The layout predicates $\varphi(w_B)$, $\varphi(w_B, w')$ and $\varphi(w', w_B)$ of Eq. (2) are not translated in $\Phi^{\varepsilon}_R(k)$ because they trivially hold ($w_B = \varepsilon$). When translating $B' \Rightarrow_* w'$ of Eq. (2), we must be careful that w' might be empty:

- in $\Phi^{\varepsilon}_R(k)$, w' is always empty so the translation is $\mathit{null}(B')$;
- in $\Phi^{\not\varepsilon}_R(k)$, we use the "if-then-else" predicate $\mathit{ite}(c, \Phi_1, \Phi_2)$, a shorthand for $(c \wedge \Phi_1) \vee (\neg c \wedge \Phi_2)$, to cover both cases.

Lemma 5.3. *If $m \models \Phi_D(k) \wedge \Phi^{\not\varepsilon}_R(k) \wedge \Phi^{\varepsilon}_R(k)$, then for every $B \in N$, if $m(\mathcal{R}^B_{\varepsilon}) = \mathit{true}$, then $(S, w^m) \to_* (B, \varepsilon)$.*

Proof Sketch. By well-founded induction on $\succ$ (note this is the reverse of $\prec$). Apply Lemma 5.1 where necessary. □

Lemma 5.4. *If $m \models \Phi_D(k) \wedge \Phi^{\not\varepsilon}_R(k)$, then for every $B \in N$, $\delta > 0$, $x + \delta \le k$, if $m(\mathcal{R}^B_{x,\delta}) = \mathit{true}$, then $(S, w^m) \to_* (B, w^m_{x,\delta})$.*

Proof Sketch. By well-founded induction on $k - \delta$ and $\succ$ (note this is the reverse of $\prec$). Apply Lemma 5.1 where necessary. □

Example 5.5. The formula $\Phi^{\varepsilon}_R(3)$ iterates over the nonterminals. One of them is the following:

$$\mathcal{R}^{\mathsf{block}}_{\varepsilon} \Leftrightarrow \bot \vee \bot$$
$$\vee [(\mathcal{R}^{\mathsf{block}}_{\varepsilon} \wedge \mathit{null}(\mathsf{stmt})) \vee (\mathcal{R}^{\mathsf{block}}_{0,1} \wedge \mathcal{D}^{\mathsf{stmt}}_{0,1}) \vee (\mathcal{R}^{\mathsf{block}}_{0,2} \wedge \mathcal{D}^{\mathsf{stmt}}_{0,2}) \vee (\mathcal{R}^{\mathsf{block}}_{0,3} \wedge \mathcal{D}^{\mathsf{stmt}}_{0,3})$$
$$\vee (\mathcal{R}^{\mathsf{block}}_{1,1} \wedge \mathcal{D}^{\mathsf{stmt}}_{1,1}) \vee (\mathcal{R}^{\mathsf{block}}_{1,2} \wedge \mathcal{D}^{\mathsf{stmt}}_{1,2}) \vee (\mathcal{R}^{\mathsf{block}}_{2,1} \wedge \mathcal{D}^{\mathsf{stmt}}_{2,1})]$$
$$\vee \ldots$$

Its right-hand side encodes the possible conditions for $(\mathsf{block}, w^m) \to_* (\mathsf{block}, \varepsilon)$:

- the first disjunct is false ($\bot$) because $k = 3 \neq 0$;
- the second disjunct is false ($\bot$) because no unary production rule can produce block;

- the third disjunct encodes the possibilities using "block $\longrightarrow$ block $\|$ stmt", which is further divided into 7 sub-disjuncts: among them, the first one corresponds to the case when stmt is nullable ($\mathit{null}(\text{stmt})$); the rest requires $(\text{block}, w^m_{x,\delta})$ is reachable and stmt produces $w^m_{x,\delta}$ for each (x, δ) such that $0 < x + \delta \leq 3$ (in total 6 possibilities);
- there is one more disjunct encodes the possibilities using "r $\longrightarrow$ s block", which is similar to the above case (to avoid redundancy, this formula is elided).

Example 5.6. The formula $\Phi^{\not\emptyset}_R(3)$ iterates over the nonterminals and all pairs of x and δ such that $\delta > 0 \wedge x + \delta \leq 3$. One of them is the following:

$$\mathcal{R}^{\text{block}}_{0,3} \Leftrightarrow \top \vee (\mathcal{R}^{\text{block}}_{0,3} \wedge \mathit{null}(\text{stmt}) \wedge \top) \vee (\mathcal{R}^{\text{block}}_{0,3} \wedge \mathit{null}(\text{s}) \wedge \top).$$

Its right-hand side encodes the possible conditions for $(\text{block}, w^m) \longrightarrow_* (\text{block}, w^m_{0,3})$:

- the first disjunct is true ($\top$) because block is the start symbol, $x = 0$, and $\delta = 3$;
- the second disjunct encodes the possibility using "block $\longrightarrow$ block $\|$ stmt": stmt is nullable, $(\text{block}, w^m_{0,3})$ is reachable, and the alignment constraint trivially holds ($\top$) as stmt is nullable;
- the last disjunct encodes the possibility using "r $\longrightarrow$ s block", which is similar to the above case.

