Resolvable Ambiguity

ANONYMOUS AUTHOR(S)

Text of abstract ....

Additional Key Words and Phrases: keyword1, keyword2, keyword3

1 INTRODUCTION

Text of paper ...

1.1 Motivating Ambiguity in Programming Languages

Consider the following nested match expression in OCaml:

```
match 1 with
  | 1 -> match "one" with
         | str -> str
  | 2 -> "two"
```

The OCaml compiler, when presented with this code, will give a type error for the last line:

```
Error: This pattern matches values of type int
       but a pattern was expected which matches
       values of type string
```

The compiler sees the last line as belonging to the inner **match** rather than the outer, as was intended. The fix is simple; put parentheses around the inner match:

```
match 1 with
  | 1 -> (match "one" with
          | str -> str)
  | 2 -> "two"
```

The connection between the error message and the fix is not a clear one however; adding parentheses around an expression does not change the type of anything.

To come up with an alternative error to present in this case we look to the OCaml manual for inspiration. It contains an informal description of the syntax of the language[1], in the form of an EBNF-like grammar. Below is an excerpt of the productions for expressions, written in a more standard variant of EBNF:

⟨*expr*⟩ ::= 'match' ⟨*expr*⟩ 'with' ⟨*pattern-matching*⟩

⟨*pattern-matching*⟩ ::= ('|' ⟨*pattern*⟩ '->' ⟨*expr*⟩)+

[1] https://caml.inria.fr/pub/docs/manual-ocaml/language.html

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

Note that ⟨*pattern-matching*⟩ is slightly simplified, the original grammar supports **when** guards and makes the first '|' optional. If we use this grammar to parse the nested match we find an ambiguity: the last match arm can belong to either the inner match or the outer match. The OCaml compiler makes an arbitrary choice to remove the ambiguity, which may or may not be the alternative the user intended.

We instead argue that the grammar should be left ambiguous for this sort of corner cases that are likely to trip a user, allowing the compiler to present an ambiguity error, which lets the user select the intended alternative.

1.2 Unresolvable Ambiguity

Unfortunately, not all ambiguities can be resolved by adding parentheses. Again, looking to the informal OCaml grammar:

⟨*expr*⟩ ::= ⟨*expr*⟩ ';' ⟨*expr*⟩
| '[' ⟨*expr*⟩ (';' ⟨*expr*⟩)* ';'? ']'
| ⟨*constant*⟩

The first production is sequential composition, the second is lists (the empty list is under ⟨*constant*⟩). Now consider the following expression: "`[1; 2]`".

We find that it is ambiguous with two alternatives:

(1) A list with two elements.
(2) A list with one element, namely a sequential composition.

We can select the second option by putting parentheses around "`1; 2`", but there is no way to select the first. If the user intended the first option we have a problem: we can present an accurate error message, but there is no way for an end-user to solve it; it requires changes to the grammar itself.

To prevent the possibility of an end-user encountering such an error we must ensure that the grammar cannot give rise to an unresolvable ambiguity. It is worth mentioning here that statically checking if a context-free grammar is ambiguous has long been known to be undecidable [Cantor [n. d.]]. Unresolvable ambiguity, however, turns out to be decidable[2].

Note that, as for ambiguity, the shape of the grammar is important, since the property considers parse trees rather than merely words. For this paper, we consider context-free grammars with EBNF operators.

1.3 Contributions

- Building on the work by Palmkvist and Broman [[n. d.]], which merely isolates ambiguities, an algorithm that suggests solutions to ambiguity errors.
- A formalization of the unresolvable ambiguity property for context-free EBNF grammars.
- An algorithm for deciding if a grammar is unresolvably ambiguous or not.

2 PRELIMINARIES

This section briefly describes the theoretical foundations we build upon. Sections 2.3 and 2.5 describe context-free grammars and various forms of automata, the latter with some non-standard notation to make later sections easier to read. Section 2.6 then describes visibly pushdown languages, which enable the analyses described in Sections 5 and 6. Finally, Sections 2.7 and 2.4 describe trees and ambiguity, the former with some non-standard notation and the latter with a slightly wider definition than normal.

[2]With some caveats, I'll talk more about this during the meeting.

2.1 Bags

A *bag* (alternatively known as a *multiset*) is a generalization of a set, it is an unordered collection where each element may appear multiple times. The number of times an element a appears in a bag B is called the *multiplicity* of a in B and is written $m_B(a)$. A bag A is included in another bag B, written $A \subseteq B$, iff $\forall x.\ m_A(x) \leq m_B(x)$. The underlying set of elements of a bag B is called the *support* of B, and is given by $Supp(B) = \{x \mid m_B(x) > 0\}$.

2.2 Regular Expressions

A regular expression r over alphabet Σ is defined inductively:

$$r ::= \epsilon \mid a \mid r \cdot r \mid r + r \mid r^*$$

where $a \in \Sigma$. The language of a regular expression r is given by $L(r)$:

$$\begin{aligned} L(t) &= \{a\} \\ L(\epsilon) &= \{\epsilon\} \\ L(r_1 \cdot r_2) &= \{w_1 \cdot w_2 \mid w_1 \in L(r_1), w_2 \in L(r_2)\} \\ L(r_1 + r_2) &= L(r_1) \cup L(r_2) \\ L(r^*) &= L(\epsilon + r \cdot r^*) \end{aligned}$$

We will denote the set of regular expressions with alphabet Σ as $Reg(\Sigma)$.

2.3 Context-Free Grammars

A context-free grammar (CFG) G is a 4-tuple (V, Σ, P, S) where V is a set of non-terminals; Σ a set of terminals, disjoint from V; P a finite subset of $V \times (V \cup \Sigma)^*$[3], i.e., a set of productions; and $S \in V$ the starting non-terminal.

A word $w \in \Sigma^*$ is recognized by G if there is a sequence of steps starting with S and ending with w, where each step replaces a single non-terminal using a production in P. Such a sequence is called a *derivation*. The set of words recognized by G is written $L(G)$[4].

2.4 Ambiguity

The standard definition of ambiguity, given a context-free grammar G, is expressed in terms of *left-most derivations*. A left-most derivation is a derivation where the non-terminal being replaced is always the left-most one.

Definition 2.1. A word $w \in L(G)$ is ambiguous if there are two distinct left-most derivations of that word.

2.5 Automata

A nondeterministic finite automaton (NFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states; Σ a finite set of terminals; δ a transition function from $Q \times \Sigma$ to finite subsets of Q ; $q_0 \in Q$ an initial state; and $F \subseteq Q$ a set of final states.

A successful run is a sequence of states $r_0, \ldots, r_n$ and a word $a_0 \cdots a_n$ such that:

- $r_0 = q_0$.
- $\forall i \in \{0, 1, \ldots, n-1\}.\ r_{i+1} \in \delta(r_i, a_i)$.
- $r_n \in F$.

[3]Where * is Kleene-star.

[4]We will always name a regular expression r (possibly with a subscript) and a CFG G (possibly with a subscript), to lessen the risk of confusion

We say that the automaton accepts the word $a_0 a_1 \dots a_n$ iff there is such a succesful run.

