Statically Scheduling Circular Remote Attribute Grammars

Seyedamirhossein Hesamian

University of Wisconsin-Milwaukee

Follow this and additional works at: https://dc.uwm.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

https://dc.uwm.edu/etd/3383

This Dissertation is brought to you for free and open access by UWM Digital Commons. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of UWM Digital Commons. For more information, please contact scholarlycommunicationteam-group@uwm.edu.
STATICALLY SCHEDULING CIRCULAR REMOTE ATTRIBUTE GRAMMARS

by

Seyedamirhossein Hesamian

A Dissertation Submitted in
Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
in Engineering

at
The University of Wisconsin-Milwaukee
December 2023
ABSTRACT

STATICALLY SCHEDULING CIRCULAR REMOTE ATTRIBUTE GRAMMARS

by

Seyedamirhossein Hesamian

The University of Wisconsin-Milwaukee, 2023
Under the Supervision of Professor Dr. Boyland

Classical attribute grammars invented by Knuth have been the subject of extensive study. Over the years there have been various extensions introduced, each with the goal of making attribute grammar more useful for applications such as program analysis. The first extension described here is circular attribute grammar by Farrow. It is followed by remote attribute grammar, which was introduced separately by Boyland and Hedin. More recently, Hedin introduced circular remote attribute grammars and a proof of concept implementation with demand evaluation. Remote attribute grammars make it possible for semantic rules to access attributes of nodes that are not local, and circular attribute grammars make it possible to define and solve problems that are circularly defined. Combining these two extensions opens the door that enables users to define various circular tree problems in attribute grammars. Having the means to evaluate circular remote attribute grammars efficiently both in terms of space and time complexity is important for real-world applications. Unfortunately, demand evaluation methods lack in both aspects. However, another type of static evaluation called visit sequence evaluation was introduced earlier by Kastens for $l$-ordered subset of classical attribute grammars. This evaluation method is efficient as schedules are generated statically and not dependent on a particular derivation. The goal of this thesis is to statically schedule and evaluate circular remote attribute grammars, and lastly, implement these changes into APS which is a declarative attribute grammar system introduced by Boyland.
© Copyright by Seyedamirhossein Hesamian, 2023
All Rights Reserved
# TABLE OF CONTENTS

1 Introduction ............................................. 1
   1.1 Motivation ........................................... 1
   1.2 Research Aims and Objectives ...................... 1
   1.3 Examples ............................................. 4
   1.4 Structure of Thesis .................................. 8

2 Related Work ............................................ 11
   2.1 Visit Sequence Evaluator ............................ 11
   2.2 Circular Attribute Grammar (CAG) .................. 12
   2.3 Remote Attribute Grammars (RAG) ................... 14
   2.4 Circular Remote Attribute Grammars (CRAG) ....... 15
   2.5 Program Analysis Tools Based on AGs ............... 16
       2.5.1 JastAdd ......................................... 17
       2.5.2 APS ............................................ 18
   2.6 Other Declarative Program Analysis Tools .......... 20
       2.6.1 Datalog with Binary Decision Diagrams ....... 22
       2.6.2 FLIX, Declarative Language for Fixed Points on Lattices .... 23

3 Definitions .............................................. 27
   3.1 Context-Free Grammar ................................ 27
   3.2 Classical Attribute Grammar ........................ 30
       3.2.1 Schedule ........................................ 37
3.2.2 Demand Evaluation .................................................. 41
3.2.3 Dependency Graph .................................................. 44
3.2.4 1-SWEEP ............................................................... 47
3.2.5 Ordered Attribute Grammars ..................................... 51
3.3 Remote Attribute Grammar ............................................ 58
3.3.1 Schedule ............................................................... 59
3.4 Circular Attribute Grammars .......................................... 62
3.4.1 Schedule ............................................................... 67
3.4.2 Ordered Circular Attribute Grammars ............................ 71
3.5 Circular Remote Attribute Grammars ................................. 73
3.5.1 Schedule ............................................................... 74
3.5.2 Ordered Circular Remote Attribute Grammars ................. 79

4 Methods 82

4.1 Visit Sequence ............................................................ 83
4.1.1 Visit Sequence for Classical AGs ................................. 84
4.1.2 Visit Sequence For Circular Remote AGs ....................... 95
4.1.3 Visit Sequence Evaluation ......................................... 105
4.2 Group Scheduling .......................................................... 107
4.3 Fiber Cycle Breaking ..................................................... 114
4.4 SCC Chunk Scheduling ................................................... 115

5 Implementation 121

5.1 Prerequisite Changes .................................................... 122
5.1.1 Overview of Canonical Types and Signatures .................... 123
5.1.2 Canonical Types ....................................................... 126
5.1.3 Canonical Signature ................................................... 129
5.1.4 Canonical Signature Set .............................................. 129
5.1.5 Code Generation Modification ........................................ 131
5.1.6 Fiber Cycle Breaking .................................................. 132
5.2 Loops in Code Generation ............................................... 132
5.3 SCC chunk Scheduler ..................................................... 133
  5.3.1 Intrinsic Complexity of Static Scheduler ......................... 135
  5.3.2 Failed Attempts ..................................................... 137
  5.3.3 SCC Scheduling Proposal .......................................... 138
  5.3.4 Concession ......................................................... 144

6 Evaluation ................................................................. 148

7 Conclusion ................................................................. 153
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Structure of a context-free grammar in APS</td>
<td>6</td>
</tr>
<tr>
<td>1.2</td>
<td>APS function to describe “.&quot; operation defined in Equation (1.1)</td>
<td>7</td>
</tr>
<tr>
<td>1.3</td>
<td>Assigning attributes to non-terminals</td>
<td>8</td>
</tr>
<tr>
<td>1.4</td>
<td>First function implementation in APS</td>
<td>9</td>
</tr>
<tr>
<td>2.1</td>
<td>Nullable implementation in JastAdd</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>Assigning attributes to non-terminal for Nullable example</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>Nullable implementation in APS</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>Nullable implementation in Prolog</td>
<td>23</td>
</tr>
<tr>
<td>2.5</td>
<td>Nullable implementation in Datalog</td>
<td>24</td>
</tr>
<tr>
<td>2.6</td>
<td>Nullable implementation in FLIX</td>
<td>26</td>
</tr>
<tr>
<td>3.1</td>
<td>Example of context-free grammar and its trivial derivation</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Equivalent derivation tree for derivation in Figure 3.1</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>Example of context-free grammar with recursive production</td>
<td>30</td>
</tr>
<tr>
<td>3.4</td>
<td>Equivalent derivation tree ( \tilde{T} ) for derivation defined in Figure 3.1</td>
<td>33</td>
</tr>
<tr>
<td>3.5</td>
<td>Purely synthesized attribute grammar with local attribute</td>
<td>34</td>
</tr>
<tr>
<td>3.6</td>
<td>Example of purely synthesized attribute grammar without local attribute</td>
<td>34</td>
</tr>
<tr>
<td>3.7</td>
<td>Instantiated pure-S AG defined in Figure 3.6</td>
<td>35</td>
</tr>
<tr>
<td>3.8</td>
<td>Syntax tree and flow of attribute values for derivation defined in Equation (3.4)</td>
<td>35</td>
</tr>
<tr>
<td>3.9</td>
<td>Example of classical attribute grammar</td>
<td>38</td>
</tr>
<tr>
<td>3.10</td>
<td>Instantiated AG defined in Figure 3.9 using derivation defined in Equation (3.5)</td>
<td>39</td>
</tr>
<tr>
<td>3.11</td>
<td>Example of classical attribute grammar</td>
<td>44</td>
</tr>
</tbody>
</table>
3.12 Example of classical non-circular AG

3.13 Augmented dependency graph $DG^*_{p:A_0\rightarrow A_1a}$ for non-circular AG defined in Figure 3.12

3.14 Example of 1-SWEEP attribute grammar

3.15 Example of attribute grammar that is not 1-SWEEP

3.16 Summary graph used to test Figure 3.14 and Figure 3.15 for the 1-SWEEP property

3.17 Augmented dependency graph for Figure 3.14 constructed using Figure 3.16

3.18 Augmented dependency graph for Figure 3.15 constructed using Figure 3.16

3.19 Example of $l$-ordered attribute grammar [5]

3.20 Correct summary graph used to test Figure 3.19 for the $l$-ordered property

3.21 Incorrect summary graph used to test Figure 3.19 for the $l$-ordered property

3.22 Augmented dependency graph for Figure 3.19 constructed using Figure 3.20

3.23 Augmented dependency graph for Figure 3.19 constructed using Figure 3.21

3.24 Cycle involving multiple “instantiated” augmented dependency graphs

3.25 A simple remote attribute grammar definition

3.26 Instantiated RAG defined in Figure 3.26 using derivation defined in Figure 3.1

3.27 Dependency graph for the trivial derivation defined in Figure 3.1 of remote AG defined in Figure 3.25

3.28 Example of circular attribute grammar

3.29 Dependency graph $DG$ for circular derivation $S \rightarrow AA \rightarrow aA \rightarrow aa$ of circular AG described in Figure 3.28 that results in a cycle between attribute instances

3.30 Dependency graph $DG^*_p$ for non-circular derivation $S \rightarrow AA \rightarrow aA \rightarrow ab$ of circular AG described in Figure 3.28 that does not result in a cycle between attribute instances

3.31 Quasi total order described in Example 3.4.1 expressed as a directed graph

3.32 Cycle involving multiple “instantiated” augmented dependency graphs
3.33 Example of a circular remote attribute grammar with all monotone operations.
76
3.34 Instantiated CRAG defined in Figure 3.33 using the derivation defined in
Figure 3.1

3.35 Dependency graph for derivation S → AB → aB → ab of circular remote AG

described in Figure 3.33 that does not result in a cycle in-between attribute
instances.

3.36 A circular remote attribute grammar with some non-monotone operations.
78
3.37 Instantiated CRAG defined in Figure 3.36 using derivation defined in Fig-
ure 3.1

4.1 Visit sequence evaluator for classical AG defined in Figure 3.9
93
4.2 Example of non-circular AG that is not l-ordered
94
4.3 Instantiated form of AG described in Figure 4.2 using trivial derivation
94
4.4 Invalid visit sequence for Figure 4.2
95
4.5 Visit sequence for circular remote AG defined in Figure 3.33
106
4.6 append top-level match of First example.
110
4.7 Items phylum phase assignments
111
4.8 Attribute occurrences used in append with their (ph, ch) assignment
112
4.9 Static schedule for append top-level match
113
4.10 Dependency graph DG_p for top-level match append
114
4.11 Flow chart describing different pieces of SCC chunk scheduling for CRAG.
119

5.1 OrLattice and AndLattice type declarations
124
5.2 Simple APS code showing the use of integer max lattice behaving as an integer
125
5.3 APS code defining MAX_LATTICE module
125
5.4 Type alias example in APS
126
5.5 C struct definition of canonical types
127
5.6 C definition of canonical signature
129
ACKNOWLEDGEMENTS

I would like to thank Professor Boyland for his invaluable advice, continuous support, and patience during the Ph.D. study and I am very thankful to have an opportunity to study under his supervision. Professor Boyland’s immense knowledge and great experience in the field of compilers and attribute grammar was a major source of encouragement during my academic research and daily life. I also would like to express my gratitude to my parents: Mr. Aref Hesamian and Mrs. Shahnaz Amirani. Without their tremendous understanding and encouragement over the past few years, it would have been impossible for me to complete my studies. I would like to thank Microsoft Corporation especially my manager Dwayne Need for enabling me and for support throughout this journey. Finally, I must thank all the faculty members at UW-Milwaukee College of Engineering & Applied Science who guided me during my bachelor’s, master’s, and Ph.D. studies, their continuous support, and a world-class level of education. Thank you.
Chapter 1

Introduction

1.1 Motivation

The motivation behind this thesis is to efficiently run analysis over context-free grammars (CFG). More specifically circularly defined problems and one of these circularly defined problems is First and Follow functions. The end goal of this thesis is to define these or any other circularly defined problems in a declarative manner, and then statically schedule the attribute grammar so these functions can be evaluated statically using a visit sequence evaluator given any formal grammar. Regarding the implementation, the objective of this thesis is to build upon on previous work of Boyland in [6, 8] and modify APS to add support for circular remote attribute grammars. To accomplish this goal, several modules of APS had to be modified and these changes will be discussed thoroughly in the following chapters.

1.2 Research Aims and Objectives

Attribute grammars are a fundamental formalism of modern Computer Science. Since 1968 when Knuth [18] introduced the basic concept, they have been the subject of much research including Boyland [8] and Hedin [21], proving the importance of the area and the broad range of its applications. One of these applications is to define the semantics, meaning of some sort,
to a program, expression, or phrase, in a language. Attribute grammars define semantics by specifying that certain attributes (or values) will be associated with certain nodes in the syntax tree, and providing equations that specify how the values of these attributes are to be computed. As previously mentioned, the goal of this thesis is to statically schedule circular remote attribute grammars, and to accomplish it, this research uses APS as a starting point. APS is an attribute grammar system that is declarative, typed, and has a well-defined syntax and (to some extent) semantics. This chapter describes the problem and what makes it so important, why this thesis chose APS as a starting point, and finally the approach this research took to extend APS to support statically scheduling circular remote attribute grammars.

Declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow. It makes it possible to describe the logic briefly and succinctly with a high level of abstraction. This is important in the context of attribute grammar as it makes it possible to define the tree and semantics. Then the attribute grammar system generates the necessary code to find the result of those semantics given a derivation of production rules for a given context-free grammar. The declarative nature of APS allows code generation to many target languages such as C++ and Scala. Also, the fact that programming language features used in APS are declarative and not syntactically very close to one specific programming language, simplifies onboarding of any new target language and code generation.

In mathematics, it is natural to have recursive definitions, but usually, it's cumbersome to bring over those definitions in computer science without modification to the problem definition. This extends to attribute grammars as well, and circular attribute grammars make it possible to describe circular problems succinctly and as is. Attribute grammars have been around for some time and have been studied extensively but circular attribute grammar usually has been avoided. Traditionally, starting from Knuth's paper, circular dependencies are seen as an error in attribute grammar. The classical definition of “well-formed” attribute
grammar (AG) which will be described in later chapters disallows them. Even some demand evaluators will simply fault and error out if a circular dependency is encountered. More specifically, recent attribute grammar systems including Hedin [21] try to find the result of circular attributes defined on a formal grammar using demand evaluation. The intrinsic issue with these approaches is the evaluation’s time and space complexity. Basically, in a demand evaluation, there is a need for having a runtime dependency graph and topological sort to determine what attribute instance needs to be evaluated next. This problem is solved in static evaluation.

If dependencies between attributes for a particular derivation are non-circular, the attribution can be obtained simply by applying the semantic functions in topological order, provided that evaluating semantic functions terminates. If dependencies between attributes for all possible derivations are non-circular, the attribute grammar is said to be non-circular. Classical AGs are required to be non-circular, but there are also extensions such as Farrow’s [13] that allow circular dependencies. The usual requirement for such grammars is that the values in the domain of an attribute in a cyclic dependency chain can be arranged in a lattice of finite height and that all semantic functions are monotonic with respect to these lattices. Monotonic functions in the context of set functions map subsets of the domain to non-decreasing values of the codomain. In particular, if $f: X \rightarrow Y$ is a set function from a collection of sets $X$ to an ordered set $Y$, then $f$ is said to be monotone if whenever $A \subseteq B$ as elements of $X$, $f(A) \leq f(B)$. For such circular attribute grammars, one can assign the attributes in the cycle a bottom value (from the lattice) as a starting value, and the attribution or evaluation can be obtained by iteratively applying the semantic functions. The use of monotone set functions in circular attribute grammars was first introduced by Farrow [13].

In the context of classical attribute grammar (non-circular), an important advantage of a runtime schedule or total order on instantiated rules is that evaluation takes place in linear time. Since the scheduler has assured us that the rules will be executed in a legal order, it
is not necessary to test this property at run-time. However, generating a schedule is costly both in terms of time and space as a schedule only works for a single derivation. Static evaluation methods such as visit sequence evaluation on the other hand are generated once and will work for all derivations of an appropriate attribute grammar. Additionally, they run in linear time for classical AGs. However, as there can be cycles in the class of circular (remote) attribute grammars, by definition, having cycles will make linear time complexity not always achievable. But one can modify visit sequences to add fixed point loops that run until attribute values in a cycle reach a fixed point. The goal of this thesis is to statically schedule circular remote attribute grammars. To accomplish this task, this research did not want to start from the ground up. This thesis uses APS [6] which is an attribute grammar system that is declarative, typed, has a well-defined syntax and semantics, and already supports scheduling of remote attribute grammars.

1.3 Examples

This subsection showcases APS syntax and semantics by writing a simple utility that given any context-free grammar as input, would then resolve the First and Follow sets. These functions are an example of circularly defined problems over context-free grammars. These two functions are commonly used in LL(k) parsers and the construction of a predictive parser and they allow filling in the entries of a predictive parsing table. Implementing these functions in APS will showcase the usefulness of APS in writing tools for program language analysis. Equation (1.1) describes the formal definitions based on Compilers textbook [1] and Boyland’s lecture notes. Informally, the First set for a non-terminal is the set of terminals that begin the strings derivable from this non-terminal. The Follow set for a non-terminal is the set of terminals that can appear immediately after the derivation of this non-terminal in strings of the language (non-terminals do not appear in strings).

The first step is to define the structure of the context-free grammar in APS and this is
done in Figure 1.1. This definition will be used as a basis for the rest of the thesis. APS is a tool that allows defining the structure and semantics of context-free grammar and assists with deriving the semantic values. However, defining First and Follow functions in APS requires the definition of a context-free grammar using an abstract syntax tree (AST) which demonstrates the meta-level nature of these functions.

APS uses terms such as **phylum**, **type** (which is not used in this particular example), **constructor**, and top-level match to define the structure and semantic rules of a context-free grammar. In short, a **phylum** is used to declare a syntactic type one which can be attributed. A tree node belongs to a phylum if it was constructed by the constructor of a phylum whereas a top-level match is used to define semantic rules for a production. More specifically, a **phylum** keyword has a feature where all its instances may be matched using a top-level match. The phyla are disjoint so no subtree belongs to more than one phylum. A **constructor** is a function that creates a tree node and labels it with the name of the constructor. Each constructor is associated with exactly one phylum, although multiple constructors may share the same phylum. Lastly, **root.phylum** is a pragma and it declares **Grammar** phylum as the type of CFG root in the forest of phyla. Note that pragma provides hints to analysis but does not change the semantic description. After describing the APS tree definition syntax and how the declarative nature of it enables programmers to succinctly define attribute grammars, the rest of this section continues with why circularly defined problems matter so much.
module GRAMMAR[] begin
  phylum Grammar;

  phylum Item;
  phylum Items := SEQUENCE[Item];

  phylum Production;
  phylum Productions := SEQUENCE[Production];

  constructor terminal(s: Symbol) : Item;
  constructor nonterminal(s: Symbol) : Item;
  constructor prod(nt: Symbol; children: Items) : Production;
  constructor grammar(prods: Productions) : Grammar;

  pragma root_phylum(type Grammar);
end;

Figure 1.1: Structure of a context-free grammar in APS

This code is included in grammar.aps

\[
\text{FIRST}(N) = \bigcup_{N \rightarrow \alpha} \text{FIRST}(\alpha)
\]

\[
\text{FIRST}(\epsilon) = \{\epsilon\}
\]

\[
\text{FIRST}(x\beta) = \{x\}
\]

\[
\text{FIRST}(N\beta) = \text{FIRST}(N) \cdot \text{FIRST}(\beta)
\]

\[
\text{FOLLOW}(N) = \bigcup_{N' \rightarrow \alpha N\beta} \text{FIRST}(\beta) \cdot \text{FOLLOW}(N')
\]

\[
S \cdot S' = (S - \{\epsilon\}) \cup S' \quad \text{when } \epsilon \in S
\]

\[
S \cdot S' = S \quad \text{when } \epsilon \notin S
\]

Expressing the First and Follow functions in an imperative way is not straightforward as it would require mutually recursive functions and some kind of store. However, by using a declarative pattern-matching statement, it can be solved without a need to transform the original definition. Figure 1.2 shows how the “·” function that is defined using a mathematical notation can be succinctly implemented thanks to the declarative coding style. Note that
symbols \( \setminus \) and \( \backslash \) in APS define the union and intersection set operations respectively.

\[
\text{function black\_dot(s1 :SymbolLattice; s2 :SymbolLattice): SymbolLattice begin}
\]
\[
\text{if contains\_epsilon(s1) then}
\text{result := (s1 \setminus \{ epsilon \}) \backslash s2;}
\text{else}
\text{result := s1;}
\text{endif;}
\text{end;}
\]

Figure 1.2: APS function to describe “·” operation defined in Equation (1.1)

This code is included in first.aps

Figure 1.3 is a APS code that declares attributes needed for First and we notice how some attributes are declared with circular keyword. In APS, by declaring an attribute circular it means the instance of this attribute may have circular dependencies or participate in a cycle. The circular keyword in this case was used for attributes such as item\_first and global collection variables such as firstTable to indicate attributes that may be involved in a cycle. Using a circular keyword is needed because, in a top-level match for non-terminal, it writes to item\_first a value coming from the global collection variable firstTable. Then it uses items\_first which is an aggregate of item\_first in another top-level match prod to write to the global collection variable firstTable. This circular nature of the problem even for a single non-terminal can be observed with any left recursive production, for example, \( A \rightarrow A a | \epsilon \). Lastly, APS requires that the type of a circular attribute must be a lattice. Declaring an attribute circular helps the analyzer, especially when APS tries to find the strongly connected components and cycles to find a correct static schedule and static schedule validation purposes. Also, notice this code takes advantage of global collection variables to describe the firstTable, a map between a symbol and a set of associated symbols. As a way to describe a map from Symbol to Symbols (set of symbols), this code uses a TABLE\_LATTICE and it is a type of lattice that treats items in a map as a set of key-value pairs and combines the values for the join operation of the lattice.

In Figure 1.3 each terminal and non-terminal is of type of Item and it has a circular
type Symbols := SET[Symbol];
type SymbolLattice := UNION_LATTICE[Symbol, Symbols];
type DeclarationTable := TABLE_LATTICE[Symbol, SymbolLattice];

var circular collection firstTable : DeclarationTable;
epsilon : Symbol := make_symbol("epsilon");

circular attribute Item.item_first : SymbolLattice;
circular attribute Items.items_first : SymbolLattice;
attribute Grammar.grammar_first : DeclarationTable;

pragma synthesized(item_first, items_first, grammar_first);

Figure 1.3: Assigning attributes to non-terminals
This code is included in first.aps

attribute called item_first. For the terminal, item_first is its symbol value. For non-terminal, however, one needs to look at the associated production(s) and find the first of them. That information is stored in a global collection variable which is a map of non-terminal to its first result. A sequence of one or more Item is called Items. The first of Items is a circular attribute called Items_first and is passed up and ultimately set in that global map variable inside the production top-level match rule. This APS code is semantically equivalent to the definition in Equation (1.1) and yields the correct result.

1.4 Structure of Thesis

The structure of this thesis is as follows: In the next chapter, it discusses the related works, most notably circular remote attribute grammar work by Hedin [21] which is the closest paper to this thesis. Thereafter the third chapter talks about definitions needed to understand the problem this thesis is trying to solve: definitions such as schedule, \(l\)-ordered, remote attribute grammars, and circular evaluations. Then the fourth and fifth chapter talks about the methods and implementation respectively to tackle the scheduling of circular remote attribute grammars, including visit sequence evaluation and SCC scheduler. Finally, the
module FIRST[T :: var GRAMMAR[]] extends T begin
  match ?self:Grammar=grammar(?prods: Productions) begin
    self.grammar_first := firstTable;
  end;

  match ?self:Production=prod(?nt:Symbol, ?items: Items) begin
    firstTable :> DeclarationTable$table_entry(nt, items.items_first);
  end;

  match ?self:Item=terminal(?s:Symbol) begin
    self.item_first := { s };
  end;

  match ?self:Item=nonterminal(?s:Symbol) begin
    case DeclarationTable$select(firstTable, s) begin
      match DeclarationTable$table_entry(?item_first_objs) begin
        self.item_first :> item_first_objs;
      end;
    end;
  end;

  match ?self : Items = Items$none() begin
    self.items_first :> { epsilon };
  end;

  match ?self : Items = Items$single(?item : Item) begin
    self.items_first := item.item_first;
  end;

  match ?self : Items = Items$append(?items1,?items2 : Items) begin
    self.items_first := black_dot(items1.items_first, items2.items_first);
  end;
end;

Figure 1.4: First function implementation in APS
Continuation of Figure 1.3. This code is included in first.aps
sixth chapter which is titled Evaluations discusses the benchmarks of various phases all the way from APS code to Scala-generated AG evaluator code.
Chapter 2

Related Work

This chapter discusses related works including visit sequence evaluation [17], extensions to classical attribute grammars such as: circular attribute grammars [13], remote attribute grammars [8] and circular remote attribute grammars [21] and lastly attribute grammar systems and declarative program analysis tools such as Datalog [26] and FLIX [19].

2.1 Visit Sequence Evaluator

Kastens in [17] introduced a class of attribute grammars called $l$-ordered (also called partitionable attribute grammars in [25]). This is the largest class of grammars for which “simple” multi-visit sequences can be constructed. Simple in this context refers to a fixed number of visits associated with each non-terminal. Furthermore, given a visit sequence, a static evaluation method called visit sequence evaluation can be constructed. This attribute evaluator is static in the sense that the order of evaluation at each node of the tree is determined independent of a specific derivation. Visit sequence evaluator is an interpreter-type program that can evaluate all attribute instances given a derivation. This is in contrast with demand evaluators that generate a schedule for a specific derivation and this schedule cannot be reused for other derivations. This makes visit sequence evaluators efficient and suitable for practical applications.
2.2 Circular Attribute Grammar (CAG)

In the classical attribute grammar, circularities between attributes are not allowed. That is, no attribute instance in any derivation may be defined in terms of itself. However, in mathematics or program analysis, circular (or recursive) definitions are common and essential. Given certain constraints, a fixed-points in the evaluation of circular attribute grammars is possible. Farrow [13] suggested finitely recursive attribute grammars. This is a class of attribute grammar that allows circularities in attribute dependency and it is defined as an extension of the class of non-circular AG. In essence, this class of AG imposes a constraint that results in all attributes involved in cyclic dependencies having a fixed point that can be computed with a finite number of iterations. In Farrow’s approach, values of circularly defined attribute instances are determined by means of a finite iteration of the evaluation of associated semantic rules. Broadly speaking, if there are cycles of attribute dependency and if there exists a fixed-point (or a minimum solution if the attribute type is a set) for each attribute in the cycle, then the evaluator can compute the attribute values using successive approximation. This means a guarantee that the values for each attribute in a cycle can be organized in a lattice of finite height if all the semantic functions involved in computing these attributes are monotonic in their respective lattices.

The problems of deciding whether a traditional AG is circular and identifying the attributes taking part in cycles have been addressed by several researchers. Knuth [18] developed a polynomial algorithm for circularity testing. The algorithm is conservative, i.e., circularity is always detected but some non-circular AGs may be reported to be circular. Later Knuth constructed an exact algorithm that is exponential in time and space complexity. Rodeh and Sagiv [23] extended these algorithms to deal with the problem of finding circular attributes. They developed a polynomial approximation algorithm, i.e., all circular attributes are discovered but some non-circular attributes may be reported to be circular. They also constructed an exact algorithm with exponential time complexity.

Farrow also introduced a recursive synth-function evaluator which is a type of partially
dynamic evaluator. For each synthesized attribute of each class, a function is generated. There is a total order which represents the sequence of attribute evaluation in the synth function. Farrow’s synth function could benefit from caching by using a global dictionary to store attribute instance values. Farrow’s evaluator allowed nested loops, that is if there is another cycle occurring during the iterated evaluation of circularly defined attribute values. This causes inefficiency since the number of iterations of the innermost loop becomes an exponential factor of the nested level of the loop in the worst case.

Jones [16] introduced a demand evaluation method for circular attribute grammars using strongly connected components to identify inter-dependencies among attributes and repeated evaluation of these regions until a fixed point is reached. More specifically, attribute computation is viewed as a fixed-point computation over a system of simultaneous equations, which can have a unique least fixed point under certain conditions. Efficient algorithms are presented for exhaustive and incremental evaluation of circular attributes by partitioning the attribute dependency graph into strongly connected components and ordering evaluations based on a topological sort of the resulting directed acyclic graph of components. These algorithms are efficient when considering only affected attributes and their dependencies. The algorithm guarantees that each strongly connected component is evaluated only once, resulting in global consistency. This paper also provides an efficient method for computing the fixed points of strongly connected components using monotone semantic functions and finite lattices. This algorithm iteratively evaluates the attributes in a depth-first order, considering only those attributes whose values have changed since the last evaluation. By maintaining sets of attributes to be evaluated in each iteration, the algorithm converges to the least fixed point of the strongly connected component. Furthermore, the efficiency of the algorithm depends on the size of the partitioned attribute sets and the method used for computing fixed points.
2.3 Remote Attribute Grammars (RAG)

Knuth introduced the use of “collections” in his original attribute grammar paper but attribute grammar can benefit substantially from objects and references. Popular programming languages have a concept of objects and it feels natural for developers to take advantage of them in the context of an attribute grammar. Boyland [8] and separately Hedin [14] defined an attribute grammar extension called remote attribute grammars or RAG for short that permits references to objects with fields to be passed through the attribute system. Fields may be read and written to through these references. Remote attribute grammars (RAGs) allow non-local dependencies and connect distant nodes of the derivation tree. A major disadvantage of classical attribute grammar is that the definition of AG becomes too low-level when dealing with non-local dependencies, that is, situations where an attribute of one non-terminal node instance is dependent on attributes of non-terminal nodes far away in the tree. Classical attribute grammars are well-suited for problems where the dependencies are local and follow the derivation tree structure. However, classical attribute grammars are less suited for problems with non-local dependencies, such as unused variable detection and name analysis problems. Typically, the use and declaration of an attribute can be arbitrarily far away from each other in the tree, and any information propagated between them needs to involve all intermediate nodes. There are several drawbacks to this. In RAG, attributes are allowed to be references to nodes in the derivation tree. Thus, it abandons the value semantics and introduces reference semantics. Structured attributes like sets, dictionaries, etc., may also include reference values. The use of reference values makes attribute grammars well-suited for expressing problems with non-local dependencies that do not necessarily follow the derivation tree structure.

It has been shown earlier by Hedin how RAGs support the easy specification and automatic implementation of many practical problems. For example, name and type analysis of object-oriented languages [14], execution time prediction [22], program visualization [20], and design pattern checking [11].
Boyland \[8\] described “infinite fiber construction” which is the fundamental concept that makes static scheduling of remote attribute grammars possible as it expresses the semantics of remote attribute grammars in classical terms. This is called a “fiber construction” because each object, which implicitly carries a number of separate values for the fields is seen as a “rope” which can be separated into individual fibers. A field can be (partially) written at multiple points, but whenever the field is read, it gets the final value collected from all units. This is important as it makes the construction of evaluators for RAG more involved in contrast with classical AG.