Encoding the existence of dissimilar parse trees. Our goal is to express "there exist two dissimilar parse trees that witness $H \Rightarrow_* w^m_{x,\delta}$" in an SMT formula $\Phi_{\text{multi}}(H, x, \delta)$. The key technical problem is how to encode the existence of two such dissimilar parse trees. By definition, to tell if two given parse trees are similar or not, it suffices to only compare their children of the root nodes[5]. Since every parse tree t is a visual representation of a derivation trace for $\mathit{root}(t) \Rightarrow_* \mathit{word}(t)$ (or a proof tree that evidences the validity of that derivation judgment), and that the root node with its children on the parse tree correspond to the *first step* of the trace, "being dissimilar" is essentially "having at least two derivation traces that *differ in the first step*".

Specifically, to make this distinction on two derivation traces for $H \Rightarrow_* w^m_{x,\delta}$, we use (in the first step) either two distinct production rules for H or one binary rule $H \longrightarrow B_1\ \varphi\ B_2$ in two distinct ways. For the latter case, each way splits $w^m_{x,\delta}$ into two parts where the prefix is derivable from B_1, the suffix is derivable from B_2, and the length of the two prefixes are different. Suppose we can compute the set of all possible choices for such first steps, the formula $\Phi_{\text{multi}}(H, x, \delta)$ simply states that "this set has at least two elements".

We start by defining the elements of this set—possible choices of the first step in deriving $H \Rightarrow_* w^m_{x,\delta}$. A compact *syntactic* representation could simply be a clause α, indicating the production rule $H \longrightarrow \alpha$ is chosen. In case a binary rule $H \longrightarrow B_1\, \varphi\, B_2$ is chosen, we further need to specify how $w^m_{x,\delta}$ gets split—an additional parameter, the prefix length, is enough. Thus, we introduce *choice clauses* with a similar syntax of (layout) clauses:

$$\gamma ::= \varepsilon \mid a \mid B^{\varphi} \mid B_1^{\delta'}\ \varphi\ B_2.$$

The only difference with clauses is that in the last case, the binary choice clause includes an additional parameter δ' ($0 \leq \delta' \leq \delta$), which denotes the aforementioned prefix length.

[5]This coincides with the intuitive explanation of being dissimilar—they differ on a node at level 1.

Semantically, a choice clause γ represents under which condition we can use γ as the first derivation step to eventually achieve the derivation $H \Rightarrow_* w^m_{x,\delta}$, denoted by $[\![\gamma]\!]_{x,\delta}$:

$$
\begin{aligned}
[\![\varepsilon]\!]_{x,\delta} &\triangleq \delta = 0 \\
[\![a]\!]_{x,\delta} &\triangleq \delta = 1 \wedge \mathcal{T}_x = a \\
[\![B^\varphi]\!]_{x,\delta} &\triangleq \mathit{ite}(\delta = 0, \mathit{null}(B), \mathcal{D}^B_{x,\delta} \wedge \Phi_\varphi(x,\delta)) \\
[\![B_1^{\delta'} \ \varphi \ B_2]\!]_{x,\delta} &\triangleq \mathit{ite}(\delta' = 0, \mathit{null}(B_1), \mathcal{D}^{B_1}_{x,\delta'}) \wedge \mathit{ite}(\delta = \delta', \mathit{null}(B_2), \mathcal{D}^{B_2}_{x+\delta',\delta-\delta'}) \wedge \Phi_\varphi(x,\delta',\delta)
\end{aligned}
$$

Note that all the above conditions are encodable in SMT formulae.

The set of all possible choices of the first step in deriving $H \Rightarrow_* w^m_{x,\delta}$ is given by

$$\{\gamma \mid \gamma \in \Gamma(H,\delta) \wedge [\![\gamma]\!]_{x,\delta}\},$$

where

$$
\begin{aligned}
\Gamma(H,\delta) \triangleq \{\epsilon \mid H \to \varepsilon \in P\} \cup \{a \mid H \to a \in P\} \cup \{B^\varphi \mid H \to B^\varphi \in P\} \\
\cup \{B_1^{\delta'} \ \varphi \ B_2 \mid H \to B_1 \ \varphi \ B_2 \in P, 0 \le \delta' \le \delta\}
\end{aligned}
$$

gives all choice clauses of H according to the production rules. With this, the formula $\Phi_{\text{multi}}(H,x,\delta)$ should be

$$|\{\gamma \mid \gamma \in \Gamma(H,\delta) \wedge [\![\gamma]\!]_{x,\delta}\}| \ge 2,$$

which is equivalent to the following form that uses $\mathit{Two}(\mathcal{P})$, a standard approach to encode "at least two of the propositions in the set $\mathcal{P}$ are true" in propositional logic:

$$\Phi_{\text{multi}}(H,x,\delta) \triangleq \mathit{Two}(\{[\![\gamma]\!]_{x,\delta}\}_{\gamma\in\Gamma(H,\delta)}).$$

Lemma 5.7. *If $m \models \Phi_D(k)$ and $x + \delta \le k$, then $m \models \Phi_{multi}(H,x,\delta)$ iff there exist two dissimilar parse trees that witness $H \Rightarrow_* w^m_{x,\delta}$.*

Example 5.8. Traversing the production rules, we obtain the set of possible choices for deriving $\text{stmt} \Rightarrow_* w^m_{0,2}$:

$$\Gamma(\text{stmt},2) = \varnothing \cup \{\text{nop}\} \cup \{\text{r}^{\triangleright}\} \cup \{\text{s}^0 \ \text{block}, \text{s}^1 \ \text{block}, \text{s}^2 \ \text{block}\}.$$