A deterministic finite automaton (DFA) has the same definition, except $\delta : Q \times \Sigma \to Q$, i.e., given a state and a symbol there is always a single state we can transition to. NFAs and DFAs have the same expressive power as regular expressions, i.e., for every regular expression there is an NFA and a DFA both reconizing the same language, and vice-versa.

A pushdown automaton extends a finite automaton with a stack from which transitions can push or pop symbols. Formally, a nondeterministic pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q is a finite set of states; Σ a finite set of input symbols, i.e., an input alphabet; Γ a finite set of stack symbols, i.e., a stack alphabet; δ a transition function from $Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{\lambda\}$ to finite subsets of $Q \times (\Gamma \cup \{\lambda\}$; $q_0 \in Q$ the initial state; and $F \subseteq Q$ a set of final states. λ essentially means "ignore", i.e., $\delta(q_1, \lambda, \lambda) = \{(q_2, \lambda)\}$ means: transition from state q_1 without consuming an input symbol (the first λ) and without examining or popping from the current stack (the second λ), to state q_2 without pushing a new symbol on the stack (the third λ).

A successful run is now a sequence of *configurations*, elements of $Q \times \Gamma^*$, starting with (q_0, ϵ), ending with (f, γ) for some $f \in F$ and $\gamma \in \Gamma^*$.

However, in this paper we will only consider pushdown automata with relatively limited stack manipulation, and will thus use some convenient shorthand:

- $p \xrightarrow{a} q$, a transition that recognizes the terminal a and does not interact with the stack at all, i.e., $\delta(p, a, \lambda) \supseteq \{(q, \lambda)\}$.
- $p \xrightarrow{a,+g} q$, a transition that recognizes the terminal a and pushes the symbol g on the stack, i.e., $\delta(p, a, \lambda) \supseteq \{(q, g)\}$.
- $p \xrightarrow{a,-g} q$, a transition that recognizes the terminal a and pops the symbol g from the stack, i.e., $\delta(p, a, g) \supseteq \{(q, \lambda)\}$.

2.6 Visibly Pushdown Languages

A visibly pushdown language [Alur and Madhusudan [n. d.]] is a language that can be recognized by a visibly pushdown automaton. A visibly pushdown automaton is a pushdown automaton where the input alphabet Σ can be partitioned into three disjoint sets Σ_c, Σ_i, and Σ_r, such that all transitions in the automaton has one of the following three forms:

- $p \xrightarrow{c,+s} q$, where $c \in \Sigma_c$ and $s \in \Gamma$.
- $p \xrightarrow{i} q$, where $i \in \Sigma_i$.
- $p \xrightarrow{r,-s} q$, where $r \in \Sigma_r$ and $s \in \Gamma$.

i.e., the terminal recognized by a transition fully determines the change to the stack height.

The partition names stem from their original use in program analysis, c is for *call*, i for *internal*, and r for *return*. We will instead primarily use Σ_c and Σ_r for balanced parentheses.

This partitioning gives us some useful properties. Of particular relevance to this paper are the following two points:

- Visibly pushdown languages with the same input partitions are closed under intersection, complement, and union [Alur and Madhusudan [n. d.]]. Intersection in particular is given by a product automaton, i.e., given a pair of VPDAs $(Q_1, \Sigma, \delta_1, q_0, F_1)$ and $(Q_2, \Sigma, \delta_2, q_0', F_2)$ their product automaton has the form $(Q_1 \times Q_2, \Sigma, \delta', (q_0, q_0'), F_1 \times F_2)$ where:

$$\begin{aligned}
\delta'((p_1, p_2), c, \lambda) &= \{((q_1, q_2), (g_1, g_2)) \mid (q_1, g_1) \in \delta_1(p_1, c, \lambda), (q_2, g_2) \in \delta_2(p_2, c, \lambda)\} && \text{where } c \in \Sigma_c \\
\delta'((p_1, p_2), i, \lambda) &= \{((q_1, q_2), \lambda) \mid (q_1, \lambda) \in \delta_1(p_1, i, \lambda), (q_2, \lambda) \in \delta_2(p_2, i, \lambda)\} && \text{where } i \in \Sigma_i \\
\delta'((p_1, p_2), r, (g_1, g_2)) &= \{((q_1, q_2), \lambda) \mid (q_1, \lambda) \in \delta_1(p_1, r, \lambda), (q_2, \lambda) \in \delta_2(p_2, r, \lambda)\} && \text{where } r \in \Sigma_r
\end{aligned}$$

- A visibly pushdown automaton can be trimmed [Caralp et al. [n. d.]], i.e., modified in such a way that all remaining states and transitions are part of at least one successful run, none are redundant. Furthermore, a successful run in the trimmed automaton corresponds to exactly one successful run in the original automaton, and vice-versa.

2.7 Unranked Regular Tree Grammars

Trees generalize words by allowing each terminal to have multiple ordered successors, instead of just zero or one. Most literature considers *ranked* tree languages, where each terminal has a fixed arity, i.e., the same terminal must always have the same number of successors. This is as opposed to *unranked* tree languages, where the arity of a terminal is not fixed. The sequence of successors to a single terminal in an unranked tree tends to be described by a word language (referred to as a horizontal language in [Comon et al. [n. d.]]), often a regular language.

The results and properties presented in this paper are more naturally described through unranked trees, thus all further references to trees are to unranked trees, despite ranked being more common in the literature. We further distinguish terminals used solely as leaves from terminals that may be either nodes or leaves. Since we will use unranked trees to represent parse trees, the former will represent terminals from the parsed word, while the latter represent terminals introduced as internal nodes.

An unranked tree grammar T is a tuple (V, Σ, X, P, S) where:

- V is a set of (zero-arity) non-terminals.
- Σ is a set of zero-arity terminals, used as leaves.
- X is a set of terminals without fixed arity, used as inner nodes or leaves.
- P is a set of productions, a finite subset of $V \times X \times Reg(\Sigma \cup X)$. We will write a production (N, x, r) as $N \longrightarrow x(r)$.

A tree t (containing only terminals from Σ and X) is recognized by T if there is a sequence of steps starting with S and ending with t, where each step either replaces a single non-terminal using a production in P, or replaces a regular expression r with a sequence in $L(r)$. The set of trees recognized by T is written $L(T)$[5].

Finally, $yield : L(T) \longrightarrow \Sigma^*$ is the sequence of terminals $a \in \Sigma$ obtained by a left-to-right[6] traversal of a tree. Informally, it is the flattening of a tree after all internal nodes have been removed.

3 RESOLVABLE AMBIGUITY

This section introduces our definition of *resolvable ambiguity*, and then relates it to some more standard concepts in languages.

Normally, we define a language as a set of words, i.e., a purely syntactical definition, but here we additionally require a *meaning* of each word, in some sense. We will call this meaning an *interpretation*[7] of a word. Note that a single word may have multiple interpretations. As such, we will define an interpreted language as:

- An alphabet Σ.
- A set of interpretations T. Typically, these will be abstract syntax trees.
- A function $parse : \Sigma^* \longrightarrow 2^T$ that relates words to their interpretations, where 2^T denotes the powerset of T. A word $w \in \Sigma^*$ is in the language if $parse(w) \neq \emptyset$.