### 2.4 Circular Remote Attribute Grammars (CRAG)

Circular remote attribute grammars were first introduced by Hedin \[21\]. Traditionally, an attribute is either used to propagate information upwards in the tree (synthesized attribute) or downwards in the tree (inherited attribute). In the original form, classical attribute grammar introduced by Knuth, the definition of an attribute allows depending directly only on attributes of neighbor nodes in the tree and the dependencies between attributes may not be cyclic. The first of these restrictions are lifted in remote attributed grammars (RAG) by Boyland \[8\] where an attribute may be a reference (or a pointer) to an arbitrarily distant node in the tree, and an attribute defined in a semantic function may directly access attributes of the reference. The second of the restrictions mentioned above, circular definitions, is lifted in a class of circular attribute grammars (CAG) by Farrow \[13\].

Dependencies between attribute instances in a derivation tree can be modeled as a directed graph. If the dependency graph is acyclic for every possible derivable syntax tree for a certain grammar, the grammar is said to be non-circular. For non-circular grammars, it is always possible to topologically order the dependency graphs and evaluate the attributes by applying the semantic functions in that order. Traditional AGs are required to be non-circular, but as has been shown by Farrow, grammars with circular dependencies under
certain constraints can be considered well-formed in the sense that it is possible to satisfy all semantic rules for all possible syntax trees. One way to formulate the constraints is to require that the domain of all attributes involved in cyclic chains can be arranged in a lattice of finite height and that all semantic functions for these attributes are monotonic. The evaluation of circularly defined attributes can be regarded as a special case of solving the equation $x_{i+1} = f(x_i)$. By giving $x_0$ the bottom value of the lattice (⊥) as the start value, the iterative process will converge to a least fixed point for all involved semantic rules. The problem of testing remote attribute grammars for circularity can be solved by recording all possible reference edges to the dependency graph, and then testing the dependency graph for circularity.

In short, Hedin [21] introduced a robust demand evaluation algorithm to evaluate CRAG. It evaluates strongly connected components in topological order, avoids recomputation of potentially circular attributes, and runs evaluation until a fixed point is reached.

2.5 Program Analysis Tools Based on AGs

This section describes various program analysis tools including APS, JastAdd, Datalog, and FLIX. In order to have a consistent comparison, implementation of nullable function is used. Nullable [21] is a simple to understand and implement function defined over context-free grammar. More specifically, a context-free grammar’s non-terminal is nullable if any of its production rules is nullable. A production is nullable if all of its symbols are nullable. A terminal is not nullable under any circumstances. However, a non-terminal is nullable if any of its declarations is nullable. This is formally defined as follows:

(i) Let $X$ be a non-terminal with the following declarations: $X \rightarrow \alpha_1, X \rightarrow \alpha_2, \ldots, X \rightarrow \alpha_n$. $X$ is nullable if any of its production right-hand sides is nullable. This is synonymous to an OrLattice where the bottom value $\bot = \text{false}$ and the top value $\top = \text{true}$. 

16
\( \text{nullable}(X) = \text{nullable}(\alpha_1) \lor \text{nullable}(\alpha_2) \lor \cdots \lor \text{nullable}(\alpha_n) \)

(ii) Let \( \epsilon \) be an empty sequence of terminal and non-terminal symbols (or \( \epsilon \) for short). The empty sequence is always nullable.

\[ \text{nullable}(\epsilon) = \text{true} \]

(iii) Let \( \alpha = \beta \delta \gamma \) be a possibly empty sequence of terminal and non-terminal symbols. \( \alpha \) is nullable if all three of \( \beta, \delta \) and \( \gamma \) are nullable.

\[ \text{nullable}(\alpha) = \text{nullable}(\beta) \land \text{nullable}(\delta) \land \text{nullable}(\gamma) \]

(iv) A terminal symbol \( t \) is always not nullable.

\[ \text{nullable}(t) = \text{false} \]

The above conditions are summarized in the following example production:

\[
\text{nullable}(p) = \begin{cases} 
  \text{true} & A \rightarrow B \mid a \\ 
  \text{true} & B \rightarrow \epsilon \\ 
  \text{false} & C \rightarrow ABC \mid c 
\end{cases}
\] (2.1)

2.5.1 JastAdd

JastAdd \cite{21} takes an imperative approach to defining and evaluating attribute grammars. It is built on top of the parser generator JavaCC to parse the program and build the abstract syntax tree. However, the actual implementation is completely independent of JavaCC. The attribute evaluator used in JastAdd is a type of recursive demand evaluation. If the dependencies contain cycles, these are detected during attribute evaluation. In demand
evaluation, access to attribute values is replaced by functions that compute the semantic
function for the value and then cache the computed value for potential future accesses.
JastAdd combines both declarative and imperative approaches. It allows results computed by
declarative modules to be accessed by imperative modules. Imperative modules are written
in ordinary Java code. In declarative modules, JastAdd supports remote attribute grammars
(RAG). The RAG modules are specified in an extension to Java and are translated to ordinary
Java code by the system. More recently, Hedin [21] introduced the CRAG module which
enables JastAdd to evaluate circular remote attribute grammars using demand evaluation.

Figure 2.1 is an implementation of nullable in JastAdd. Notice the use of imperative
programming constructs such as a for-loop. Using imperative semantics and a syntax close
to Java makes it challenging to come up with a static scheduler and a code generation that
is target language independent.

2.5.2 APS

Boyland [6] introduced APS, a compiler description language. APS extends the attribute
grammar formalism with pattern matching, sequences, functions, higher-order language fea-
tures, and modules. APS provides the ability to define attributes or typed values that are
associated with particular nodes. The programmer specifies the values by expressing an at-
tribute as a function of other values, including other attributes. The APS compiler schedules
the evaluation of the attributes so that no attribute is used before it is defined. Automatic
scheduling, as well as demand evaluation, are some of the main benefits of attribute-grammar-
based systems such as APS.

The APS language includes different programming language constructs such as conditionals, global and local variables, functions, and polymorphism. The conditionals are either
if-else statements or matches and they impact conditional total order (CTO) and subse-
sequently visit sequence generation. Global variables allow factoring out a common value but
they impact the dependency graph because it induces an edge from the declaration of the
aspect Nullable {
    inh boolean NDecl.nullable() circular [false];
    syn boolean Prod.nullable() circular [false];
    syn boolean Symbol.nullable() circular [false];

    eq Terminal.nullable() {
        return false;
    }

    eq NUse.nullable() {
        return decl().nullable();
    }

    eq Rule.getNDecl().nullable() {
        for (int i = 0; i < getNumProd(); i++) {
            if (getProd(i).nullable())
                return true;
        }
        return false;
    }

    eq Prod.nullable() {
        for (int i = 0; i < getNumSymbol(); i++) {
            if (!getSymbol(i).nullable())
                return false;
        }
        return true;
    }
}

Figure 2.1: Nullable implementation in JastAdd
global variable to its use. All these programming constructs allow an environment where the programmer can define semantics using familiar programming idioms and notations. Then APS analyzes them and subsequently generates code in either Scala or C++ such that given a derivation tree, it would evaluate the defined semantics.

Figure 2.2 is an APS implementation of nullable. This is declarative in contrast with JastAdd implementation and it uses a construct called the top-level match to encapsulate semantic rules for production. The APS implementation below uses a construct called OrLattice with the bottom value of false and uses logical OR as a join operation. Basically, using OrLattice makes it possible to express in a declarative way that if there is any item in production that is nullable, then the whole production becomes nullable.

```plaintext
type NullableTable := TABLE_LATTICE[Symbol, OrLattice];
var circular collection nullableTable : NullableTable;

circular attribute Item.item_nullable : OrLattice := false;
pragma synthesized(item_nullable);

circular attribute Items.items_nullable : OrLattice := false;
pragma synthesized(items_nullable);

circular attribute Grammar.grammar_nullable : NullableTable;
pragma synthesized(grammar_nullable);
```

Figure 2.2: Assigning attributes to non-terminal for Nullable example
This code is included in nullable.aps

2.6 Other Declarative Program Analysis Tools

This section briefly overviews closely related works that are not based on attribute grammars, but their motivation is to tackle program analysis in a declarative manner. Thereafter, a nullable function will be implemented for each program analysis tool.
module NULLABLE[T :: var GRAMMAR[]] extends T begin
epsilon : Symbol := make_symbol("epsilon");

match ?self:Item=terminal(?s:I) begin
    self.item_nullable := s == epsilon;
end;

match ?self:Item=nonterminal(?s:I) begin
    case NullableTable$select(nullableTable, s) begin
        match NullableTable$table_entry(?,?item_nullable_boolean) begin
            self.item_nullable := item_nullable_boolean;
        end;
    end;
end;

match ?self:Production=prod(?nt:Symbol, ?items: Items) begin
    nullableTable :> NullableTable$table_entry(nt, items.items_nullable);
end;

match ?self:Grammar=grammar(?prods: Productions) begin
    self.grammar_nullable := nullableTable;
end;

match ?self : Items = Items$none() begin
    self.items_nullable := true;
end;

match ?self : Items = Items$single(?item : Item) begin
    self.items_nullable := item.item_nullable;
end;

match ?self : Items = Items$append(?items1,?items2 : Items) begin
    self.items_nullable := items1.items_nullable and items2.items_nullable;
end;
end;

Figure 2.3: Nullable implementation in APS
Continuation of Figure 2.2. This code is included in nullable.aps
2.6.1 Datalog with Binary Decision Diagrams

Datalog is a declarative logic programming language that uses a subset of Prolog syntax. Unlike Prolog which is untyped, Datalog is weakly typed. Datalog can be used as a query language for relational databases; however, unlike core SQL, Datalog queries support recursion. One useful property of Datalog is that a Datalog program is guaranteed to terminate with the minimal model or smallest Herbrand interpretation. This means a programmer can write the constraints of a problem in Datalog and then the solver module handles the specifics of computing the solution. The programmer is assured that the analysis of constraints will terminate. Additionally, Datalog allows the programmer to choose between different solver implementations thus giving control to the programmer to select a specific optimization. These advantages make Datalog a useful language for implementing static analysis, such as Pointer Analysis of Java programs [2, 9, 24, 27]. The declarative nature of the analysis makes it easier to understand and maintain.

Lam [26] describes “bddbdoo” which is an implementation of Datalog with stratified negation, totally-ordered finite domains, and comparison operators with a focus on program static analysis. The following overview of the major optimization of Datalog has been done in this paper. The first improvement is the ability to remove rules and relations that do not directly or indirectly contribute to the output relations. This is done with the help of a predicate dependency graph (PDG) which records dependencies between rules and relations which consist of a countable set of predicate symbols. Another important contribution made in this paper is a guarantee that the relation can be fully computed before applying rules. This is built upon the fact that the ordering of the clauses is irrelevant in Datalog, in contrast to Prolog, which depends on the ordering of clauses for computing the result of the query call. Lastly, the ability to find cycles in the PDG and indicate that some rules and relations are recursively defined, thus requiring iterative application of rules within the cycles to reach a fixed-point solution. Then, the PDG is then separated into a set of strongly connected components and the Datalog solver computes the results for each strongly
connected component and non-cyclic relations according to the topological order of the PDG.

Figure 2.4 describes a succinct Prolog implementation of Nullable and it’s a good starting point for Datalog implementation as Datalog uses a subset of Prolog syntax. However, converting the code from Prolog to Datalog requires some changes as Datalog supports only atoms and variables and does not support lists and other Prolog terms. The first change is the introduction of a special symbol called nothing which is neither terminal nor a non-terminal and is hard-coded as nullable. The tricky part of conversion is expressing the fact that a production is not nullable if at least one of its symbols is not nullable. This uses an auxiliary predicate to correctly nest the negations. Figure 2.5 is a Nullable implementation in Datalog.

```
% Ys is a list of symbols
nullable_nonterminal(X) :-
    variable(X),
    rule(X,Ys),
    \+ member(X, Ys),
    nullable_nonterminals(Ys).

% The empty list is always a nullable list
nullable_nonterminals([]).

% A list is nullable if all of its components are nullable
nullable_nonterminals([X|Xs]) :-
    nullable_nonterminal(X),
    nullable_nonterminals(Xs).

% A production rule _ -> Ys is nullable if Ys is nullable
nullable_rule(rule(_,Ys)) :-
    nullable_nonterminals(Ys).
```

Figure 2.4: Nullable implementation in Prolog

2.6.2 FLIX, Declarative Language for Fixed Points on Lattices

FLIX [19] is a functional, imperative, and logic programming language that language supports algebraic data types, pattern matching, parametric polymorphism, currying, higher-
nullable_symbol(nothing).
nullable_symbol(Symbol) :-
    terminal(Symbol),
    nullable_terminal(Symbol).
nullable_symbol(Symbol) :-
    nonterminal(Symbol),
    nullable_nonterminal(Symbol).

nullable_terminal(_Symbol) :-
    false.

nullable_nonterminal(Nonterminal) :-
    rule(_Rule, _Index, Nonterminal, Production),
    nullable_production(Production).

nullable_rule(Rule) :-
    rule(Rule, _Index, _Nonterminal, Production),
    nullable_production(Production).

nullable_production(Production) :-
    not(some_nonnullable_symbol(Production)).

some_nonnullable_symbol(Production) :-
    production(Production, _Index, Symbol),
    not(nullable_symbol(Symbol)).

Figure 2.5: Nullable implementation in Datalog
order functions, extensible records, channel and process-based concurrency, and tail call
elimination. FLIX programs compile to JVM and its type system is based on a well-known
Hindley-Milner type-system with extensions including row polymorphism and boolean uni-
fication.

FLIX consists of a logic language and a functional language. The logic component is
inspired by Datalog, but unlike Datalog, it supports user-defined lattices. A functional
language, unlike Datalog, allows users to write functions. The unique features of FLIX are
user-defined lattices and monotone functions over their elements. This enables users wide
range of analyses in FLIX that cannot be expressed by Datalog easily. For example, unused
variable detection or constant propagation. Datalog is limited to constraints on relations,
and although it can simulate finite lattices, it cannot express lattices over an infinite domain.
Finally, another advantage of FLIX is that it supports interoperability with existing tools
written in general-purpose programming languages in the JVM family of languages.

FLIX shares some similarities with Bloom$^T$, a language for ensuring consistency in dis-
tributed programs [10]. Bloom$^T$ extends Bloom, a language based on Datalog, with lattices
and monotone functions. While FLIX programs compute the minimal model of a fixed-point
problem and terminate, whereas Bloom$^T$ programs may run continuously.

Figure 2.6 is an implementation of Nullable in FLIX. This example does not use the
fixed-points lattice feature of FLIX that is available as Datalog constraints because it only
supports constraints where parameters are atomic. For example, facts of the form $\text{Edge}(x,y)$
where $x$ and $y$ are atoms. For the case of a CFG, this implementation uses facts of the form
$\text{Rule}(x,y)$ with $x$ an atom but $y$ is not an atom, $y$ is a string/list that has to be decomposed.
FLIX also does not support recursion. Notice the use of $\text{List.filter}$ function to make up
for recursions.
// epsilon is represented as the empty word
def variables(): List[String] = _
def rules(): List[List[String]] = _

// gets the right hand side of a rule
def getRightHandSide(r: List[String]): String =
  // get the second element of r (if any)
  let x = List.head(List.drop(1,r));
  // then get its value. If no such value then assign ",", the empty string
  Option.getWithDefault(x,"")

// gets the right hand side of rules with v as left hand side and
// whose right hand side does not contain v
def getRules(v: String): List[String] =
  // first remove rules that do not contain v as right hand side
  let y = List.filter(x -> List.isPrefixOf(v::Nil, x), rules());
  // then get the right hand side of such rules
  let z = List.map(getRightHandSide, y);
  // and then ignore looking at right hand sides in which v appears
  List.filter(x -> List.indexOf(String.charAt(0,v),String.toList(x)) == -1, z)

// a string of symbols is nullable if all its characters are
def nullableStr(s: String): Bool =
  let cList = String.toList(s);
  List.forall(x -> nullable(x), cList)

// a rule is nullable if its right hand side is a nullable string
def nullableRule(r: List[String]): Bool =
  let x = getRightHandSide(r);
  nullableStr(x)

// a character is nullable if it is a variable and either has an epsilon rule
// or a rule in which the variable does not appear and is a nullable string
def nullable(v: Char): Bool= 
  let isVariable = List.indexOf(Char.toString(v), variables()) != -1;
  let hasEpsilonRule = List.indexOf(Char.toString(v)::"":Nil, rules()) != -1;
  let vRules = getRules(Char.toString(v));
  isVariable and (hasEpsilonRule or List.exists(x -> nullableStr(x), vRules))

Figure 2.6: Nullable implementation in FLIX
Chapter 3

Definitions

This chapter describes the formal definitions of the structures needed to allow the static scheduling of circular remote attribute grammars. More specifically, it describes context-free grammars, variations of attribute grammars including classical, circular, remote, and circular-remote, and evaluation methods of attribute grammars including demand evaluation and static evaluation. This chapter relies on [8] for definitions of classical and remote attribute grammars.

3.1 Context-Free Grammar

Definition 3.1.1. Formally a context-free grammar is a tuple of $G = (T, N, P, Z)$ where

1. $T$ is a set of terminal symbols which are the characters that appear in the language/strings generated by the grammar. Terminal symbols never appear on the left-hand side (LHS) of the production rules but may appear on the right-hand side (RHS).

2. $N$ is a set of non-terminal symbols that are placeholders for patterns of terminal symbols. These are the symbols that will always appear on the left-hand side of the production rules, though they can be included on the right-hand side. The strings that
3. \( P \) is a set of production rules which are the rules for replacing non-terminal symbols on the left-hand side of the production with other terminal or non-terminal symbols on the right-hand side of the production. Formally, \( A \to \alpha \) where \( A \in N \) and \( \alpha \in (T \cup N)^* \). As a shorthand, it is common to define a set of variables \( \Sigma = (T \cup N) \).

4. \( Z \) is a start symbol which is a special non-terminal symbol that generates the strings of the language.

A derivation for grammar is formally defined in Definition 3.1.3 as a sequence of grammar rules starting from the start symbol, and then repeatedly replacing a non-terminal with the right-hand side of a production rule. Figure 3.1 is an example of context-free grammar and its sole trivial derivation.

Definition 3.1.2. The symbol \( ( \rightarrow ) \) is a relation between strings of tokens and non-terminals; \( \alpha B \gamma \rightarrow \alpha \beta \gamma \) such that \( \alpha, \beta, \gamma \in \Sigma^* \) where \( (\cdot)^* \) is a Kleene star and \( B \in N \) when there is a production \( (B \to \beta) \in P \).

Definition 3.1.3. A sequence of steps \( \alpha_0 \rightarrow \alpha_1 \rightarrow \cdots \rightarrow \alpha_n \) where \( \alpha_0 = Z \) is called a Derivation \( \mathcal{D} \) and it is written as \( Z \xrightarrow{\mathcal{D}} \alpha_n \) such that \( \alpha_n \in T^* \).

\[
T = \{a, b\}, \quad N = \{S, A, B\}, \quad P = \{S \to AB, \quad A \to a, \quad B \to b\}, \quad Z = S
\]

\[
\mathcal{D} : S \rightarrow AB \rightarrow aB \rightarrow ab
\]

Figure 3.1: Example of context-free grammar and its trivial derivation
The notations are according to Definition 3.1.1

A derivation tree \( (\mathcal{T}) \) is a graphical representation for the derivation of the given production rules of CFG. It is a simple way to show how the derivation can be done to obtain some string from a given set of production rules. In a derivation tree, the root is the starting non-terminal \( Z \), all internal nodes are labeled with non-terminals, while all leaves are
labeled with terminals. The children of an internal node are labeled from left to right with the right-hand side of the production used. In practice, a derivation and derivation tree are used interchangeably. Figure 3.2 is an example of a derivation tree.

Figure 3.2: Equivalent derivation tree for derivation in Figure 3.1

Definition 3.1.4. If we have a derivation sequence: \( \alpha_0 \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_n \), where each \( \alpha_i \) is a string in \( \Sigma^* \), every occurrence of a non-terminal symbol in each \( \alpha_i \) is a non-terminal instance that can be replaced according to the production rules in \( P \) of CFG.

Definition 3.1.5. A derivation can be described as \( \hat{D} = \hat{\alpha}_0 \Rightarrow \hat{\alpha}_1 \Rightarrow \cdots \Rightarrow \hat{\alpha}_n \) where \( \hat{\alpha}_i \in (\Sigma \cup (N \times N)) \) is a string of terminals and instances of non-terminals with the form of \((X, i)\) where \( i \) corresponds to the index of transition step in \( \hat{D} \). For each \( \hat{\alpha}_i \Rightarrow \hat{\alpha}_{i+1} \) such that \( \hat{\alpha}_i = \hat{\beta}(X, i)\hat{\gamma} \) and \( \hat{\alpha}_{i+1} = \hat{\beta}\hat{\delta}\hat{\gamma} \) and exists \( p = X \rightarrow \delta \) is in \( P \) and \( |\hat{\delta}| = \delta \). Where operation \( |(X, i)| = X, |t| = t \) for non-terminals and terminals respectively, and then we lift the operation to strings of symbols. This way \( |\hat{\delta}| = \delta \) means that \( \delta \) is \( \hat{\delta} \) with instances removed.

In Figure 3.2, \( S, A, \) and \( B \) in the derivation tree are non-terminal instances, and this is described by the \((X, i)\) syntax, where \( X \in N \), and \( i \) denotes the belonging \( \alpha_i \) where in \( \alpha_{i+1} \) the non-terminal \( X \) will be expanded by its associated production.

A production is called recursive when there exists a production rule in \( P \) of the form: \( X \rightarrow \beta X \gamma \) where \( X \in N \), and \( \beta, \gamma \in \Sigma^* \). Since \( X \) appears on the left-hand side of the production rule with the right-hand side also including \( X \), this is called a recursive production. Figure 3.3 is an example of such grammar. To resolve the ambiguity, we assign an index \( j \) to each non-terminal \( X \) in a production, for instance, \((X:j)\). Similarly, \((X:0)\)
indicates the LHS non-terminal of a production. As a result, a particular non-terminal instance involved in a recursive production $p:X_0 \rightarrow X_1 \ldots X_m$ is described by the $(X:j,i)$ syntax where $X \in N$, $j$ denotes the non-terminal index in production $p$ and $i$ denotes the belonging $\alpha_i$.

For example, for a trivial derivation as described in Figure 3.1, $D$ as defined in Definition 3.1.3 is: $D = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3$ where $\alpha_0 = S$, $\alpha_1 = AB$, $\alpha_2 = aB$ and $\alpha_3 = ab$. Then $\hat{D}$ as defined in Definition 3.1.5 is $\hat{\alpha}_0 \Rightarrow \hat{\alpha}_1 \Rightarrow \hat{\alpha}_2 \Rightarrow \hat{\alpha}_3$ where $\hat{\alpha}_0 = (S,0)$, $\hat{\alpha}_1 = (A,1)(B,2)$, $\hat{\alpha}_2 = a(B,2)$ and $\hat{\alpha}_3 = ab$.

$$T = \{a\}, N = \{A\}, P = \{A \rightarrow AA, A \rightarrow a\}, Z = A$$
$$\mathcal{D} : A \Rightarrow AA \Rightarrow aA \Rightarrow aa$$
$$\hat{\mathcal{D}} : \sigma_0 \Rightarrow \sigma_1 \Rightarrow \sigma_2 \Rightarrow \sigma_3$$
where $\sigma_1 = (A:0,0)$, $\sigma_1 = (A:1,1)(A:2,2)$, $\sigma_2 = a(A:2,2)$, $\sigma_3 = aa$

Figure 3.3: Example of context-free grammar with recursive production
The notations are according to Definition 3.1.1

### 3.2 Classical Attribute Grammar

A classical attribute grammar (or AG) is a context-free grammar with an addition of two disjoint sets of inherited and synthesized attributes for each non-terminal, a set of local attributes, and a set of semantic rules associated with each production [18]. One of the many applications of attribute grammars is to define the semantics or some kind of meaning to a program in a language. Attribute grammars define semantics by specifying that certain attributes will be associated with certain nodes in the syntax tree and providing equations that specify how the values of these attributes are to be computed.

**Definition 3.2.1.** A classical attribute grammar is a tuple $A = (G, S, I, L, R)$ where

1. $G = (T, N, P, Z)$ is a context-free grammar as defined in Definition 3.1.1
2. $S$ is a set of “synthesized attributes” for each non-terminal $X \in N$. We define a shorthand for all synthesized attributes in this attribute grammar as: $\text{Syn} = \bigcup_{X \in N} S(X)
$

3. $I$ is a set of “inherited attributes” for each non-terminal $X \in N$. We define a shorthand for all inherited attributes in this attribute grammar as: $\text{Inh} = \bigcup_{X \in N} I(X)$

4. $L$ is a set of “local attributes” for each production $p \in P$

5. $R$ is a set of semantic rules of the form $v_0 = g(v_1, \ldots, v_k)$ for each production $p \in P$, where $g$ is a function, and $v_i \in V(p)$ are “attribute occurrences” of either the form $(X:j).a$ or $l$. In the case of $(X:j).a$, $a \in (I(X) \cup S(X))$, where $(X:j)$ is a non-terminal occurrence in $p:X_0 \rightarrow \sigma_1 \ldots \sigma_n$ such that $\sigma_i \in \Sigma$. In the second case of $l$, $l \in L(p)$. We say that in $r = v_0 = g(v_1, \ldots, v_k)$ it defines $v_0$ and uses $v_1, \ldots, v_k$. If $g$ is the identity function, then the rule can be written $v_0 = v_1$ as a shorthand. $\text{DO}(p)$ is a set of defined occurrences of $p$ and $\text{DU}(p)$ is a set of used occurrences of $p$. Also, $R(p)$ must define all the attributes in $\text{DO}(p)$ exactly once and nothing else, and rules in $R(p)$ can only use the occurrences in $\text{UO}(p)$ where:

$$\text{DO}(p) = \begin{cases} (X:j).a & a \in S((X:0)) \\ (X:j).a & a \in I((X:k)) \quad j \neq 0, k \neq 0 \end{cases} \cup L(p)$$

$$\text{UO}(p) = \begin{cases} (X:j).a & a \in I((X:0)) \quad j = 0 \\ (X:j).a & a \in S((X:k)) \quad j \neq 0, k \neq 0 \end{cases} \cup L(p)$$

$$\text{AO}(p) = \text{DO}(p) \cup \text{UO}(p)$$

**Remark 1.** Set of all inherited ($\text{Inh}$) and set of all synthesized ($\text{Syn}$) attributes must be disjoint meaning that an attribute may belong to an inherited or synthesized set and no attribute may belong to both sets: $\text{Inh} \cap \text{Syn} = \emptyset$

**Remark 2.** A classical attribute grammar is well-defined if for each attribute $a$ of non-terminals $X$, or $(X:j).a$ for short, there exists exactly one equation with left-hand side $(X:j).a$
that defines it, except obviously for the inherited attribute of the root node which is not defined.

\[
\text{Attr}(X) = I(X) \cup S(X) \quad (3.2)
\]

In Equation (3.2), \(\text{Attr}(X)\) denote the set all attributes for non-terminal \(X\).

**Definition 3.2.2** formally describes instantiated AG and instantiated nodes. Given a derivation of the form defined in **Definition 3.1.5**, an attribute grammar becomes *instantiated*: non-terminals become *nodes* in the tree and semantic rules become *instantiated rules* where they now define semantics using instantiated nodes instead of non-terminals.

**Definition 3.2.2.** An instantiated attribute grammar is \(\hat{A} = (A, \hat{D}, \hat{R})\) such that

1. \(A = (G, S, I, L, R)\) is an attribute grammar as defined in **Definition 3.2.1** where \(G = (T, N, P, Z)\).

2. Derivation \(\hat{D}\) or set of transitions as defined in **Definition 3.1.5** \((X:j, i)\) is called an instantiated node.

3. \(\hat{R}\) is a set of instantiated semantic rules, defined as follows: for each step of derivation \(\alpha_i \rightarrow \alpha_{i+1}\) which has a form of \(\hat{\alpha}(X:j, i)\hat{\beta} \rightarrow \hat{\alpha}\hat{\delta}\hat{\beta}\) we have an associated production \(p:X_0 \rightarrow \delta\) that made that derivation step possible. To get \(\hat{R}\), find a function \(\sigma\) such that \(\sigma(X:j) = (X:j, i)\) for each \((X:j, i) \in \hat{\delta}\) and then use this function to go in a sequence of steps. First, we lift sigma that applies to non-terminal occurrences and local occurrences in the normal way to apply to all attribute occurrences: \(\sigma((X:j), a) = \sigma(X:j, i).a\) and \(\sigma(l) = (l, i)\). Then we lift \(\sigma\) to apply to rules: \(\sigma(v_0=g(v_1, \ldots, v_k))\) to create \(\sigma(v_0)=g(\sigma(v_1), \ldots, \sigma(v_k))\). Then we lift \(\sigma\) to apply to sets of rules: \(\hat{R} = \bigcup_{p \in P} \sigma(R(p)) = \bigcup_{p \in P} \bigcup_{r \in R(p)} \sigma(\hat{r})\).

\(\sigma\) is easy to define given \(\hat{\alpha}(X:j, i)\hat{\beta} \rightarrow \hat{\alpha}\hat{\delta}\hat{\beta}\), in particular \((X:j, i)\) means \((X:0)\) maps to \((X:j, i)\) and each non-terminal instance in \(\hat{\delta}\) is the result of mapping the corresponding
\((X:j)\). For example if \(\hat{\delta}\) is \((A, 3)(A, 8)\), then \((A : 1)\) maps to \((A, 3)\) and \((A : 2)\) maps to \((A, 8)\).

Equation (3.3) describes instantiated tree nodes \(\hat{N}\) and all attribute instances \(\hat{A}\).

\[
\hat{N} = \{(X:j, i) \mid (X:j, i) \in \hat{D}\}
\]

\[
\hat{V} = \bigcup_{\hat{r} \in \hat{R}} \{(X:j, i).a \mid \hat{v}_i \in \hat{r} \land \hat{v}_i \equiv (X:j, i).a\} \quad \text{(3.3)}
\]

One can modify the derivation tree and map each node in the tree where each node now belongs to \((T \cup \hat{N})\). This is useful to visualize the derivation \(\hat{D}\) and quickly see the structure of all instantiated nodes.