The formula $\Phi_{\text{multi}}(\text{stmt},0,2)$ is then $\mathit{Two}(\{[\![\gamma]\!]_{0,2}\}_{\gamma\in\Gamma(\text{stmt},2)})$ where the set of formulae are:

$$
\begin{aligned}
[\![\text{nop}]\!]_{0,2} &= \bot \wedge \mathcal{T}_0 = \text{nop} \\
[\![\text{r}^{\triangleright}]\!]_{0,2} &= \mathcal{D}^{\text{r}}_{0,2} \wedge \Phi_{\text{offside}}(0,2) \\
[\![\text{s}^0 \ \text{block}]\!]_{0,2} &= \top \wedge \mathit{null}(\text{s}) \wedge \mathcal{D}^{\text{block}}_{0,2} \\
[\![\text{s}^1 \ \text{block}]\!]_{0,2} &= \top \wedge \mathcal{D}^{\text{s}}_{0,1} \wedge \mathcal{D}^{\text{block}}_{1,1} \\
[\![\text{s}^2 \ \text{block}]\!]_{0,2} &= \top \wedge \mathcal{D}^{\text{s}}_{0,2} \wedge \mathit{null}(\text{block})
\end{aligned}
$$

5.3 Formal Properties

For any acyclic LS2NF and any length $k \ge 0$:

Theorem 5.9 (Soundness). *If $m \models \Phi(k)$, then $S \Rightarrow_* w^m$ is ambiguous.*

Proof Sketch. By Theorem 4.5 it suffices to show local ambiguity, which is straightforward by applying Lemmas 5.3, 5.4 and 5.7. □

Theorem 5.10 (Completeness). *If there exists a sentence w such that $|w| = k$ and $S \Rightarrow_* w$ is ambiguous, then $\Phi(k)$ is satisfiable.*

Proof Sketch. By Theorem 4.5 we have $(S, w) \to_* (H, h)$. We pick a model m where

$$
\begin{aligned}
w^m &= w \\
m(\mathcal{D}^A_{x,\delta}) &\Leftrightarrow A \Rightarrow_* w^m_{x,\delta} \\
m(\mathcal{R}^B_{\varepsilon}) &\Leftrightarrow (S, w) \to_* (B, \varepsilon) \\
m(\mathcal{R}^B_{x,\delta}) &\Leftrightarrow (S, w) \to_* (B, w^m_{x,\delta})
\end{aligned}
$$

By definition, $\Phi_D(k)$, $\Phi^{\varepsilon}_R(k)$ and $\Phi^{\notin}_R(k)$ are satisfiable. Given that $(S, w) \to_* (H, h)$, h must be a subsentence of w, say $h = w_{x_h,\delta_h}$ for some x_h, δ_h. By Lemma 5.7, $\Phi_{\text{multi}}(H, x_h, \delta_h)$ is satisfiable. □

Space Complexity. For every length k, $O(k^2)$ SMT variables are created, and for each variable, an enumeration loop similar to CYK [Younger 1967] creates $O(k)$ terms, where we also use $O(n)$ space to go through the n production rules. Thus, the space complexity for $\Phi(k)$ is $O(n \cdot k^3)$.

5.4 The Bounded Loop

Our bounded ambiguity checker is facilitated by a bounded loop where the bound is configurable. In the k-th iteration (initially $k = 1$), logical constraints for finding an ambiguous sentence with length k are encoded as an SMT formula $\Phi(k)$. We rely on a backend SMT solver (e.g., Z3) to check its satisfiability: if it is satisfiable under some model, say $m \models \Phi(k)$, then we recover an ambiguous sentence w^m from m (recall the auxiliary variables $\mathcal{T}_i$, $\mathcal{L}_i$, and C_i encode this ambiguous sentence), and the loop exits immediately; otherwise, we go to the next iteration until the bound is reached. In this way, the *shortest* nonempty ambiguous sentence is obtained. Additionally, since our encoding is complete, if our checker does not find any ambiguous sentence bounded by a certain length, then we have proved that the input grammar is unambiguous within this bound.

6 COQ MECHANIZATION

All the definitions and theorems presented in §4 and §5 were mechanized in the Coq proof assistant. Our proof consists of 10 Coq files and ~2 k lines of code (excluding comments and blanks). On a MacBook Pro with an Apple M1 chip and 16 GB memory, a complete verification took ~12 s.

There was only one axiom we made: in the type "`grammar Σ N`" for LS2NFs, the terminal set Σ is assumed to be *nonempty*, introduced by the type class constraint `!Inhabited Σ`. This makes sense because a language with an empty alphabet is trivially empty and thus not interesting at all, though, in theory, we could avoid this axiom (see our artifact for more details).

7 EVALUATION

We developed an ambiguity detector, called Lamb[6], based on the bounded ambiguity checking loop and SMT encodings described in §5. This tool was written in Python 3 and it used Z3 as the backend SMT solver via the PySMT library.

To examine the effectiveness of Lamb, we evaluated several grammars (and grammar fragments) and their variants selected from the language syntax specification of five popular programming languages used in various application fields, including the full grammar of Python and SASS.

7.1 Experimental Setup

Dataset. We studied the syntax specification of five widely used programming languages: Python, SASS, YAML, F# and Haskell.

