**Discovery of Visible Semantic Predicates Omitted from**

*LL(1)*: The Foundation of the ANTLR Parser Generator

Authors omitted for blind review

1. **Introduction**

The formal semantics of predicated grammars and analysis algorithm in the submitted paper require that disambiguating predicates appear at the left edge of ambiguous productions. This is cumbersome in practice and forces programmers to duplicate predicates. Fortunately, grammar analysis can automatically discover and hoist predicates from productions further down the derivation chain into parsing decisions without predicates. For example, it’s common to define a `Typename` production that specifies both semantics and syntax:

```
Typename → {isType(next symbol)}? id
```

References to `Typename` behave as if inlined, automatically making the predicate visible to parsing decisions. If we restrict analysis to `k = 1` for demonstration purposes, the DFA for rule

```
Decl → Typename id | id
```

is `D_0 \rightarrow D_1, D_1 \rightarrow \text{isType} f_1, D_1 \rightarrow \text{isType} f_2`. Edge `\text{isType}` is inferred; see function `resolveWithPreds` in Algorithm 5.

Because our DFA construction algorithm operates on grammars that can have predicates and actions anywhere on the right-hand side, hoisting predicates into parsing decisions introduces a semantic hazard. We cannot hoist predicates over actions because they might be a function of that action. For any derivation sequence `(S, uAδ) \rightarrow^∗ (S', uAδ')`, `λ_i \in I` is visible if `∃ \lambda_j \in M` for `j < i` and `\lambda = \lambda_1 \ldots \lambda_n`.

**Definition 1.** Semantic predicate transition `q \xrightarrow{a} q'` is visible from state `p_{A,i}` if the ATN can transition from `p_{A,i}` to `q` without consuming input and without encountering an action transition. The set of predicates visible between states `p` and `q` is:

```
Visible(p, q) = \{ \lambda_i | (S, p, w, γ) \xrightarrow{a} (S, q, v, γ) \text{ where } \lambda_i \in I \text{ for } i < j \text{ if } \exists \lambda_j \in M \}
```

For example, if `p` is from position `A → · (π_1)` then `Visible(p, q) = \{π_1, π_2\}`.

At its most complex, the visible semantic context is a “sum of products.” For example, in grammar `A → [π_1]? B \{π_2\}? a` then `Visible(p, q) = \{π_1, π_2\}`.

**Definition 2.** DFA state `D` is `D_0 \xrightarrow{a} D_1`, `D_1 \xrightarrow{\text{isType}} f_1, D_1 \xrightarrow{\text{isType}} f_2`. Our DFA construction algorithm relies on the following definitions to compute semantic context.

**Definition 3.** Semantic context `π` in ATN configuration `(q, i, γ, π)` is `π = \bigwedge_{p \in \text{Visible}(p, q)} \pi_j` where `p` is the ATN state derived from alternative `i`’s left edge.

**Definition 4.** Alternative production `i` is sufficiently covered with predicates if we must evaluate a predicate for every derivation leading to an ambiguous sequence `x \in C(α_i) \cap C(α_j)` for `i \neq j`. Visible `p_{A,i}` and every transition `q \xrightarrow{a} q'` such that `(p_{A,i}, xw, γ) \rightarrow^{a+} (q, xw, γ')`.

For example, the first alternative of nonterminal `A` in the following grammar is insufficiently covered because it can match ambiguous sequence `b` without evaluating a predicate via the second alternative of `B`.

```
A → B \{π_1\}? b
B → \{π_2\}? b | c
```

Specifically, we have `A \Rightarrow B \Rightarrow b` but also `A \Rightarrow B \Rightarrow \varepsilon`.

2. **DFA construction algorithm with predicate hoisting**

Here we present the same DFA construction algorithm as in the submitted paper but with visible predicate hoisting.

As closure passes predicates, it “ands” them into new configuration `c`’s semantic context. We do not hoist semantic predicates derived from syntactic predicates in another nonterminal’s submachine.