[5] Again, to distinguish from regular expressions and context-free languages, all trees will be named T, possibly with a subscript.

[6] Preorder, postorder, or inorder does not matter since terminals in Σ only appear as leaves

[7] Not to be confused with the more common use of the word "interpreter": a program that runs other programs.

Definition 3.1. A word $w \in \Sigma^*$ in an interpreted language given by $parse : \Sigma^* \to 2^T$ is ambiguous if $\exists t_1, t_2 \in parse(w).\ t_1 \neq t_2$.

We can see that this definition agrees rather nicely with the definition of ambiguity for context-free grammars by choosing *parse* to be the function that associates words with their left-most derivations for some particular grammar.

The opposite, an *unambiguous* word, is a word that is in the language, but is not ambiguous. Notably, this is not quite the negation of Definition 3.1, since "in the language" requires that $parse(w) \neq \emptyset$. Thus:

Definition 3.2. A word $w \in \Sigma^*$ in an interpreted language given by $parse : \Sigma^* \to 2^T$ is unambiguous if $\neg\exists t_1, t_2 \in parse(w).\ t_1 \neq t_2$. Equivalently, w is unambiguous iff $\exists t \in T.\ parse(w) = \{t\}$.

A *resolvably* ambiguous word is a word where all its interpretations can be written in an unambiguous way, or more formally:

Definition 3.3. A word $w \in \Sigma^*$ in an interpreted language given by $parse : \Sigma^* \to 2^T$ is resolvably ambiguous if $\forall t \in parse(w).\ \exists w' \in \Sigma^*.\ parse(w') = \{t\}$.

We can now immediately state a few things:

- An unambigous word w is trivially resolvably ambiguous, since its only interpretation t can be written unambiguously with w itself ($parse(w) = \{t\}$). The set of resolvably ambiguous words is thus a superset of the unambiguous words.
- An interpreted language given by an ambiguous context-free grammar is unresolvably ambiguous, since two distinct words must have distinct left-most derivations. In general, $\forall t \in T.\ |\{w \mid t \in parse(w)\}| \leq 1$ implies that a word is unresolvably ambiguous iff it is ambiguous.
- If $\forall t \in T.\ \exists w \in \Sigma^*.\ parse(w) = \{t\}$ then all words are resolvable. Furthermore, if $T = \bigcup_{w \in \Sigma^*} parse(w)$ then the two statements are equivalent.

The second point suggests that resolvable ambiguity is only an interesting property if an element of T does not uniquely identify an element of Σ^*. Intuitively, this only happens if *parse* discards some information present in its argument when constructing an individual interpretation. Fortunately, this is generally true for parsing in commonly used programming languages; they tend to discard, e.g., grouping parentheses and whitespace. In general, whatever information *parse* discards can be used by an end-user to disambiguate an ambiguity encountered at parse-time, without changing the interpretation.

We thus propose to loosen the common "no ambiguity" restriction on programming language grammars, and instead only require them to be resolvably ambiguous. However, merely having an arbitrary function *parse* gives us very little to work with, and no way to decide whether the language it defines is resolvably ambiguous or not. The remainder of this paper will thus consider *parse* functions defined with a particular formalism, introduced in Section 4, that gives us some decidable properties.

4 PARSE-TIME DISAMBIGUATION

This section describes our chosen language definition formalism, and motivates its design.

The primary purpose of this formalism is, as describen in the previous section, to produce a *parse* function, i.e., to describe a word language and assign one or more interpretations to each word. The interpretations will be unranked trees, intended to be somewhat reminiscent of the abstract syntax trees used in most compilers. Section 3 suggests that *parse* discard some information, to enable the resolution of some ambiguities; we here choose to discard grouping parentheses.

Terminals	$t \in \Sigma$
Non-terminals	$N \in V$
Labels	$l \in L$
Marks	$m \subseteq L$
Regular expressions	$r ::= t \mid N_m \mid r \cdot r \mid r + r \mid \epsilon \mid r^*$
Labelled productions	$N \longrightarrow l : r$

Fig. 1. The abstract syntax of a language definition.

$$
\begin{array}{lcll}
E & \longrightarrow & l: & \texttt{'['}\ (E\ (\texttt{';'}\ E)^* + \epsilon)\ \texttt{']'} \\
E & \longrightarrow & a: & E\ \texttt{'+'}\ E \\
E & \longrightarrow & m: & E_{\{a\}}\ \texttt{'*'}\ E_{\{a\}} \\
E & \longrightarrow & n: & N
\end{array}
$$

Fig. 2. The input language definition used as a running example, an expression language with lists, addition, and multiplication, with precedence defined, but not associativity. Assumes that N matches a numeric terminal.

With that in mind, we define a language definition D as a set of labelled productions, as described in Figure 1. Note that we require the labels to uniquely identify the production, i.e., there can be no two distinct productions in D that share the same label. Also note that the right-hand side of a production is here a regular expression, rather than the theoretically simpler sequence used in a context-free grammar. Each non-terminal appearing in the regular expression of a production carry a *mark* m, which is a set of labels whose productions may *not* replace that non-terminal. To lessen clutter, we will write $E_\emptyset$ as E. As an example, consider the language definition in Figure 2, which will be used as a running example. In the production describing multiplication (m) both non-terminals are marked with $\{a\}$, which thus forbids addition from being a direct child of a multiplication. By "direct child" we mean "without an intermediate node", most commonly grouping parentheses, thus this enforces conventional precedence.

From D we then generate four grammars: G_D, T_D, G'_D, and T'_D. Technically, only G'_D and T_D are required, G'_D is used as the defined word language and T_D as the interpretations, but the remaining two grammars help the presentation.

- G_D represents a word language describing all semantically distinct programs.
- T_D represents a tree language describing the parse trees of words in $L(G_D)$.
- G'_D is essentially a modified version of G_D, e.g., adding parentheses and other forms of disambiguation (i.e., the result of marks).
- T'_D represents a tree language describing the parse trees of words in $L(G'_D)$.

Figure 3 contains the four grammars generated from our running example in Figure 2. The context-free grammars are produced by a rather standard translation from regular expressions to CFGs, while the primed grammars get a new non-terminal per distinctly marked non-terminal in D, where each new non-terminal only has the productions whose label is not in the mark. For example, the non-terminal $E_{\{a\}}$ in Figure 3b has no production corresponding to the a production in Figure 2.

Examples of elements in each of these four languages can be seen in Figure 4, along with visualizations of the syntax trees. Each element corresponds to the word "$(1 + 2) * 3$" in $L(G'_D)$. Note that the word in $L(G_D)$ as ambiguous, and that there are other words in $L(G'_D)$ that correspond to the same element in $L(T_D)$, e.g., "$((1 + 2)) * 3$" and "$(1 + 2) * (3)$". As a memory aid, the prime versions (G'_D and T'_D) contain disambiguation (grouping parentheses, precedence, associativity, etc.) while the unprimed versions (G_D and T_D) are the (likely ambiguous) straightforward translations from D. We will generally refer to elements of $L(G_D)$ as w, $L(G'_D)$ as w', $L(T_D)$ as t, and $L(T'_D)$ as t'.