[6]Meaning "Layout-sensitive ambiguity detector".

```
block-node → tokens | block-sequence | block-map
off-node → token | block-map
off0-node → block-sequence
tokens → token*
token → t
block-sequence → ||sequence-item||+
sequence-item → (- start)▷
block-map → key-val
key-val → explicit-key-val | implicit-key-val
explicit-key-val → explicit-key || explicit-val
explicit-key → off-explicit-key | off0-explicit-key
off-explicit-key → (? off-node)▷
off0-explicit-key → (? off0-node)⩥
explicit-val → off-explicit-val | off0-explicit-val
off-explicit-val → (: off-node)▷
off0-explicit-val → (: off0-node)⩥
implicit-key-val → off-implicit-key-val | off0-implicit-key-val
off-implicit-key-val → implicit-key off-node
off0-implicit-key-val → implicit-key off0-node
implicit-key → ⟪tokens :⟫
```

Fig. 4. The studied YAML grammar fragment (#3-0).

Python is self-described as a powerful yet easy-to-learn language. It is popular in the statistics and machine-learning community. Its indentation syntax has influenced the syntax design of many later languages such as Scala 3. SASS is self-described as "Syntactically Awesome Style Sheets" and has become the industrial standard in *cascading style sheets* (CSS) processors used in web development. A survey [Coyier 2012] shows that among all developers who use a CSS preprocessor, SASS took 41% of the market, being the second most popular option. As a preprocessor language, SASS has a rich feature set competitive to general-purpose programming languages, such as nested blocks, mixins, and control structures, which makes it Turing-complete. For these two languages, we extracted their *full grammars* from their language manuals as our dataset: the Python grammar (numbered #1-0) consists of 83 rules in EBNF and 1009 rules in (the desugared) LS2NF; the SASS grammar (numbered #2-0) consists of 83 rules in EBNF and 564 rules in LS2NF.

YAML is a human-readable serialization format for both describing configurations and exchanging data between systems/applications. In our dataset, we concentrate on YAML's layout-sensitive fragment (numbered #3-0). This fragment covers a rich set of layout-sensitive features and is conceptually complex due to its deeply nested lists and maps. This grammar is presented in Fig. 4; it consists of 21 rules in EBNF and 37 rules in LS2NF.

F# and Haskell are two popular functional languages that employ a rich set of layout-sensitive features including alignment, indentation and offside. Our dataset contains two small fragments

```
start → expr+
expr → l-expr | m-expr | e
bind → (let id = expr)▷
m-expr → m-with rules
m-with → match id with
rules → ‖(| id -> expr)▷‖+
```

Fig. 5. The studied F# grammar fragment (#4-0).

```
document → ‖stmt‖+
stmt → instance | valdef
valdef → (decl where?)▷
decl → ⟪e = e⟫
instance → (instance Eq a where)▷
e → id | id id
where → where ‖decl‖+
```

Fig. 6. The studied Haskell grammar fragment (#5-0).

of each: the F# fragment (numbered #4-0, Fig. 5) exhibits two typical features of ML languages—let-binding and pattern matching; and the Haskell fragment (numbered #5-0, Fig. 6) exhibits two "killer features" of Haskell—where-declaration and type classes.

Since the above five grammars were extracted from mature languages, it is very unlikely that they have ambiguity issues; indeed, we didn't detect any ambiguity among them. To better demonstrate Lamb's power of ambiguity detection, for each of the five original grammars (referred to *seeds*), we created four *variants* (numbered from #x-1 to #x-4 for the seed grammar #x-0) by randomly removing 3 – 5 layout constraints from the seed grammar. In total, our dataset consists of 25 grammars: 5 seed grammars and 20 variants. All of these grammars can be found in our open-source artifact linked at the end of this paper.

Method. In executing Lamb on each grammar, we set both (meaning the execution will be aborted when either condition is reached) a timeout of 1 h and an upper bound (i.e., sentence length) of 15. When an ambiguous sentence is found, we are interested in how users can benefit from it. For this, we manually inspect the ambiguous sentence along with all its parse trees, try to understand the cause of ambiguity, and add layout constraints to resolve it.

Environment. All experiments were conducted on a machine with Intel(R) Core(TM) i7-12700K CPU and 64 GB memory, running Ubuntu 22.04 and Python version 3.10.6.

7.2 Results

Table 1 presents the following metrics we collected:

- the SMT formulae construction times;
- the SMT formulae solving times;

Table 1. Overall evaluation results. In the last column, "–" means no ambiguous sentence was found.

#	Lang.	Formula construction time	Solving time	# LS2NF rule	Ambig. sentence length
#1-0	Python	1.5 min	timeout	1009	—
#1-1	Python	43.7 s	3452.5 s	1009	10
#1-2	Python	20.1 s	515.9 s	1009	7
#1-3	Python	15.1 s	236.8 s	1008	6
#1-4	Python	47.8 s	3135.3 s	1009	10
#2-0	SASS	68.8 s	timeout	695	—
#2-1	SASS	10.7 s	111.7 s	700	6
#2-2	SASS	38.4 s	2321.7 s	695	11
#2-3	SASS	48.7 s	2381.7 s	695	11
#2-4	SASS	12.9 s	214.2 s	695	7
#3-0	YAML	18.4 s	712 s	48	—
#3-1	YAML	50 ms	3 ms	46	2
#3-2	YAML	743 ms	294 ms	46	6
#3-3	YAML	587 ms	287 ms	46	6
#3-4	YAML	86 ms	14 ms	49	2
#4-0	F#	11 s	116 s	22	—
#4-1	F#	328 ms	97 ms	20	7
#4-2	F#	677 ms	151 ms	21	7
#4-3	F#	651 ms	171 ms	22	7
#4-4	F#	702 ms	197 ms	21	7
#5-0	Haskell	17.6 s	180.7 s	33	—
#5-1	Haskell	2.82 ms	2.54 ms	32	11
#5-2	Haskell	642 ms	332 ms	33	7
#5-3	Haskell	1.64 s	1.85 ms	32	10
#5-4	Haskell	1.65 ms	2.14 ms	32	10

- the numbers of LS2NF rules; and
- the lengths of ambiguous sentences (if found).