Function `resolveWithPreds` encodes the definitions above. It first collects configurations by conflicting alternative number and then “ors” together predicates associated with each conflicting alternative. If there exists a conflicting alternative that has fewer predicates than configurations, then at least one configuration isn’t covered by a predicate (`resolve` reports this later). If there are `n - 1` predicate contexts for `n` alternatives, conjure up the `n^{th}` context as “not the and” of the other contexts. If there are fewer than `n - 1` predicate contexts, return and indicate we couldn’t resolve `D`. If we have `n` contexts, choose a representative configuration, `c`, and set `c.π` to the combined context “or’d” together for `c`’s alternative held in `preds` array.
Alg. 1: createDFA(ATN State pA) returns DFA
work := []; Δ := {}; D₀ := {};
F := {fᵢ | fᵢ := new DFA state, 1...numAlts(A)};
Q := F;
DFA.p₀ := p₀; // save ATN start state in DFA
D₀ := closure(D₀, A, {(pA, i, [], [-]) | edge i is
p₀ \rightarrow p}).true);
work += D₀; Q += D₀;
DFA := DFA(−, Q, T ∪ Π, Δ, D₀, F);
foreach D ∈ work do
  for each input symbol a ∈ T do
    D′ := closure(D, A, move(D, a), false);
    if D′ /∈ Q then
      resolve(D′);
      switch findPredictedAlt(D′) do
        case None: work += D′;
        case Just j: fⱼ := D′;
      endsw
      Q += D′;
      Δ += D \rightarrow D′;
    end
  end
  if wasResolved(D) then
    foreach c ∈ D such that wasResolved(c) do
      Δ += D \rightarrow fᵢ.c;
    end
  end
  work -= D;
end
return DFA;

Algorithm 2: move(DFA State D, a ∈ T)
returns set of configurations
return {(q, i, γ, π) | (p, i, γ, π) ∈ D, p \rightarrow a q};

Alg. 3: resolve(DFA State D)
conflicts := the conflict set of D;
if |conflicts| = 0 and not overflowed(D) then return;
resolved := resolveWithPreds(D, conflicts);
if resolved and insufficientlyCovered(i) then
  report i insufficiently covered with predicates;
if not resolved then
  resolve by removing all c ∈ D such that
c.i ∈ conflicts and c.i ≠ min(conflicts);
end
if overflowed(D) then report recursion overflow;
else report grammar ambiguity;

Alg. 4: closure(DFA State D, c = (p, i, γ, π),
boolean collectπ) returns set closure
if c ∈ D.busy then return {c}; else D.busy += c;
closure := {c};
if p = p′A (i.e., p is stop state) then
  if γ = p′γ then
closure += closure(D, (p′, i, γ′, π), collectπ);
else
  closure += \bigcup closure(D, (p₂, i, [], π), collectπ);
endsw
foreach transition t emanating from ATN state p do
  switch t do
    case p \rightarrow a transition and A′ is synpred:
      // make sure pred is not syn pred in another rule
      if collectπ and (S, DFA.p₀, w, γ) \rightarrow (S, p, w, γ) then
        π′ := π ∧ π.A′;
        else π′ := π;
        add closure(D, (q, i, γ, π′), collectπ) to closure;
    case p \rightarrow q transition:
      if collectπ then π′ := π ∧ π.p;
      else π′ := π;
        add closure(D, (q, i, γ, π′), collectπ) to closure;
    case p \rightarrow p′ transitions to nonterminal A:
      depth := number of occurrences of A in γ;
      if depth = 1 then
        add i to D.recursiveAlts;
      if |D.recursiveAlts| > 1 then
        throw LikelyNonLLRegularException;
      end
    end
  if depth ≥ m, the max recursion depth then
    mark D to have recursion overflow;
    return closure;
    closure += closure(D, A, (pA, i, p′γ, π), collectπ);
  case p := q, p := q:
    closure += closure(D, A, (q, i, γ, π), collectπ);
endsw
end
return closure;
Alg. 5: \texttt{resolveWithPreds(DFA State } D, \text{ set } \texttt{conflicts)}

\textbf{return}s boolean

\texttt{preds} := \{\}; // \texttt{preds}[i] is predicate for alt \textit{i}
\texttt{configs} := \{\}; // configurations for alt \textit{i}

\textbf{foreach} \textit{i} \in \texttt{conflicts} \textbf{do}

\texttt{configs}[i] := \{c \in D | c.i = i\};

\texttt{preds}[i] := \bigvee_{c \in \texttt{configs}[i]} c.\pi

\textbf{if} 0 < |\texttt{preds}[i]| < |\texttt{configs}[i]| \textbf{then}

mark alt \textit{i} as \textit{insufficiently covered};

\textbf{end}

\textbf{if} |\texttt{preds}| < |\texttt{conflicts}| - 1 \textbf{then return false};

\textbf{if} |\texttt{preds}| = |\texttt{conflicts}| - 1 \textbf{then}

let \textit{j} be the alt with missing predicate;

\texttt{preds}[j] = \neg(\bigwedge_{i \neq j} \texttt{preds}[i]); // not the others

\textbf{end}

\textbf{foreach} \textit{i} \in \texttt{conflicts} \textbf{do}

\texttt{remove} all but one representative \textit{c} = (i, \pi) \in D;

\texttt{c.\pi} := \texttt{preds}[i]; // reset to combined \texttt{preds}

mark \textit{c} as \textit{wasResolved};

\textbf{end}

mark \textit{D} as \textit{wasResolved};

\textbf{return} \textit{true};