At this point we also note that the shape of D determines where the final concrete syntax permits grouping parentheses; they are allowed exactly where they would surround a complete production. For example, G_D in Figure 3c can be seen as a valid language definition (if we generate new unique labels for each of the productions). However, starting with that language definition would allow the

$$
\begin{array}{lcl}
E & \to & l(\ \texttt{'['}\ (\epsilon + E(\texttt{';'}\ E)^*)\ \texttt{']'}\ ) \\
E & \to & a(\ E\ \texttt{'+'}\ E\ ) \\
E & \to & m(\ E\ \texttt{'*'}\ E\ ) \\
E & \to & n(\ N\ )
\end{array}
$$

(a) T_D, the parse trees of G_D.

$$
\begin{array}{lcl}
E & \to & l(\ \texttt{'['}\ (\epsilon + E(\texttt{';'}\ E)^*)\ \texttt{']'}\ ) \\
E & \to & a(\ E\ \texttt{'+'}\ E\ ) \\
E & \to & m(\ E_{\{a\}}\ \texttt{'*'}\ E_{\{a\}}\ ) \\
E & \to & n(\ N\ ) \\
E & \to & g(\ \texttt{'('}\ E\ \texttt{')'}\ ) \\
\hline
E_{\{a\}} & \to & l(\ \texttt{'['}\ (\epsilon + E(\texttt{';'}\ E)^*)\ \texttt{']'}\ ) \\
E_{\{a\}} & \to & m(\ E_{\{a\}}\ \texttt{'*'}\ E_{\{a\}}\ ) \\
E_{\{a\}} & \to & n(\ N\ ) \\
E_{\{a\}} & \to & g(\ \texttt{'('}\ E\ \texttt{')'}\ )
\end{array}
$$

(b) T'_D, the parse trees of G'_D.

$$
\begin{array}{lcl}
E & \to & \texttt{'['}\ E_{l1}\ \texttt{']'} \\
E & \to & E\ \texttt{'+'}\ E \\
E & \to & E\ \texttt{'*'}\ E \\
E & \to & N \\
\hline
E_{l1} & \to & \epsilon \\
E_{l1} & \to & E\ E_{l2} \\
\hline
E_{l2} & \to & \epsilon \\
E_{l2} & \to & \texttt{';'}\ E\ E_{l2}
\end{array}
$$

(c) G_D, the generated abstract syntax.

$$
\begin{array}{lcl}
E & \to & \texttt{'['}\ E_{l1}\ \texttt{']'} \\
E & \to & E\ \texttt{'+'}\ E \\
E & \to & E_{\{a\}}\ \texttt{'*'}\ E_{\{a\}} \\
E & \to & N \\
E & \to & \texttt{'('}\ E\ \texttt{')'} \\
\hline
E_{\{a\}} & \to & \texttt{'['}\ E_{l1}\ \texttt{']'} \\
E_{\{a\}} & \to & E_{\{a\}}\ \texttt{'*'}\ E_{\{a\}} \\
E_{\{a\}} & \to & N \\
E_{\{a\}} & \to & \texttt{'('}\ E\ \texttt{')'} \\
\hline
E_{l1} & \to & \epsilon \\
E_{l1} & \to & E\ E_{l2} \\
\hline
E_{l2} & \to & \epsilon \\
E_{l2} & \to & \texttt{';'}\ E\ E_{l2}
\end{array}
$$

(d) G'_D, the generated concrete syntax.

Fig. 3. The generated grammars.

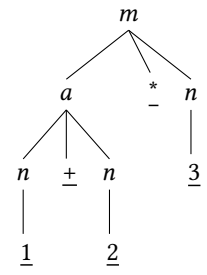
expression "[1(; 2)]", which makes no intuitive sense; grouping parentheses should only be allowed around complete expressions, but "; 2" is not a valid expression.

Finally, we require a function $unparen : L(T'_D) \to L(T_D)$ that removes grouping parentheses from a parse tree, i.e., it replaces every subtree $g(\ \texttt{'('}\ t\ \texttt{')'}\ )$ with t. The relation between the four grammars in terms of *yield* and *unparen* can be seen in Figure 5. With this we can define $parse : L(G'_D) \to 2^{L(T_D)}$, along with its inverse $words : L(T_D) \to 2^{L(G'_D)}$:

$$
\begin{array}{rcl}
parse(w') & = & \{unparen(t') \mid t' \in L(T'_D) \wedge yield(t') = w'\} \\
words(t) & = & \{w' \mid t \in parse(w')\}
\end{array}
$$

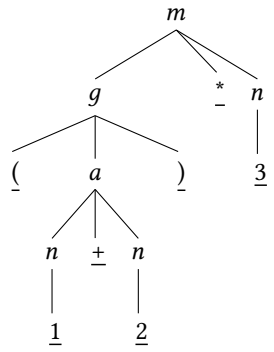
The latter is mostly useful in later sections, but *parse* allow us to consider some concrete examples of resolvable and unresolvable ambiguities. For example, in our running example (Figure 2), the word `'1 + 2 + 3'` is ambiguous, since $parse(\texttt{'1 + 2 + 3'}) = \{t_1, t_2\}$ where

$$
\begin{array}{l}
t_1 = a(\quad n(\ \texttt{'1'}\ )\ \texttt{'+'}\ a(\ n(\ \texttt{'2'}\ )\quad \texttt{'+'}\ n(\ \texttt{'3'}\ )\ )\ ) \\
t_2 = a(\ a(\ n(\ \texttt{'1'}\ )\ \texttt{'+'}\quad n(\ \texttt{'2'}\ )\ )\ \texttt{'+'}\ n(\ \texttt{'3'}\ )\quad)
\end{array}
$$



$m(a(n(\underline{1}) \underline{+} n(\underline{2})) \underline{*} n(\underline{3}))$

(a) Example tree in $L(T_D)$ and visualization.



$m(g(\underline{(}\, a(n(\underline{1}) \underline{+} n(\underline{2}))\, \underline{)}) \underline{*} n(\underline{3}))$

(b) Example tree in $L(T'_D)$ and visualization.

$1 + 2 * 3$

(c) Example word in $L(G_D)$.

$(1 + 2) * 3$

(d) Example word in $L(G'_D)$.

Fig. 4. Example with elements from each generated language that correspond to each other. The leaf terminals in the tree languages appear underlined to distinguish the two kinds of parentheses.

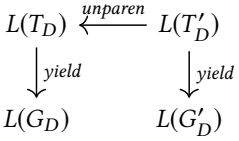


Fig. 5. The grammars considered, and their relation to each other.