Above all, no ambiguous sentences were found in any of the seed grammars: Lamb timed out for the two full grammars (#1-0 and #2-0) and reached the maximum bound 15 for the three grammar fragments (#3-0, #4-0 and #5-0), indicating they are bounded unambiguous. Ambiguous sentences were found in every variant, with the ambiguous sentence lengths varying from 2 to 11. These results reveal that Lamb can apply to grammars of different sizes and can find ambiguous sentences within a relatively small bound.

Fig. 7. The ambiguous sentence found in grammar #3-2. Spaces are denoted as "␣".

Regarding the solving time, a larger grammar (in particular, a full grammar) and a longer ambiguous sentence lead to a greater amount of time cost. This is well-understood in SMT solving: the solver usually slows down as the complexity of the formula increases. Given that optimizing mature SMT solvers (like the Z3 solver we used) is extremely challenging these days, the only thing we could do is to *minimize* the complexity of the constructed SMT formulae: as mentioned in §5.3, the number of nodes in our SMT encoding is proportionate to $O(kn^3)$—where k is the bound and n is the number of LS2NF rules—and this is indeed the time complexity of the fastest parsing algorithm (namely CYK) we have ever known. To this end, we believe our encoding is efficient for bounded ambiguity detection.

7.3 Empirical Analysis of Ambiguity

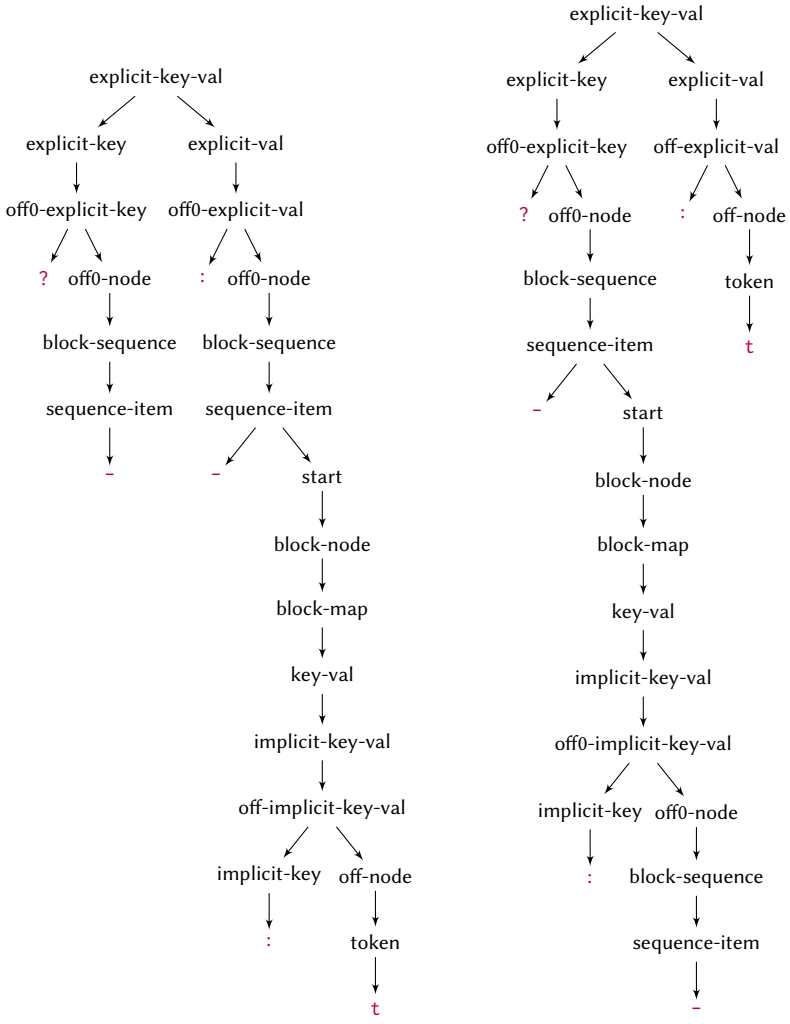
For grammars where ambiguous sentences were found, we are interested in how users can benefit from them. We conducted an empirical analysis of the parse trees of the ambiguous sentences and try to figure out the cause of the ambiguity and a possible way to resolve it. Let us consider grammar #3-2 as an example. This grammar is a variant of #3-0 (shown in Fig. 4); mutated production rules are listed below (in total four layout constraints were removed):

$$\begin{aligned}
\text{off-explicit-key} &\longrightarrow \texttt{?}\ (\text{off-node})^{\triangleright} \\
\text{off0-explicit-key} &\longrightarrow \texttt{?}\ \text{off0-node} \\
\text{off-explicit-val} &\longrightarrow \texttt{:}\ (\text{off-node})^{\triangleright} \\
\text{off0-explicit-val} &\longrightarrow \texttt{:}\ \text{off0-node}
\end{aligned}$$

Lamb found an ambiguous sentence of length 6 (shown in Fig. 7), and presented all of its two parse trees as depicted in Fig. 8. Note that our tool presents users with de-binarized parse trees that correspond to the input EBNF (rather than binary trees that correspond to the desugared LS2NF) for better readability. In the first parse tree (Fig. 8a), the mapping key is a one-element list, containing exactly a `null`. In the second parse tree (Fig. 8b), the mapping key is also a one-element list but its element is no longer `null`. To resolve this ambiguity issue, we added the offside-align constraint ($\cdot^{\triangleright}$) to the right-hand side of off0-explicit-key, off0-implicit-key-val and off0-explicit-val. In this way, the grammar becomes unambiguous.