![Figure 3.4: Equivalent derivation tree \(\hat{\mathcal{T}}\) for derivation defined in Figure 3.1](image)

The term bottom-up propagation is used when synthesized attribute values of the children are computed and passed up to the parent node. On the other hand, the term top-down propagation is when inherited attributes are set by the parent node and passed down to the child node(s). Figure 3.5 describes semantic rules over the CFG defined in Figure 3.1. \(l\) is a local variable and attribute \(s\) on non-terminals \(S, A\) and \(B\) is a synthesized attribute. Attribute \(s\) propagates attribute values up. Also, notice this attribute grammar uses only synthesized attributes; such an AG is also known as pure-\(S\) or purely synthesized attribute. Similarly, attribute grammar is called pure-\(I\) if it contains only inherited attributes. It can be seen that the pure-\(S\) and pure-\(I\) AG classes are too restrictive in practice as they only allow the propagation of attribute values in one direction. Figure 3.6 is another example of purely synthesized attribute grammar that is easier to understand as it sums the values of
the leaves and propagates the sum value up.

\[
\begin{align*}
S & \rightarrow A \ B \\
& A \rightarrow b \\
& B: B \rightarrow b \\
S . s &= A . s + B . s \\
& \text{local } l \\
& l = 'a' \\
& A . s = l \\
& \text{local } l \\
& l = 'b' \\
& B . s = l \\
\end{align*}
\]

Figure 3.5: Purely synthesized attribute grammar with local attribute

\[
\begin{align*}
E & \rightarrow E + T \\
& E0 . val = E1 . val + T . val \\
& T \rightarrow T * F \\
& T0 . val = T1 . val * F . val \\
E & \rightarrow T \\
& E . val = T . val \\
& T \rightarrow F \\
& T . val = F . val \\
F & \rightarrow \text{digit} \\
& F . val = \text{digit. lexical_val} \\
\end{align*}
\]

Figure 3.6: Example of purely synthesized attribute grammar without local attribute

Math expression addition & multiplication with the correct order of operation

\[
E \rightarrow E + T \rightarrow T + T \rightarrow T * F + T \rightarrow F * F + T \\
\rightarrow \text{digit * F + T} \\
\rightarrow \text{digit * digit + T} \\
\rightarrow \text{digit * digit + F} \\
\rightarrow \text{digit * digit + digit} \\
\]

(3.4)

Figure 3.7 provides an example of instantiated AG. \{n_0, \ldots, n_7\} are instantiated nodes in the tree where instance \(n_0\) and \(n_1\) corresponds to non-terminal \(E\) and \{\hat{r}_0, \ldots, \hat{r}_7\} are instantiated rules.

After defining semantic rules on context-free grammar, one important question that arises is the process of evaluating those rules given a derivation. There are two categories of at-
n0:E -> n1:E + n6:T
   r0: n0.val = n1.val + n6.val

n1:E -> n2:T
   r1: n1.val = n2.val

n2:T -> n3:T * n5:F
   r2: n2.val = n3.val * n5.val

n3:T -> n4:F
   r3: n3.val = n4.val

n4:F -> digit
   r4: n4.val = digit.lexical_val

n5:F -> digit
   r5: n5.val = digit.lexical_val

n6:T -> n7:F
   r6: n6.val = n7.val

n7:F -> digit
   r7: n7.val = digit.lexical_val

Figure 3.7: Instantiated pure-S AG defined in Figure 3.6
Instantiated rules using derivation defined in Equation (3.4)

Figure 3.8: Syntax tree and flow of attribute values for derivation defined in Equation (3.4)
tribute grammar evaluators: “dynamic” and “static.” Dynamic means the evaluator requires a runtime dependency graph generated dynamically for each derivation and this dependency graph will work only for that particular derivation, whereas static means the evaluator can be generated independently of a derivation.

In order for an attribute grammar to be well-defined, the attributes associated with non-terminals at any node in the derivation tree must be possible to be evaluated using the semantic rules for the attribute grammar. Non-trivial context-free grammars can produce an infinite number of trees, so the problem of determining an AG’s well-definedness is non-trivial. When discussing and utilizing attribute grammars, it is usually assumed that one is referring to well-defined category of attribute grammars. Otherwise, such an AG would not always provide practical value.

Definition 3.2.3. An attribute grammar is well-defined WD if there is exactly one defining semantic rule for each attribute given a derivation.

In a well-defined attribute grammar, the rules in production should define every DO exactly once and not anything else. The rules in \( R(p) \) can only define attributes in \( DO(p) \) and define each attribute exactly once.

Definition 3.2.4. A well-defined attribute grammar is canonical if furthermore for every production \( p \in P \) the following holds: \( DO(p) \cap DU(p) \subset L(p) \).

Definition 3.2.4 implies that in a canonical AG, we will not use a synthesized attribute to compute another synthesized attribute(s) or an inherited attribute to define other inherited attribute(s). If we want to reuse a semantic value, we can introduce an extra local variable to hold the result. This means that one can canonicalize any well-defined AG by holding on to the value in a local attribute.
3.2.1 Schedule

Informally, a schedule defines an order that if followed then no attribute is used before it is defined. More specifically, the schedule is a total-order on instantiated rules such that every write of an attribute instance comes before all reads of the same attribute instance. That order is reflexive, transitive, and most importantly total meaning that either \( a < b \) or \( b < a \) for \( a \neq b \). A proper attribute grammar can be evaluated for any tree by computing the attribute instances using the instantiated rules in the schedule. One advantage of a schedule is that evaluation takes place in a linear time since the scheduler has assured that the rules will be executed in the correct order. The disadvantage of schedule is that not all attribute grammars can be evaluated using a schedule evaluation scheme as a valid schedule may not exist. More importantly, finding the schedule is costly both in terms of space and time. More specifically, the time complexity is \( O(|V + E|) \) where \( |V| \) is a number of the attribute instances \( \hat{V} \), and \( |E| \) is a number of the edges where for all rules \( \hat{r} \in \hat{R} \) there are edges between all attribute instances in RHS of the rule \( \hat{r} \) as a source and attribute instance in the LHS of the rule \( \hat{r} \) as a destination.

The schedule could also be defined more concretely using \( DO \) and \( UO \) sets as defined in Definition 3.2.1. Note that \( v_i \) is an attribute occurrence and \( \hat{r}_i \) is an instantiated rule that defines attribute instance \( \hat{v}_i \). For example, for the instantiated rule of \( \hat{r}_0 = n_0.out=n_1.s \), \( DO = \{n_2.i\} \) and \( UO = \{n_1.s\} \) and the associated classical attribute grammar semantic rule is \( v_0=g(v_1) \) where function \( g \) is an identity function, \( v_0=n_2.i, v_1=n_1.s \). Intuitively, the attribute instances must be related by some kind of order over the rule instances such that any instantiated rule \( \hat{r}_i \) that defines \( \hat{v}_0 \) must precede \( \hat{r}_j \) which is any rule which uses \( \hat{v}_0 \). This intuitive order can be formalized as schedule.

Definition 3.2.5. A schedule in an instantiated classical attribute grammar is a strict total order of the instantiated rules.

Definition 3.2.6. A schedule in an instantiated classical attribute grammar is valid when
every rule that defines some attribute instance is before rules that use the same attribute instance. Formally, for all \( \hat{r}_1, \hat{r}_2 \in \hat{R} \) then \( \hat{r}_1 < \hat{r}_2 \) whenever \( \exists \hat{v}_i \in \hat{r}_1 \left( \exists \hat{v}_i \in \hat{r}_2 (\hat{v}_i \in DO(\hat{r}_1) \land \hat{v}_i \in UO(\hat{r}_2)) \right) \). This means that rule \( \hat{r}_1 \) which defines \( \hat{v}_i \) must strictly precede rule \( \hat{r}_2 \) which uses \( \hat{v}_i \).

**Equation (3.6)** describes a schedule for the instantiated classical attribute grammar defined in Figure 3.9 and its instantiated form in Figure 3.10. This schedule is valid because it first evaluates the instantiated rule \( \hat{r}_0 \) and evaluates \( n_1.i_1 \). It evaluates rule \( \hat{r}_3 \) which passes down the inherited attribute of \( n_1.i_1 \) as an inherited attribute of \( n_2.i_1 \). Then \( \hat{r}_9 \) passes the inherited attribute of \( n_2.i_1 \) up as synthesized attribute of \( n_2.s_1 \) which in turn becomes a inherited attribute of \( n_2.i_2 \) using rule \( \hat{r}_4 \) and so on. Finally, \( \hat{r}_8 \) passes the synthesized attribute of \( n_1.s_2 \) up which results in the evaluation of the synthesized attribute of \( n_0.out \) and evaluation is complete at this point, meaning that all attribute instances are evaluated.

\[
\begin{align*}
S \rightarrow A & \quad A \rightarrow A \ A & \quad A \rightarrow a \\
A.i_1 & = S.in & A1.i_1 & = A0.i_1 \\
A.i_2 & = A.s_1 & A1.i_2 & = A1.s_1 \\
S.out & = A.s_2 & A0.s_1 & = A1.s_2 \\
& & A2.i_1 & = A0.i_2 \\
& & A2.i_2 & = A2.s_1 \\
& & A0.s_2 & = A2.s_2 \\
\end{align*}
\]

Figure 3.9: Example of classical attribute grammar
This AG includes both inherited and synthesized attributes.

\[
S \rightarrow A \rightarrow AA \rightarrow a \ A \rightarrow aa
\]

\[
\left\{ \hat{r}_0 < \hat{r}_3 < \hat{r}_9 < \hat{r}_4 < \hat{r}_{10} < \hat{r}_5 < \hat{r}_1 < \hat{r}_6 < \hat{r}_{11} < \hat{r}_7 < \hat{r}_{12} < \hat{r}_8 < \hat{r}_2 \right\}
\]

One way to find the schedule which will be used together with the schedule evaluator is to perform a topological sort on the attribute dependencies using attribute instances. It
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n0:S -&gt; n1:A</td>
<td>n1:A -&gt; n2:A n3:A</td>
</tr>
<tr>
<td>r0 : n1.i1 = n0.in</td>
<td>r3 : n2.i1 = n1.i1</td>
</tr>
<tr>
<td>r1 : n1.i2 = n1.s1</td>
<td>r4 : n2.i2 = n2.s1</td>
</tr>
<tr>
<td>r2 : n0.out = n1.s2</td>
<td>r5 : n1.s1 = n2.s2</td>
</tr>
<tr>
<td>r6 : n3.i1 = n1.i2</td>
<td>r7 : n3.i2 = n3.s1</td>
</tr>
<tr>
<td>r8 : n1.s2 = n3.s2</td>
<td>r12: n3.s2 = n3.i2 + 2</td>
</tr>
</tbody>
</table>

Figure 3.10: Instantiated AG defined in Figure 3.9 using derivation defined in Equation (3.5)

then obtains the evaluation order and then evaluates attributes according to this order using the corresponding attribute evaluation rules associated with the respective productions.

Evaluation using a schedule is a process of evaluating each of the instantiated rules according to the order they appear in the schedule.

**Definition 3.2.7.** A schedule evaluator for classical attribute grammar is an interpreter-style program that follows a pre-computed order of instantiated rules to evaluate all attribute instances and runs in \( O(|\hat{R}|) \). Schedule evaluation is guaranteed to terminate.

**Theorem 3.** An evaluator will complete the AG evaluation without errors if the schedule is valid.

**Proof.** Trivial. In order to prove this, one needs to prove that no two instantiated rules can define the same attribute. However, we already put a requirement that the AG must be well-defined. This means by definition, the schedule evaluator will not find the definition of attribute instance missing or is being defined more than once. Therefore, if the schedule or total order of instantiated rules is valid, then according to the definition of a valid schedule, all rules that use an attribute instance should come after the rule that defines the same attribute instance. Then by evaluating the instantiated rules which are nothing but applying a map from attribute instance(s) to a semantic value, and most importantly in a sequence according to the schedule, the evaluation will be complete and valid.

After finding a valid schedule for a classical attribute grammar, the evaluation using
a schedule is straightforward. Code 1 describes a rule evaluator for a given instantiated semantic rule in a classical attribute grammar. Notice the check that ensures that attribute instance values are ready before the evaluation proceeds. Later on in this chapter, it will be extended to support extensions to attribute grammars. Code 2 describes the schedule evaluator which takes a totally ordered set of all instantiated semantic rules as an input and uses that to evaluate the rules.

Code 1 Rule evaluator for classical attribute grammar

```plaintext
procedure AG_EVAL(\( \hat{r}, Val \))
    if \( \hat{r} \equiv \hat{v}_0 = g(\hat{v}_1, \ldots, \hat{v}_n) \) then
        for \( \forall i \in [1, n] \) do
            if \( \hat{v}_i \notin Val \) then
                Error: Cannot read attribute instance value before it's ready.
            end if
        end for
        if \( \hat{v}_0 \in Val \) then
            Error: re-assignment of the attribute instance not allowed
        end if
        Val(\( \hat{v}_0 \)) ← g(Val(\( \hat{v}_1 \)), \ldots, Val(\( \hat{v}_n \)))
    end if
end procedure
```

Code 2 Schedule evaluator for classical AG using pre-defined order on rules

```plaintext
procedure AG_SCHEDULEREVAL( \( \hat{R}, < \), Val = dict() )
    for \( \hat{r} \leftarrow (\hat{R}, <) \) do
        AG_EVAL(\( \hat{r}, Val \))
    end for
end procedure
```

Definition 3.2.8. An attribute grammar is well-formed WF if it is well-defined, and there is a unique solution that can be computed for each attribute given any derivation out of all possible derivations.

\( WF \subset WD \)
In well-formed attribute grammar, no circular dependencies are allowed. There should be no circular chains of attribute dependencies for any derivation, meaning that attributes should not depend on each other in a way that creates an infinite evaluation loop. In fact, by definition, for every derivation of all well-formed non-circular attribute grammars, there exists a valid schedule.

Both well-defined and well-formed attribute grammars are essential for creating reliable and efficient attribute-based systems, such as compilers and language processors, and are crucial for maintaining correctness and predictability during attribute evaluation. As a matter of fact, Definition 3.2.1 describes a well-formed attribute grammar as it requires all attribute occurrences to be defined exactly once in order to be evaluated using a schedule. Well-formed will be important later on in this chapter where circular extension of attribute grammar will be discussed.

3.2.2 Demand Evaluation

Evaluation of attribute values is a process of resolving or evaluating instantiated semantic rules and their associated values given a derivation. To evaluate the instantiated attribute grammar in Figure 3.7, one needs to resolve all attribute values. This particular evaluation is straightforward because attribute grammar consists of only synthesized attributes and all attribute instances can be evaluated using a single bottom-up propagation. An instantiated attribute grammar can be evaluated using a demand evaluation which is of type of dynamic evaluation where the evaluator starts from root synthesized attributes and dynamically evaluates the required attributes needed to evaluate that root attributes. This process is done recursively until the evaluation is complete. A schedule evaluator, on the other hand, follows a pre-computed plan that specifies the order of evaluation of instantiated rules. These plans are created for each instantiated attribute grammar during the construction of the grammar evaluator. At run-time, the evaluator applies evaluations for the instantiated semantic rule using the plans associated with each production instance. The advantage of schedule
evaluators is that they are more efficient than demand evaluators in both time and space. However, finding the pre-computed plan to be used in schedule evaluation requires computations. Both demand and schedule evaluations are examples of “dynamic” evaluation since they work with the instantiated rules.

**Definition 3.2.9.** Demand evaluation is a kind of attribute grammar evaluation where each attribute instance access requires a call to evaluate the corresponding instantiated semantic rule. In practice, one can add a caching mechanism to store the value of already calculated attribute instances. We do that to avoid exponential time complexity which is caused by re-evaluating attribute instances.

**Observation 4.** *Demand evaluation gets stuck if it tries to get a value of an attribute instance that has not been defined yet causing the evaluator to stop evaluating and ultimately fail.*

Algorithm 1 describes a demand evaluator where it takes an attribute instance $\hat{v}_i$ and a set of all instantiated semantic rules $\hat{R}$ as an input, and then tries to follow a depth-first traversal to evaluate the associated instantiated rules. This type of evaluation usually starts from one of the root node’s synthesized attribute instances and goes backward trying to resolve the dependencies. Finally, it will evaluate the original instantiated rule that defined the synthesized attribute of the root when all used attribute instances are evaluated. In practice, the implementation of demand evaluation uses a cache of attribute values to avoid exponential evaluation time from duplicate evaluations. This improvement in time complexity negatively impacts the space complexity as the evaluator now has to keep track of which rules have been evaluated so far as well as cached attribute values. The evaluator does not know when the attribute instance value will be reused next and because of this, it needs to hold on to the cached value until the evaluation terminates. In fact, in Code 1 and Algorithm 1, Val is used as a caching mechanism.

Algorithm 1 takes as input an attribute instance $\hat{v}_i$ to evaluate and a set of all instantiated rules $\hat{R}$. It also uses two optional arguments visited and Val that are not needed when depth-
first traversal starts but will be used in the recursive calls. The visited which is initialized to \( \emptyset \) is a set of all attribute instances such that the recursive evaluator function has visited so far and is used to detect circularities between attribute instances. Lastly, Val which is initialized to an empty map is an injection (or a map) between an attribute instance and its value.

**Algorithm 1** Naive demand evaluator for classical AG

```plaintext
procedure DEMAND_EVAL(\( \hat{v}_i, \hat{R}, visited = \emptyset, Val = \text{dict()} \))
\( \hat{r} \leftarrow \exists \hat{r} \in \hat{R}, \text{LHS}(r') = \hat{v}_i \)
if \( \hat{r} \in visited \) then
   Error: Circularity detected.
end if
for \( \hat{v}_j \leftarrow \text{RHS}(\hat{r}) \) do
   DEMAND_EVAL(\( \hat{v}_j, \hat{R}, visited \cup \{\hat{r}\}, Val \))
end for
AG_EVAL(\( \hat{r}, Val \))
end procedure
```

Figure 3.9 describes a more involved AG with both inherited and synthesized attributes. In this example, the set of attribute instances are \( \hat{A} = \{n_2.i_1, n_1.i_2, n_0.out, \ldots \} \). The first step of the evaluation is to define the instantiated nodes and semantic rules which are both defined in Figure 3.10. The goal of the evaluation schedule is to find all attribute values of non-terminal nodes. Evaluation of this AG could be done using demand evaluation by calling DEMAND_EVAL with \( n_0.out \) to start the depth-first traversal from the root synthesized attribute. The evaluation would use the fact that for \( n_0.out \) to be evaluated \( n_1.s_2 \) needs to be evaluated and for \( n_1.s_2 \) to be evaluated \( n_3.s_2 \) needs to be evaluated and so on. Demand evaluation is done recursively by calling evaluation on each attribute instance without a pre-planned path to evaluate the instantiation rules. This means the path the evaluator follows is determined during the evaluation runtime and if there is any issue with the attribute grammar semantics, it will be revealed during the runtime as well.
3.2.3 Dependency Graph

Summary dependency graphs and augmented-dependency graphs are useful for visualizing dependencies for all the rules for a given production and in determining the order of semantic rules in production, especially during the generation of visit sequence evaluator (Section 3.2.5) for an attribute grammar. Informally, the dependency graph for a production \( p : X_0 \rightarrow X_1 \ldots X_n \), \( DG_p \), is a directed graph where the vertices are attributes occurrences in a particular production and edges indicate the flow of propagation of value. For example, in the semantic rule \( B.i = A.s \) in Figure 3.11, there is an edge \( A.s \rightarrow B.i \). The augmented dependency graph \( DG^*_p \) augments the dependency information as defined in the dependency graph by adding information concerning the parent node’s attributes and attributes of the sub-trees rooted in the RHS of the production. It is obtained from the \( DG_p \) graph by adding an edge \((v_i, v_j)\) whenever such an edge exists in \( SDG_X \) where \( X \in p \).

\[
\begin{align*}
S & \rightarrow A \ B \\
A & \rightarrow a \\
B & \rightarrow b \\
B.i & = A.s \\
A.s & = g() \\
B.t & = f(B.i) \\
A.j & = B.t \\
A.r & = h(A.j) \\
S.x & = A.r
\end{align*}
\]

Figure 3.11: Example of classical attribute grammar

If there is a path between two attribute occurrences \((X, i).a \) and \((X, j).b \) then there has to be a dependency \((X, i).a \rightarrow (X, j).b \), for all non-terminal derivation tree nodes \( X \) and knowing that \(|(X, i)| = |(X, j)|\), in order to capture all possible ways in which production rules can be “glued” together and taken into account. We have to be pessimistic and consider all such “induced” dependencies.

**Definition 3.2.10.** Given an attribute grammar \( A = (G, S, I, L, R) \), the dependency graph for a production \( p : X_0 \rightarrow X_1 \ldots X_k \in P \) is the directed graph \( DG_p = (V, E) \), where \( V \) is a set of all attribute occurrences of non-terminals used in the production or \( V = V(p) \) for short (note that \( V \) used in \( V(p) \) is referring to the set of attribute occurrences, this is defined in **Definition 3.2.1**), and \( E = \{(v_i, v_j) \mid \forall r \in R(p) \ (\exists v_j \in r \ (\exists v_i \in r \ (v_j \in DO(r) \land v_i \in \ldots))\}} \)
\{UO(r))\} meaning that \(v_i\) and \(v_j\) are attribute occurrences used in semantic rules for a production \(p\), and the value of \(v_i\) is used for computing the value of \(v_j\) in a semantic rule associated with production \(p\).

**Definition 3.2.11.** A summary dependency graph \(SDG_X\) is a directed graph for every attribute of non-terminal \(X \in N\).

\[
SDG_X = (V, E) \text{ where } V \subseteq Attr(X), \ E \subseteq (Attr(X) \times Attr(X))
\]

**Definition 3.2.12.** Augmented dependency graph using a summary dependency graph \(SDG_{X_i}\) for production \(p: X_0 \rightarrow X_1 \ldots X_k \in P\) is a directed graph of attribute occurrences \(DG^*_p = (V, E^*)\), where the vertices are borrowed from \(DG_p\) and edges are dependency graph edges and also extra edges from the summary dependency graph. More specifically,

\[
DG^*_{p: X_0 \rightarrow X_1 \ldots X_k \in P} = (V, E^*) \text{ where } DG_p[X_0, \ldots, X_k]=(V, E)
\]

and \(E^* = E \cup \bigcup_{i \in [0, \ldots, k]} \{(X:i).a, (X:i).a') \mid (a, a') \in SE \text{ where } SDG_{X_i}=(SV, SE)\}\)

**Definition 3.2.13.** Instantiated dependency graph \(IDG: (DG, \mathcal{D}) \rightarrow (V, E)\) is a function that takes take a family of the dependency graph \(DG\) and derivation \(\mathcal{D}\) and returns a directed graph where vertices and edges are brought over from the dependency graph and “instantiated”. More specifically, for a particular step of the derivation \((\alpha_i \rightarrow \alpha_{i+1}) \in \mathcal{D}\) which has a form of \(\hat{\alpha}(X_0, j)\hat{\beta} \rightarrow \hat{\alpha}\hat{\delta}\hat{\beta}\) we have an associated production \(p: X_0 \rightarrow \alpha\) that made that derivation step possible. For each \((X:j, i) \in \hat{\delta}\) find a function \(\sigma\) such that \(\sigma(X:j) = (X:j, i)\) and then use this function to go in a sequence of steps. First, we lift sigma that applies to non-terminal occurrences and local occurrences in the normal way to apply to all attribute occurrences: \(\sigma((X:j), a) = \sigma((X:j, i)).a\). The result is a map from attribute occurrences to attribute instances. Applying this function to the vertices and edges of the dependency graphs results in the instantiated form of the dependency graph.
\[ IDG(DG_p, (\alpha_i \rightarrow \alpha_{i+1})) = (\sigma(V), \sigma(E)) \]

where \( DG_p = (V, E) \) and \( \exists \sigma \) for any given \( \alpha_i \rightarrow \alpha_{i+1} \in \mathcal{D} \)

\[
\begin{align*}
\sigma(V) &= \{ \sigma(v) \mid v \in V \} \\
\sigma(E) &= \{ (\sigma(v), \sigma(v')) \mid (v, v') \in E \}
\end{align*}
\]

**Definition 3.2.14.** Instantiated augmented dependency graph is a function \( IADG : (AG, \mathcal{D}) \rightarrow (V, E) \) that takes attribute grammar \( (AG) \) and derivation \( (\mathcal{D}) \) and returns a directed graph where vertices and edges are brought over from augmented dependency graph \( DG_p^* \) for each production \( p \) used in derivation \( \mathcal{D} \) and then instantiated by applying \( IDG \) function. This set of instantiated augmented dependency graphs is naturally connected together because neighboring subgraphs share the same attribute instances. The result is \( IADG \), which is a single directed graph between attribute instances.

\[
IADG(AG, \mathcal{D}) = \bigcup_{(\alpha_i \rightarrow \alpha_{i+1}, p, \sigma) \in \mathcal{D}} DG_p^* \text{ where } DG_p^* = IDG(DG_p^*, \alpha_i \rightarrow \alpha_{i+1})
\]

**Figure 3.12: Example of classical non-circular AG**

Algorithm\( ^2 \) describes a simple greedy scheduling algorithm that can be used to find a schedule in a classical attribute grammar. The required space is \( n = |\hat{R}| \) as it has to keep track of visited nodes. Concerning time complexity, in the first iteration and in the worst
Figure 3.13: Augmented dependency graph $DG^*_{p,A_0\rightarrow A_1a}$ for non-circular AG defined in Figure 3.12

case scenario the inner “for” loop has to run $n$ times. In the second run of the inner loop, in the worst case, it will loop $n - 1$ times. Thus, one can conclude this algorithm has a time complexity of $O(n^2)$ which is not efficient. This further shows the need for static evaluation (or visit sequence evaluator) where it runs in $O(|V + E|)$ where $V = \hat{V}$ and $E = |\hat{V} \times \hat{R}|$.

### 3.2.4 1-SWEEP

An evaluation class of attribute grammar as defined by [5] is called 1-SWEEP. This class operates under the assumption that in the summary graph, all synthesized attributes rely only on all inherited attributes, and then foreach augmented dependency graph using this summary graph, it should not have any cycles. In 1-SWEEP it is feasible to evaluate all attributes in a single traversal of the tree, hence the name 1-SWEEP.

```
A -> A A
   A0.out = A1.out
   A2.in = A0.in
   A1.in = A2.out

A -> a
   A0.out = A0.in
```

Figure 3.14: Example of 1-SWEEP attribute grammar
Algorithm 2 Simple greedy total order of instantiated rules generator for classical AG

```plaintext
function GEN_SCHEDULE(\( \hat{R} \))
    schedule ← array(|\( \hat{R} |))
    \( \triangleright \) while there is an instantiated rule that has not been scheduled yet
    while (\( \bigcup \) schedule) \( \neq \) \( \hat{R} \) do
        changed ← false
        for \( \hat{r} \) ← \( \hat{R} \) do
            if \( \hat{r} \) \( \not\in \) schedule then
                if \( \forall \hat{v}_i \in UO(\hat{r}) (\exists \hat{r}' \in \text{schedule} \text{ such that } \hat{v}_i = \text{DO}(\hat{r}')) \) then
                    changed ← true
                    append(schedule, \( \hat{r} \))
                end if
            end if
        end for
        if changed = false then
            Error: Schedule does not exist.
        end if
    end while
    return schedule
end function
```

A -> A A
A0.out = A0.in
A2.in = A1.out
A1.in = A2.out

A -> a
A0.out = A0.in

Figure 3.15: Example of attribute grammar that is not 1-SWEEP

Figure 3.16: Summary graph used to test Figure 3.14 and Figure 3.15 for the 1-SWEEP property
Figure 3.17: Augmented dependency graph for Figure 3.14 constructed using Figure 3.16. Notice that the augmented graph does not have any cycles.

Figure 3.18: Augmented dependency graph for Figure 3.15 constructed using Figure 3.16. Notice that the augmented graph has a cycle.
Alternatively, the 1-SWEEP class can also be viewed as an encompassing class of attribute grammars where each synthesized attribute, for every non-terminal, can be evaluated as a direct function of all inherited attributes. Furthermore, within each production, it is possible to arrange the children in a way that facilitates the sequential evaluation of attributes for each child. The concept of 1-SWEEP (or 1-visit) attribute grammars was initially introduced by Engelfriet [12]. The determination of whether a classical attribute grammar belongs to the 1-SWEEP class is polynomial in relation to the size of the attribute grammar. This is because we need to examine each summary graph and, for each augmented dependency graph using a summary graph, ensure that the constraints are being upheld.

Figure 3.14 and Figure 3.15 are examples of attribute grammars where they both share the same summary graph Figure 3.16 and synthesized attribute depend only on the inherited attribute. However, after constructing the augmented dependency graph in Figure 3.17 and Figure 3.18 respectively we can see only one of them has a non-circular augmented dependency graph. This implies that this AG with a non-circular augmented dependency graph belongs to the 1-SWEEP evaluation class.

**Lemma 5.** Every 1-SWEEP AG is well-formed

*Proof.* Deferred. **Lemma 5** means one can find a valid schedule for all possible derivations \( \overset{\hat{D}}{D} \) if AG is 1-SWEEP. However, the proof is deferred as 1-SWEEP is a subset of \( l \)-ordered AG, and the well-formedness lemma for \( l \)-ordered AGs is discussed in **Lemma 10**. Therefore, that proof is applicable to **Lemma 5** as well.

Furthermore, a class of attribute grammars that are a subset of 1-SWEEP where finding order is naturally defined is called \( L \)-attributed attribute grammars. This subclass of attribute grammars can be evaluated in a single depth-first left-to-right pass over the input. In this class of AG, each synthesized attribute of the LHS symbol depends only on the attributes of its RHS symbols. Each inherited attribute of an RHS symbol by definition of \( L \)-attributed depends only on the inherited attributes of the LHS symbol or on synthesized
or inherited attributes of symbols to its left. Definition 3.2.15 formally defines this subclass of 1-SWEEP.

**Definition 3.2.15.** An attribute grammar is $L$-attributed if each inherited attribute of $X_j$, $1 < j < n$, on the right side of $X_0 \rightarrow X_1X_2\ldots X_n$, depends only on either

1. attributes of the symbols $X_1, X_2,\ldots, X_{j-1}$ to the left of $X_j$ in the production and

2. inherited attributes of $X_0$

### 3.2.5 Ordered Attribute Grammars

Engelfriet [12] defined the evaluation class of $l$-ordered attribute grammars and it is sometimes called simple-multi-visit grammars (or $m$-visit). The evaluation class of $l$-ordered attribute grammars is a larger class than 1-SWEEP or in fact any $n$-SWEEP. The definition of $l$-ordered is given a summary graph, this graph must reflect a total order of attributes for that non-terminal, and then foreach augmented dependency graph using this summary graph should not have cycles. For $l$-ordered grammars, there is a fixed evaluation order of all the attributes of all the non-termsinals. This evaluation order can be applied to all the derivation trees derived from the attribute grammar to create a schedule and evaluate each derivation in $\mathcal{D}$.

\[
\begin{align*}
A & \rightarrow A A \\
A.1.i1 &= A0.i1 \\
A.1.i2 &= A1.s1 \\
A0.s1 &= A1.s2 \\
A2.i1 &= A0.i2 \\
A2.i2 &= A2.s1 \\
A0.s2 &= A2.s2
\end{align*}
\]

\[
\begin{align*}
A & \rightarrow a \\
A0.s1 &= A0.i1 \\
A0.s2 &= A0.i2
\end{align*}
\]

**Figure 3.19:** Example of $l$-ordered attribute grammar [5]

Given a summary graph deciding whether it is $l$-ordered takes a polynomial time. Not every non-circular attribute grammar belongs to the $l$-ordered. Deciding whether a grammar
Figure 3.20: Correct summary graph used to test Figure 3.19 for the $l$-ordered property

Figure 3.21: Incorrect summary graph used to test Figure 3.19 for the $l$-ordered property

Figure 3.22: Augmented dependency graph for Figure 3.19 constructed using Figure 3.20

Notice that the augmented graph does not have a cycle.
is \( l \)-ordered is not a polynomial time problem because we have to guess the summary graph and its order. The question of whether an attribute grammar is \( l \)-ordered is certainly in NP since total orders \( T(X), X \in N \), can be guessed and then checked for compatibility in polynomial time. Furthermore, it has been proven that this problem is not only in NP, but it is in NP-complete [12]. A subclass of the \( l \)-ordered grammars called ordered attribute grammars (OAGs), has been identified that allows a polynomial-time greedy algorithm [17], and it’s commonly used for membership test of an attribute grammar. Visit sequence is an evaluation technique that is based on OAG and such efficient evaluators are called visit sequence evaluators. This evaluation order can be interpreted as a sequence of abstract computations to be performed on semantic rules of production. To summarize, by definition, there exists a visit sequence for \( l \)-ordered attribute grammars but the membership test is NP-complete so it’s common to use an OAG test which is a greedy algorithm that runs in polynomial time.
Remark 6. Two partial orders over the same set are compatible if their union is also a partial order.