This is a resolvable ambiguity, since $parse(\texttt{'1 + (2 + 3)'}) = \{t_1\}$ and $parse(\texttt{'(1 + 2) + 3'}) = \{t_2\}$. To demonstrate the unresolvable case, we add the production $E \longrightarrow s : E\ \texttt{';'}\ E$, at which point we find that the word `'[1 ; 2]'` is unresolvably ambiguous; $parse(\texttt{'[1 ; 2]'}) = \{t_3, t_4\}$ where:

$$t_3 = l(\ \texttt{'['}\quad n(\ \texttt{'1'}\ )\ \texttt{';'}\ n(\ \texttt{'2'}\ )\quad \texttt{']'}\ )$$
$$t_4 = l(\ \texttt{'['}\ s(\ n(\ \texttt{'1'}\ )\ \texttt{';'}\ n(\ \texttt{'2'}\ )\ )\ \texttt{']'}\ )$$

In this case, t_4 has an unambigous word (namely `'[(1 ; 2)]'`), but t_3 does not. The solution is to modify the language definition in Figure 2 so that both non-terminals in the production l are marked with s (i.e., they look like $E_{\{s\}}$), at which point $parse(\texttt{'[1 ; 2]'}) = \{t_3\}$.

We can now state the two central problems we wish to consider:

- Static resolvability analysis: determine if a given language definition D is resolvably ambiguous, otherwise produce an unresolvably ambiguous word $w' \in L(G'_D)$. Section 5 provides a partial solution to this problem.
- Dynamic resolvability analysis: given a language definition D and a word $w' \in L(G'_D)$, determine if it is resolvably ambiguous, and if so, present an unambiguous word for every tree $t' \in parse(w')$. Section 6 provides a full solution with one caveat: it only considers languages with balanced parentheses.

Additionally, on the dynamic side, there is another property of great practical relevance; we want ambiguities to be minimal and independent. In a practical setting, it is rarely helpful to merely

state that an entire program is ambiguous. Instead, we wish to minimize the reported ambiguous portion of the program to, e.g., a single expression. Section 7 addresses this.

5 STATIC RESOLVABILITY ANALYSIS

To determine if a given language definition D is resolvably ambiguous we attempt to show the contrapositive: find a tree $t \in L(T_D)$ such that there is no $w' \in L(G'_D)$ for which $parse(w') = \{t\}$, or prove that no such tree exists. Or, more briefly put: find a tree that has only ambiguous words or show that none exist.

Before an actual algorithm, we split the problem into three different versions of increasing difficulty. We then lay the groundwork for our correctness proofs, eventually culminating in the algorithm, which we show to be sound and complete for version 1 and sound for version 2.

The three versions are as follows:

Version 1 $\Sigma \cap \{$ `'('`, `')'` $\} = \emptyset$ and no non-terminals appear marked, i.e., for all non-terminals N_m appearing on the right-hand side of the labelled productions in D, we have $m = \emptyset$.

Version 2 $\Sigma \cap \{$ `'('`, `')'` $\} = \emptyset$.

Version 3 There are no constraints on D.

The first restriction in version 1 states that D cannot contain parentheses. This implies that all parentheses present in G'_D are grouping parentheses. The second restriction implies that no parentheses are required (as a counterexample, assuming normal precedence, the parentheses in "$(1 + 2) * 3$" are required and removing them would produce a different interpretation).

The reason for this split stems from the following insight: double grouping parentheses do not matter, in the sense that they do not change the interpretation of the word, e.g., $parse($`'((1 + 2)) * 3'`$) = parse($`'(1 + 2) * 3'`$)$. Version 1 and 2 have only grouping parentheses, meaning that no double parentheses matter. What follows is a brief outline of the remainder of this section, which details our static analysis for version 1 and 2, building from that observation. Note that this outline is unlikely to be easy to understand initially, but is rather there to serve as context for later sections.

- We can consider an alternate representation of words that splits non-parenthesis terminals from the ranges covered by parentheses, the former as a word, the latter as a multiset or a bag (Section 5.1).
- This alternate representation gives rise to a lattice per tree in $L(T_D)$, where a tree is unresolvably ambiguous iff its lattice is entirely covered by the lattices of other trees (Section 5.2).
- These lattices can be encoded as words, leading to the construction of a VPDA that we can examine to determine the existence of a tree whose lattice is covered by the lattice of a *single* other tree, which is sufficient for completeness in version 1, but not in version 2 (Section 5.3).

5.1 An Alternative Word View

This section introduces an alternative (isomorphic) definition of a word, heavily used in sections 5.2 and 5.3. The method by which we generate G_D and G'_D limits the possible differences between them significantly. In particular, no new terminals are introduced, except '(' and ')', and they are always introduced in a well-balanced fashion.

If we thus delimit ourselves to only consider languages where words have no unbalanced parentheses[8] we can give the following alternative definition of a word: a word is a two-tuple containing a sequence of non-parenthesis terminals and a bag (or multiset) of ranges covered by parentheses. For example, the word "$(1+2)*3$" is equivalent to ("$1+2*3$", $\{1\text{–}3\}$), while "$((1+2))*3$" is equivalent to ("$1 + 2 * 3$", $\{1\text{–}3, 1\text{–}3\}$). We will refer to the first component to as a *basic word*, the second as a *range bag*.

[8] I.e., the vast, vast majority of programming languages currently in use.

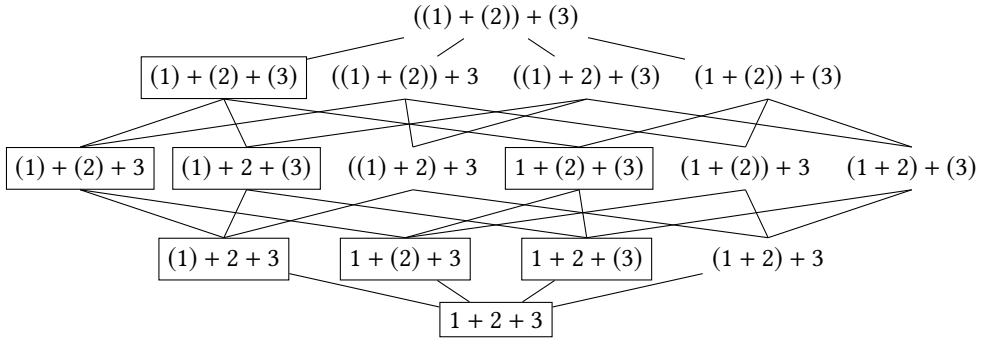


Fig. 6. The lattice of words for the tree $a(a(n(1) + n(2)) + n(3))$. The boxed words are shared with Figure 7.

We denote the function producing the alternative two-tuple by *alt*, while *basic* and *rangebag* directly produce the basic word and rangebag, respectively.

We will now note a few things about the words in $words(t)$ for some given $t \in L(G_t)$. First off:

Lemma 5.1. $\forall w'_1, w'_2 \in words(t).\ basic(w'_1) = basic(w'_2)$, *i.e., all* w' *have the same basic word.*

Intuitively, *words* first applies the "inverse" of *unparen*, i.e., adding some number of grouping parentheses nodes between pre-existing nodes, then *yield*, which removes internal nodes. The only terminals changed in this process are parentheses, i.e., w' can only differ in their range bag. This also implies that two trees that share a word (i.e., two trees t_1 and t_2 that have a word w' such that $parse(w') \supseteq \{t_1, t_2\}$) must also share a basic word.

Lemma 5.2. $\exists R$ *such that* $\exists w' \in words(t).\ rangebag(w') = R$ *and* $\forall w' \in words(t).\ R \subseteq Supp(rangebag(w'))$, *i.e., some parentheses are required.*

For example, removing the parentheses in "$(1 + 2) * 3$" changes the interpretations produced. Each required range is a direct consequence of a mark on a non-terminal in D. We will write the word implied by the first quatified expression as $w'_{\perp}$. It is unique, since its basic word is fixed by t, and its rangebag is a set.