The above process was repeated on every other ambiguous sentence that Lamb found. For all of them, we found it not hard to first understand the cause of ambiguity by inspecting the parse trees, and then figure out a set of additional layout constraints for disambiguating the grammar.

In practice, one may easily miss some layout constraints that are indeed necessary for unambiguity, particularly in large grammars. Hence, detecting potential ambiguity automatically is beneficial. As shown in Table 1, despite the large size of the Python grammar and its variants, Lamb was still able to find ambiguous sentences as short as six tokens. These shortest ambiguous sentences



(a) parse tree t_1 of #3-2 (b) parse tree t_2 of #3-2

Fig. 8. The parse trees of the ambiguous sentence found in grammar #3-2.

are undoubtedly more productive than a random (and usually longer) ambiguous sentence for the language designer to debug and fix design flaws.

8 RELATED WORK

Layout-sensitive languages & parsing. In 1966, Landin [1966] first introduced layout-sensitive languages, which influenced the syntax design of many later programming languages. As an extension to CFG, *Indentation-sensitive CFG* (ISCFG) [Adams 2013] can express layout rules and derive LR(k) and GLR algorithms for parsing these layout-sensitive grammars. In ISCFG, symbols are annotated with the numerical relation that the indentation of every nonterminal must have with that of its children. Although ISCFG has a formal theory and is parser-friendly, it takes too much effort for a human to manually specify those annotations.

In a declarative layout specification [Erdweg et al. 2013], layout constraints are expressed with primitives that support direct access to the position of a certain "border" of a code block. To realize layout-sensitive parsing, a naive approach is to generate every possible parse tree with GLR parsing first while neglecting layouts, and then filter with layout constraints. This approach is, however, inefficient for practical applications. To improve performance, they identify a subset of constraints that are independent of context-sensitive information and enforce them at parse time. Later, another version with high-level specification was proposed by Amorim et al. [2018]. Our grammar declarative specification notations are inspired by theirs. The difference lies in how we regard layout constraints: in their specification, constraints are attached to a normal layout-free production rule, whereas in ours, constraints are part of the rule.

Apart from generalized parsing, Iguana [Afroozeh and Izmaylova 2015, 2016] is a novel parsing framework based upon *data-dependent grammars* [Jim et al. 2010], which extends a CFG with arbitrary computation, variable binding and logical constraint. Iguana translates high-level declarations into equations that are expressive in data-dependent grammars.

Brzozowski derivatives [Brzozowski 1964] have been rediscovered to simplify the description of parsers. Brachthäuser et al. [2016] propose a new parser combinator library with first-class derivatives, gaining fine-grained control over an input stream. It is still an open question whether their framework can implement alignment and offside rules modularly, which, for example, are seen in Haskell's grammar.

Automata-based ambiguity detection. Basten [2016] proposes an approach of converting ambiguity detection problems into the search for an accepting path in the multi-stack pushdown automata constructed from the input grammar. However, this method cannot be directly extended to layout-sensitive grammars, because they are not expressible using PDAs (which are equivalent to context-free languages). Based on our knowledge, it is still an open question of how to build automata that are equivalent to layout-sensitive languages.

Random enumeration. Madhavan et al. [2015] propose a practical approach to check the equivalence of CFGs, based on random enumeration of parse trees and words. To generate words of a fixed length, they first transform the input CFG into a restricted CFG that can only derive words of the given length, and then apply the random enumeration to it. It is also possible to apply random enumeration techniques to the detection of ambiguous sentences. This can be done by filtering out the ambiguous ones in a set of randomly generated valid sentences. Vasudevan and Tratt [2013] utilized this approach by first generating valid parse trees that conform to the grammar, then unparsing each into a sentence, and finally, parsing each sentence again to see if it is ambiguous.

To apply this approach to layout-sensitive grammars, two issues have to be resolved: (1) how to generate random sentences that not only meet the syntax but also the layout constraints specified in the input grammar, and (2) how to produce all parse trees of a randomly-generated sentence. For (2), one could use an existing generalized LL parsing technique. But for (1), how to efficiently obtain line/column numbers that fulfill the layout constraints is not obvious. Random enumeration may not be a good idea for this step due to the low chances of hitting the right combination of line/column numbers. Using constraint solving to generate line/column numbers might perform better, but we have no idea how much overhead the solvers will incur since they may be invoked a lot of times. Overall, it seems to be difficult to extend this method to layout-sensitive grammars.

Grammar synthesis. Is it possible to generate a parser from examples? Mernik et al. [2003] raised this question and attempted genetic programming methods to answer it. In the recent decade, the research community has had significant interest in *programming by examples* [Gulwani 2011; Polozov and Gulwani 2015] and novel approaches have been proposed.