For example, two ordered set \(X\) and \(Y\) using a binary relation \(<\) is defined as \(\forall x, y \in X, (x < y) \in (X, <)\), then exists \((x < y) \in (Y, <)\).

Lemma 7. Two total orders on the same set are compatible with each other if they are the same total order.

Proof. Proof by contradiction. If two total orders \(T(X) = (X, <)\) and \(T'(X) = (X, <)\) on the same set \(X\) are compatible with each other and not equal, that is \(T(X) \neq T'(X)\), then it would mean that there would be an element \(\alpha\) between some \(x, y \in X\) such that \(x < \alpha < y\) but as two sets have the same elements, it would be impossible to have such \(\alpha\). Hence two total orders must be equal, \(T(X) = T'(X)\).

Theorem 8 ([17]). An attribute grammar belongs to the class of Ordered Attribute Grammar (OAG) if, for every production \(p \in P\), the graph \(DG_p^*\) is cycle-free.

A visit sequence evaluator is a type of “static” evaluation method that traverses the derivation tree and evaluates the attribute instances. It starts at the root node and moves from the node to the adjacent node. A subtree is entered when it is reached with a downward move and it is exited with an upward move from the root of the subtree. During the traversal, attribute instances are computed. The decoration of a tree is completed when all its attribute instances are computed and the root node is exited. It was proven in [12] that in visit-sequence evaluator for a \(l\)-ordered attribute grammar, the number of visits to a non-terminal child node is fixed, hence the term “simple” in simple \(m\)-visit evaluator. Furthermore, each visit to a non-terminal has a fixed interface that consists of a set of inherited attributes of the root that may be used during the visit and later visits, and a set of synthesized attributes of the root that are guaranteed to be computed by the visit. The time complexity of running the visit sequence evaluator for classical AG is the same as the time complexity
of the schedule evaluator for classical AG, that is $O(|\hat{N}|)$ or guaranteed linear time with respect to the number of nodes in the derivation tree.

The foundation that makes the visit sequence evaluator and static evaluation possible is the concept of $l$-ordered attribute grammars and it is described in the following:

**Definition 3.2.16.** An attribute grammar is called $l$-ordered if there exists a summary dependency graph $SDG_X$ for every non-terminal $X \in N$ that is totally ordered and an augmented dependency graph $DG_p^*$ for each production $p \in P$ using those summary graphs that are cycle free.

**Lemma 9.** 1-SWEEP implies $l$-ordered (or 1-SWEEP $\subseteq l$-ordered)

*Proof sketch:* To establish that 1-SWEEP implies $l$-ordered (Lemma 9), we begin with a canonical attribute grammar. The crucial step involves demonstrating that the summary graph constitutes a total order. The challenge lies in constructing a summary graph, maintaining this total order, derived from the summary graph used in 1-SWEEP. The solution is to linearize the summary graph into a total order (e.g., $v_1 < v_2$ if there is an edge between $v_1$ and $v_2$ in the summary graph). The outcome is a total order without cycles or the introduction of new dependencies. Any such introduction of dependencies resulting in a cycle would imply a contradiction, suggesting a dependency between two inherited attributes or two synthesized attributes, which is not possible since we started from a canonical attribute grammar.

**Lemma 10.** $l$-ordered attribute grammars are well-defined.

*Proof.* Proof by contradiction.

1. Given a $l$-ordered classical attribute grammar with total order $T(X)$ of attributes for each non-terminal $X \in N$, and acyclic (cycle-free) augmented dependency graph $DG_p^*$ for all productions $p \in P$
2. For a particular derivation $\mathcal{D}$, we instantiate respective augmented dependency graphs for each production of the derivation to create instantiated augmented dependency graph or $IADG$ (see Definition 3.2.14).

3. Suppose there was a cycle in $IADG$ then the cycle that spans multiple instantiated augmented dependency graphs $\hat{DG}_p^*$ and not local to one augmented dependency graph (see Figure 3.24). This would mean that we must have an edge from one instantiated augmented dependency graph $\hat{DG}_p^*$ instance to another $\hat{DG}_{p'}^*$.

4. However, the cycle should include a path between attribute instances of the same tree node because attribute values come into the augmented dependency graph and go out through the attribute instances of the same tree node, and should come back again all through attribute instances of the same tree node. This means there is such path $(X:i,j).a \rightarrow \cdots \rightarrow (X:i,j).a' \rightarrow \cdots \rightarrow (X:i,j).a$ in the cycle.

5. Additionally, there exist the same total order of attributes as the other instantiated augmented dependency graph that it touches.

6. So then we can skip the part of the cycle that goes down to its children attribute instances and then up back to its root by “summarizing” with the summary edge $(X:i,j).a' \rightarrow (X:i,j).a$ edge. As a result, there is now a cycle in the $IADG$ that uses fewer instantiated augmented dependency graphs to describe the same dependency graph. This process of representing a cycle using fewer instantiated augmented dependency graphs can be done as many times as possible until we have a $IADG$ that a cycle using the smallest number of the instantiated individual augmented dependency graph.

7. If we lift the attribute occurrence from the attribute instance $(X:i,j).a \mapsto (X:i).a$, then we should get a cycle path that looks like this: $(X:i).a \rightarrow \cdots \rightarrow (X:i).a' \rightarrow \cdots \rightarrow (X:i).a$.

56
8. We know that \textit{IADG} has a cycle and every dependency is reflected in \textit{IADG}. This would mean that the original \textit{DG}^\ast_p has to be cyclic because again augmented dependency graph includes summary edges but that’s not possible because we assumed they were acyclic (cycle-free). Hence, proof by contradiction.

\textbf{Observation 11.} In \textbf{Lemma 10}, we relied on the fact that no two total orders on the same set are compatible unless they are the same (see \textbf{Lemma 7}), and we showed that if all non-circular-augmented-dependency-graphs of an attribute grammar can be ordered then the dependency graph \textit{IADG} for any derivation \(D\) can be ordered too.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure324.png}
\caption{Cycle involving multiple “instantiated” augmented dependency graphs The root of one “instantiated” augmented dependency graph is a child of another hence the overlap as they are the same instantiated tree node.}
\end{figure}

\textbf{Corollary 11.1.} All 1-SWEEP AGs are well-formed because 1-SWEEP AGs are l-ordered as well (see \textbf{Lemma 9}).

\textbf{Figure 3.22} and \textbf{Figure 3.23} are augmented dependency graphs constructed using summary graphs \textbf{Figure 3.20} and \textbf{Figure 3.21} of l-ordered attribute grammar defined in \textbf{Figure 3.19}. It is important to note that one of these augmented dependency graphs has a cycle. \textbf{Chapter 4} describes a technique for implementing l-ordered attribute grammars that avoids the need to build dependency graphs at run-time. This method is called static scheduling.
3.3 Remote Attribute Grammar

Remote attribute grammar (or RAGs) is an extension of classical attribute grammar by adding a set of objects and fields. It has been introduced separately in both [4, 8] and [14]. Informally, in addition to the capabilities of classical AG, it allows the passing of object references and object fields as attributes.

**Definition 3.3.1.** A remote attribute grammar (RAG) is a tuple \((G, S, I, L, R, B, F)\) where \(B\) is a set of objects declared at each production and \(F\) is a set of fields that each object has. Equation (3.7) describes the \(DO\) and \(OU\) for RAG.

\[
DO(p) = \left\{ \begin{array}{c}
(X:j).a \\
\{ (X:j).a \mid a \in S((X:0)) \quad j = 0 \\
\quad a \in I((X:k)) \quad j \neq 0, k \neq 0 \}
\end{array} \right\} \cup L(p)
\]

\[
UO(p) = \left\{ \begin{array}{c}
(X:j).a \\
\{ (X:j).a \mid a \in I((X:0)) \quad j = 0 \\
\quad a \in S((X:k)) \quad j \neq 0, k \neq 0 \}
\end{array} \right\} \cup L(p) \cup \{O_0 \mid O_0 \in B(p)\}
\]

In a remote attribute grammar, there are two forms of rules in addition to the classical form \(v_0=g(v_1, \ldots, v_n)\). A rule \(r \in R(p)\) may be of the form \(v=w.f\) for \(f \in F\), which is called a field read, or it may be of the form \(w.f \equiv v\), a partial field write. In \(v=w.f\) example, \(v\) is defined and \(w\) is used: \(DO(r) = \{v\}\), \(UO(r) = \{w\}\). In the \(w.f \equiv v\) example, the rule uses both \(v\) and \(w\); it does not define anything local: \(DO(r) = \emptyset, UO(r) = \{v, w\}\).

Notice that the partial field write operation uses \((\equiv)\) operator as opposed to the equals sign \(=\). For example, in \(w.f \equiv v\). In this case, the partial field write operation contributes the value of an attribute \(v\) to the field \(f\) of the object \(w\). The final value of the field is the combination of all the values that are written into it. Thus \(w.f \equiv v\) means that for every object reference \(o\) in \(w\) that is a reference to an instance of an object declaration in the RAG, all the object references in \(v\) are added to field \(f\) of \(o\).

We can express the semantics of remote attribute grammar in classical terms using a
process called “fiber construction” (see [8]). But the problem with that is it can result in attribute grammar with an infinite number of attribute instances because objects can be attributes and attributes can be on objects too and so on and so forth. Boyland [8] introduced a process called “fiber approximation” that results in the representation of the RAG in classical AG with a finite number of attribute instances. This process can be used to define \( l \)-ordered for remote attribute grammars by applying it to the definition of \( l \)-ordered for classical attribute grammars (see Definition 3.2.16).

3.3.1 Schedule

Generally speaking, a schedule for an AG is valid if it ensures that attributes and fields are assigned before they are used. However, for RAG this is complicated by the fact that an object may be created as a part of semantics rules for one production. It then may be passed around and its field gets written to or read in as part of semantic rules for different productions. This is different from classical attribute grammar where only values (no references) are passed around. So the schedule should account for the accessibility of objects and it has to be done before object fields are read or written to. There may be attribute instance \( v \) used as an argument to the classical rule \( \hat{v}_0=g(\hat{v}_1, \ldots, \hat{v}_n) \) that passes through an object reference. On the other hand, it may simply use the object references without passing them through. This is represented using \( L_{gi} \) where \( i \) indicates the instance of \( \hat{v} \) and it has two possible values: 0 indicates a copy of value and \( u \) indicates the use of value. Given a derivation tree \( t \), Equation (3.8) defines the sets of reaching object instance references \( B(\hat{v}) \subseteq B(t) \) for each attribute instance \( v \) of an instantiated remote attribute grammar instance. An object instance’s reference is in the set if there is some path through the semantic rules of the remote attribute grammar that can take a reference to that object instance and then to that attribute instance. \( B(\hat{v}) \) set is equal to the upper bound on the contents of attribute instance \( \hat{v} \) or formally defined as the least fixed-point solution of the following equations generated by the rule instances \( \hat{R} \). The third equation in Equation (3.8) denotes that object instances
are assumed instantiated before the schedule begins.

\[
\begin{aligned}
B(v_0) \supseteq B(v_i) & \quad (v_0=g(v_1, \ldots, v_n)) \in \hat{R} \text{ when } L_{gi} = 0 \\
B(v) \supseteq B(v') & \quad (v=w.f), (w'.f \supseteq v') \in \hat{R} \text{ when } B(w) \cap B(w') \neq \emptyset \quad (3.8) \\
B(o) = \{o\} & \quad o \in B(t)
\end{aligned}
\]

Note that remote attribute grammar introduced two new operations of partial write and read of object fields to interact with the object’s field(s). The schedule for remote attribute grammar is formally defined in [Definition 3.3.2]. There are two conditions that a valid schedule has to satisfy. The first condition is almost identical to the definition of schedule in classical attribute grammars which was formally described in [Definition 3.3.3].

**Definition 3.3.2.** A schedule for a remote attribute grammar is a strict total order (\(\prec\)) on the instantiated rules.

**Definition 3.3.3.** A schedule for a remote attribute grammar is valid when the two conditions below are met.

1. Every rule that uses some attribute instance must be scheduled after any rule that defines the same attribute instance. Formally, for parse tree \(t\) and for all \(\hat{r}_1, \hat{r}_2 \in \hat{R}\) then \(\hat{r}_1 \prec \hat{r}_2\) iff \(\exists \hat{v}_i \in \hat{r}_1 (\exists \hat{v}_i \in \hat{r}_2 (\hat{v}_i \in DO(\hat{r}_1) \land \hat{v}_i \in UO(\hat{r}_2) \land \hat{v}_i \notin B(t)))\). This means that rule \(\hat{r}_1\) which defines \(\hat{v}_i\) must strictly precede rule \(\hat{r}_2\) which uses \(\hat{v}_i\).

2. For two rules \(\hat{r}_1 = (v=w.f), \hat{r}_2 = (w'.f \supseteq v')\) if these rules may be referring to the same object, that is \(B(w) \cap B(w') \neq \emptyset\), then the partial field write must strictly precede the field read or \(\hat{r}_2 \prec \hat{r}_1\).

In [Definition 3.3.3] the first condition indicates that the definition of attribute instance should precede its use unless it is an object whose order in the schedule does not matter as objects are assumed instantiated before the schedule begins. The second condition similarly extends the first condition as RAG introduces the two new rules. In order for the evaluation
of the RAG class of attribute grammars to be valid the partial write operation must precede
the read operation if the fixed-point set of references of partial write and read of attribute
objects are not completely disjointed, meaning that they may be referring to the same object.

Figure 3.25 describes a remote attribute grammar over the CFG which itself is defined
in Figure 3.1 The (only) instantiated form is defined in Figure 3.26 Equation (3.9) cal-
culates the B sets for each attribute instance and finally, a valid schedule is defined in
Equation (3.10)

\[
\begin{align*}
S \rightarrow & \ A \ B \\
A \rightarrow & \ a \\
B \rightarrow & \ b
\end{align*}
\]

<table>
<thead>
<tr>
<th>S</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>S.x = A.r</td>
<td>A.s = o</td>
<td>A.r = o.f</td>
</tr>
<tr>
<td>B.i = A.s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.25: A simple remote attribute grammar definition

\[
\begin{align*}
n0: S \rightarrow & \ n1: A \ n2: B \\
r0: & \ n2.i = n1.s \\
r1: & \ n0.x = n1.r \\
n1: A \rightarrow & \ a \\
r2: & \ n1.s = o0 \\
r3: & \ n1.r = o0.f \\
n2: B \rightarrow & \ b \\
r4: & \ 10 = n2.i \\
r5: & \ 10.f \leftarrow 'b'
\end{align*}
\]

Figure 3.26: Instantiated RAG defined in Figure 3.26 using derivation defined in Figure 3.1

\[
\begin{align*}
\hat{r}_0 \rightarrow \hat{r}_1 \\
\hat{r}_2 \rightarrow \hat{r}_3 \\
\hat{r}_4 \rightarrow \hat{r}_5
\end{align*}
\]

Figure 3.27: Dependency graph for the trivial derivation defined in Figure 3.1 of remote AG
defined in Figure 3.25
\[ B(l_0) \supseteq B(n_2.i) \supseteq B(n_1.s) \supseteq B(o_0) = \{ o_0 \} \]  
\[ B(n_0.x) \supseteq B(n_1.r) = \emptyset \] (3.9)

The root non-terminal attributes are \( n_0.x \) and the following is a valid schedule because the partial write of field \( f \) is done prior to the read and the last rule assigns the synthesized attribute of the root node.

\[ \{ \hat{r}_2 < \hat{r}_0 < \hat{r}_4 < \hat{r}_5 < \hat{r}_3 < \hat{r}_1 \} \] (3.10)

The schedule evaluation of remote attribute grammar uses the same approach as the schedule evaluation for classical attribute grammar. However, as remote attribute grammars introduce two new rules in addition to classical rule, thus, one needs to modify \textsc{Schedule Eval} defined in \textbf{Code 2} and replace \textsc{AG Eval} call with \textsc{RAG Eval} which would then be able to handle the evaluation of RAG semantic rules. Notice that as the object’s attributes are of type of a \textit{set}, hence the union operation (\( \cup \)) is used for the evaluation of partial field write operation.

\textbf{Code 3} Rule evaluator for remote attribute grammar

```
procedure RAG_EVAL(\( \hat{r} \), Val)
  if \( \hat{r} \equiv \hat{v}_0 = g(\hat{v}_1, \ldots, \hat{v}_m) \) then \( \triangleright \) classical rule
    AG_EVAL(\( \hat{r} \), Val)
  else if \( \hat{r} \equiv \hat{v}_i = o_j.f \) then \( \triangleright \) field read
    Val(\( \hat{v}_i \)) \leftarrow Val(o_j.f)
  else if \( \hat{r} \equiv o_i.f \supseteq \hat{v}_j \) then \( \triangleright \) partial field write
    Val(o_i.f) \leftarrow Val(o_i.f) \cup Val(\hat{v}_j)
  end if
end procedure
```

### 3.4 Circular Attribute Grammars

This section describes circularity in attribute grammars which traditionally has been considered an error even in the original Knuth paper [18]. Knuth devised an algorithm that
decided circularity in AG in an intrinsically exponential number of steps. Jazayeri in [15] was
the first to prove that the problem was intractable. Circularity in AG is formally defined in
Definition 3.4.1 but informally it means that given a derivation tree, in order to evaluate the
attribute values one attribute instance may end up depending on itself. Conversely, in order
to show that attribute grammar is circular, one needs to find a derivation tree such that its
dependency graph contains a cycle. In other words, assume that the attribute grammar is
non-circular and then find a counter-example. However, it can get complicated and costly
as there can be a derivation of attribute grammar that results in a cyclic dependency graph
and other derivations that do not result in a cyclic dependency graph. To make matters
worse, a practical AG that can be used for real-world applications usually has an infinite
number of derivations.

**Definition 3.4.1.** An attribute grammar is *circular* if there exists a derivation tree of the
context-free grammar whose attribute dependency graph has a cycle.

Circularity in classical attribute grammar directly negates well-formedness. However,
Farrow [13] showed that a particular class of grammars with circular dependencies under cer-
tain constraints can be considered well-formed by requiring that the domain of all attributes
involved in a cycle can be arranged in a lattice of finite height where all uses of attribute
instances in a cycle are monotone. This result is built upon the well-known Knaster-Tarski
fixed-point theorem. It is important to make a clear distinction between circularities in
classical attribute grammar where such an AG would not be well-formed as attribute types
are not sets and evaluating such attribute grammar is not possible, as opposed to circular
attribute grammars where attribute types are sets and attribute grammar can be evaluated
upon finite repeated evaluation and a fixed-point of attribute instance value can be reached.

The evaluation of circularly defined attributes begins with giving the attribute instance
the bottom value and the iterative process (or a loop) will converge to a least fixed point for
all involved semantic rules. This uses an ascending chain condition and a fixed-point loop
will terminate because only monotone uses are involved in cycles. Farrow also introduced a
partially dynamic evaluation technique to compute the fixed point for attributes that may have circular dependencies. The evaluation technique used by Farrow utilizes a group of mutually recursive functions along the derivation tree which are called *synth-function*. Each function body determines to which production the subtree belongs, evaluates a subset of the attributes for the production, and returns the desired synthesized attribute. Code 4 is an example synth-functions for non-circular AG defined in Figure 3.9. In practice, a cache of attribute values is used to avoid exponential evaluation time.

**Code 4** Synth functions evaluators for AG defined in Figure 3.9

```plaintext
function EVAL_A_s1(i₁)
    if PRODUCTION = A → ϵ then
        return i₁ + 1
    else if PRODUCTION = A → AA then
        return EVAL_A_s2(EVAL_A_s1(i₁))
    end if
end function

function EVAL_A_s2(i₂)
    if PRODUCTION = A → ϵ then
        return i₂ + 2
    else if PRODUCTION = A → AA then
        return EVAL_A_s2(EVAL_A_s1(i₂))
    end if
end function

function EVAL_S_R(in)
    return EVAL_A_s2(EVAL_A_s1(in))
end function
```

Figure 3.28 is an example of a circular attribute grammar. Figure 3.29 and Figure 3.30 visualize dependency graph of trivial derivations, demonstrating the fact that not all possible derivations of a circular attribute grammar have cycles. More specifically, the partial derivation $S → AA$ does not always cause a cycle and neither does $A → a$, just when the two are put together, $D: S → AA → aA → aa$, it results in a cycle.

In circular attribute grammars, to distinguish a monotone use of an attribute instance from a simple (or non-monotone) use one needs to define some semantics. If an attribute
S → A A
A0.i1 = A1.s2
A0.i2 = A1.s1
A1.i1 = A0.s1
A1.i2 = A0.s2
S.s = A0.s1

A → a
A.s1 = A.i1
A.s2 = A.i2

A → b
A.s1 = A.i1
A.s2 = 'b'

Figure 3.28: Example of circular attribute grammar

Figure 3.29: Dependency graph $DG$ for circular derivation $S \rightarrow AA \rightarrow aA \rightarrow aa$ of circular AG described in Figure 3.28 that results in a cycle between attribute instances.

Figure 3.30: Dependency graph $DG_p$ for non-circular derivation $S \rightarrow AA \rightarrow aA \rightarrow ab$ of circular AG described in Figure 3.28 that does not result in a cycle between attribute instances.
instance used at index $j$ of the semantic function $g$ (classical AG rule) belongs to a cycle in
a dependency graph, then $M_{gj}$ should be true meaning that the use of an attribute instance
must be monotone.

**Definition 3.4.2.** If $f:X \rightarrow Y$ is a set function from a collection of sets $X$ to an ordered
set $Y$, then $f$ is said to be monotone if whenever $A \subseteq B$ as elements of $X$, $f(A) \leq f(B)$.

**Definition 3.4.3.** Given an attribute grammar rule $v_0=g(v_1,\ldots,v_j,\ldots,v_n)$, $M_{gj} =$ true or
false indicates whether use of attribute instance at argument $j$ of function $g$ is monotone
if it yields true. If it yields false then it is not non-monotone.

The formal definition for $v_0=g(v_1,v_2)$ monotone in argument $v_1$ or simply $M_{g1} =$ true:

$$M_{g1} = \forall v_2 (\forall v_1 \subseteq v_1' \rightarrow g(v_1,v_2) \subseteq g(v_1',v_2))$$

The previous equation can be generalized for $k$ arguments, and argument $v_1$ to $v_j$. Let
us first keep $v_j$ as $v_1$:

$$M_{g1} = \forall v_2,v_3,\ldots,v_k (\forall v_1 \subseteq v_1' \rightarrow g(v_1,v_2,\ldots,v_k) \subseteq g(v_1',v_2,\ldots,v_k))$$

Now, to generalize $v_1$ to $v_j$ is trivial, but somewhat cumbersome, as one needs to list all
$v_i$ except $v_j$.

$$M_{gj} = \forall v_1,\ldots,v_{j-1},v_{j+1},\ldots,v_k (\forall v_j \subseteq v_j' \rightarrow g(v_1,\ldots,v_k) \subseteq g(v_1,\ldots,v_{j-1},v_j',v_{j+1},\ldots,v_k))$$

(3.11)

**Definition 3.4.4.** A circular attribute grammar (CAG) is a tuple $(G, S, I, L, R, M)$ where
$M$ is a function $M:(g,j) \rightarrow \text{bool}$ that indicates whether the use in the semantic function
$g \in R(p)$ for some $p \in P$ is monotone or not at $j$th argument.

**Remark 12.** The dependency edges in the classical attribute grammars are all “simple”
edges, but in circular attribute grammars, there may be either simple or “monotone” edges. Combining a simple and a monotone edge results in a simple edge (conservatively).

Remark 13. The identity function \(I\) defined in Definition 3.2.1 is monotone in its first argument, that is \(M_{11} = \text{true}\) (for example, the use of \(B.y\) in semantic rule \(A.x = B.y\) is monotone).

Lastly, \(UO(p)\) or used occurrence was defined in Equation (3.1) but for CAG it needs to be defined more specifically because only monotone-use occurrences should be used in a cycle. \(SUO(p)\) is referring to simple (non-monotone) use occurrences and \(MUO(p)\) refers to monotone use occurrences.

\[
SUO(r) = \{v_j \mid M_{gj} = \text{false} \text{ for } 1 \leq j \leq k \text{ in classical rule } v_0 = g(v_1, \ldots, v_k)\}
\]

\[
MUO(r) = UO(r) - SUO(r)
\]

\[
UO(r) = SUO(r) \cup MUO(r)
\]

3.4.1 Schedule

The schedule for circular attribute grammars uses a relation \((\preceq)\) on the set of instantiated rules, the relation is transitive and total. Let’s call such a relation as quasi total order. This is very similar to a well-known “total pre-order” relation except it does not include reflexivity. In a quasi total order, two values are always comparable (either \(a \preceq b\) or \(b \preceq a\) or both). Unlike total pre-order, quasi total order need not to be reflexive or irreflexive, this is what distinguishes quasi total order from total pre-order. Formally, a quasi total order relation \(\preceq\) satisfies the following properties:

1. For all \(x, y\) and \(z\), if \(x \preceq y\) and \(y \preceq z\) then \(x \preceq z\) (transitivity).

2. For all \(x, y\), then \(x \preceq y\) or \(y \preceq x\) must be true or both (strong connectedness or total).
**Lemma 14.** Quasi total order on a set $S$ can be expressed as a total order on a partition of $S$ where singleton sets can be either loops or not.

**Example 3.4.1.** The following set of binary quasi total order relationship: $\{a \preceq b, b \preceq a, b \preceq c, c \preceq c\}$ can be expressed as total order on the partition of $S$: $\{a, b\} < \{c\}$.

In order to prove Lemma 14, first let’s prove a similar variant for well-known total pre-order, that is: a total pre-order on a set $S$ can be expressed as a total order on a partition of $S$. Note that for convenience we use the ($\preceq$) operator for the total pre-order as well.

**Proof.** First, we observe that a total pre-order is not a total order only because different elements can be “equal” under it, i.e. $a \preceq b$ and $b \preceq a$ do not imply $a = b$. To demonstrate this, consider the relation:

$$r \sim s \iff r \preceq s \land s \preceq r$$

This is an equivalence relation. It’s clearly symmetric, and it inherits reflexivity and transitivity from the total pre-order. It partitions $S$ into equivalence classes of elements that are “equal” under the total pre-order. The total pre-order induces a total order on the equivalence classes of $S$ (i.e. partition) under this equivalence relation:

$$\{r\} < \{s\} \iff r \preceq s$$

This is well-defined since it does not depend on the representatives: If $r \preceq s$, $r \sim t$ and $s \sim u$, then $t \preceq r \preceq s \preceq u$ and thus by transitivity $t \preceq u$. It’s a total order because it inherits reflexivity, transitivity, and totality from the pre-order, and it’s anti-symmetric: If $\{r\} < \{s\}$ and $\{s\} < \{r\}$, then by the definition of ($<$) we have $r \preceq s$ and $s \preceq r$, and then by the definition of ($\sim$) we have $r \sim s$ and thus $\{r\} = \{s\}$.

Conversely, every total order ($<$) on a partition of $S$ induces a total pre-order on $S$, again via $r \preceq s \iff \{r\} < \{s\}$. Reflexivity, transitivity, and totality are inherited from ($\preceq$), but anti-symmetry is not (since $\{r\} = \{s\}$ does not imply $r = s$).
These two transformations are inverses of each other, so the total pre-orders on \( S \) and the total orders on partitions of \( S \) are in one-to-one correspondence.

A similar proof can be done for quasi total order with a small modification where we need to add a \textit{diagonal} turn to first turn it into a pre-order and then take the corresponding total order on a partition. Then we notice that singleton sets can be either loops or not.

\textbf{Definition 3.4.5.} \textit{Loops} of quasi total order are partitions that are strongly connected and these partitions can have one or more elements in them.

\textbf{Observation 15.} Quasi total order \( T = (S, \preceq) \) can be expressed as a directed graph \( G=(V, E) \) where \( V=S \) and \( E=\{(v, v') \mid (v \preceq v') \in T\} \) (see Figure 3.31). This means we can borrow operations such as (graph) isomorphism \((\simeq)\) and use it to compare two quasi total orders.

\textbf{Remark 16.} In the following, \((<)\) operator is a short-hand for a comparison when \( A \preceq B \land B \not\preceq A \).

\textbf{Definition 3.4.6.} A \textit{schedule} for a circular attribute grammar is a quasi total order \((\preceq)\) on the set of instantiated rules.

\textbf{Definition 3.4.7.} A schedule \((\preceq)\) for a circular attribute grammar is \textit{valid} when every rule that “simple” uses some attribute instance must be scheduled after any rule that defines the same attribute instance. Formally, for all \( \hat{r}_i, \hat{r}_j \in \hat{R} \) then \( \hat{r}_i \prec \hat{r}_j \text{ or } \hat{r}_i \preceq \hat{r}_j \iff \exists \hat{v}_k \in \hat{r}_i (\exists \hat{v}_k \in \hat{r}_j (\hat{v}_k \in DO(\hat{r}_i) \land \hat{v}_j \in UO(\hat{r}_j))) \). This means that rule \( \hat{r}_i \) which defines \( \hat{v}_k \) has a binary relation with \( \hat{r}_j \) which uses \( \hat{v}_k \). The following defines the condition of when \( \prec \) or \( \preceq \) is used:
\[
\begin{cases}
\hat{r}_i < \hat{r}_j, & \text{if } \hat{v}_k \in SUO(\hat{r}_j) \\
\hat{r}_i \preceq \hat{r}_j, & \text{otherwise } \hat{v}_k \in UO(\hat{r}_j) \land \hat{v}_k \notin SUO(\hat{r}_j) \text{ (or } \hat{v}_k \in MUO(\hat{r}_j) \text{ for short)}
\end{cases}
\]

In summary, the schedule for circular attribute grammar is valid when it ensures that “simple” dependencies are respected. That is, \( \forall \hat{v}_k \in \hat{r}_i (\exists \hat{v}_k \in \hat{r}_j (\hat{v}_k \in DO(\hat{r}_i) \land \hat{v}_j \in SUO(\hat{r}_j))) \) then \( r_i < r_j \). Similarly, monotone uses also have to be respected in the schedule, that is \( \forall \hat{v}_k \in \hat{r}_i (\exists \hat{v}_k \in \hat{r}_j (\hat{v}_k \in DO(\hat{r}_i) \land \hat{v}_j \in MUO(\hat{r}_j))) \) then \( r_i \preceq r_j \).

The schedule for CAG has the following form: \( T = (\hat{R}, \preceq) \). We define \( \hat{r}_1 \sim \hat{r}_2 \) whenever \( r_1 \preceq r_2 \) and \( r_2 \preceq r_1 \). Because quasi total order is transitive and total, \( (\sim) \) is an equivalence relation. Then \( T/\sim \) is the partition of \( T \) where equivalent instantiated rules are placed in the same set, and the quasi total order can be seen as a total order over the equivalent set.

