CompoGrammar: A Framework for Grammar Composition

Selim Ciraci

OutLine

• Why grammar composition?
• Current approaches
• CompoGrammar Framework
  – Syntactic composition operators
  – Ambiguity detection
  – Semantic composition operators

Why use Grammar composition?

• Software development with multiple programming languages
  – UI with C#, Java, libraries in C
  – Domain Specific Languages embedded in a host language (like C)
  – JavaDoc
  – Analysers, parsers, transformers need to detect/parse both languages
• Model composition
  – Textual syntaxes developed for models also need to be composed.
• Problems: the parser for the combined grammar should be developed.
  – Parser generators don't support composition.
  – No support for reuse.
  – Syntactic generators for DSLs cannot be used.
  – Modifying host language grammar is a complex procedure.
  – Automatic DSL to host conversion cannot be used.
• Terms: Host grammar and donor grammar.

Current Approaches

• Parser generators has limited support for Grammar composition.
  – Grammar inheritance: limited reuse.
    • Requires modifications to the base grammar.
    • No reuse for donor grammar definitions.
    • Multiple rule modifications are problematic.
  – Island grammars: Requires modifications to the parser generator itself.
    • E.g. Parser generated by Antlr needs to be modified in the right places > can introduce errors.
  – Sophisticated parser generators in literature.
    • Not based on Antlr > cannot make use of the grammar database of Antlr.

CompoGrammar Framework

• Provided automated grammar composition for Antlr. Aims:
  – Reuse Antlr's grammar database, grammars generated by DSL tools.
  – No modifications to the grammar files.
• CompoGrammar borrows ideas for aspect oriented programming.
  – The compositions are programmed through a set of rules.
  – Support for multiple rule modifications.

CompoGrammar Framework

• Challenge in grammar composition: the donor grammar rules should respect the tokens of the base grammar.
  – Lexer ambiguities cannot be resolved.
  – Antlr gives errors and does not generate
  – E.g FOR can be a token in both languages. Donor grammar needs to be modified.
  – Donor grammar rules needs to be rewritten in terms the tokens of the base language.
  – Detect ambiguities in tokens.
CompoGrammar Framework

• What about ambiguities in high order rules?
  – Antlr does not give errors -> the ambiguity may actually be desired.
  – CompoGrammar can be used to detect it. But resolution of the ambiguity is out of scope (cause Antlr does not give errors 😐).

CompoGrammar Framework

• Compositions are written as prolog programs.
  – E.g.: grammar('grammars/Java.g','Java',G),grammar('grammars/ANTLRv3.g','antlr',A),
    rule(A,'finallyClause',R),addrule(R,G),writegrammar('grammars/deneme.g',G)
  • Predicates are grouped into three:
    1. I/O predicates: read/write of grammar files.
    2. Syntactic predicates: modifications to grammar rules. Following Antlr’s own grammar.

Syntactic Predicates

• Syntactically correct rule composition.
  • addrule(R,G): adds the rule R and its dependencies to the grammar G. All other syntactic predicates use this predicate.
  • before(R1,R2): adds a reference to the rule R1 from R2 such that R1 is called before any alternative of R2.
  • after(R1,R2): adds a reference to the rule R1 from R2 such that R1 is called after any alternative of R2.
  • around(R1,R2): replaces R2 with R1.
  • beforeAlternative(R1,R2,A1), afterAlternative(R1,R2,A1),
    aroundAlternative(R1,R2,A1).
  • addAlternative(R1,R2): adds an alternative to the rule R2 which calls the rule R1.
  • island(R1,R2,'island rule'): adds an alternative to R2 which first matches the island rule and then calls R1.
  • ...

Syntactic Predicates

• Example: adding drop statements from SQL to Java.
  – grammar('grammars/Java.g','Java',G),grammar('grammars/Sql.g','sql',A),
    rule(A,'drop_view_stmt',R),addrule(R,G),writegrammar('grammars/deneme.g',G)

Ambiguity Detection

• Syntactic predicates ignore the semantics -> introduce ambiguities to the grammar.
• The ambiguities between rules should be detected.
  – Paves the way for ambiguity resolution strategies.
• Ambiguity detection is a well studied problem:
  – LL(k) parsing, horizontal/vertical parsing... -> very sophisticated detection methods.
  – CompoGrammar uses a very basic one, generates all strings possible from a rule.
    • As we work with tokens sophisticated proof are not needed. Length can be limited.

Ambiguity Detection

• CompoGrammar uses state-space generation to generate all strings possible with a rule.
  – State-space generation is based on GROOVE: control programs have very similar semantics as grammar rules.
    • Closures (*) and positive closures (+) are unrolled.
    • Strings up to a user specified length is generated.
  – generateTerminals(R,S): generates all terminals possible with rule R and stores in S.
  – getAmbiguities(S1,S2,A): returns the intersection of S1,S2. Depending on this intersection we can build-up resolution strategies.
Ambiguity Detection

• Grammar rule to Control program mapping is straightforward.
  – 'a' -> addChar(a)
  – 'a'..'z' -> addCharRange('a', 'z').
  – A | B -> choice(A) or B.
  – A... -> function A(){...}
  – (A)? -> choice(A) or { epsilon }
  – (A)* -> choice{ A choice{ A} or { epsilon } } or{ epsilon }.

Ambiguity Detection

• Example: java grammar’s identifier rule converted to a control program.

Ambiguity Detection

• After generating all strings possible with R1 and R2, the detection algorithm compares each string and returns their intersection.
  – R1(i) and R2(i) is a character return the character if they are equal.
  – R1(i) and R2(i) is a character range return their overlap.
  – ...
  – Example: (AZ)(AZ)(AZ)(BZ)(AZ) UNI(AK)(HV) returns UNI(BK)(HV)

Semantics Predicates

• Deals with ambiguity resolution.
  • keyword(R1,SR1,R2,SR2): Used when tokens overlap with string literals, identifiers. A token keyword should match before these rules. Hence this predicate places the rule R1 before R2. Checks of SR1 and SR2 intersect.
  • intersectionRule(R1,SR1,R2,SR2): removes the alternatives both in SR1 and SR2 from R1 and R2. Creates a third rule R3 containing the intersection of SR1 and SR2. Arranges the references to R1 and R2 to (R1 | R3) and (R2 | R3).
    – Uses the GTS to find the alternatives.
    – Limited rule reconstruction but can be made very clever.
    – Currently doesn’t work with rule references.

Conclusion

• CompoGrammar aims at reuse.
  – Based on Antlr.
  – Grammar database of Antlr can be used as is.
  – Grammar generators based on Antlr can be used as is.
• CompoGrammar tries to be easy.
  – No grammar modification.
  – Ambiguity detection and warnings/guidelines based on the results.
• CompoGrammar cannot do type checking.
  – Because it is based on Antlr.
  – Conflict of types cannot be resolved. TXL can do that.