Parsify [Leung et al. 2015] is a graphical, interactive system for synthesizing and testing parsers from user-provided examples (a set of sentences). They rely on a GLL parser to identify ambiguous grammars and a few *disambiguation filters* [Klint and Visser 1994; Thorup 1996] to eliminate the well-known associatively and priority issues in grammars of binary expressions. They disambiguate in an interactive manner: possible parse trees are presented to the user, and only one of them is accepted. It is an interesting problem to study whether such an interactive manner applies to resolving the ambiguities in layout-sensitive grammars.

Glade [Bastani et al. 2017] is an oracle-based grammar inference system. The algorithm synthesizes a CFG that encodes the language of valid program inputs, beginning with a small set of the target language that the user provides as seed inputs. Although grammar synthesis is a promising direction that can ease the design of grammars and parsers, we still have not seen any work on the automated synthesis of layout-sensitive grammars and parsers.

Grammar-based fuzzing. To improve the test coverage on programs whose inputs are highly structured, like compilers and interpreters, the concept of grammar-based fuzzing [Zeller et al. 2019] has been studied to leverage user-defined grammars for generating syntactically valid inputs. Black-box fuzzing has been integrated with manually specified grammars to test C compilers [Lindig 2005; Yang et al. 2011], find bugs in PHP and JavaScript interpreters [Holler et al. 2012], and generate plausible inputs with the help of a parser merely [Mathis et al. 2019]. On white-box techniques, Godefroid et al. [2008] use a handwritten grammar in combination with a custom grammar-based constraint solver to fuzz a JavaScript interpreter. CESE [Majumdar and Xu 2007] combines exhaustive enumeration of valid inputs with symbolic execution. Interestingly, it is possible to consider extra constraints too (like in data-dependency grammars): from Boolean combinations of patterns [Gopinath et al. 2021] to first-order logic [Steinhöfel and Zeller 2022]. In comparison, our goal is to find an ambiguous sentence not just a random one.

9 CONCLUSION AND FUTURE DIRECTIONS

Detecting ambiguity in layout-sensitive grammars is an important problem in language design. Traditionally, language designers have to check ambiguity by hand, which is tedious and error-prone. To ease this process, this paper presents the *first* approach (to the best of our knowledge) that can *automatically* detect ambiguous sentences via SMT solving. The generated ambiguous sentences are the shortest ones, which makes it more productive for language designers to debug and fix design flaws. Our evaluation demonstrates the effectiveness of identifying ambiguous sentences in real-world grammar variants and the ability to help users resolve such ambiguities.

In the future, we are interested in investigating whether it is possible to further automate the process of adding layout constraints to disambiguate layout-sensitive grammars via user interaction, based on the produced ambiguous sentences and parse trees. Moreover, it is interesting to see how ambiguity detection could benefit data-dependency grammar users.

DATA-AVAILABILITY STATEMENT

The software that supports §4 – §7 is available on Software Heritage [Zhu and Liu 2023a] and Zenodo [Zhu and Liu 2023b].

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China (Grant No. 62072267 and Grant No. 62021002).

REFERENCES

Michael D. Adams. 2013. Principled Parsing for Indentation-sensitive Languages: Revisiting Landin's Offside Rule. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. ACM, New York, NY, USA, 511–522. https://doi.org/10.1145/2429069.2429129

Ali Afroozeh and Anastasia Izmaylova. 2015. One Parser to Rule Them All. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (Pittsburgh, PA, USA) *(Onward! 2015)*. ACM, New York, NY, USA, 151–170. https://doi.org/10.1145/2814228.2814242

Ali Afroozeh and Anastasia Izmaylova. 2016. Iguana: A Practical Data-dependent Parsing Framework. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. ACM, New York, NY, USA, 267–268. https://doi.org/10.1145/2892208.2892234

Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers: Principles, Techniques, and Tools. *Addison wesley* 7, 8 (1986), 9.

Luís Eduardo de Souza Amorim, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. 2018. Declarative Specification of Indentation Rules: A Tooling Perspective on Parsing and Pretty-printing Layout-sensitive Languages. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering* (Boston, MA, USA) *(SLE 2018)*. ACM, New York, NY, USA, 3–15. https://doi.org/10.1145/3276604.3276607

Roland Axelsson, Keijo Heljanko, and Martin Lange. 2008. Analyzing Context-Free Grammars Using an Incremental SAT Solver. In *Automata, Languages and Programming*, Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 410–422.

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 95–110. https://doi.org/10.1145/3062341.3062349

Hendrikus J. S. Basten. 2016. Context-Free Ambiguity Detection Using Multi-stack Pushdown Automata. In *Developments in Language Theory - 20th International Conference, DLT 2016, Montréal, Canada, July 25-28, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9840)*, Srecko Brlek and Christophe Reutenauer (Eds.). Springer, 1–12. https://doi.org/10.1007/978-3-662-53132-7_1

Jonathan Immanuel Brachthäuser, Tillmann Rendel, and Klaus Ostermann. 2016. Parsing with First-class Derivatives. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. ACM, New York, NY, USA, 588–606. https://doi.org/10.1145/2983990.2984026

Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (Oct. 1964), 481–494. https://doi.org/10.1145/321239.321249

N Chomsky and MP Schützenberger. 1963. The Algebraic Theory of Context-Free Languages. In *Studies in Logic and the Foundations of Mathematics*. Vol. 35. Elsevier, 118–161.