**Definition 3.4.8.** A circular attribute grammar is well-formed if it has a valid schedule for all derivation trees.

**Definition 3.4.9.** The trivial schedule of CAG is a schedule where all instantiated rules belong to the same set or \( \{\{\hat{r}_0, \ldots, \hat{r}_n\}\} \).

**Observation 17.** The trivial schedule will be a valid schedule when all uses are monotone. If a CAG has all monotone functions, then all of its derivation trees have valid schedules by putting all instantiated rules belonging to a single set.

**Proof.** The proof of **Observation 17** uses the fact that in CAG all attributes are declared circular meaning that the scheduler can assume there is a cycle involving all attribute instances and as all uses are monotone then it is enough to know that trivial schedule is valid. Thus proving the existence of a (trivial) schedule.

**Remark 18.** A schedule will rarely have a singleton loop in it, but it could occur, for example, if the CAG has the following rule in it: \( x = x + y \).
3.4.2 Ordered Circular Attribute Grammars

**Definition 3.4.10.** We define the restriction function (|) of a graph as follows:

\[ G \mid_{(X;i)} = G \mid_{\{(X;i), a \mid x \in \text{Attr}(X)\}} \]

**Definition 3.4.11.** A circular attribute grammar is called l-ordered if there exists a summary dependency graph \( SDG_X \) that is quasi total ordered for every non-terminal \( X \in N \), let’s call this quasi total order \( Q(X) \). And for each augmented dependency graph \( DG^*_p : X_0 \rightarrow X_1 \ldots X_k \) of production \( p \in P \) we first form a transitive closure (which is a quasi total order and let’s call it \( Q(p) \)). Then the following must be true:

1. If any dependency between two things that are in an equivalence group of the quasi total order \( Q(p) \), that dependency has to be monotone.
2. The restriction of the quasi-total-transitive closure of the augmented dependency graph for each \( X_i \in p \), that is \( Q(p) \mid_{(X;i)} \), must be isomorphic to \( Q(X_i) \).

**Remark 19.** The linearization (total) operation of a graph will never introduce dependencies that result in a new cycle.

**Lemma 20.** l-ordered circular attribute grammars are well-formed.

*Proof.** Constructive proof and contradiction. We instantiate the augmented dependency graph and ask for the smallest dependency graph showing the cycle involving the parent and child (see Figure 3.32). There are two possibilities:

- If there was a summary edge from the parent inherited attribute to the parent synthesized attribute then it would have meant the child edge going from parent inherited to child attributes and back up to parent synthesized attribute was unnecessary but we asked for the smallest dependency graph showing the cycle and it was not the smallest dependency graph as it could have been simplified further, hence proof by contradiction.

71
• If that summary edge was not there then it would mean the summary edge was missing as the edge from parent inherited to child attributes and back up to parent synthesized attribute was not reflected in the summary graph so it was an invalid summary graph.

So in either case this is not possible.

The following describes how to update the $l$-ordered well-formedness proof for classical AG to work with CAG $l$-ordered. In short, the goal of Lemma 10 and its proof was to show that when we instantiate the tree, we have a total order of instantiation. However, we do have a quasi total order for every single production, we want to use that to make quasi total order of instances (augmented dependency graph) and show that it does not break the monotonicity requirements. Basically, we start with copies of each the quasi total order of the augmented dependency graph and it interacts with other instantiated augmented dependency graphs. And the question is if we put all these instantiated augmented dependency graphs together, the result will be transitive and total.

Next, we want to show that the transitive total closure will not break the monotonicity by bringing simple edges to be involved in a cycle. Quasi-total closure of the relation, will not add any new cycles, it will make two things that are unrelated by making them related, but not forming a cycle between them. When we form quasi-total closure, the transitive closure operation may cause a cycle and we proved in Lemma 10 it does not, but the total part (linearizing) will not cause cycles. Once we get a quasi-order that respects dependencies then we automatically have a quasi-total order that respects dependencies by putting in arbitrary linearization. Going from quasi to quasi-total order will not introduce any cycles and we will never have problems with monotonicity. This means they are in the same equivalence class. When we put things together, at every place we connect these things, as we are using an isomorphic quasi-total order that is respected on both sides, and direct dependencies are already respected. The only potential problem is if transitivity can cause two connected regions that were not connected before to be connected after. But it is impossible, since if
they were not connected before, then it means they could not possibly have a dependency; hence, there is nothing to be concerned about.

Figure 3.32: Cycle involving multiple “instantiated” augmented dependency graphs
The smallest example of showing a cycle involving parent and child.

3.5 Circular Remote Attribute Grammars

Circular remote attribute grammars (or CRAG) are a generalized form of remote attribute grammars where the use of fiber attributes is assumed monotone, and circular dependencies in a cycle must be monotone. Recall that remote attribute grammars introduce two additional rules: object field read and partial write in addition to a classical rule. These two rules do not have to be changed going from RAG to CRAG because they operate on sets and are monotone operations. Additionally, validating the monotonicity of a (semantic) function is an undecidable problem.

**Definition 3.5.1.** Circular remote attribute grammars extend remote attribute grammars and have the same form as remote attribute grammars, except all attributes are declared circular and some functions are declared monotone in some arguments. This monotonicity is declared as $M_{gi} = \text{true}$, meaning that function $g$ is monotone in argument $i$.

**Remark 21.** A common lattice type attribute used in CRAG is a set or union lattice.
Remark 22. The partial field write operation ($\sqsubseteq$) as defined in the definition of remote attribute grammar is a monotone operation.

3.5.1 Schedule

The following uses the notations as described in [8]. Informally, schedule for a circular remote attribute grammar is a modified version of the definition of schedule for a remote attribute grammar.

Remark 23. In the following, ($<$) operator is a short-hand for a comparison operator when $A \subseteq B \land B \not\subseteq A$.

Definition 3.5.2. A schedule for a circular remote attribute grammar is a quasi total order ($\subseteq$) on the set of instantiated rules.

Definition 3.5.3. A schedule ($\subseteq$) for a circular remote attribute grammar is valid when:

1. Every rule that “simple” uses some attribute instance must be scheduled after any rule that defines the same attribute instance. Formally, for parse tree $t$ and for all $\hat{r}_i, \hat{r}_j \in \hat{R}$ then $\hat{r}_i < \hat{r}_j$ or $\hat{r}_i \subseteq \hat{r}_j$ iff $\exists \hat{v}_k \in \hat{r}_i (\exists \hat{v}_k \in \hat{r}_j (\hat{v}_k \in DO(\hat{r}_i) \land \hat{v}_j \in UO(\hat{r}_j) \land \hat{v}_k \not\in B(t)))$. This means that rule $\hat{r}_i$ which defines $\hat{v}_k$ has a binary relation with $\hat{r}_j$ which uses $\hat{v}_k$. The following defines the condition of when $<$ or $\subseteq$ is used:

$$
\begin{align*}
\hat{r}_i &< \hat{r}_j, \quad \text{if } \hat{v}_k \in SUO(\hat{r}_j) \\
\hat{r}_i &\subseteq \hat{r}_j, \quad \text{otherwise } \hat{v}_k \in UO(\hat{r}_j) \land \hat{v}_k \not\in SUO(\hat{r}_j) \quad \text{(or } \hat{v}_k \in MUO(\hat{r}_j) \text{ for short)}
\end{align*}
$$

2. For two rules $\hat{r}_i = (v=w.f)$, $\hat{r}_j = (w'.f \sqsubseteq v') \in R(t)$ if these rules are potentially referring to the same object or $B(w) \cap B(w') \neq \emptyset$, then the partial field write must precede the field read, meaning $\hat{r}_j \subseteq \hat{r}_i$ since the partial field write and field read are monotone with respect to each other.
In summary, the schedule for circular remote attribute grammar is valid when it ensures that “simple” dependencies are respected. Given parse tree $t$, $\forall \hat{v}_k \in \hat{r}_i (\exists \hat{v}_k \in \hat{r}_j (\hat{v}_k \in DO(\hat{r}_i) \land \hat{v}_j \in SUO(\hat{r}_j) \land \hat{v}_k \notin B(t)))$ then $\hat{r}_i < \hat{r}_j$. Similarly, monotone uses also have to be respected in the schedule, that is $\forall \hat{v}_k \in \hat{r}_i (\exists \hat{v}_k \in \hat{r}_j (\hat{v}_k \in DO(\hat{r}_i) \land \hat{v}_j \in MUO(\hat{r}_j) \land \hat{v}_k \notin B(t)))$ then $\hat{r}_i \preceq \hat{r}_j$.

In Definition 3.5.2, $v \in B(t)$ is referring to the object instance itself and the fact that object instances are assumed instantiated before the schedule gets generated so their position in the schedule does not matter. Also, in the first condition, if the use occurrence is simple (or not monotone), then a strict less-than operator ($<$) is used to indicate that declaration has to strictly precede use and can never be in the same cycle. If however, the use is not simple then they can potentially be in the same cycle. In the second condition, as discussed previously, the two operations of field read and partial field write that were introduced in remote attribute grammar extensions are indeed monotone. Therefore, this condition points out that partial field write has to precede field read but they can potentially be in the same cycle.

Figure 3.33 and Figure 3.36 define two CRAG examples and the following explores schedule in circular remote attribute grammars. For example, Figure 3.33 consists of all monotone operations and its schedule would not be interesting because, by definition of schedule in CRAG, one can follow a trivial evaluation schedule and evaluate all the attribute instances without any order over and over again until a fixed-point is reached. The Figure 3.36 is an example of introducing non-monotone function $h$ (e.g. set subtraction, but it is monotone in the first operand, but not in the second) which in turn makes finding the schedule more challenging as a trivial schedule would not be valid anymore.

Given a trivial derivation defined in Figure 3.1, the Equation (3.13) shows the calculation of $B$ sets for each attribute instance.
S -> A B
A -> a
B -> b

local l
l = A.r
B.i = l
A.i = B.s
S.x = l.f

Figure 3.33: Example of a circular remote attribute grammar with all monotone operations.

n0:S -> n1:A n2:B
n1:A -> a
n2:B -> b

local l0
l0 = n1.r
l1 = B.i

r0: l0 = n1.r
r1: n2.i = l0
r2: n1.i = n2.s
r3: n0.x = l0.f
r4: o0.f <- n1.i
r5: n1.r = o0
r6: l1 = n2.i
r7: n2.s = l1.f

Figure 3.34: Instantiated CRAG defined in Figure 3.33 using the derivation defined in Figure 3.1

Figure 3.35: Dependency graph for derivation $S \rightarrow AB \rightarrow aB \rightarrow ab$ of circular remote AG described in Figure 3.33 that does not result in a cycle in-between attribute instances.
\[ B(n_1,i) \supseteq B(n_2,s) \supseteq B(l_1,f) = \emptyset \]

\[ B(l_0) \supseteq B(n_1,r) = \{ o_0 \} \] (3.13)

\[ B(l_1) \supseteq B(n_2,i) = \{ o_0 \} \]

\[ \{ \hat{r}_5 \preceq \hat{r}_0 \preceq \hat{r}_1 \preceq \hat{r}_6 \preceq \{ \hat{r}_4, \hat{r}_7, \hat{r}_2 \} \preceq \hat{r}_3 \} \] (3.14)

As previously mentioned, the trivial but yet valid schedule is \( \{ \hat{r}_0, \ldots, \hat{r}_7 \} \) as all functions are monotone, however, a more efficient schedule is defined in the Equation (3.14). By efficient, we refer to the fact that a subset of instantiated rules has to be evaluated over and over again until their attribute instances reach a fixed-point value instead of all instantiated rules as in the trivial schedule. Notice the field \( f \) of \( o_0 \) is being read in \( \hat{r}_7 \) and written to in \( \hat{r}_4 \). The read requires the write to be done first and then write requires the read to be done first, and this is indeed a cycle. The use of curly braces in the schedule means that instantiated rules could be evaluated in any order but have to be evaluated over and over again until a fixed point is reached as they are in a cycle or the same equivalence group. This schedule is valid because:

1. \( \hat{r}_5 \) passes reference to object \( o_0 \) up to \( n_0 \). Note that objects are assumed instantiated before the schedule begins.

2. \( \hat{r}_0 \) assigns reference to object \( o_0 \) to a local variable \( l_0 \)

3. \( \hat{r}_1 \) passes down the reference to object \( o_0 \) to node \( n_2 \)

4. \( \hat{r}_6 \) assigns reference to object \( o_0 \) to a local variable \( l_1 \)

5. \( \hat{r}_4 \) partial write to field \( f \) of \( o_0 \) using inherited attribute of \( n_1 \) which is yet to be assigned

6. \( \hat{r}_7 \) reads the field \( f \) of \( o_0 \) and passes it up to node \( n_0 \)

7. \( \hat{r}_2 \) finally assigns inherited attribute of \( n_1 \)
8. $\hat{r}_3$ assigns the synthesized attribute of the root node. The evaluation is complete.

The previous example (Figure 3.33) is modified in Figure 3.36 by introducing non-monotone function $h$ outside of the cycle in order to disallow trivial schedule. In this example, the only valid schedule is Equation (3.14).

$$S \rightarrow A \ B
\quad A \rightarrow a
\quad B \rightarrow b
\begin{align*}
&\text{local } l \\
&l = h(A.r) \\
&B.i = h(l) \\
&A.i = B.s \\
&S.x = h(l.f)
\end{align*}

Figure 3.36: A circular remote attribute grammar with some non-monotone operations $h$ is a non-monotone function.

$$n0:S \rightarrow n1:A \ n2:B
\quad n1:A \rightarrow a
\quad n2:B \rightarrow b
\begin{align*}
&\text{local } l \\
&r0: l = h(n1.r) \\
&r1: n2.i = h(l) \\
&r2: n1.i = n2.s \\
&r3: n0.x = h(l.f)
\end{align*}

Figure 3.37: Instantiated CRAG defined in Figure 3.36 using derivation defined in Figure 3.1

The schedule for a circular remote attribute is a quasi total order on the set of instantiated rules. Recall that quasi total order can be represented as a total order on the partition where some of the singleton sets can be looped or not depending on whether they are strongly connected. Note that each partition represents a cycle in the dependency graph (see Definition 3.4.5).

Before CRAG evaluation begins, circular attribute instances belonging to partitions of the instantiated rules in the schedule are initialized to the bottom value ($\bot$), for example, if the attribute type is a set then it is initialized to an empty set ($\emptyset$). Again, not all attributes are initialized to the bottom value because for instance re-evaluating the singleton instantiated rule that is declared non-circular is just unnecessary, inefficient, and wrong as it indicates a
bigger problem is going on, problems such as simple (non-monotone) use being involved in a cycle.

Furthermore, during the evaluation of CRAG, all rules in each partition of the schedule have to be evaluated over and over again until a fixed point is reached. However, the order of evaluation of individual attribute instances in each partition has no impact on the evaluation as all uses and functions involved in the cycle have to be monotone. The difference between the evaluation of RAG and CRAG is the use of fixed-point loops. More specifically, just following the schedule or the order of evaluation of instantiated rules is not enough for CRAG to be evaluated. So rules involving circular attribute instances have to be repeatedly evaluated until their values reach a fixed point or there is no change. This is required because as it is noted in the definition of circular attribute grammar, it is valid in CRAG for an attribute instance to be read before any write has taken place, and use occurrences of attribute instances in a cycle have to be monotone. Code 5 describes the implementation of this schedule evaluator for CRAG.

Some anomalies can happen during evaluation and they can be detected during the evaluation and the following describes the possibilities. First, the use of attributes that are involved in a cycle has to be monotone. Considering that in this research circular attribute types are assumed as sets, that means during the evaluation, the current semantic value $v$ is a superset of the previous $v'$ or $v' \subseteq v$. But if results get smaller after evaluation or something completely different then it would no longer mean that the use occurrence of the attribute instance was monotone. However, the scheduler in an attribute grammar system trusts the programmer that functions are indeed monotone and it will not test the monotonicity.

### 3.5.2 Ordered Circular Remote Attribute Grammars

As previously described in the remote attribute grammar section, the semantics of remote attribute grammars can be described in terms of classical attribute grammars using a method called “fiber approximation” (see [8]). By definition the use of all fiber attributes is monotone,
Code 5 Schedule evaluation of circular remote attribute grammar

1:  procedure CRAG_EVAL(\(\hat{r}\), Val)
2:     if \(\hat{r} \equiv \hat{v}_0 = g(\hat{v}_1, \ldots, \hat{v}_n)\) then \(\triangleright\) classical rule
3:         for \(\forall i \in [1, n]\) do
4:             \(a \leftarrow |\hat{v}_i|\)
5:             attributeCircular \(\leftarrow\) DECLARED_CIRCULAR(\(a\))
6:             if \(\hat{v}_i \notin\) Val then
7:                 if \(\text{attributeCircular} = \text{true}\) then
8:                     \(\triangleright\) default bottom value will be used for this circular attribute
9:                     continue
10:                else
11:                    Error: Cannot read attribute instance value before it’s ready.
12:                end if
13:         end if
14:     end for
15:     a \(\leftarrow\) \(|\hat{v}_0|\)
16:     attributeCircular \(\leftarrow\) DECLARED_CIRCULAR(\(a\))
17:     if \(\text{attributeCircular} = \text{false} \land \hat{v}_0 \in\) Val then
18:         Error: re-assignment of the non-circular attribute instance not allowed
19:     end if
20:     \(\triangleright\) already asserted non-circular attributes must have a value defined
21:     prev \(\leftarrow\) null
22:     if \(\text{attributeCircular} = \text{true}\) then
23:         \(\triangleright\) use bottom value if the value is not defined yet if the attribute is circular
24:         prev \(\leftarrow\) Val(\(\hat{v}_0,\text{default}=\bot\))
25:     end if
26:     Val(\(\hat{v}_0\)) \(\leftarrow\) \(g(\text{Val}(\hat{v}_1,\text{default}=\bot), \ldots, \text{Val}(\hat{v}_n,\text{default}=\bot))\)
27:     if \(\text{attributeCircular} = \text{true} \land \text{Val}(\hat{v}_0) \not\supseteq\) prev then
28:         Error: non-monotonicity detected
29:     end if
30:     \(\triangleright\) field read
31:     else if \(\hat{r} \equiv \hat{v}_i = o_j.f\) then
32:         Val(\(\hat{v}_i\)) \(\leftarrow\) Val(\(o_j.f\))
33:     end if
34:     else if \(\hat{r} \equiv o_i.f \subseteq \hat{v}_j\) then \(\triangleright\) partial field write
35:         Val(\(o_i.f\)) \(\leftarrow\) Val(\(o_i.f\)) \(\cup\) Val(\(\hat{v}_j\))
36:     end if
37: end procedure
procedure CRAG_SCHEDULE_EVAL( (\( \hat{R}, \preceq \)), Val = dict() )

▷ loop through each partition of instantiated rules in the schedule

for \( R' \leftarrow (\hat{R}, \preceq) \) do

    do

    changed \( \leftarrow \) false

    for \( \hat{r} \leftarrow R' \) do

        prev \( \leftarrow \) Val

        CRAG_EVAL(\( \hat{r}, Val \))

        if prev \( \neq \) Val then

            changed \( \leftarrow \) true

        end if

    end for

end do

while changed \( \land R' \) is a circular partition

end for

end procedure

and this means that the definition of ordered circular attribute grammars (Definition 3.4.11) and well-formedness lemma (Lemma 20) that was previously described can be extended to circular remote attribute grammars without any additional restriction.
Chapter 4

Methods

This chapter describes the methods and algorithms used in this research to implement static scheduling of circular remote attribute grammars. More specifically, it describes the changes to cycle breaking algorithm and visits sequence evaluator generation. The methods described in this chapter improve Boyland’s work by not requiring extra fiber cycle breaking and simplifying it because the generated fiber approximation dependencies are monotone and fiber cycles can be handled by the new CRAG static scheduler. It also improves Hedin’s work by making CRAG evaluation efficient by using a static schedule evaluation instead of a demand evaluation.

Previously, to eliminate fiber cycles in attribute grammar, Boyland introduced a novel approach and implemented it in APS. In summary, it first uses union-find to find all strongly connected regions of fibered attributes. For every non-terminal whose fibered attributes participate in the cycle, it creates two new attributes, a UP attribute, and a DOWN attribute. These attributes are used to break the cycle: all elements of the strongly connected region are removed; all dependencies into the strongly connected region are moved to the up attribute of the left-hand-side (LHS) non-terminal, and all dependencies out of the strongly connected region are moved to come from the down attribute of the left-hand-side (LHS) non-terminal. The resulting dependency graphs are resubmitted to the DNC test,
which should now be completed successfully. Next, the dependency graphs are submitted to the OAG test to find a scheduling order. If none is found, the remote attribute grammar is rejected. Once the order is found, a conditional total order of the attribute instances in each production is found and this is used as the basis of code generation. However, reusing the same OAG scheduler for circular remote AGs turned out to be problematic. APS detects a child phase change (and generates a child visit call) when it encounters a synthesized attribute of the child in the CTO. But if a child phase does not have a synthesized attribute, the visit sequence code generation never generates the visit. It gets worse because cyclic dependencies were broken in functions and locals because their dependencies were missing as they were removed by the UP-DOWN cycle breaking. Also, local attributes may end up being scheduled before their dependencies.

The solution we ended up with was rewriting the scheduler to embrace cycles instead of breaking them and scheduling attribute instances as a group or a block. A block is ready to be scheduled if all attribute instances are ready. Local attributes, on the other hand, should be scheduled according to the total order. This generalized version of the scheduler fixed the issues with the old scheduler and became much simpler implementation-wise. This static scheduling method will be discussed in depth in this chapter.

4.1 Visit Sequence

Visit sequence is a static evaluation method consisting of a series of recursive functions. The $l$-ordered attribute grammars showed that there is no need to construct a runtime dependency graph for certain attribute grammars. However, the $l$-ordered attribute grammars do not discuss the implementation of the evaluator and how the evaluation works, and it’s only concerned with capturing the dependencies statically (not during the runtime). This allows us to run evaluations without the need to create a large and costly runtime dependency graph. Additionally, visit sequence enables incremental evaluation of attribute grammars
The $l$-ordered attribute grammars were defined in the previous chapter and this section discusses the construction of visit sequence for both classical attribute grammars and circular remote attribute grammars. A visit sequence $\mathcal{V}(p)$ for production $p$ uses an interface or protocol $\Pi(X)$ for each of its non-terminal $X$ to recursively call their visit subroutine for a particular phase. Basically, before the child visits call, the parent sets the child’s inherited attributes for a particular phase, and after the visit, the parent can read the child’s synthesized attributes for the same phase. This heuristic is defined by the protocol for that child’s non-terminal and directs the visit sequence by marking which child attribute belongs to which child phase.

4.1.1 Visit Sequence for Classical AGs

**Definition 4.1.1.** A Protocol $\Pi(X)$ for non-terminal $X \in N$ of classical attribute grammar is a sequence of subsets of all attributes of that non-terminal $\text{Attr}(X)$ where it can include an empty set or multiple synthesized or inherited attributes and every attribute is in exactly one of those sets.

In canonical attribute grammar, there is a natural order between inherited and synthesized attributes of any non-terminal because, in a well-defined attribute grammar, a synthesized attribute cannot define other synthesized attributes. Similarly, the inherited attribute of non-terminal cannot be used to define other inherited attributes. In fact, within each inner set of protocols for a non-terminal, all synthesized attributes must follow all inherited attributes. For a non-terminal $X$, $\Pi(X)$ can be converted into a $T(X)$ and compatibility can be verified in the same way compatibility is defined in [Definition 3.2.16](#) for $l$-ordered AG.

For example, for non-terminal $A$ for AG defined in [Figure 3.9](#) $\Pi(A)$ is the following:

$$\Pi(A) = \left\{ \{A.i_1, A.s_1\} < \{A.i_2, A.s_2\} \right\}$$
Observation 24. For a non-terminal $X \in N$, a half-set of synthesized or inherited attributes are $\Pi_I(X)$ and $\Pi_S(X)$ respectively such that they contain only inherited or synthesized attribute occurrences. This is defined as follows:

$$
\Pi_I(X) = \bigcup_{\pi \in \Pi(X)} \{ X.a \mid X.a \in \pi \wedge a \in I(X) \}
$$

$$
\Pi_S(X) = \bigcup_{\pi \in \Pi(X)} \{ X.a \mid X.a \in \pi \wedge a \in S(X) \}
$$

$$
\Pi(X) = \left\{ \Pi_I(X)[k] \cup \Pi_S(X)[k] \mid k \in \max(|\Pi_I(X)|, |\Pi_S(X)|) \right\}
$$

For example, for non-terminal $A$ of AG defined in Figure 3.9, a half-set of inherited and synthesized attribute occurrences are the following:

$$
\Pi_I(A) = \left\{ \{ A.i_1 \} < \{ A.i_2 \} \right\}
$$

$$
\Pi_S(A) = \left\{ \{ A.s_1 \} < \{ A.s_2 \} \right\}
$$

$$
\Pi(A) = \left\{ \{ A.i_1, A.s_1 \} < \{ A.i_2, A.s_2 \} \right\}
$$

Definition 4.1.2. A visit-sequence for a production $p.X \rightarrow \alpha \in P$ of a $l$-ordered attribute grammar is $\mathcal{V}(p)$, a sequence of visits where each visit consists of rules that define local attributes, rules that define child inherited attributes or child visit call with occurrence number. Also, $\mathcal{V}(p)$ has to be compatible with $R(p)$ and $\Pi(X_i)$ for all $X_i \in (p \cap T)$. To show compatibility we would need to “flatten” the visit sequence by replacing the visit calls with the rules that defined attribute occurrences. For example, in visit($((X:i, j), k)$, $k$ is a visit number corresponding to $\Pi(X_i)[k]$.

- $\mathcal{V}(p)$ is compatible with $\Pi(X_i)$ if for each visit (outer set) in $\mathcal{V}(p)$ we replace all visit calls visit($((X:i, j), k)$ with just rules that define occurrence $X_i.a$ for some attribute $a$. And then lift the attribute part from the occurrence (i.e. $X_i.a \mapsto a$) to get an ordered set of attributes. Then would see that the ordered set of attributes is identical to the
associated inner set in $\Pi(X_i)$ (i.e. $\Pi(X_i)[k]$).

- $\mathcal{V}(p)$ is compatible with $R(p)$ if we replace all visit calls $visit((X:i,j), k)$ with just rules that define occurrence $X_i.a$ for some attribute $a$. And then flattening of this ordered set of sets so we just end up with rules. We can see that for any two $r, r' \in R(p)$ which themselves are a part of $\mathcal{V}(p)$, then $(r < r') \in \mathcal{V}(p)$ whenever $v_i \in DO(r)$ and $v_i \in UO(r')$.

For production $p:A \rightarrow AA$ of AG defined in Figure 3.9 a visit sequence is defined in Equation (4.1) and it is made of two outer sets indicating that there are two visits. The first visit consists of all rules that are precursors to a rule that assigns LHS synthesized attribute of $s_1$. Similarly, the second visit does the same for $s_2$.

\[
\mathcal{V}(p:A_0 \rightarrow A_1A_2) = \begin{cases} 
\text{first visit corresponding to first set in } \Pi(A) \\
\{visit(A_1, 1) < visit(A_1, 2)\} < \\
\text{second visit corresponding to second set in } \Pi(A) \\
\{visit(A_2, 1) < visit(A_2, 2)\}
\end{cases}
\]  

(4.1)

Remark 25. Any total order of attributes of non-terminal $T(X)$ in a non-circular attribute grammar can be converted to a protocol $\Pi(X)$.

Proof. There exists a function $f$ that can map between the visit sequence for classical AGs and the definition of $l$-ordered for non-circular AGs (see Definition 3.2.16). The function $f$ is a non-injective surjective function (surjection, not a bijection).

- Forward direction: the protocol $\Pi(X)$ can be mapped into a summary graph total order $(SDG_X, <)$, and the visit sequence $\mathcal{V}(p)$ can be mapped into augmented dependency graph total order $(DG^*_p, <)$ by removing the visit partitions and mapping each semantic rule in the visit sequence to an attribute occurrence being defined on LHS of the semantic rule. There can be empty visits or different combinations of a grouping of
the attributes in a single visit of the protocol that maps to the same summary graph total order. Hence a surjection and not a bijection.

- Backward direction: the summary graph total order \((DG_p^*, <)\) can be mapped to one or more different protocols \(\Pi(X)\) by putting the attributes into partitions that are called “visits” which act as an interface for visit sequence. The total order of the augmented dependency graph \((DG_p^*, <)\) can be mapped to a visit sequence \(\mathcal{V}(p)\) by mapping an attribute occurrence to a semantic rule that defines the attribute occurrence and inserting a child visit call of the appropriate phase between child-inherited and child-synthesized attributes such that the resulting visit sequence is compatible with the protocol interface and this visit call serves as the recursive evaluation call that evaluates the child’s attributes. The algorithm for this is defined in Algorithm 3 and it’s well-formed since all dependencies used for static scheduling are lifted from the augmented dependency graph total-order, hence dependencies are compatible with each other.

Remark 26 tries to make connection between well-formedness and validity of protocol and visit sequence.

**Remark 26.** A protocol that is unworkable will prevent any visit sequence from being valid. Similarly, a visit sequence can be well-formed if it has the right form, but invalid if protocol dependencies are not respected.

**Definition 4.1.3.** Visit sequence evaluator is a static attribute grammar evaluator that is a recursively defined function and takes as argument: an instance of a non-terminal node of LHS of production \((X:i, j)\), a visit sequence \(\mathcal{V}_{p,X_i \rightarrow \alpha}\) and derivation \(\mathcal{D}\) and uses them to evaluate the attribute instances.