Lemma 5.3. $\exists R$ *such that* $\exists w' \in words(t).\ rangebag(w') = R$ *and* $\forall w' \in words(t).$ $Supp(rangebag(w')) \subseteq R$, *i.e., there is a finite set of possible parentheses.*

As mentioned in Section 4 grouping parentheses can only be added if they exactly cover a production, which amounts to an internal node in t, and each tree has a finite amount of nodes. We will write the word implied by the first quantified expression $w'_{\top}$.

Lemma 5.4. $\forall w'_1, w'_2 \in words(t).\ Supp(rangebag(w'_1)) = Supp(rangebag(w'_2)) \rightarrow parse(w'_1) = parse(w'_2)$, *i.e., duplicated parentheses do not matter.*

We first note that in version 1 and 2 there are no parentheses in D, i.e., all parentheses in G'_D are introduced as grouping parentheses. Each pair of parentheses thus corresponds to a single grouping node g in a tree in $L(T'_D)$. Duplicated grouping parentheses correspond to nested grouping nodes, all of which are removed by *unparen*, thus producing identical sets of trees in $L(T_D)$. For example, $((1 + 2)) * 3$ has the same interpretations as $(1 + 2) * 3$.

5.2 A Lattice of Word Partitions

Lemma 5.4 suggests a partition of the words in $words(t)$ for any given $t \in L(T_D)$; group words w' by their *rangeset*, where $rangeset(w') = Supp(rangebag(w'))$. These partitions can be partially

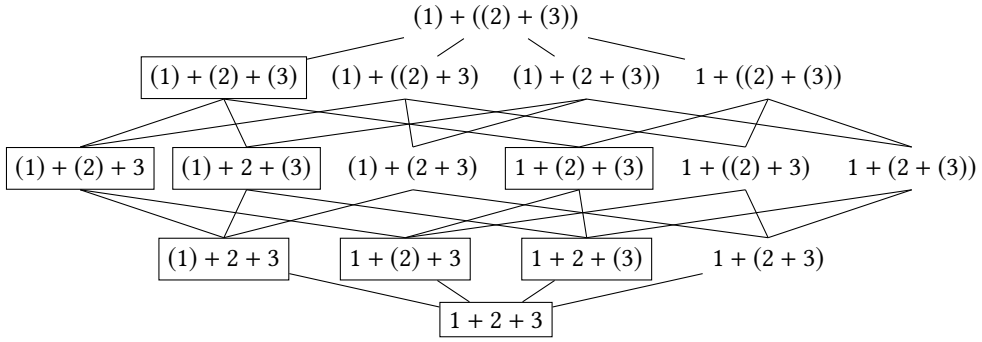


Fig. 7. The lattice of words for the tree $a(n(1) + a(n(2) + n(3)))$. The boxed words are shared with Figure 6.

ordered by subset on the rangeset, resulting in a lattice of word partitions per tree. This lattice is bounded, with top and bottom elements given by lemmas 5.3 and 5.2 respectively. Lemma 5.1 further states that all words share the same basic word. For example, Figure 6 contains the lattice for the tree $a(a(n(1) + n(2)) + n(3))$ (from our running example, defined in Figure 2 on page 7). Each partition is represented by the word whose rangebag is a set. To reduce clutter, we do not draw the partitions that have grouping parentheses around the entire word. This outermost possible pair is uninteresting since it is always allowed, and would double the size of the figure if it was included.

To show the connection between resolvable ambiguity and these lattices, consider the word "1 + 2 + 3". It has two interpretations in our running example, $a(a(n(1) + n(2)) + n(3))$, which we would normally write as $(1 + 2) + 3$, and $a(n(1) + a(n(2) + n(3)))$, which we would normally write $1 + (2 + 3)$. The lattices for these two trees are given in Figures 6 and 7 respectively. The partitions that appear in both lattices are represented as boxed words, the others are unboxed. The shared partitions thus represent words that are ambiguous between these particular trees. Finding an unambiguous word is thus the same as finding a partition that is not shared with any other tree. In this particular case, there is no ambiguity with any other tree at all, thus the unboxed words are all valid resolutions of the ambiguity.

At this point it is clear that a given tree has an unambiguous word iff its lattice has at least one partition that is not shared with any other tree.

Next, we note that each lattice is uniquely determined by its top and bottom elements, it contains all elements between them, i.e.:

Lemma 5.5. *Given* $t \in L(T_D)$, $w'_\top, w'_\bot \in words(t)$ *such that* $\forall w'.\ w' \in words(t) \rightarrow rangebag(t'_\bot) \subseteq rangeset(w') \subseteq rangebag(w'_\top)$ *the following holds:*
$\forall w'.\ rangebag(w'_\bot) \subseteq rangeset(w') \subseteq rangebag(w'_\top) \rightarrow w' \in words(t)$.

Lemmas 5.3 and 5.2 guarantee the existance of two such words $w'_\top$ and $w'_\bot$.

We now have the tools we require to motivate our approach.

Theorem 5.6. *In version 1, for a given* $t \in L(T_D)$, $\exists w'.\ parse(w') = \{t\}$ *iff* $parse(w'_\top) = \{t\}$ *(where* $w'_\top$ *is given by Lemma 5.3).*

5.3 A Lattice as a Word, and an Algorithm

Our initial limitations / assumptions are as follows:

(1) The input language definition contains no parentheses. This implies that all parentheses in G'_w are grouping parentheses, thus we only need to consider words $w' \in L(G'_w)$ whose range bag is a set.

(2) No non-terminals in the input language definition are marked, i.e., there are no forbidden children. This implies that there are no required grouping parentheses.
(3) No production has a right-hand side that matches a single non-terminal, i.e., $\forall(N \longrightarrow i : r) \in D.\ V \cap L(r) = \emptyset$[9].

The first two limitations allow us to place all words in $words(t)$ for some given t in a lattice, whose structure is formed by the subset ordering of the range sets. For example, the two trees in Figure 9 have the lattices of words seen in Figures 6 and 7 respectively.

The bottom word corresponds to the set of required parentheses (i.e., the empty set, in this sub-problem), while the top word corresponds to the set of possible parentheses ranges. All trees that can be ambiguous must thus share the same bottom word, while the top may differ.

Of particular use is to examine whether the top word (call it w') is ambiguous. There are two cases:

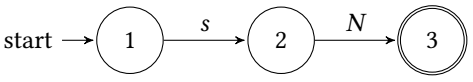
- The top word is unambiguous, then this is not a tree we are looking for (since it has at least one unambiguous word).
- It is ambiguous. That means that there is some other lattice for some other tree that also contains the word. Call the top word of this other lattice w'_2. For w' to be in the lattice whose top is w'_2 its rangeset must be a subset of the rangeset of the latter. But this is true also for all the other words in the lattice, thus there are no unambiguous words in the lattice, i.e., this tree is what we are looking for.