Chris Coyier. 2012. https://css-tricks.com/poll-results-popularity-of-css-preprocessors/

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2013. Layout-Sensitive Generalized Parsing. In *Software Language Engineering*, Krzysztof Czarnecki and Görel Hedin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 244–263.

Clark C Evans et al. 2014. Yaml: Yaml ain't markup language.

Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. ACM, New York, NY, USA, 206–215. https://doi.org/10.1145/1375581.1375607

Rahul Gopinath, Hamed Nemati, and Andreas Zeller. 2021. Input Algebras. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 699–710. https://doi.org/10.1109/ICSE43902.2021.00070

John Gruber. 2012. Markdown: Syntax. *URL http://daringfireball.net/projects/markdown/syntax. Retrieved on June* 24 (2012), 640.

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. ACM, 317–330. https://doi.org/10.1145/1926385.1926423

Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Bellevue, WA) *(Security'12)*. USENIX Association, Berkeley, CA, USA, 38–38. http://dl.acm.org/citation.cfm?id=2362793.2362831

John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News* 32, 1 (2001), 60–65.

Trevor Jim, Yitzhak Mandelbaum, and David Walker. 2010. Semantics and Algorithms for Data-dependent Grammars. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid,

Spain) *(POPL '10)*. ACM, New York, NY, USA, 417–430. https://doi.org/10.1145/1706299.1706347

Paul Klint and Eelco Visser. 1994. Using filters for the disambiguation of context-free grammars. In *Proc. ASMICS Workshop on Parsing Theory*. Citeseer, 1–20.

Donald E Knuth. 1965. On the translation of languages from left to right. *Information and control* 8, 6 (1965), 607–639.

P. J. Landin. 1966. The Next 700 Programming Languages. *Commun. ACM* 9, 3 (March 1966), 157–166. https://doi.org/10.1145/365230.365257

Martin Lange and Hans Leiß. 2009. To CNF or not to CNF? An efficient yet presentable version of the CYK algorithm. *Informatica Didactica* 8, 2009 (2009), 1–21.

Alan Leung, John Sarracino, and Sorin Lerner. 2015. Interactive Parser Synthesis by Example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 565–574. https://doi.org/10.1145/2737924.2738002

Christian Lindig. 2005. Random Testing of C Calling Conventions. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging* (Monterey, California, USA) *(AADEBUG'05)*. ACM, New York, NY, USA, 3–12. https://doi.org/10.1145/1085130.1085132

Ravichandhran Madhavan, Mikaël Mayer, Sumit Gulwani, and Viktor Kuncak. 2015. Automating Grammar Comparison. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 183–200. https://doi.org/10.1145/2814270.2814304

Rupak Majumdar and Ru-Gang Xu. 2007. Directed Test Generation Using Symbolic Grammars. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, Georgia, USA) *(ASE '07)*. ACM, New York, NY, USA, 134–143. https://doi.org/10.1145/1321631.1321653

Simon Marlow et al. 2010. Haskell 2010 language report. *Available online http://www.haskell.org/(May 2011)* (2010).

Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-directed Fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 548–560. https://doi.org/10.1145/3314221.3314651

Marjan Mernik, Goran Gerlič, Viljem Žumer, and Barrett R. Bryant. 2003. Can a Parser Be Generated from Examples?. In *Proceedings of the 2003 ACM Symposium on Applied Computing* (Melbourne, Florida) *(SAC '03)*. ACM, New York, NY, USA, 1063–1067. https://doi.org/10.1145/952532.952740

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices* 50, 10 (2015), 107–126.

Dominic Steinhöfel and Andreas Zeller. 2022. Input Invariants. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 583–594. https://doi.org/10.1145/3540250.3549139

Don Syme, Luke Hoban, Tao Liu, Dmitry Lomov, James Margetson, Brian McNamara, Joe Pamer, Penny Orwick, Daniel Quirk, Chris Smith, et al. 2010. The F# 2.0 language specification. *Microsoft, August* (2010).

Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

Mikkel Thorup. 1996. Disambiguating grammars by exclusion of sub-parse trees. *Acta Informatica* 33, 6 (1996), 511–522.

Guido Van Rossum and Fred L Drake. 2011. *The python language reference manual*. Network Theory Ltd.

Naveneetha Vasudevan and Laurence Tratt. 2013. Detecting Ambiguity in Programming Language Grammars. In *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8225)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Springer, 157–176. https://doi.org/10.1007/978-3-319-02654-1_9

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

Daniel H Younger. 1967. Recognition and parsing of context-free languages in time n3. *Information and control* 10, 2 (1967), 189–208.

Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. The fuzzing book.

Fengmin Zhu and Jiangyi Liu. 2023a. *Artifact of paper "Automated Ambiguity Detection in Layout-Sensitive Grammars"*. https://archive.softwareheritage.org/swh:1:rev:6a08a4b9a6321aeb44d5bde19c1a62dd4e5fd2a2;origin=https://github.com/lay-it-out/OOPSLA23-Artifact;visit=swh:1:snp:2135b15883c8028549ce5a460c745b782bdc2544

Fengmin Zhu and Jiangyi Liu. 2023b. *Artifact of paper "Automated Ambiguity Detection in Layout-Sensitive Grammars"*. https://doi.org/10.5281/zenodo.8329981

Received 2023-04-14; accepted 2023-08-27