Code 6 is an implementation of the visit sequence evaluator for classical attribute grammar. Notice that Code 6 also defines a helper function \(EVAL_TREE_ATTRS\) and it is used
Algorithm 3 Chunk visit sequence generator for classical attribute grammar

procedure AG_CREATE_CHUNKS( \( AG, p : X_0 \rightarrow X_1 \ldots X_k, \Pi \) )

locals \( \leftarrow \{ \{ l \} \mid l \in L(p) \} \) \( \triangleright \) collect each local as its own chunk

halfLefts \( \leftarrow \text{array}(|\Pi(X_0)|) \)

halfRights \( \leftarrow \text{array}(|\Pi(X_0)|) \) \( \triangleright \) collect parent’s half left and half right chunks

for phase \( \leftarrow 1 \ldots |\Pi(X_0)| \) do

\[ \text{halfLefts}[\text{phase}] \leftarrow \{ X.a \mid a \in \text{Inh}(X_0) \land a \in \Pi(X_0)[\text{phase}] \} \]

\[ \text{halfRights}[\text{phase}] \leftarrow \{ X.a \mid a \in \text{Syn}(X_0) \land a \in \Pi(X_0)[\text{phase}] \} \]

end for

childVisits \( \leftarrow \text{array}(k) \) \( \triangleright \) collect each child’s visits for each of their phases

for childIndex \( \leftarrow 1 \ldots k \) do

\[ \text{childVisits[childIndex]} \leftarrow \text{array}(|\Pi(X_{\text{childIndex}})|) \}

for phase \( \leftarrow 1 \ldots |\Pi(X_{\text{childIndex}})| \) do

\[ \text{childVisits[childIndex][phase]} \leftarrow \{ X_{\text{childIndex}.a} \mid a \in \Pi(X_{\text{childIndex}})[\text{phase}] \} \]

end for

end for

chunks \( \leftarrow \) locals \( \cup \left( \bigcup \text{halfLefts} \right) \cup \left( \bigcup \text{halfRights} \right) \cup \left( \bigcup \text{childVisits} \right) \)

return (locals, halfLefts, halfRights, childVisits, chunks)

end procedure

procedure AG_LIFT_CHUNK_DEPENDENCY_GRAPH( \( AG, p : X_0 \rightarrow X_1 \ldots X_k, T: (DG^*_p, <) \), state )

(locals, halfLefts, halfRights, childVisits, chunks) \( \leftarrow \) state

\( \triangleright \) adjacency graph with \(|\text{chunks}|\) number of vertices

chunkGraph \( \leftarrow \text{graph}(|\text{chunks}|) \)

\( \triangleright \) lift attribute occurrence dependencies to chunk dependency graph

for chunk\(_1 \leftarrow \) chunks do

for chunk\(_2 \leftarrow \) chunks do

for \( v_1 \in \) chunk\(_1 \) do

for \( v_2 \in \) chunk\(_2 \) do

if \((v_1, v_2) \in (DG^*_p, \langle)\) then

edgeKind \( \leftarrow \text{EDGE_KIND}(v_1, v_2) \)

chunkGraph \( \leftarrow (\text{chunk}_1, \text{chunk}_2, \text{edgeKind}) \)

end if

end for

end for

end for

end for

return chunkGraph

end procedure
procedure AG_ADD_GUIDING_DEPENDENCIES(AG, p: X₀ → X₁...Xₖ, Π, state, chunkGraph)
  (locals, halfLefts, halfRights, childVisits, chunks) ← state
  ▷ add dependency between parent inherited and synthesized of the same phase
  for phase ← 1...|Π(X₀)| do
    chunkGraph ← (halfLefts[phase], halfRights[phase], indirectTransitiveDependency)
  end for
  ▷ add dependency between parent synthesized and inherited of the next phase
  for phase ← 1...(|Π(X₀)| − 1) do
    chunkGraph ← (halfRights[phase], halfLefts[phase + 1],
                  indirectTransitiveDependency)
  end for
  ▷ add a dependency to ensure child visits are sequential for the same child
  for childIndex ← 1...k do
    for phase ← 1...(|Π(X[childIndex]| − 1) do
      chunkGraph ←
                  (childVisits[childIndex][phase], childVisits[childIndex][phase + 1],
                    indirectTransitiveDependency)
    end for
  end for
  return chunkGraph
end procedure

procedure AG_GENERATE_CHUNK_VISITS(AG, p: X₀ → X₁...Xₖ, chunkPartition)
  visitPartition ← array(|chunkPartition|)
  for v ∈ chunkPartition do
    r ← ∃r ∈ R(p) such that LHS(r) = v
    append(visitPartition, r)
  end for
  return visitPartition
end procedure
procedure AG_CHUNK_READY_TO_SCHEDULE( chunkGraph, scheduled, chunk )
(chunks, dependencies) ← chunkGraph
for dependentChunk ← dependencies(chunk) do
  if dependentChunk \notin scheduled then
    return false
  end if
end for
return true
end procedure

procedure AG_FIND_NEXT_CHUNK_TO_SCHEDULE( chunkGraph, scheduled, except )
(chunks, dependencies) ← chunkGraph
for chunk ← (chunks \setminus (scheduled \cup except)) do
  if AG_CHUNK_READY_TO_SCHEDULE(chunkGraph, scheduled, chunk) then
    return chunk
  end if
end for
return null
end procedure

to evaluate rules that define child (i.e. non-terminal instances on the RHS of production) inherited attributes just before recursive visit call or parent (i.e. non-terminal instance on the LHS of production) synthesized attributes after a visit is ended. The corresponding visit sequence evaluator for production \( p : A \rightarrow AA \) is in the Figure 4.1.

A simple \( m \)-visit (multi-visit) evaluator where \( m > 0 \) as defined in [12], for an attribute grammar \( G \) is an attribute evaluator where the evaluation strategy is completely determined by a totally ordered partition \( \text{Attr}_1(X), \ldots, \text{Attr}_{\phi(X)}(X) \) of \( \phi(X) \in [0, m] \) over the attributes \( \text{Attr}_i(X) \) of each \( X \in N \). An integer \( \phi(X) \) is the total number of visits to any instance of non-terminal \( X \). A simple \( m \)-visit evaluator may visit any number of children of an instantiated non-terminal in any order.

**Theorem 27** ([12 Theorem 3.3]). An AG is \( l \)-ordered if it is a simple multi-visit.

**Remark 28.** If a generated visit sequence evaluator is used to evaluate a particular derivation of an attribute grammar that is circular, then the visit sequence will evaluate the attributes out of order, that is reading an attribute value before writing takes place.
procedure AG_VISIT_SEQUENCE_GEN_HELPER( AG, p: X₀ → X₁ ... Xₖ, state, chunkGraph, scheduled, visits, scheduledChildPhases, phase )

(locals, halfLefts, halfRights, childVisits, chunks) ← state

visit ← array_list()

parentInheritedReady ← AG_CHUNK_READY_TO_SCHEDULE(chunkGraph, scheduled, halfLefts[phase])

if parentInheritedReady = false then
    Error: expected parent inherited chunk for parent phase to be ready
end if

▷ there is no rule in R(p) that defines parent inherited attributes so its skipped

append(scheduled, halfLefts[phase])

while true do
    ▷ check dependencies of half right are scheduled

    chunk ← AG_FIND_NEXT CHUNK_TO_SCHEDULE(
        chunkGraph, scheduled, {halfRights[phase]})

    if chunk = null then
        break ▷ there is no more chunk to schedule in this visit
    end if

    append(scheduled, chunk)

    if chunk ∈ locals then
        appendAll(visit, AG_GENERATE CHUNK VISITS(AG, p, chunk))
    else if chunk ∈ (∪ childVisits) then
        (childPhase, childIndex) ← chunk
        inherited ← {v | v ∈ chunk ∧ v ≡ X.a ∧ a ∈ Inh(X(childIndex))}
        scheduledChildPhases[childIndex] ← childPhase
        ▷ there is no rule in R(p) that defines child synthesized attributes so it is safely skipped

        appendAll(visit, AG_GENERATE CHUNK VISITS(AG, p, inherited))
        append(visit, visitMarker(childPhase, childIndex)) ▷ child visit marker
    end if
end while

parentSynthesizedReady ← AG_CHUNK_READY_TO_SCHEDULE(chunkGraph, scheduled, halfRights[phase])

if parentSynthesizedReady = false then
    Error: expected parent synthesized chunk for parent phase to be ready
end if

append(scheduled, halfRights[phase])

appendAll(visit, AG_GENERATE CHUNK VISITS(AG, p, halfRights[phase]))

append(visit, visitMarker(phase, -1)) ▷ end of parent phase visit marker

visits[phase] ← visit

end procedure
procedure AG_VISIT_SEQUENCE_GEN(AG, p:X_0 \rightarrow X_1 \ldots X_k, \Pi, T:(DG_p^*, <) )

state \leftarrow \text{AG_CREATE_CHUNKS}(AG, p, \Pi)
(locals, halfLefts, halfRights, childVisits, chunks) \leftarrow state
chunkGraph \leftarrow \text{AG_LIFT_CHUNK_DEPENDENCY_GRAPH}(AG, p, T:(DG_p^*, <), state)
chunkGraph \leftarrow \text{AG_ADD_GUIDING_DEPENDENCIES}(AG, p, \Pi, state, chunkGraph)
chunkGraph \leftarrow \text{transitive_closure}(chunkGraph)
visits \leftarrow \text{array}(|\Pi(X_0)|)
scheduled \leftarrow \emptyset
childPhases \leftarrow [0] \times \text{array}(k) \quad \triangleright \text{array of size } k \text{ initialized to 0}

for \ phase \leftarrow 1 \ldots |\Pi(X_0)| \ do
\text{AG_VISIT_SEQUENCE_GEN_HELPER}(AG, p, state, chunkGraph, scheduled, visits, childPhases, phase)
end for

if scheduled \neq chunks then
\textbf{Error: generated visit sequence is missing some chunks}
else if childPhases \neq (|\Pi(X_1)|, \ldots, |\Pi(X_k)|) then
\textbf{Error: generated visit sequence does not corresponds with its children protocols}
end if
return visits
end procedure

It’s important to understand that not all attribute grammars belonging to the class of non-circular attributes are l-ordered, meaning that one can construct a valid and consistent visit sequence evaluator. Figure 4.2 is an example of such attribute grammar and its instantiated form using trivial derivation defined in Figure 4.3. Equation (4.2) is a valid schedule for this attribute grammar and yet the visit sequence evaluator described in Figure 4.4 is not valid because of inconsistencies. Notice in the body of the visit sequence evaluator for non-terminal S, it calls the visit subroutine for its children in an out-of-order fashion. The first visit_A_part1 is called and then visit_A_part2 is called subsequently afterward for the first child, but in reverse order for the second child. These inconsistencies arise from the fact that \( \gamma_{p,A \rightarrow AA} \) is not compatible with \( \Pi(A) \). As there is no other way to arrange visit calls to be compatible with \( \Pi(A) \) and for the schedule to be valid both at the same time we can conclude that this AG is not l-ordered.
procedure AG_VISIT_SEQUENCE_EVAL_HELPER(AG, \mathcal{V}, \hat{\mathcal{D}}, (X:i, j), phase, Val)

\[ (p: X \rightarrow \alpha) \leftarrow \hat{\mathcal{D}}[j] \]  
\[ V \leftarrow \mathcal{V}(p)[phase] \]  
\[ \text{for } I \leftarrow V \text{ do} \]
\[ \text{if } I \equiv \text{visit}(X':n, \text{childPhase}) \text{ then} \]
\[ q \leftarrow \exists q \in [j + 1, \ldots, |\hat{\mathcal{D}}|] \text{ such that } (\alpha_q \rightarrow \alpha_{q+1} \equiv \hat{\mathcal{D}}[q] \wedge \alpha_q \equiv \hat{\alpha}(X':n,q)\hat{\beta} \wedge \alpha_{q+1} \equiv \hat{\alpha}\hat{\beta}) \]
\[ \text{AG_VISIT_SEQUENCE_EVAL_HELPER}(AG, \mathcal{V}, \hat{\mathcal{D}}, (X':n, q), \text{childPhase}, Val) \]
\[ \text{else if } I \equiv r \text{ then} \]
\[ \hat{r} \leftarrow \sigma(r) \]
\[ \text{AG_EVAL}(\hat{r}, Val) \]
\[ \text{end if} \]
\[ \text{end for} \]
\[ \text{end procedure} \]

procedure AG_VISIT_SEQUENCE_EVAL(AG, \mathcal{V}, \hat{\mathcal{D}}, Val = \text{dict}())

\[ S \leftarrow AG \]
\[ \text{for } phase \leftarrow 1 \ldots |\Pi(S)| \text{ do} \]
\[ \text{AG_VISIT_SEQUENCE_EVAL_HELPER}(AG, \mathcal{V}, \hat{\mathcal{D}}, (S, 0), \text{phase}, Val) \]
\[ \text{end for} \]
\[ \text{end procedure} \]

**Figure 4.1:** Visit sequence evaluator for classical AG defined in Figure 3.9
\[ S \rightarrow A \ A \]
\[
A1.i1 = S.i \\
A2.i2 = S.i \\
A1.i2 = A1.s1 \\
A2.i1 = A2.s2 \\
S.s = A1.s2 + A2.s1
\]

\[ a \rightarrow A \]
\[
A.s1 = A.i1 \\
A.s2 = A.i2
\]

Figure 4.2: Example of non-circular AG that is not \( l \)-ordered

\[
\{ \hat{r}1 < \hat{r}8 < \hat{r}3 < \hat{r}7 < \hat{r}0 < \hat{r}5 < \hat{r}2 < \hat{r}6 < \hat{r}4 \} \quad (4.2)
\]

Schedule in Equation (4.2) is valid because it starts with setting the second child’s inherited attribute \( n2.i2 \) and then gets the resulting synthesized attribute \( n2.s2 \) from the same child. Then it sets \( n2.s2 \) as an inherited attribute \( n2.i1 \) for the same child. Then it picks up the resulting synthesized attribute \( n2.s1 \) from the second child and uses it to calculate to resolve \( n0.s \) but in reverse order for the first child. Finally, it resolves the root node’s synthesized attribute \( n0.s \) using the synthesized attributes of children.

A \( l \)-ordered attribute grammar, by definition, says that given a protocol \( \Pi(X) \), each non-terminal has a complete order of attributes that respects all dependencies. Then using a visit sequence method, we evaluate the constructed visit sequence to create a runtime schedule that will respect all the dependencies too. This runtime schedule is valid for all derivation trees \( (\text{Lemma 29}) \).

In summary, the runtime schedule generated by the protocols of non-terminal(s) of attribute grammar is compatible with the linearization of the attribute grammar dependency...
This visit sequence is invalid because child visits of the same non-terminal $A$ are invoked in an out-of-order fashion; this means that the visit sequence is incompatible with the protocol $\Pi(X)$.

**Lemma 29.** Given a protocol for which the derived attribute grammar dependency graph is all linearizable, then the AG is l-ordered

**Proof.** This can be proven using contradiction. Assuming schedule $\text{Ord}(\hat{R})$ and schedule generated by $\mathcal{V}(p)$ for all $p \in P$ are not compatible then it would mean there exists a production $p$ such that visit sequence $\mathcal{V}(p)$ is not compatible with $R(p)$. And by definition [Definition 4.1.2](#), this is not possible. Hence, a contradiction.

### 4.1.2 Visit Sequence For Circular Remote AGs

The first thing we notice when we define visit sequence for (circular) remote attribute grammar is it may contain partial field write and field read in addition to the rules that define local attributes. But as described in the previous chapter, the schedule for circular remote AGs may have cycles in them but it’s not an intrinsic feature of the set to be cyclic which is why we explicitly mark the sets as circular or non-circular in the following.

**Definition 4.1.4.** A protocol $\Pi(X)$ for each non-terminal $X \in N$ of circular remote attribute grammar is a sequence of possibly empty sets of attributes of non-terminal where some of the
sets are marked as cyclic. In fact, the difference between the summary graph and protocol in CRAG is the existence of empty sets.

**Definition 4.1.5.** A protocol \( \Pi(X) \) for non-terminal \( X \in N \) of a class of circular remote attribute grammar is well-formed if all attributes of non-terminal \( X \) exist in the protocol without duplication and no circular visit is ever next to another circular visit or at the start or at the end.

**Observation 30.** In the protocol for circular attribute grammars, a visit has to be marked circular or non-circular but there is no need for that in non-circular attribute grammars.

Protocol compatibility is defined by instantiating the protocol on LHS and RHS, and the result should be a quasi total order where there are only monotone dependencies among things in cycles.

**Definition 4.1.6.** A *visit-sequence* for a production \( p.X \rightarrow \alpha \in P \) of a \( l \)-ordered circular remote attribute grammar is \( \mathcal{V}(p) \), an ordered set of visits similar to visit sequence for classical attribute grammar but inside in each visit there can additionally be a sub-sequence of rules or child visits. Inside this subsequence, there cannot be another level subsequence. Each visit of each child occurs exactly once and they appear in the exact order they appear in the protocol. Additionally, \( \mathcal{V}(p) \) has to be compatible with \( R(p) \) and \( \Pi(X_i) \) for all \( X_i \in (p \cap T) \). To show compatibility we would need to flatten the visit sequence by replacing the visit calls with the rules that defined attribute occurrences.

- \( \mathcal{V}(p) \) is compatible with \( \Pi(X_i) \) if for each visit (outer set) in \( \mathcal{V}(p) \) we replace all visit calls \( \text{visit}((X:i, j), k) \) with just rules that define occurrence \( X_i.a \) for some attribute \( a \). And then lift the attribute part from the occurrence (i.e. \( X_i.a \rightarrow a \) ) to get quasi total order set of attributes. Then we would see that the quasi total order set of attributes is identical to the associated inner set in \( \Pi(X_i) \) (i.e. \( \Pi(X_i)[k] \)).

- \( \mathcal{V}(p) \) is compatible with \( R(p) \) if we replace all visit calls \( \text{visit}((X:i, j), k) \) with just rules that define occurrence \( X_i.a \) for some attribute \( a \). And then flattening of this quasi total
order set of sets so we just end up with rules. We can see that for any two \( r, r' \in R(p) \) which themselves are a part of \( \mathcal{V}(p) \), then \( (r < r') \in \mathcal{V}(p) \) whenever \( v_i \in DO(r) \) and \( v_i \in SUO(r') \). Similarly, \( (r \preceq r') \in \mathcal{V}(p) \) whenever \( v_i \in DO(r) \) and \( v_i \in MUO(r') \).

**Remark 31.** Attribute instances declared non-circular are not allowed in circular visits.

**Remark 32.** Protocols do not have subsequences in them, and subsequences are allowed only in non-circular visits of visit sequences.

**Remark 33.** Any quasi total order of attributes of non-terminal \( T(X) \) in a circular attribute grammar can be converted to a protocol \( \Pi(X) \).

**Proof.** There exists a function \( f \) that can map between the visit sequence for circular AGs and the definition of \( l \)-ordered for circular AGs (see [Definition 3.4.11](#)). The function \( f \) is a non-injective surjective function (surjection, not a bijection). This proof is similar to the proof of [Remark 25](#).

- **Forward direction:** the protocol \( \Pi(X) \) can be mapped into a summary graph quasi total order \( (SDG_X, \preceq) \), and the visit sequence \( \mathcal{V}(p) \) can be mapped into augmented dependency graph quasi total order \( (DG^*_p, \preceq) \) by removing the visit partitions and mapping each semantic rule in the visit sequence to an attribute occurrence being defined on LHS of the semantic rule. There can be empty visits or different combinations of a grouping of the attributes in a single visit of the protocol that maps to the same summary graph quasi total order. Hence a surjection and not a bijection.

- **Backward direction:** the summary graph quasi total order \( (DG^*_p, \preceq) \) can be mapped into one or more different protocols \( \Pi(X) \) by putting the attributes into partitions that are called “visits” which act as an interface for visit sequence. The quasi total order of the augmented dependency graph \( (DG^*_p, \preceq) \) can be mapped into a visit sequence \( \mathcal{V}(p) \) by mapping an attribute occurrence to a semantic rule that defines the attribute occurrence and inserting a child visit call of the appropriate phase between child-inherited
and child-synthesized attributes such that the resulting visit sequence is compatible with the protocol interface and this visit call serves as the recursive evaluation call that evaluates the child’s attributes. The algorithm for this is defined in Algorithm 4 and it’s well-formed since all dependencies used for static scheduling are lifted from the augmented dependency graph quasi total order, hence dependencies are compatible with each other.

Algorithm 4 SCC Chunk visit sequence generator for circular remote attribute grammar

```
procedure CRAG_CHUNK_READY_TO_SCHEDULE( chunkGraph, scheduled, chunkComponent, chunk )

(chunks, dependencies) ← chunkGraph
▷ look for unsatisfied direct dependencies inside of the component
for dependentChunk ← (dependencies(chunk) ∩ chunkComponent) do
    dependencyEdgeKind ← EDGE_KIND( dependentChunk, chunk )
    if dependentChunk ∉ scheduled ∧ dependencyEdgeKind = directEdge then
        if chunk ∈ dependencies(dependentChunk) then
            forwardEdgeKind ← EDGE_KIND( chunk, dependentChunk )
            if forwardEdgeKind = directEdge then
                continue
                ▷ bidirectional direct dependency detected and ignored
            end if
        end if
    end if
return false
end for
return true
end procedure

procedure CRAG_GENERATE_CHUNK_VISITS( AG, p:X_0 → X_1...X_k, chunkPartition )

visitPartition ← array(|chunkPartition|)
for v ∈ chunkPartition do
    for r ∈ R(p) do
        if LHS(r) = v then
            append(visitPartition, r)
        end if
    end for
end for
return visitPartition
end procedure
```
procedure CRAG_FIND_NEXT_CHUNK_TO_SCHEDULE( chunkGraph, scheduled, chunkComponent, except )

(chunks, dependencies) ← chunkGraph
availableChunks ← (chunkComponent − (scheduled ∪ except))
for chunk ← availableChunks do
    chunkReady ← CRAG_CHUNK_READY_TO_SCHEDULE(chunkGraph, scheduled, chunkComponent, chunk)
    if chunkReady = true then
        return chunk
    end if
end for
return null
end procedure

procedure CRAG_FIND_NEXT_INNER_COMPONENT_TO_SCHEDULE( chunkGraph, scheduled, chunkComponents )

(chunks, dependencies) ← chunkGraph
for chunkComponent ← chunkComponents do
    ▷ exclude components containing parent inherited or parent synthesized chunks
    if componentReady ∩ ((∪ halfLefts) ∪ (∪ halfRights)) ∩ (scheduled ∪ except) = ∅ then
        continue
    end if

    componentReady ← CRAG_CHUNK_COMPONENT_READY_TO_SCHEDULE(chunkGraph, scheduled, chunkComponent)
    if componentReady = true then
        return chunkComponent
    end if
end for
return null
end procedure

99
procedure ASSERT_CHUNK_COMPONENTS($p: X_0 \rightarrow X_1 \ldots X_k, \Pi, state,\newline$ \hspace{0.5cm} chunkGraph, chunkComponents)\newline

$(locals, halfLefts, halfRight, childVisits, chunks) \leftarrow state$\newline
$(chunks, dependencies) \leftarrow chunkGraph$\newline

for phase $\leftarrow 1 \ldots |\Pi(X_0)|$ do\newline

circularPhase $\leftarrow IS_CIRCULAR(\Pi(X_0)[phase])$\newline

if circularPhase $=$ true then\newline

chunkComponent $\leftarrow \forall chunkComponent \in chunkComponents,\newline
\exists$ chunkComponent such that\newline

halfLeft[phase] $\in$ chunkComponent $\lor$\newline
halfRight[phase] $\in$ chunkComponent\newline

▷ find all parent chunks (half-left and half-right) in this component\newline

parentChunks $\leftarrow ((\bigcup_{\text{phase}} \text{halfLeft}) \cup (\bigcup_{\text{phase}} \text{halfRight})) \cap$ chunkComponent\newline

if halfLefts[phase] $\not\in$ chunkComponent then\newline

Error: component is missing parent inherited chunk for the phase\newline
else if halfRights[phase] $\not\in$ chunkComponent then\newline

Error: component is missing parent synthesized chunk for the phase\newline
else if $(\text{parentChunks} - \{\text{halfLeft[phase]}, \text{halfRight[phase]}) \neq \emptyset$ then\newline

Error: component cannot involve more than one parent phase\newline
end if\newline

for chunk $\in$ chunkComponent do\newline

if chunk $\in (\bigcup \text{childVisits})$ then\newline

(childPhase, childIndex) $\leftarrow$ chunk\newline

if IS_CIRCULAR(\Pi(X_{childIndex})[childPhase]) $=$ false then\newline

Error: non-circular child attributes are involved in a cycle\newline
end if\newline
else if chunk $\in (\bigcup \text{locals})$ then\newline

▷ check whether chunk depends on itself, if so its circular,\newline
otherwise, it’s non-circular\newline

if chunk $\not\in$ dependencies(chunk) $\in$ then\newline

Error: non-circular local chunk is involved in a cycle\newline
end if\newline
end if\newline
end for\newline
end if\newline
end for\newline
end procedure
this function is only used by non-circular phase scheduler to schedule subsequences

```
procedure CRAG_GENERATE_INNER_COMPONENT_VISITS( AG, p, state, chunkGraph, scheduled, chunkComponent, scheduledChildPhases )
(locals, halfLefts, halfRights, childVisits, chunks) ← state
result ← array_list()
while true do
    chunk ← CRAG_FIND_NEXT_CHUNK_TO_SCHEDULE(chunkGraph, scheduled, chunkComponent)
    if chunk = null then
        break▷ there is no more chunk to schedule in this visit
    end if
   ▷ there is no rule in R(p) that defines parent inherited attributes so its skipped
    append(scheduled, chunk)
    if chunk ∈ locals then
        append(result, CRAG_GENERATE_CHUNK_VISITS(AG, p, chunk))
    else if chunk ∈ (∪∪ childVisits) then
        (childPhase, childIndex) ← chunk
        inherited ← {v | v ∈ chunk ∧ v ≡ X.a ∧ a ∈ Inh(X[childIndex])}
        childPhases[childIndex] ← childPhase
       ▷ there is no rule in R(p) that defines child synthesized attributes
        so it is safely skipped
        append(result, AG_GENERATE_CHUNK_VISITS(AG, p, inherited))
        append(result, visitMarker(phase, childIndex))▷ child visit marker
    end if
end while
return result
end procedure
```

```
procedure COMPONENT_IS_CIRCULAR(chunkGraph, chunkComponent)
(chunks, dependencies) ← chunkGraph
for chunk ← component do
    if chunk ∈ dependencies(chunk) then
        return true
    end if
end for
return false
end procedure
```
procedure CRAG_GENERATE_COMPONENT_VISITS_NON_CIRCULAR(AG, p, Π, state, chunkGraph, scheduled, chunkComponent, scheduledChildPhases, visits, phase)

(locals, halfLefts, halfRights, childVisits, chunks) ← state

visit ← array_list()

parentInheritedReady ← CRAG_CHUNK_READY_TO_SCHEDULE(chunkGraph, scheduled, halfLefts[phase])

if parentInheritedReady = false then
    Error: expected parent inherited chunk for parent phase to be ready
end if

▷ there is no rule in R(p) that defines parent inherited attributes so its skipped
append(scheduled, halfLefts[phase])

while true do

    chunkComponent ← CRAG_FIND_NEXT_INNER_COMPONENT_TO_SCHEDULE(chunkGraph, scheduled, chunkComponents)

    if chunkComponent = null then
        break
        ▷ there is no more chunk component to schedule in this visit
    end if

    result ← CRAG_GENERATE_INNER_COMPONENT_VISITS(AG, p, state, chunkGraph, scheduled, chunkComponent, scheduledChildPhases)

    if COMPONENT_IS_CIRCULAR(chunkGraph, chunkComponent) = true then
        ▷ append all the component visit result (no sub-sequence)
        appendAll(visit, result)
    else
        ▷ append the component visit result as a sub-sequence
        ▷ this will be wrapped in a fixed-point loop during the evaluation
        append(visit, result)
    end if
end while

parentSynthesizedReady ← CRAG_CHUNK_READY_TO_SCHEDULE(chunkGraph, scheduled, halfRights[phase])

if parentSynthesizedReady = false then
    Error: expected parent synthesized chunk for parent phase to be ready
end if

append(scheduled, halfRights[phase])
appendAll(visit, CRAG_GENERATE_CHUNK_VISITS(AG, p, halfRights[phase]))
append(visit, visitMarker(phase, −1)) ▷ end of parent phase visit marker

visits[phase] ← visit

end procedure
procedure CRAG_GENERATE_COMPONENT_VISITS_CIRCULAR( CRAG, p, Π, state, chunkGraph, scheduled, chunkComponent, scheduledChildPhases, visits, phase )

(locals, halfLefts, halfRights, childVisits, chunks) ← state
parentInheritedReady ← CRAG_CHUNK_READY_TO_SCHEDULE(chunkGraph, scheduled, halfLefts[phase])

if parentInheritedReady = false then
    Error: expected parent inherited chunk for parent phase to be ready
end if
▷ there is no rule in $R(p)$ that defines parent inherited attributes so its skipped
append(scheduled, halfLefts[phase])

while true do
    chunk ← CRAG_FIND_NEXT_CHUNK_TO_SCHEDULE(chunkGraph, scheduled, chunkComponent, {halfRight[phase]})
    if chunk = null then
        break
    end if
    append(scheduled, chunk)

if chunk ∈ locals then
    appendAll(visits[phase], AG_GENERATE_CHUNK_VISITS(AG, p, chunk))
else if chunk ∈ (∪ childVisits) then
    (childPhase, childIndex) ← chunk
    inherited ← \{v | v ∈ Inh(X_{childIndex}) ∧ v ∈ chunk\}
    childCircular ← IS_CIRCULAR( Π(X_{childIndex})[childPhase] )
    childPhases[childIndex] ← childPhase
    if childCircular = false then
        Error: non-circular visit cannot exist inside circular parent visit
    end if
    ▷ there is no rule in $R(p)$ that defines child synthesized attributes so it is safely skipped
    appendAll(visits[phase], CRAG_GENERATE_CHUNK_VISITS(CRAG, p, inherited))
    append(visits[phase], visitMarker(phase, childIndex)) ▷ child visit marker
end if
end while

parentSynthesizedReady ← CRAG_CHUNK_READY_TO_SCHEDULE(chunkGraph, scheduled, halfRights[phase])

if parentSynthesizedReady = false then
    Error: expected parent synthesized chunk for parent phase to be ready
end if
append(scheduled, halfRights[phase])
appendAll(visit, CRAG_GENERATE_CHUNK_VISITS(CRAG, p, halfRights[phase]))
append(visit, visitMarker(phase, −1)) ▷ end of parent phase visit marker
visits[phase] ← visit
end procedure
procedure CRAG_VISIT_SEQUENCE_GEN( CRAG, \( p : X_0 \rightarrow X_1 \ldots X_k \), \( \Pi, T : (CRAG^*_{\leq}) \) )
state \leftarrow \text{AG_CREATE_CHUNKS}(AG, p, \Pi)
(loals, halfLefts, halfRights, childVisits, chunks) \leftarrow state
chunkGraph \leftarrow \text{AG_LIFT_CHUNK_DEPENDENCY_GRAPH}(CRAG, p, T : (DG^*_{\leq}), state)
chunkGraph \leftarrow \text{AG_ADD_GUIDING_DEPENDENCIES}(CRAG, p, \Pi, state, chunkGraph)
chunkGraph \leftarrow \text{transitive_closure}(chunkGraph)
chunkComponents \leftarrow \text{graph_components}(chunkGraph)
\triangleright\text{} ensure chunk components are formed correctly
\text{ASSERT_CHUNK_COMPONENTS}(p, \Pi, state, chunkGraph, chunkComponents)
visits \leftarrow \text{array}(|\Pi(X_0)|)
scheduled \leftarrow \emptyset
scheduledChildPhases \leftarrow [0] \times \text{array}(k) \quad \triangleright\text{} array of size \( k \) initialized to 0
\triangleright\text{} start the scheduler from the first visit of the parent
for phase \in 1 \ldots |\Pi(X_0)| \ do
visits[phase] \leftarrow \text{array_list}() \quad \triangleright\text{} initialize each visit to a dynamically sized list
if \text{IS_CIRCULAR}(\Pi(X_0)[phase]) = \text{false} \ then
\text{CRAG_GENERATE_COMPONENT_VISITS_NON_CIRCULAR}(CRAG, p, \Pi, state, chunkGraph, scheduled, scheduledChildPhases, visits[phase], phase)
\text{MARK_CIRCULAR}(visits[phase], false)
\text{else}
\triangleright\text{} already asserted to make sure such chunk component exists
chunkComponent \leftarrow \forall chunkComponent \in \text{chunkComponents},
\exists chunkComponent \text{} such that
halfLeft[phase] \in chunkComponent
\wedge halfRight[phase] \in chunkComponent
\text{CRAG_GENERATE_COMPONENT_VISITS_CIRCULAR}(CRAG, p, \Pi, state, chunkGraph, scheduled, chunkComponent, scheduledChildPhases, visits, phase)
\text{MARK_CIRCULAR}(visits[phase], true)
\text{end if}
\text{end for}
if scheduled \neq \text{chunks} \ then
\text{Error: generated visit sequence is missing some chunks}
\text{else if scheduledChildPhases} \neq (|\Pi(X_1)|, \ldots, |\Pi(X_k)|) \ then
\text{Error: generated visit sequence does not corresponds with its children protocols}
\text{end if}
\text{return visits}
\text{end procedure}
Equation (4.3) and Equation (4.4) describe the protocols and visit sequence for circular remote attribute grammar defined in Figure 3.33. This visit sequence is used as an input for a visit sequence evaluator for CRAG in Figure 4.5.

\begin{align*}
\Pi(S) & = \begin{cases} 
\text{non-circular visit} & \{S.x\} \\
\text{empty non-circular visit} & \{\}\end{cases} \\
\Pi(A) & = \begin{cases} 
\text{empty non-circular visit} & \{A.r\} < \\
\text{circular visit} & \{A.i\} \langle \\
\text{empty non-circular visit} & \{\}\end{cases} \\
\Pi(B) & = \begin{cases} 
\text{empty non-circular visit} & \{B.i\} < \\
\text{circular visit} & \{B.s\} \langle \\
\text{empty non-circular visit} & \{\}\end{cases}
\end{align*}

\begin{align}
(4.3)
\end{align}

\begin{align*}
\nu(p:S \rightarrow AB) = \begin{cases} 
\text{subsequence cycle (fixed-point loop needed)} & \{\text{visit}(A, 2), \text{visit}(B, 2), (A.i=B.s)\} \langle \\
\text{single parent visit corresponding to } \Pi(S)[0] & \{\text{visit}(A, 1) < (l=A.r) < (B.i=l) < \text{visit}(B, 1) < \\
\text{visit}(A, 3) < \text{visit}(B, 3) < (S.x=l.f)\}
\end{cases}
\end{align*}

\begin{align}
(4.4)
\end{align}

\begin{align*}
\nu(p:A \rightarrow a) = \begin{cases} 
\text{non-circular visit} & \{(A.r=a)\} < \\
\text{visit} & \{o.f \equiv A.i\} \langle \\
\text{empty non-circular visit} & \{\}\end{cases}
\end{align*}

\begin{align*}
\nu(p:B \rightarrow b) = \begin{cases} 
\text{non-circular visit} & \{(l=B.i)\} < \\
\text{visit} & \{(B.s=l.f)\} \langle \\
\text{non-circular visit} & \{\}\end{cases}
\end{align*}

\subsection{Visit Sequence Evaluation}

The visit sequence evaluator for circular remote attribute grammar differs from the visit sequence evaluator for classical attribute grammar in the sense that it has to take cycles into account. More specifically, it uses markings defined in Definition 4.1.6 to wrap the contents of the visit from the start of the cycle to the end of a cycle in a fixed-point loop.