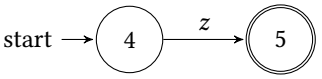
The algorithm thus looks for two trees, where the set of possible parentheses for one is a superset of the other. To do this, we use a linear representation of each lattice: namely the top word. If two trees have the same top word, that implies that their sets of possible parentheses are equal. If we can add parentheses to one word and get the other word, that implies that the rangeset of the former is a subset of the latter.

We will now construct a pushdown automaton that recognizes these top words in such a way that there is a bijection between successful runs and trees in $L(G_t)$. As a running example, we will use the following (very simple) language definition D:

$$\begin{array}{lcl} \langle N \rangle & \longrightarrow & \text{Succ}: \text{ 's' } \langle N \rangle \\ \langle N \rangle & \longrightarrow & \text{Zero}: \text{ 'z'} \end{array}$$

We begin by constructing a DFA per production. This can be done in the standard way by constructing an NFA, then determinizing it, and optionally minimizing it.



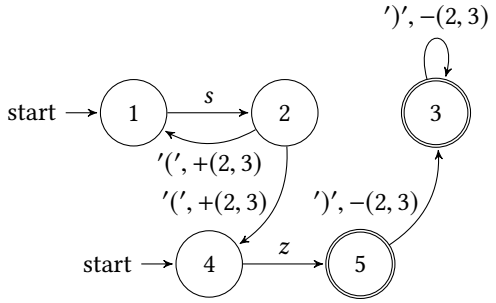


We then combine them into a single pushdown automata by replacing each edge with a non-terminal label $p \xrightarrow{N} q$ with:

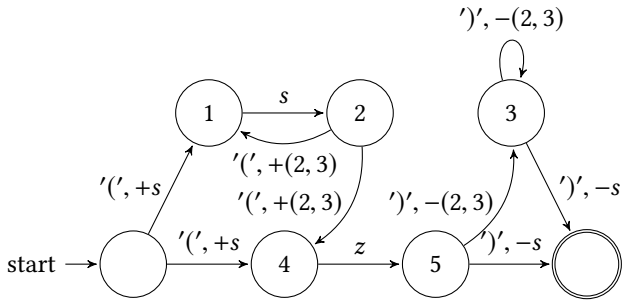
- an edge $p \xrightarrow{'(',+(p,q)} p'$ for every initial state p' in some DFA belonging to non-terminal N, and
- an edge $q' \xrightarrow{')',-(p,q)} q$ for every final state q' in some DFA belonging to non-terminal N,

[9] I haven't really written a proper description for the language definition, but I'm here using D for it, and considering it essentially as a set of "productions".

Intuitively, we parse a child node, but put parentheses around it, then return. This is where we use assumption 3, without it we might introduce double parentheses here.

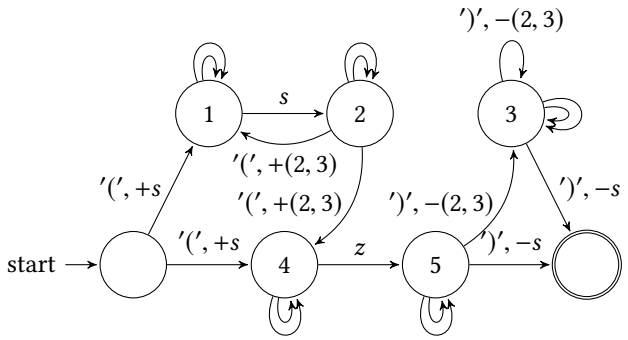


Finally, we add a new initial state and a new final state, and connect them with the initial and final states belonging to the starting non-terminal:



The resulting pushdown automaton has only a single source of non-determinism: the edges labelled '('. Each one corresponds to one of the allowable child productions at that point in the parse tree.

We now have a pushdown automaton (call it $A_{()}$) that recognizes the top word for each tree in $L(G_t)$. Next we need to be able to add arbitrary parentheses, to produce a word with a rangeset superset. To do this we create a copy of $A_{()}$ with one modification: for every state s in $A_{()}$ (except the initial and final states), add two transitions $s \xrightarrow{'(',+p} s$ and $s \xrightarrow{')',-p} s$. To avoid cluttering the graph too much, these transitions are shown unlabeled below:



We will call this automaton $A'_{()}$. Successful runs in this automaton have a surjection to runs in $A_($ (ignore transitions along the newly added edges), and thus also have a surjection to parse trees in $L(G_t)$. We can also note that every sucessful run in $A_{()}$ is also a successful run in $A'_{()}$, since the latter has all states and transitions of the former. Furthermore, two distinct runs, one in $A_{()}$ (call it

p) and one in $A'_{()}$ (call it p'), that both recognize the same word must represent different parse trees in $L(G_t)$. To see why, we consider two cases:

(1) p' only uses transitions present in $A_{()}$. This is a successful run in $A_{()}$, and distinct from p. But there is a bijection between runs in $A_{()}$ and parse trees in $L(T)$, thus p' represents a different parse tree.
(2) p' uses at least one transition added in $A'_{()}$. Using the surjection between runs in $A'_{()}$ and $A_{()}$ we find a new successful run that produces a different word (at least one fewer pair of parentheses). Since this run produces a different word, it must be distinct from p, and thus represent a different parse tree.

Two distinct successful runs that accept the same word thus represent two trees where one permits a superset rangeset of the other. To find such runs we construct a product automaton and trim it. We can construct a product automaton since both $A_{()}$ and $A'_{()}$ are visibly pushdown, with the same partition of the input alphabet (push on open parenthesis, pop on close parenthesis, do nothing otherwise). We can trim the product since it retains the same partitioning and thus is also visibly pushdown.

In this product automaton, if any transition pushes a stack symbol (a, b) where $a \neq b$, or transitions to a state (p, q) where $p \neq q$, then there is a successful run that corresponds to two distinct runs through $A_{()}$ and $A'_{()}$ (since the automaton is trim).

6 DYNAMIC RESOLVABILITY ANALYSIS

7 LOCAL AMBIGUITIES

8 EVALUATION

8.1 OCaml

9 RELATED WORK

[Afroozeh et al. [n. d.]]'s operator ambiguity removal patterns bear a striking resemblance to the marks presented in this paper. However, in special-casing (what in this paper would be) marks on left and right-recursions in productions they correctly cover the edge case discussed in Section 8.1.

$$\begin{array}{ccc} L(G_w) & \xleftarrow[yield]{} & L(G_t) \\ & & \uparrow{\scriptstyle unparen} \\ L(G'_w) & \xleftarrow[yield]{} & L(G'_t) \end{array}$$

Fig. 8. The generated grammars and their relation to each other.

$$1 + 2 + 3$$

(a) Example word in $L(G_w)$.

$$\begin{array}{l} [_S[_S[_N1]_N + \ \ [_N2]_N]_S + [_N3]_N \ \ ]_S \\ [_S \ \ [_N1]_N + [_S[_N2]_N \ \ + [_N3]_N]_S]_S \end{array}$$

(b) Example words in $L(G_T)$.

$$1 + 2 + 3$$

(c) Example word in $L(G'_w)$.

$$\begin{array}{l} [_S[_S[_N1]_N + \ \ [_N2]_N]_S + [_N3]_N \ \ ]_S \\ [_S \ \ [_N1]_N + [_S[_N2]_N \ \ + [_N3]_N]_S]_S \end{array}$$

(d) Example words in $L(G'_T)$.

