Resolvable Ambiguity

Anonymous Author(s)

Abstract

Text of abstract ....

Keywords 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

PL'18, January 01–03, 2018, New York, NY, USA
2018.

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. 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 PADL-PAPER, which merely isolates ambiguities, an algorithm that suggests solutions to ambiguity errors.

[2] cross your fingers

Terminals	$t \in \mathbb{T}$
Non-terminals	$N \in \mathbb{N}$
Regular expressions	$r ::= t \mid N \mid r \cdot r \mid r + r \mid \epsilon \mid r^*$
Productions	$N \longrightarrow r$

Figure 1. Context-free EBNF grammars

- 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

A context-free EBNF grammar is a tuple $(S, P, \mathbb{N}, \mathbb{T})$. $S \in \mathbb{N}$ is the starting symbol, P is a set of productions, as given in Figure 1, $\mathbb{N}$ and $\mathbb{T}$ are disjoint sets of non-terminals and terminals, respectively.

The (word) language of a given non-terminal N in a grammar $G = (S, P, \mathbb{N}, \mathbb{T})$ is given by $L_G(N)$:

$$
\begin{aligned}
L_G(t) &= \{t\} \\
L_G(N) &= \bigcup \{L_G(r) \mid (N \longrightarrow r) \in P\} \\
L_G(r_1 \cdot r_2) &= \{w_1 \cdot w_2 \mid w_1 \in L_G(r_1), w_2 \in L_g(r_2)\} \\
L_G(r_1 + r_2) &= L_G(r_1) \cup L_G(r_2) \\
L_G(\epsilon) &= \{\epsilon\} \\
L_G(r^*) &= \{\epsilon\} \cup L_G(r) \cup L_G(r \cdot r) \cup \ldots
\end{aligned}
$$

The word language of a grammar G, written $L(G)$, is thus $L_G(S)$. We will omit the subscript whenever the intended grammar is clear from the context.

A Appendix

Text of appendix …