The fundamental idea is that only circular visits inside of non-circular visits get iterated.
to a fixed point. However, one needs to ensure child non-circular visits do not get evaluated repeatedly if their parent visit is wrapped in a fixed-point loop. Similarly, circular visits inside of a circular visit do not need to be wrapped in a fixed-point loop because it would cause unnecessary looped evaluation inside of a looped evaluation. The schedule or the order of evaluation of semantic rules in CRAG visit sequence evaluation is equivalent to the schedule generated during the demand evaluation for CRAG and it is valid as attributes in a cycle are evaluated over and over again until their values reach a fixed-point. However, the difference is visit sequence evaluation does not need a runtime dependency graph as opposed to demand evaluation and this substantially improves its time complexity. Table 4.1 is a case-by-case description of each scenario where the fixed-point loop is needed.

Algorithm 4 modifies Code 6 to construct a visit sequence evaluator subroutine for CRAG. Notice that it passes a Boolean value of false as an argument for parentInCycle to denote that visit sequence evaluation is not initially called from a circular visit.
Visit Type | Fixed-Point Loop?
--- | ---
Circular visit inside of circular visit | No fixed-point loop. Since the parent visit repeats the evaluation, its loop includes the child visit as well.
Non-circular visit in a circular visit | Not allowed.
Non-circular visit in a non-circular visit | No fixed-point loop. The visit has to be evaluated only once.
A circular visit in a non-circular visit | Needs a fixed-point loop.

Table 4.1: case-by-case analysis of loop requirement in CRAG visit sequence evaluator

```plaintext
Code 7 Visit sequence evaluator for circular remote attribute grammar

procedure CRAG_VISIT_SEQUENCE_EVAL(CRAG, ℱ, ℷ, Val = dict())
    ▷ start the recursive visit sequence evaluation at the first visit of the root phylum
    for phase ← 1..|Π(X)| do
        V ← ℱ(p)[phase] ▷ evaluate only this phase
        isCircular ← IS_MARKED_CIRCULAR(V)
        CRAG_VISIT_SEQUENCE_EVAL_HELPER(CRAG, ℱ, ℷ, (S, 0), phase, Val, isCircular)
    end for
end procedure
```

4.2 Group Scheduling

This research introduces a novel approach to schedule attributes. In this approach, attributes are put into groups and scheduled accordingly as a group, instead of being scheduled individually. This means a group is ready to be scheduled if all attributes in that group are ready to be scheduled. This was needed to simplify the existing visit-sequence-evaluator generator in APS and make it possible to accommodate circular remote AGs.

The following discusses the new group scheduling algorithm. Assuming a total number of attribute instances is \( n \), APS assigns a tuple \( (ph \times ch) \) for each attribute instance such that \( ph \geq 1 \) and \( ch \geq -1 \) for all attribute instances where \( ph \) refers to the phase of visit and \( ch \) refers to the child visit number. For the local attributes, the value is \( (0, 0) \). The new scheduler schedules attribute as a group or block meaning that it schedules the whole block if all the attributes in a block are ready to be scheduled. That is their dependency and transitive dependencies are scheduled already or will be scheduled in the same group.
procedure CRAG_VISIT_SEQUENCE_EVAL_HELPER(CRAG, V, \( \mathcal{D} \), (X:i, j), Val, V, parentIsCircular)

\( (p:X \rightarrow \alpha) \leftarrow \mathcal{D}[j] \) ▷ get production from the derivation step

for \( I \leftarrow V \) do

if \( I \equiv \text{visit}(X':n, \text{childPhase}) \) then ▷ child visit marker

\( (p', q) \leftarrow \exists q \in [j + 1, \ldots, |\mathcal{D}|] \) such that (\( p':\alpha_q \rightarrow \alpha_{q+1} \equiv \mathcal{D}[q] \land \alpha_q \equiv \hat{\alpha}(X':n,q)\hat{\beta} \land \alpha_{q+1} \equiv \hat{\alpha}\hat{\delta}\hat{\beta} \))

▷ check whether child visit is circular or not

\( V' \leftarrow V(p')[\text{childPhase}] \)

\( \text{childIsCircular} \leftarrow \text{IS_MARKED_CIRCULAR}(V') \)

▷ recursive child visit evaluation call

CRAG_VISIT_SEQUENCE_EVAL_HELPER(CRAG, V, \( \hat{\mathcal{D}} \), (X':n, q), childPhase, Val, childIsCircular)

else if \( I \equiv r \) then ▷ semantic rule

\( \hat{r} \leftarrow \sigma(r) \)

▷ instantiate the rule before evaluating it

CRAG_EVAL(\( \hat{r}, Val \))

else if \( I \equiv (\ldots) \) then ▷ nested sub-sequence

if parentIsCircular = true then

Error: expected fixed-point loop for subsequence to appear only in non-circular parent phase.

end if

changed \( \leftarrow false \)

do

\( \text{changed} \leftarrow false \) ▷ copy over attribute instance value dictionary

\( \text{prev} \leftarrow \text{Val} \) ▷ recursively evaluate the sub-sequence

CRAG_VISIT_SEQUENCE_EVAL_HELPER(CRAG, V, \( \hat{\mathcal{D}} \), (X:i, j), Val, I, parentIsCircular)

if \( \text{Val} \neq \text{prev} \) then

\( \text{changed} \leftarrow \text{true} \)

end if

end while

end if

end if

end for

end procedure
For the locals, it also makes sure they are scheduled according to the total order of the augmented dependency graph. Note that all attribute instances in a group are scheduled together which is the key differentiation of the new scheduler. Lastly, it uses two types of markers to mark the visit sequence for the code generation module. The first one is marking where a child’s visit happens (between the inherited and synthesized attributes of that visit for the child). This is of the form \((ph, ch)\). Second marking is where a visit of the parent ends (and potentially where the next visit starts), which is after any synthesized attribute of the parent for that visit and just before the inherited attributes of the next visit. This is of the form \((ph, -1)\). See [Equation (4.5)] for the case-by-case analysis. In summary, the group scheduler schedules the attributes in this order:

1. A whole group of inherited attributes of the parent (start of a phase) marked by \((-ph, -1)\). This group could be empty if there is no parent-inherited attribute belonging to this phase.

2. A whole group of attributes including both inherited and synthesized attributes for a child phase marked by \((ph, ch)\).

3. A whole group of synthesized attributes of the parent (end of the phase or visit) marked by \((ph, -1)\). This group could be empty if there is no parent-synthesized attribute belonging to this phase.

Between each block, the group scheduler also assigns visit markers as a way to signal the code generation or any other module that uses the static schedule. There are two types of visit markers:

- mark where a child’s visit happens (between the inherited and synthesized attributes for that visit for the child). This is of the form \((ph, ch)\).

- mark where a visit of the parent ends (and potentially where the next visit starts), which is after any synthesized attribute of the parent for that visit and just before the
inherited attributes of the next visit. This is of the form \((ph, -1)\).

\[
ph = \begin{cases} 
  < 0 & \text{attribute is inherited} \\
  > 0 & \text{attribute is synthesized}
\end{cases}
\]

\[
ch = \begin{cases} 
  -1 & \text{parent attribute} \\
  \geq 0 & \text{child attribute}
\end{cases}
\]

The following schedules the Figure 4.6 using the new group scheduler algorithm. Notice that we assign new \((ph, ch)\) to each attribute instance.

match ?self : Items = Items$append(?items1,?items2 : Items) begin
  self.items_first := black_dot(items1.items_first, items2.items_first);
end;

Figure 4.6: append top-level match of First example

The first step is to find phyla(s) used under append and associate an alphabetic counter for each child (starting from zero) which will be used in Figure 4.8. APS already has the functionality to find these and by observation, it can be double-checked. Two phyla used under append are: Items and black dot and the parent node is of Items phyla.

The following are the attributes used in Items phyla:

(a) \(G[\text{Items}].\\text{sharedinfo} (\text{inherited})\)
(b) \(G[\text{Items}].\\text{sharedinfo}\\text{$\epsilon$} (\text{inherited})\)
(c) \(G[\text{Items}].\\text{sharedinfo}\\text{$\epsilon$} \text{firstTable} (\text{inherited})\)
(d) \(\text{Items}._{\text{first}} (\text{synthesized})\)

The following are the attributes used in black dot phyla:

(e) \(G[\text{black dot}]\\text{sharedinfo} (\text{inherited})\)
(f) \(G[\text{black dot}]\\text{sharedinfo}\\text{$\epsilon$} (\text{inherited})\)
(g) \(\text{s1} (\text{inherited}): \text{first formal of the black dot function call}\)
(h) \(\text{s2} (\text{inherited}): \text{second formal of the black dot function call}\)
(i) \(\text{result} (\text{synthesized})\)
The next step is to find the total order for `append` top-level-match which requires phase assignment for each attribute occurrence. APS already has the functionality to calculate it, however, it does not work for static circular evaluation because it does not separate out circular and non-circular attributes in different phases. This is important as circular and non-circular attributes should not be mixed together in one phase. To solve this issue we modified the total order to start with non-circular attributes and try to include as many non-circular attributes of a phylum in a single phase as possible without breaking their dependencies. Then repeat the process for circular attributes for the second phase. Then schedule non-circular attributes for the third phase and so on.

It’s important to know that an attribute is considered circular if it is declared so. There are checks in the implementation to catch and throw out errors when the attribute is not declared circular but is involved in a cycle. Similarly, our implementation logs warning messages if the attribute is declared circular but it does not involve in any cycle.

Figure 4.7 shows the phase assignments for phylum `Items`. Notice that the first phase is dedicated to non-circular attributes and the second phase is just circular attributes:

```
// non-circular phase: 1
1- G[Items]’shared_info
1- G[Items]’shared_info$epsilon

// circular phase: 2
2- G[Items]’shared_info$firstTable
2+ items_first

// empty non-circular phase: 3
```

Figure 4.7: `Items` phylum phase assignments

Figure 4.9 is the static schedule for `append` top-level match. Notice the use of visit markers and two-parent visits. Also, the order of locals like function formals does not really matter for the code generation but what is important is the result of the function call precedes the parent synthesized attribute `self.items_first`. 

111
Figure 4.8: Attribute occurrences used in append with their \((ph, ch)\) assignment

The sanity of a static schedule can be verified by checking the completeness of the child and parent phase and the consecutiveness of child visits. By completeness, we mean that if the parent phylum has two \(n\) phases then the schedule should also have \(n\) parent visits (and \(n\) end of parent phase visit markers). The same goes for its children. If its children have \(m\) phases then the schedule should include \(m\) child visit calls for that particular child. By consecutiveness of child visits, we are referring to the fact that if a child phylum has \(k\) phases then the first visit of that child should be for phase 1 and the second visit of that child should be for phase 2, and so on until visit for phase \(k\). Lastly, the schedule can be verified by making sure that at the start of each parent visit all parent-inherited attributes for that phase are scheduled and before the end of the visit all parent-synthesized attributes for that phase are scheduled. Similarly, before child visit calls, all inherited attributes for that phase are scheduled and then followed by child visit call, and then all child synthesized attributes of that phase.
Visit 1:
self.G[Items]'shared_info: <-1,-1>
self.G[Items]'shared_info$epsilon: <-1,-1>
    items1.G[Items]'shared_info: <-1,0>
    items1.G[Items]'shared_info$epsilon: <-1,0>
    visit(1, 0) // visit marker: between child inherited and synthesized
    <1,0> // empty placeholder, no synthesized attribute of child
    items2.G[Items]'shared_info: <-1,1>
    items2.G[Items]'shared_info$epsilon: <-1,1>
    visit(1, 1) // visit marker: between inherited and synthesized
    <1,1> // empty placeholder, no synthesized attribute of child
    <1,-1> // visit marker: done with parent visit 1

Visit 2:
self.G[Items]'shared_info$firstTable: <-2,-1>
    items1.G[Items]'shared_info$firstTable: <-2,0>
    visit(2, 0) // visit marker: between inherited and synthesized
    items1.items_first: <2,0>
    items2.G[Items]'shared_info$firstTable: <-2,1>
    visit(2, 1) // visit marker: between inherited and synthesized
    items2.items_first: <2,1>
    black_dot(...):55.G[black_dot]'shared_info: local (<0,0>)
    black_dot(...):55.G[black_dot]'shared_info$epsilon: local (<0,0>)
    black_dot(...):55.s1: local (<0,0>)
    black_dot(...):55.s2: local (<0,0>)
    black_dot(...):55.result: local (<0,0>)
    self.items_first: <2,-1>
    <2,-1> // visit marker: done with parent visit 2

Visit 3:
visit(3, 0) // visit marker: between inherited and synthesized
visit(3, 1) // visit marker: between inherited and synthesized
<3,-1> // visit marker: done with parent visit 3

Figure 4.9: Static schedule for append top-level match
4.3 Fiber Cycle Breaking

The goal of this research is to statically schedule circular remote attribute grammars and it’s very important to be able to visualize cycles in a dependency graph first before diving into the fiber cycle breaking algorithm. For the sake of visualization, Figure 4.10 is the visualization of the dependency graph for `append`. Boyland in [8] denotes objects as fibers because they act as a “rope” as they implicitly carry values in their fields. In the same work, an algorithm to break cycles between fibered attributes was introduced. Informally, for every non-terminal in the cycle, it creates two new attributes, an `up` attribute, and a `down` attribute. First, it would first run a union-find algorithm to find strongly connected regions in the dependency graph. Then all nodes of the strongly-connected region are removed, all dependencies into the strongly-connected region are moved to the `up` attribute of the LHS non-terminal, and all dependencies out of the strongly-connected region are moved to the `down` attribute of the LHS non-terminal. Thereafter, the DNC test would be used against the dependency graph to detect cycles that should pass indicating cycle(s) have successfully been removed. A trivial observation one can make is this setup always preserves edges or dependency between synthesized and then inherited in the dependency graph.
The previous algorithm successfully removed the cycles given a circular reference AGs but ended up creating a resulting augmented dependency graph that was not valid anymore, which means some of the direct dependencies were removed unintentionally. This is because the nature of fiber cycles breaking in RAG and CRAG are very different. Unlike cycles that may appear in RAG, the cycles in CRAG are actually carrying value, and evaluating them may require running the same path finite times and breaking the edges to cut the cycle will break the dependency graph. This research had to rewrite that algorithm because it was creating two dummy nodes and it was complicating the static schedule, and re-writing it ended up simplifying the algorithm in the process.

The new fiber cycle-breaking algorithm is described in Algorithm 5. Note that graph is a directed graph and consists of a list of tuples $i, j$ and a dependency $d$. Notice that no dummy attribute is created, instead, individual attributes are considered either up or down depending on the fiber direction. This change resulted in simplifying the algorithm itself and making it easy to understand, simplifying the result of the algorithm by removing special dummy nodes and achieving the same result as the original algorithm described in [8].

Interestingly, due to the nature of the new proposed CRAG static scheduler, cycles are embraced so it means there is no need to run the spurious fiber cycle breaking algorithm even for AGs with pure fiber cycles because the new CRAG static scheduler can handle fiber cycles as well. This is because fiber dependencies are monotone by definition and so the effort to improve the UP-DOWN algorithm was fruitless and we did not use the UP-DOWN for static scheduling of any type of AGs.

4.4 SCC Chunk Scheduling

SCC-chunk scheduling is a proposed static scheduling method that generates a schedule for circular remote attribute grammar. More specifically, it uses strongly connected graph components of chunk and then it builds upon the group scheduler and uses it as a subroutine.
Algorithm 5 New add_up_down function implemented into APS

1:  \textbf{procedure} \textsc{add_up_down}(\hat{N}, C, \text{Dependency})
2:   \textbf{for} c \leftarrow C \textbf{ do} \quad \triangleright \text{for each cycle } c \text{ in } C
3:     \textbf{for} i \leftarrow \hat{N} \textbf{ do} \quad \triangleright \text{for each attribute occurrence } i
4:        \text{if } i \notin c \textbf{ then}
5:          d \leftarrow 0
6:          \textbf{for} j \leftarrow c \cap (\hat{N} \setminus \{i\}) \textbf{ do}
7:             \triangleright \text{Accumulate the dependencies from } i \text{ to attributes in cycle } c
8:              d \leftarrow d + \text{Dependency}(i, j)
9:          \textbf{end for}
10:          \text{if } d \neq 0 \textbf{ then}
11:             \textbf{for} j \leftarrow c \cap (\hat{N} \setminus \{i\}) \textbf{ do}
12:                \triangleright \text{Assign the accumulated dependency for edge } i \rightarrow j
13:                   \text{Dependency}(i, j) \leftarrow d
14:             \textbf{end for}
15:          \textbf{end if}
16:          d \leftarrow 0
17:          \textbf{for} j \leftarrow c \cap (\hat{N} \setminus \{i\}) \textbf{ do}
18:             \triangleright \text{Accumulate the dependencies from attributes in cycle } c \text{ to } i
19:              d \leftarrow d + \text{Dependency}(j, i)
20:          \textbf{end for}
21:          \text{if } d \neq 0 \textbf{ then}
22:             \textbf{for} j \leftarrow c \cap (\hat{N} \setminus \{i\}) \textbf{ do}
23:                \triangleright \text{Assign the accumulated dependency for edge } j \rightarrow i
24:                   \text{Dependency}(j, i) \leftarrow d
25:          \textbf{end for}
26:      \textbf{end if}
else if $i \in c \land DIRECTION(i) = \text{UP}$ then
  if instance_is_up(instance) then
    $d \leftarrow 0$
    for $j \leftarrow c \cap (\hat{N} - \{i\})$ do
      ▷ Accumulate the dependencies from $i$ to attributes in cycle $c$
      $d \leftarrow d + \text{Dependency}(i, j)$
    end for
    if $d \neq 0$ then
      for $j \leftarrow c$ do
        if $DIRECTION(j) = \text{DOWN}$ then
          $\text{graph} \leftarrow (i \Rightarrow j)$
        end if
      end for
    end if
  end if
  for $j \leftarrow c$ do
    if $DIRECTION(j) = \text{UP}$ then
      $\text{graph} \leftarrow \text{graph} - \{(i \Rightarrow j)\}$
    end if
  end for
else if $i \in c \land DIRECTION(i) = \text{DOWN}$ then
  for $j \leftarrow c$ do
    $\text{graph} \leftarrow \text{graph} - \{(i \Rightarrow j)\}$
  end for
end if
end for
end procedure
It's a greedy scheduler that does topological sorting but unlike a classical topological sort algorithm that does not notice something depends on itself, this greedy scheduler is designed to handle cycles. Unlike the OAG algorithm that works backward from the dependencies to construct the static schedule, this greedy scheduler starts from the parent inherited attribute of the first phase and moves forward which also helps simplify the implementation as there could be conditionals (or branches) along the way that splits the path into two positive and negative.

**Definition 4.4.1.** SCC-chunk scheduling is a method that generates a visit sequence for circular remote attribute grammar. More specifically, it builds upon the group scheduler and uses it as a subroutine.

- First phylum attributes have to be scheduled and put into phases. That is for all \( X \in N \), for each \( X.a \in \text{Attr}(X) \), it starts by greedily selecting the largest group of non-terminal attributes \( \{a_i, \ldots, a_j\} \) if any such that it does not conflict with rest of the graph that has not been scheduled yet. Then it continues with selecting a set of attributes \( \{a'_i, \ldots, a'_j\} \) such that they belong to a cycle. And repeat. The result is a sequence of sets of attributes for each non-terminal. This is also called a protocol \( \Pi(X) \).

\[
\begin{align*}
\{a_i, \ldots, a_j\} &< \{a'_i, \ldots, a'_j\} < \{\ldots\} \ldots &\text{circular phase} \\
\nonumber &\text{non-circular phase} &\text{non-circular phase} &\text{non-circular phase} &\text{circular phase} \\
\{\ldots\} &< \{\ldots\} < \{\ldots\} &\text{non-circular phase} &\text{non-circular phase} &\text{non-circular phase} \\
\end{align*}
\]

- Then a visit sequence for production \( X \rightarrow \alpha \) is generated where each visit \( k \) begins and ends with chunks or a set of rules associated with parent inherited and synthesized attributes corresponding to \( \Pi_i(X)[k] \) and \( \Pi_s(X)[k] \).
Figure 4.11: Flow chart describing different pieces of SCC chunk scheduling for CRAG
**Definition 4.4.2.** The schedule for circular remote attribute grammar is called *desirable* when it ensures that all direct dependencies are respected. This is only important for scoping purposes in the code generation as in the fixed-point loop (or partitions in the quasi total order schedule), the order of the valuation is arbitrary.

**Remark 34.** Desirability is only defined for the class of circular remote attribute grammar (CRAG) because of the quasi total order ($\leq)$ on the set of instantiated rules. A desirable schedule in well-defined non-circular attribute grammars is always a valid schedule. Figure 5.16 is an example of why desirability is important for scoping purposes.
Chapter 5

Implementation

This chapter overviews the implementation that makes scheduling circular remote attributes possible. We divide this chapter into two sections, prerequisites, and a new scheduler. In the prerequisites section, we describe the preliminary changes done to APS that are not considered breaking change but they improved the APS in such a way that made it possible for the new scheduler would be a drop-in replacement to the old greedy scheduler. Then in the new scheduler section, it will discuss all new SCC chunk scheduler and how it divides a single “visit” into ordered chunks and runs group scheduling on them.

APS project is mainly written in C/C++ and it is almost self-contained as the only dependencies are Flex and Bison which are used for lexing and parsing respectively. APS project source code is publicly available and “may be freely used if the user does not claim authorship”. APS takes an as input .aps files which combine the syntax and semantics of attribute grammar and outputs code in Scala that can be used in combination with basic.handcode.scala (handwritten glue code) to evaluate attributes of any derivation of a CFG grammar. APS project is structured as follows:

- **aps-type**: this module’s responsibility is to type-check the semantics used to define the attributes, and since APS is a typed attribute grammar system, these static types assist the user with catching unwanted errors statically.
• **aps-dnc**: this module’s responsibility is to analyze the AST (Abstract Syntax Tree) of the provided APS program file, create attribute occurrences, record the dependencies, and construct a summary graph and augmented dependency graph.

• **aps-fiber**: this module may be used by **aps-dnc** to instantiate fiber attributes and run fiber construction and apply fiber approximation if attribute grammar includes fibers (i.e. its RAG or CRAG).

• **aps-cycle**: this module can be used to run UP-DOWN fiber cycle breaking if invoked. This module was not used by the new chunk scheduler since it was no longer necessary as fiber attribute dependencies are monotone and RAG can also be handled by the new CRAG static scheduler.

• **aps-oag**: old pure greedy scheduler that does not support circular (remote) attribute grammars. This module was replaced with **aps-schedule** which has the same interface but can handle circular remote attribute grammars.

• **static-impl**: old code generation module that can take a conditional total order and output visit sequences and its evaluator in Scala. The generated Scala code works alongside **basic.handcode.scala** to evaluate attributes given a derivation. This module was later replaced with **static-scc-impl** which has the same interface but takes a conditional total order, visit markers, and circular flags indicating whether visits are circular or not and generates a visit sequence evaluator in Scala.

### 5.1 Prerequisite Changes

One important thing we did early on long before even starting to design the new static scheduler was that we hand-wrote a program in Scala that would have been code generated by APS to evaluate circular remote attribute grammar using visit sequence. This helped us to determine the road map for the necessary changes that had to be done before we could even
start working on the static scheduler. This section describes the preliminary non-breaking changes done to APS including the canonical types, canonical signatures, UP-DOWN fiber cycle breaking, and design of fixed-point loops.

### 5.1.1 Overview of Canonical Types and Signatures

APS can generate Scala code for either dynamic or static evaluation, but it was created mainly as a proof of concept. It was quickly observed the precursor to adding supports to loops was to fix the code generation to support lattices or any module that extends its own argument. This resulted in the introduction of canonical types and canonical signatures into APS and this along with other necessary changes in the code generation module will be discussed later in the following.

APS provides the ability to define typed attributes that are associated with particular nodes. The programmer specifies the values by expressing an attribute as a function of other values, including other attributes. The APS compiler schedules the evaluation of the attributes so that no attribute is used before it is defined. Automatic scheduling is one of the main benefits of attribute-grammar-based systems such as APS. However, the APS-type system is unique in the sense that a module can extend its own arguments. The APS code generation module was written as a proof of concept and it can generate syntactically sound code in Scala. However, this type-system feature of APS is not similar to any feature that is supported out-of-the-box in the code generation target language like Scala. To fix this issue in the generated Scala code, this research used Scala traits. Note that a Scala class can implement many traits but it can extend only one class (similar to Java). In Figure 5.1, OrLattice and AndLattice are examples of such type declarations. Notice that type declaration is set to the result of MAKE_LATTICE module instantiation and MAKE_LATTICE module extends its argument L.

The precursor to coming up with Scala traits in the code generation phase was first figuring out that given a OrLattice or AndLattice what type declarations it consists of.
module MAKE_LATTICE[L :: BASIC[]](default : L;
  comparef,
  compare_equalf : function(_,_:L) : Boolean;
  joinf,
  meetf : function (_,_:L) : L)
  :: COMBINABLE[], LATTICE[] extends L
begin
end;

type OrLattice := MAKE_LATTICE[Boolean](false,cand,implies,(or),(and));
type AndLattice := MAKE_LATTICE[Boolean](true,andc,revimplies,(and),(or));

Figure 5.1: OrLattice and AndLattice type declarations
This APS code is included in `basic.aps` which defines the built-in types and modules.
Notice MAKE_LATTICE is extending its own generic parameter L which means OrLattice and AndLattice are Boolean as well.

However, APS did not have any built-in functionality to resolve this. To fix this issue, this research used the concept of canonical types and canonical signatures. Canonical in this context means that two things are equal if they have the same reference equality. Implementing this missing feature to APS code generation was crucial because lattices in APS are modules that extend their own argument and if APS does not have this feature, then it cannot be shown whether the generated Scala visit sequence evaluator code actually runs successfully. Thereafter, the end goal of this effort would be that each type instance declaration in APS would have an associated set of canonical signatures that will then end up as Scala traits. The following tries to describe the type of problem using an example. Figure 5.2 is a simple APS code that represents an integer max lattice that starts from 0 as its bottom value (⊥) and it is being used as an integer for basic arithmetic addition. That is because IntegerMaxLattice is technically an Integer as well according to Figure 5.1. However, the code generation module was unable to transform this to a semantically correct Scala code that compiles.

To understand how it is working, one needs to understand the signatures of MAX_LATTICE and MAKE_LATTICE. Figure 5.3 is the corresponding APS code. Notice that MAX_LATTICE is a module phylum such that its generic parameter should extend ORDERED[] module and it
module TEST[] begin
  type IntegerMaxLattice := MAX_LATTICE[Integer](0);
  1 : IntegerMaxLattice := 1 + 3;
end;

Figure 5.2: Simple APS code showing the use of integer max lattice behaving as an integer
is exporting (or its result type) is a type declaration of MAKE_LATTICE.

module MAX_LATTICE[TO :: ORDERED[]](min_element : TO)
  MaxLattice := MAKE_LATTICE[TO](min_element,(<),(<=),max,min);

Figure 5.3: APS code defining MAX_LATTICE module
This APS code is included in basic.aps which defines the built-in types and modules.

MAKE_LATTICE is a module phylum where its generic parameter extends BASIC[], and also
it extends COMBINABLE[] and LATTICE[] as well as its generic parameter indirectly through
resulting type declaration. Because this does not create a new type and only extends its
generic parameter, it is considered a non-generating module. Going back to the original
problem, one can see that MAX_LATTICE of an Integer is in fact an integer with additional
functionalities so it is valid to treat two integer MAX_LATTICE as integers and add them
together.

An important distinction that needs further clarification is the difference between re-
strictive and additive signatures. It was just concluded that MAX_LATTICE[Integer] is at
least an Integer. However, the question is whether parents of modules are also included
in the signature or only the result of the module is included. Figure 5.4 is an example
type alias in APS and it helps to better understand the difference. In a restrictive man-
ner, the signatures of T are only NUMERIC[] but in an additive manner, the type of T is
NUMERIC[], MAX_LATTICE[INTEGER], MAKE_LATTICE[INTEGER], and INTEGER[] and their re-
spective parents of each one. Going with restrictive or additive routes will have a major
effect on implementation and the resulting signature set.