Fig. 9. Example with an ambiguous word in $L(G'_w)$ with corresponding words from the other grammars.

10 PARSETIME AMBIGUITY REPORTING

11 PROPERTIES WE MAY WANT TO PROVE

11.1 Static Stuff

Lemma 11.1. *There is an injective function (alt) from well-balanced words to tuples of basic words and range bags.*

Lemma 11.2. $\forall t \in L(G_t), w_1, w_2 \in L(G'_w).\ \{w_1, w_2\} \subseteq words(t) \longrightarrow basic(w_1) = basic(w_2)$

Lemma 11.3. $\forall t \in L(G_t).\ words(t) \neq \emptyset$

Lemma 11.4. $\forall t_1, t_2 \in L(G_t).\ words(t_1) \cap words(t_2) \neq \emptyset \longrightarrow basic(t_1) = basic(t_2)$
(We're using 11.2 and 11.3 to (ab)use "basic" on trees as well)

Definition 11.5. A language definition is in version 1 or 2 if $\Sigma \cap par = \emptyset$.

Theorem 11.6. *In version 1 and 2:* $\forall w_1, w_2 \in L(G'_w).\ altset(w_1) = altset(w_2) \longrightarrow parse(w_1) = parse(w_2)$, *where altset(w) is the same as alt(w), but with the second component (the rangebag) replaced with a corresponding set.*

Lemma 11.7. *In version 1 and 2, given a* $t \in L(G_t)$, *the words in* $words(t)$ *can be partitioned into a lattice. Each partition contains the words that are equal by altset, and ordering is by subset on the rangeset. Furthermore, this lattice is bounded, and uniquely described by its bottom and top elements; it contains all elements between bottom and top. We denote this lattice as* $lattice(t)$.

Definition 11.8. A language definition is in version 1 if it contains no marked non-terminals and $\Sigma \cap par = \emptyset$.

Lemma 11.9. *In version 1: the bottom element of the lattice for a tree has a rangeset that is the empty set.*

Theorem 11.10. *Given a tree* $t \in L(G_t)$ *in version 1,* $\exists w \in L(G'_w).\ parse(w) = \{t\} \longleftrightarrow \neg\exists t' \in L(G_t).\ t \neq t' \wedge lattice(t) \subseteq lattice(t')$, *i.e., a tree is resolvable iff there is no other tree that entirely covers its lattice.*

Lemma 11.11. *(Version 1 and 2) We can construct a VPDA that recognizes a linear encoding of the lattices of trees in* $L(G_t)$, *such that there is a bijection between successful runs and trees.*

Lemma 11.12. *(Version 1 and 2) Using 11.11, we can construct a VPDA that reconizes a linear encoding of "superlattices" of trees in* $L(G_t)$ *(i.e., lattices that completely cover the original lattices), where each successful run corresponds to exactly one tree.*

Theorem 11.13. *(Version 1 and 2) Using 11.11, 11.12, and their product automaton, we can construct a VPDA such that there is a bijection between successful runs and pairs of trees in* $L(G_t)$ *such that the lattice of one is entirely covered by the other.*

Theorem 11.14. *We can construct a sound, decidable algorithm that takes a language definition in version 1 or 2 and answers "resolvably ambiguous", "unresolvably ambiguous", or "unknown". Additionally, in version 1, this algorithm will never answer "unknown", i.e., it is then complete.*

Definition 11.15. Version 2.5 allows a language definition to use parentheses, but restricts it to never produce double parentheses, and changes G'_w to also never recognize double parentheses.

Theorem 11.16. *11.7 holds also for version 2.5, thus we can extend 11.14 to cover version 2.5 as well.*

11.2 Dynamic Stuff

This section assumes that we at no point produce an infinite amount of parse trees. My intuition (corroborated by some quick googling) suggests that this can only happen if there is a productive, accessible nonterminal N such that $N \Rightarrow^+ N$. This is not checked at the moment.

Lemma 11.17. *$\forall t \in L(G_t)$. words(t) is a visibly pushdown language.*

Theorem 11.18. *Given a set of trees in $L(G_t)$, we can construct a corresponding set of VPDAs that recognize the words that are unique to each tree (i.e., isn't in words(t) for some other tree t in the set).*

Theorem 11.19. *We can construct an algorithm localize(F) that takes a finite parse forest $F \subseteq L(G_t)$ and selects a set of subforests[10] such that:*

- *The unselected parts are identical across the different syntax trees, including range (i.e., we report all ambiguities).*
- *No selected subforest is contained in another (i.e., we only report one ambiguity per range).*
- *No selected subforest can be replaced by zero or more of its children while still satisfying the points above (i.e., the selected subforests are minimal[11]).*

Theorem 11.20. *Given an ambiguous word $w \in L(G'_w)$, a localized ambiguity $F' \in$ localize$($parse$(w))$ covering the subword $w' \in L_{G'_w}(N)$ (i.e., $\text{parse}_N(w') = F'$), and a resolution $w'' \in L_{G'_w}(N)$ such that $\text{parse}_N(w'') = \{t\} \subset \text{parse}_N(w')$ for some $t \in L_{G_t}(N)$, the following holds:*

- *localize$($parse$(w[w'/w'']))$ = localize$($parse$(w)) \setminus F'$*

where $w[w'/w'']$ denotes replacing a subword w' with w''.

REFERENCES

Ali Afroozeh, Mark van den Brand, Adrian Johnstone, Elizabeth Scott, and Jurgen Vinju. [n. d.]. Safe Specification of Operator Precedence Rules. In *Software Language Engineering* (2013) *(Lecture Notes in Computer Science)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Springer International Publishing, 137–156.

Rajeev Alur and P. Madhusudan. [n. d.]. Visibly Pushdown Languages. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing* (2004) *(STOC '04)*. ACM, 202–211. https://doi.org/10.1145/1007352.1007390

David G. Cantor. [n. d.]. On The Ambiguity Problem of Backus Systems. 9, 4 ([n. d.]), 477–479. https://doi.org/10.1145/321138.321145

Mathieu Caralp, Pierre-Alain Reynier, and Jean-Marc Talbot. [n. d.]. Trimming Visibly Pushdown Automata. In *Implementation and Application of Automata* (2013) *(Lecture Notes in Computer Science)*, Stavros Konstantinidis (Ed.). Springer Berlin Heidelberg, 84–96.

H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. [n. d.]. Tree Automata Techniques and Applications. Available on: http://www.grappa.univ-lille3.fr/tata. ([n. d.]). release October, 12th 2007.

Viktor Palmkvist and David Broman. [n. d.]. Creating Domain-Specific Languages by Composing Syntactical Constructs. In *Practical Aspects of Declarative Languages* (2019) *(Lecture Notes in Computer Science)*, José Júlio Alferes and Moa Johansson (Eds.). Springer International Publishing, 187–203.

[10] This is a term that I have introduced, and so it will need to be described. Basically a parse forest for a part of a word, where each tree is a subtree of the original parse forest.

[11] By covered range, not by count.