In APS, Declaration, Expression, Type, and Signature are all AST nodes. Resolving
these signatures will result in creating AST nodes outside of the parser which is an anti-
type T::NUMERIC[ ] := IntegerMaxLattice;

Figure 5.4: Type alias example in APS
Aliasing type declaration IntegerMaxLattice as T and restricting it to be at least NUMERIC.

pattern in compilers. The solution is to develop an alternative structure to resolve these signatures such that it is relatively fast and efficient.

Hashconsing is a technique to store objects such that it attempts to reuse “cons” cells that have been constructed before, thus avoiding the performance penalty of memory allocation. This means that if two things are equal then only one instance of them exists in memory. This also enables us to use the reference equality operator (==) instead of the memberwise equality test. The implementation used in this thesis also used hashtables to store the cons along with the double hashing and twin-prime technique for resolving the collisions during open addressing. During the rest of this paper, the use of the word “canonical” refers to a hash-consed structure.

5.1.2 Canonical Types

APS has the following variations of type for Declaration: type_use, type_inst, no_type, remote_type, private_type and function_type. Canonical types have the following three variations: canonical_use, canonicalQualifier, and canonical_function_type. Canonical types simplify remote_type, type_use and function_type and map them to one of the possible canonical type variations. In short, the most important thing that canonical types hope to accomplish is to eliminate the APS’s ambiguously defined type_use and this will help us to find the type Declaration where the type was actually defined. The following are the variations of canonical types in detail:

- **canonical_use** is the first and simplest variation of the canonical type that only wraps the type declaration. So given an AST Type we can map to its type declaration.

- **canonicalQualifier** is a variation of the canonical type that has two fields. The source
and decl. The source is another canonical type and decl is a Declaration AST node. This abstracts out the qual_use AST node.

- The last variation is a canonical_function_type which basically abstracts out the function_type AST node in APS.

Figure 5.5 is the struct definition of canonical types. Note that the key property will help us determine the variation of the canonical type struct as they all share this property in their struct layout.

```c
struct canonicalTypeBase
{
  int key;
};
typedef struct canonicalTypeBase CanonicalType;

struct Canonical_use
{
  int key; /* KEY_CANONICAL_USE */
  Declaration decl;
};

struct Canonical_qual_type
{
  int key; /* KEY_CANONICAL_QUAL */
  Declaration decl;
  CanonicalType *source;
};

struct Canonical_function_type
{
  int key; /* KEY_CANONICAL_FUNC */
  int num_formals;
  CanonicalType *return_type;
  CanonicalType *param_types[];
};
```

Figure 5.5: C struct definition of canonical types
This code is included in canonical-type.h. Notice all the structs share the same layout starting with int key.
Subsequently, we created a recursive utility function that takes a type AST node and returns a canonical type. Note that this function only accepts `remote_type`, `type_use` and `function_type` because there is no way to find the declaration of the type for the other variation of type AST (e.g. `private_type`, `no_type`). Note that the recursion stops when the utility function encounters a type declaration and it does not go any further than that. For example, the canonical type of `IntegerMaxLattice` is a `canonical_use` with the declaration being `IntegerMaxLattice`. In summary, given:

- `type_use` inspects the base type variation of the use until it reaches a type declaration
- `remote_type` returns a `canonical_type` of base type of remote
- `function_type` returns the `canonical_function_type`

We also created another utility function called `canonical_type_base_type` that takes a canonical type and returns a canonical base type. In short, this function goes one level deeper into type declaration and if the type declaration module is “generating” then it stops. Otherwise, it uses the return declaration of the module and returns the `canonical_type` of that declaration. For example, the canonical type of `IntegerMaxLattice` is itself because its a type declaration but the canonical base type of `IntegerMaxLattice` is a `canonical_use` with the declaration being `Integer`. Because `MAX_LATTICE` is a “non-generating” module and this research used the result of that which is `MAKE_LATTICE` type declaration. That itself is also a “non-generating” module which extends its formal and after the substitution, it results in `Integer`.

One of the important internal helper functions that deserve to be discussed which makes it possible to run the above functions is `canonical_type_join`. It takes two canonical types and then tries to use a case-by-case analysis to join them together and simplify them. One unavoidable operation that is needed due to the left recursive nature of `canonical_qual` is left refactoring the canonical types and this function accomplishes it. Basically upon joining two qual canonical types if there is no way to reduce the two canonical types, one needs to
create a `canonical_qual` that recursively pushes the second one to the left. This extra left recursion is accomplished by `canonical_type_join` helper function.

### 5.1.3 Canonical Signature

Canonical signature abstracts out the signature instance AST node in APS. However, it only uses canonical types, unlike signature instance which uses the type AST node, canonical signature applies `canonical_type_base_type` for each actual signature instance. Figure 5.6 is the definition of the struct.

```c
struct CanonicalSignature_type {
    bool is_input;
    bool is_var;
    Declaration source_class;
    int num_actuals;
    CanonicalType *actuals[];
};

typedef struct CanonicalSignature_type CanonicalSignature;
```

Figure 5.6: C definition of canonical signature
This code is included in `canonical-signature.h`

### 5.1.4 Canonical Signature Set

A canonical signature set is an ordered set of canonical signatures. Adding the order to the set will ensure that two sets are equal if they have the same canonical values and enable us to use the reference equality operator (==). Figure 5.6 is the definition of the struct and Figure 5.7 is a function that is used to generate a set of canonical signatures and these are added during the code generation as a scala trait for each APS type declaration.
CanonicalSignatureSet * infer_canonical_signatures(CanonicalType * ctype) {
    bool any_change = true;
    CanonicalSignatureSet result = empty_canonical_signature_set;
    do {
        result = union_canonical_signature_set(
            result,
            find_canonical_signature(ctype));
        ctype_t = canonica_type_base_type(ctype);
        if (ctype == ctype_t)
            { any_change = false; }
    } while (any_change);

    return result;
}

Figure 5.7: C function that returns signature set by iteratively calling base type
This code is included in canonical-signature.c.
Notice the iterative loop stops until the fixed point is reached and there is no new
canonical signature to collect.
5.1.5 Code Generation Modification

The last piece of this effort is code generation where it needs to use the inferred canonical signatures set collected for each type declaration and translate it to Scala code that is semantically correct. Basically, using “trait” as a means to generate the additional signatures, because in Scala, one cannot define a class such that it extends its own formal generic parameters and also in Scala similar to Java, a class can extend at most one other class. Our approach is to code generate the services in the canonical signature set of the canonical base type of the type declaration if the module is “non-generating”. Note that adding all the signatures of a module will not cause any semantic error, but they may be unnecessary because some functionalities that are added to a module as traits may be unused throughout the generated program. Figure 5.8 is the resulting generated code. Notice the four traits added to the M_MAX_LATTICE type declaration and how the implemented methods of the trait use the t_Integer parameter’s methods instead.

```scala
val t_IntegerMaxLattice = new M_MAX_LATTICE[T_Integer]
  ("IntegerMaxLattice",t_Integer,0)
  with C_TYPE[T_Integer]
  with C_PRINTABLE[T_Integer]
  with C_ORDERED[T_Integer]
  with C_NUMERIC[T_Integer] {
    override val v_string = t_Integer.v_string;
    override val v_zero = t_Integer.v_zero;
    override val v_one = t_Integer.v_one;
    override val v_plus = t_Integer.v_plus;
    override val v_minus = t_Integer.v_minus;
    override val v_times = t_Integer.v_times;
    override val v_divide = t_Integer.v_divide;
    override val v_unary_plus = t_Integer.v_unary_plus;
    override val v_unary_minus = t_Integer.v_unary_minus;
    override val v_unary_times = t_Integer.v_unary_times;
    override val v_unary_divide = t_Integer.v_unary_divide;
  }
```

Figure 5.8: Scala code-gen result of integer max lattice type declaration
Notice the signature sets are code generated as Scala traits and the implementation of the trait methods comes from the t_Integer parameter.
5.1.6 Fiber Cycle Breaking

The final prerequisite to allow the scheduling of circular remote attribute grammar in APS was modifying the spurious fiber cycle breaking algorithm. The old algorithm introduced two artificial (dummy) new attributes (i.e. UP and DOWN) and then tried connecting all attribute instances to those two, whereas the modified algorithm just treats all attributes as either UP or DOWN by looking at their fiber direction. The fiber cycle-breaking algorithm should not be applied when the cycle does not just involve fibers. The reason for it is that the non-fiber dependencies actually carry values and doing so will break the direct dependencies and result in a schedule that is not sound. The re-written fiber cycle breaking module has the same signature as the older implementation so it was a drop-in replacement change. The algorithm is described in Algorithm 5. However, all this effort turned out to be fruitless because fiber dependencies are monotone and the new SCC scheduler can handle fiber cycles as well so there is no need to run UP-DOWN to break spurious fiber cycles. This is good news because as always the less complicated the code, the better.

5.2 Loops in Code Generation

The final modification was modifying the visit sequence code generation to add fixed point loops around attributes of the circular chunks that are scheduled in the non-circular parent phase. Note that each phylum phase is either circular or non-circular and when child visit is invoked visit sequence evaluator knows the phase and phylum of both parent and child so it just uses Table 4.1 to add the necessary do-while loop surrounding the child attributes. Similarly, code generation includes the assertion to throw an error statically during the code generation (not runtime) if there is a non-circular visit happening inside of a circular parent visit. These loops repeat the evaluation of attributes inside the circular chunk as many times as needed while the value of any attribute instances in the cycle has changed and stops if there is no change. Similarly, to prevent running non-circular visits inside circular visits
more than once, the concept of guards was introduced to prevent additional visits and throw errors statically (i.e. not during the runtime evaluation).

There were two challenges in the code generation. Fiber attributes are not actually included in the code generation as they are merely used to guide the scheduler. The static schedule is just the head of the CTO node linked-list. Note that the CTO node can have branches for the positive and negative of conditional statements so it’s possible to have a fixed point loop with an empty body because it contains all fiber attributes. One solution is to follow the linked-list and see if there is any non-fiber attribute and in that case, go back and code generate. But a better approach is to dump the head of the fixed-point loop in a buffer, if by the end of the linked-list there is no non-fiber attribute then empty the buffer because the body of the loop would be empty. The second complexity in the implementation is there may be more than one circular chunk inside the non-circular parent phase and in that case, they require the previously fixed-point loop to end before a new one may begin.

5.3 SCC chunk Scheduler

One fundamental issue with the original implementation was the static schedule code generation module had to guess when the child visit was happening and when the parent visit ended. This did not work well in practice and also this information should have been passed down to code generation from the static scheduler to prevent any guessing. So the new static scheduler now also includes visit markers and passes them down to code generation. This would help simplify code generation and any potential ambiguity is resolved thanks to the visit markers because it was hard for code generation to pull out visits. The following is the basic overview of how the old greedy static scheduler algorithm in APS generally works:

The scheduler starts from parent-inherited attribute instances in the first phase and schedules those. Then it tries to find the next set of attributes belonging to the same group or \((ph, ch)\) that are ready to be scheduled, that is their dependencies have been already
// Boilerplate code to support static circular evaluation
// by building upon circular evaluation structure
var changed: Boolean = false;

  override def set(newValue : ValueType): Unit = {
    val prevValue = value;
    super.set(newValue);
    changed |= prevValue != value;
  }

  override def check(newValue : ValueType): Unit = {
    if (value != null) {
      if (!lattice.v_equal(value, newValue)) {
        if (!lattice.v_compare(value, newValue)) {
          throw new Evaluation.CyclicAttributeException("non-monotonic "+ name);
        }
      }
    }
  }

  // Do not treat multiple assignments of attributes as errors
  checkForLateUpdate = false;
}

// Example of how child visits can be wrapped inside of the do-while loop
// Backup previous (global) changed value and run the code block at least once
val prevChanged_2_0 = changed;
do {
  changed = false;
  visit_4_2(v_prods);
} while (changed);
// Restore the changed value to ensure the fixed-point loop does not interfere
// with potential other fixed-point loops somewhere higher in the tree
changed = prevChanged_2_0;

Figure 5.9: do-while loop in generated Scala code
This template Scala code for fixed-point loop is included in dump-scala.cc.
Notice how the changed boolean variable is backed up before the circular chunk and restored afterward.
scheduled. Upon starting to schedule child synthesized attributes or finishing scheduling of child inherited attributes it will add a visit marker and then continue with the schedule. Locals may come in any order and our code schedules them when they are ready to be scheduled. Then upon scheduling parent synthesized attributes it will add a visit marker to indicate the end of the parent visit. This process repeats for the next parent phase if any. After the static scheduling is finished then new function takes over to validate the sanity of the schedule. More specifically ensuring that child visits appear in the order. When validation is complete and no error has been found with the static schedule then code generation takes over to dump the visits.

5.3.1 Intrinsic Complexity of Static Scheduler

It can be observed that statically scheduling any attribute grammar is inherently not straightforward, NP-complete to be specific. The following are the variables that directly impact the complexity of the problem. Any conditional attribute will result in a branch (either positive or negative) in the schedule, and each attribute instance has a condition associated with it which means the condition of the branch and the condition of the attribute should be compatible for the attribute to be scheduled in a particular branch. This means both the order of attributes and count of attributes may be different in positive or negative of the branch. The order may be different because merging the condition up to the attribute instance and the condition of the attribute instance itself may be impossible under one branch of conditionals. The count may be different because it is possible that merging the conditions is impossible and in this case, we do not include it in the conditional total order because it would be unreachable. To achieve this outcome a greedy scheduler may start with no positive or negative condition and tries to find the next available attribute that is ready to be scheduled.

The next complexity is specific to remote-attribute-grammar because of the fibers and the potential of having fiber cycles. In this case, the cycle is detected by using the union-
find of attribute occurrences in the augmented dependency graph. Then each phylum in the augmented dependency graph is investigated to see if there is an edge from the attribute occurrence of the phylum to itself. If there is then combine the strongly connected component of the phylum attribute occurrence with the strongly connected component of the attribute occurrence in the augmented dependency graph. The result is a cycle that contains both phylum attribute occurrence and augmented dependency graph attribute occurrence. This distinction is important because fibers do not carry any value but cycles in CRAG may carry a value.

Another complexity is group scheduling which is not specific to any particular extension of a classical attribute grammar. The idea is attributes of a phylum are divided into one or more subsets of attributes or visits with the constraint of trying to put as many attributes that can be scheduled into a visit such that all attributes in the visit may appear in any order that is compatible with the total order of phylum attributes (i.e. protocol Π, see Definition 4.1.1). Then upon scheduling the augmented dependency graph, we look at the visits of the parent phylum and try to fit in as many children attributes as possible in each visit. The group scheduling says for each parent visit, at the beginning of the visit should be parent-inherited attributes, and at the end of the visit should be parent-synthesized attributes. Then just before each child’s visit’s recursive call should be all of its inherited attributes for the phase of the child visit and right after that should be the child’s synthesized attributes of the same phase. Lastly, parent and child visits have to appear in consecutive order. For example, the scheduler starts with the first parent phase and after it finishes it goes to the next parent phase, and a similar idea applies to the child visits. Locals which include conditionals and locally defined attributes may appear in any order as long as it is compatible with the total order of attributes.

The final complexity specific to circular remote attribute grammar is the existence of cycles that are not just fiber cycles. These are the cycles that actually carry a value and cannot be broken because if cycles are broken then we would not know during the code
5.3.2 Failed Attempts

The following are some of the attempts as we tried to schedule circular remote attribute grammar. It is important to revisit them and understand why they failed so we do not repeat the same mistakes.

The first attempt was trying to modify the UP-DOWN fiber cycle breaking logic to use DOWN-UP instead. This means that preserving DOWN to UP edges if accumulated dependency of all augmented dependency graphs indicates the existence of a cycle that is not just a fiber cycle (i.e. a cycle is carrying a value). However, this attempt was not successful because it unavoidably tries to cut edges that are maybe direct which causes the code generated from the static schedule to access values out of scope. We tried going further to make sure direct edges were not removed by cycle breaking algorithm but the resulting dependency graph still had cycles in it which means applying the greedy static scheduler would not be possible.

The next attempt was to schedule within cycles that are found during the UP-DOWN logic’s union-find algorithm. However, these cycles combine the phylum attribute dependencies with augmented graph dependencies. This ultimately failed because in some cases the cycle contained all attribute instances in the augmented dependency graph but that was in fact false. Another failed attempt was finding all strongly connected components but using the fact that attributes belong to one SCC if they are in a cycle and schedule within the SCCs. But this attempt was not successful because first, the implementation became complex and also due to the nature of the greedy scheduler as it does not see the bigger
picture and it may start working on an SCC too early thus breaking the group scheduling rules. The follow-up attempt was topological sorting inside the SCCs and the SCCs among themselves. This ultimately failed as it turns out topological sorting doesn’t really impact the greedy scheduler in a meaningful way because the greedy scheduler itself uses topological ordering.

5.3.3 SCC Scheduling Proposal

This proposed scheduling approach builds up on the last failed attempts by using the concept of the “chunk”. More specifically, four types of chunks: locals, parent inherited, parent synthesized, and child visit. Each local attribute belongs to its own chunk so a local chunk always contains a single local attribute. Parent-inherited and parent-synthesized attribute chunks have a phase number to distinguish parent visit numbers and each of them contains a set of parent-inherited or parent-synthesized attributes for a particular phase. Lastly, visit chunks contain a set of child-inherited and synthesized attributes for a particular phase of a child. More specifically, visit chunks have a phase number and a child number to differentiate the visit and the child they belong to. Chunk scheduling also required additional summary graph scheduling changes. More specifically, the new summary graph scheduler has to flip-flop between greedy scheduling as many non-circular phylum attribute instances as possible if any, then greedy scheduling as many circular phylum attribute instances if any belong to the same cycle, and then repeat until there is no more phylum attribute instance to schedule. Also, we require an empty (dummy) phase between two consecutive phases to make sure the cycles do not get merged together during the evaluation phase, and there exists a non-circular phase at the start and at the end.

The next step is to create chunks and put attribute instances into chunks if any regardless of whether there is an attribute belonging to it or not. Note that when scheduling the phylum graph, we may have created an empty non-circular phase between two circular phases and also at the start or end. This means there can be empty phases such that no attribute
instance belongs to them. For instance, if a particular child has $n$ phases then there should be $n$ visit chunks of phases $1 \ldots n$ even though some may be empty (e.g. between two circular phases or after a final circular phase).

The next step is creating a chunk dependency graph and reflecting the augmented dependency graph in a chunk graph by making sure that if two attribute occurrences in the augmented dependency graph have an edge between them then there is an edge between the corresponding chunks they belong to. Thereafter, we put *guiding dependencies* between chunks because as already mentioned, some chunks can be empty and have no attribute occurrence and this means they do not have any edges coming to and from them so we need guiding dependencies to make sure the chunks appear in the right order. The guiding dependency is added between the visit chunk of the same child to make sure child visits appear in consecutive order, and also between inherited and synthesized parent attributes to make sure for any parent phase, the inherited parent attribute comes first and then the parent synthesized attribute. Lastly, guiding edges are added between the synthesized parent attribute chunk and the inherited parent attribute chunk for the next phase to make sure parent visits appear sequentially. Finally, the SCC component of chunks is calculated using a chunk dependency graph that was just created and it is fed to the greedy scheduler.

The starting point is the SCC component scheduler subroutine which is initially fed a SCC component containing a chunk of parent inherited attributes for phase 1. This finds an SCC component of chunks to schedule such that all its chunks are ready to go meaning that there is no dependency from chunks of other SCC components that are not already scheduled to chunks of the current component. Then SCC component is fed into the chunk scheduler that continues the group schedule by going through the chunks inside the SCC one by one but now using only direct dependency between chunks as a way to know if a chunk inside a component is ready to go, this because we are scheduling inside of the SCC component. This is important because if we use any dependency other than direct dependency (e.g. transitive dependency) to schedule chunks inside the SCC component then
scheduling would not be possible as by definition attributes inside the SCC component are in a cycle. More specifically, unlike any other type of dependency, direct dependency (i.e. value directly flowing between attributes) is not transitive. For example, direct dependency from $a \rightarrow b$ and $b \rightarrow c$ results in indirect transitive dependency from $a \rightarrow c$. This means the only way direct dependencies are recorded and added to the graph is when the top-level match in APS is analyzed and direct dependencies are recorded. Lastly, when there is no more chunk to schedule inside the SCC component, then the algorithm goes back to the SCC component scheduler and repeats the process again.

Let’s analyze Figure 5.11 (graph drawn in Figure 5.15) which is a top-level match in APS. The first step is to find all chunks and it’s done in Figure 5.12. Notice there are chunks that do not have any attribute instance in them (for example, parent inherited chunk for phase 2). This is because chunks are created automatically without checking whether there is an attribute belonging to them or not. The next step is in Figure 5.13 where we find the strongly connected components of chunks. Notice there are eight components and one of them has more than one chunk. This component contains a cycle involving parent attributes. The final step is finding the schedule and it is done in Figure 5.14 where we see the group scheduling technique has been followed.
match ?ss0:Stmts=xcons_stmts(?ss1:Stmts,?s:Stmt) begin
  ss1.stmts_assigned_in := ss0.stmts_assigned_in;
  s.stmt_assigned_in := ss1.stmts_assigned_out;
  ss0.stmts_assigned_out := s.stmt_assigned_out;
end;

Figure 5.11: xcons_stmts of nested-cycles in APS

Parent inherited for phase: 1
  ss0.G[Stmts]'shared_info <-1,-1>
Parent synthesized for phase: 1
Parent inherited for phase: 2
  ss0.stmts_assigned_in <-2,-1>
Parent synthesized for phase: 2
  ss0.stmts_assigned_out <+2,-1>
Parent inherited for phase: 3
Parent synthesized for phase: 3
  ss0.G[Stmts]'shared_info$all_names! <+3,-1>
  ss0.G[Stmts]'shared_info$msgs! <+3,-1>
Child 0 non-circular visit for phase: 1
  ss0.G[Stmts]'shared_info <-1,+0>
Child 0 circular visit for phase: 2
  ss1.stmts_assigned_out <+2,+0>
  ss1.stmts_assigned_in <-2,+0>
Child 0 non-circular visit for phase: 3
  ss1.G[Stmts]'shared_info$all_names! <+3,+0>
  ss1.G[Stmts]'shared_info$msgs! <+3,+0>
Child 1 non-circular visit for phase: 1
  s.G[Stmt]'shared_info <-1,+1>
Child 1 circular visit for phase: 2
  s.stmt_assigned_out <+2,+1>
  s.stmt_assigned_in <-2,+1>
Child 1 non-circular visit for phase: 3
  s.G[Stmt]'shared_info$all_names! <+3,+1>
  s.G[Stmt]'shared_info$msgs! <+3,+1>

Figure 5.12: List of chunks for xcons_stmts of nested-cycles in APS
=> component #0
  (0) Parent inherited for phase: 1
=> component #1
  (9) Child 1 non-circular visit for phase: 1
=> component #2
  (6) Child 0 non-circular visit for phase: 1
=> component #3
  (1) Parent synthesized for phase: 1
=> component #4
  (2) Parent inherited for phase: 2
  (3) Parent synthesized for phase: 2
  (7) Child 0 circular visit for phase: 2
  (10) Child 1 circular visit for phase: 2
=> component #5
  (11) Child 1 non-circular visit for phase: 3
=> component #6
  (8) Child 0 non-circular visit for phase: 3
=> component #7
  (4) Parent inherited for phase: 3
=> component #8
  (5) Parent synthesized for phase: 3

Figure 5.13: Components of chunks of xcons_stmts in nested-cycles
The number in parenthesis indicates the chunk number.
Schedule for xcons_stmts (2 children):

ss0.G[Stmts]'shared_info <-1,-1>
ss1.G[Stmts]'shared_info <-1,+0>
-> visit marker <+1,+0>
s.G[Stmt]'shared_info <-1,+1>
-> visit marker <+1,+1>
<- visit marker (1) <+1,-1>

ss0.stmts_assigned_in <-2,-1>
ss1.stmts_assigned_in <-2,+0>
-> visit marker <+2,+0>
ss1.stmts_assigned_out <+2,+0>
s.stmt_assigned_in <-2,+1>
-> visit marker <+2,+1>
s.stmt_assigned_out <+2,+1>
ss0.stmts_assigned_out <+2,-1>
<- visit marker <+2,-1>

-> visit marker <+3,+0>
ss1.G[Stmts]'shared_info$all_names! <+3,+0>
ss1.G[Stmts]'shared_info$msgs! <+3,+0>
-> visit marker <+3,+1>
s.G[Stmt]'shared_info$all_names! <+3,+1>
s.G[Stmt]'shared_info$msgs! <+3,+1>
ss0.G[Stmts]'shared_info$all_names! <+3,-1>
ss0.G[Stmts]'shared_info$msgs! <+3,-1>
<- visit marker <+3,-1>

Figure 5.14: Static schedule for xcons_stmts for $\texttt{nested-cycles}$
Figure 5.15: \textit{xconsStmts} in \texttt{nested-cycles}

5.3.4 Concession

One concession in the implementation was no support for conditional cycles. There are two situations that can arise when using conditions with circular attribute grammars. The first is conditions inside the cycle, which showcases the need to use direct edges for scheduling chunks because of potential scoping issues. Figure 5.16 is an example of such attribute grammar where a declarative \texttt{For}-loop was used and a locally scoped variable (\texttt{v}) is used to denote each iterable item. The second situation is conditional cycles where the condition sits outside controlling the cycles, this is where our APS implementation is not supported. Figure 5.17 is an example of such attribute grammar.
module CYCLE_SERIES[T :: var TINY[]] extends T begin
  type IntegerSet := SET[Integer];
  type IntegerSetLattice := UNION_LATTICE[Integer,IntegerSet];

  circular attribute Wood.ins : IntegerSetLattice;
  circular attribute Wood.out : IntegerSetLattice;
  attribute Root.answer : IntegerSet;

  pragma synthesized(answer, out);
  pragma inherited(ins);

  match ?r=root(?w) begin
    w.ins := {0};
    r.answer := w.out;
  end;

  match ?w=branch(?x,?y) begin
    c : IntegerSet := y.out;
    x.ins := y.out;
    y.ins := x.out \ { 0 };
    w.out := w.ins \^- c;
  end;

  match ?l=leaf(?x) begin
    circular collection c : IntegerSetLattice;
    c := l.ins;
    for v in l.ins begin
      if (v < x) then
        c := {v+1};
      endif;
    end;
    l.out := c;
  end;
end;

Figure 5.16: Circular ordered attribute grammar cycle-series
Demonstrating scoping requirement of statically scheduling cycle
module TINY_COAG[T : var TINY[]] extends T begin
    attribute Wood.ai1 : Integer;
    attribute Wood.ai2 : Integer;
    attribute Wood.as1 : Integer;
    attribute Wood.as2 : Integer;
    attribute Wood.bi : Integer;
    attribute Wood.bs : Integer;

    pragma inherited(ai1,ai2,bi);
    pragma synthesized(as1,as2,bs);

    match ?r0=root(?w) begin
        end;

    match ?l0=branch(?l1,?l2) begin
        if odd(l0.ai1) then
            l1.ai1 := l0.ai1;
            l1.ai2 := l1.as1;
            l0.as1 := l1.as2;
            l2.bi := l0.ai2;
            l0.as2 := l2.bs;
        else
            l2.bi := l0.ai1;
            l0.as1 := l2.bs;
            l1.ai1 := l0.ai2;
            l1.ai2 := l1.as1;
            l0.as2 := l1.as2;
        endif;
    end;

    match ?l0=leaf(?n) begin
        l0.as1 := l0.ai1;
        l0.as2 := l0.ai2;
    end;
end;

Figure 5.17: Conditional circular ordered attribute Grammar \texttt{tiny-coag}
Variant of \texttt{simple-coag} attribute grammar but using \texttt{tiny} instead of \texttt{simple} as AST
Figure 5.18: Dependency graph of leaf top-level match in cycle-series.
The edge from $v$ to $c$ is a simple dependency edge (non-monotone).

Figure 5.19: Dependency Graph for branch top-level match in tiny-coag.
The conditional attribute has been omitted to simplify the graph. The red edges are for positive and blue edges are for the negative branch of the conditional.
Chapter 6

Evaluation

One of the early end goals that were set during this research is to compare and contrast the implementation of the Follow implementation with the work of Magnusson in [21]. Moreover, compare the time it takes to find the schedule and compare the evaluation of the schedule and then try to create an apples-to-apples comparison. However, the implementation in Magnusson’s paper utilizes demand evaluation and this thesis uses a static schedule so a fair comparison of them may not be possible. For benchmarking purposes, we used a desktop workstation with i5-8500 CPU, 32GB of RAM, Ubuntu 22.04-LTS with Scala 2.13.10 using -Xmx16g JVM flag. That is 16GB JVM heap size to prevent evaluation running out of memory and repeated garbage collector calls to free the memory. We did not use any optional optimization flags.

Figure 6.1 is the benchmark comparison of evaluating nested-cycles on APS using demand and visit sequence given AST with different depths. Notice that as the depth of evenly growing AST increases, the time and space complexity improvement becomes more distinctive. The memory cannot get larger than 16GB, hence the repeated garbage collection calls and memory swaps are happening behind the scenes to prevent the running of memory failure. However, the demand evaluation finally crashes when evaluating attributes of 12-level deep randomly generated AST. The visit sequence evaluation is at most $\approx 30$ percent
faster compared to demand evaluation as the evaluator is following a pre-determined path. Similarly, the memory usage is reduced by at most \( \approx 60 \) percent which makes sense as the evaluator in the visit sequence is not finding the schedule during the evaluation. Even though we did not optimize the implementation in any way, it’s safe to conclude that visit sequence evaluation is substantially faster than demand evaluation for the same AST in both time and space complexity for obvious reasons.

<table>
<thead>
<tr>
<th>AST depth</th>
<th>Nodes</th>
<th>Visit sequence evaluation</th>
<th>Demand evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ET in sec.</td>
<td>Memory in MB</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0.009</td>
<td>2.971</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>0.003</td>
<td>2.921</td>
</tr>
<tr>
<td>3</td>
<td>111</td>
<td>0.005</td>
<td>3.646</td>
</tr>
<tr>
<td>4</td>
<td>354</td>
<td>0.019</td>
<td>7.872</td>
</tr>
<tr>
<td>5</td>
<td>1083</td>
<td>0.097</td>
<td>13.091</td>
</tr>
<tr>
<td>6</td>
<td>3270</td>
<td>0.044</td>
<td>46.332</td>
</tr>
<tr>
<td>7</td>
<td>9831</td>
<td>0.064</td>
<td>57.778</td>
</tr>
<tr>
<td>8</td>
<td>29514</td>
<td>0.501</td>
<td>155.338</td>
</tr>
<tr>
<td>9</td>
<td>88563</td>
<td>2.276</td>
<td>198.215</td>
</tr>
<tr>
<td>10</td>
<td>265710</td>
<td>7.439</td>
<td>940.484</td>
</tr>
<tr>
<td>11</td>
<td>797151</td>
<td>27.578</td>
<td>1368.750</td>
</tr>
<tr>
<td>12</td>
<td>2391474</td>
<td>128.611</td>
<td>3279.699</td>
</tr>
</tbody>
</table>

Figure 6.1: Benchmarks of running `nested-cycles` example in APS

The exponential nature of the demand evaluation becomes evident as number of the AST nodes gets larger.

Figure 6.4 is the benchmark comparison of evaluating `Follow` using both APS and JastAdd given a randomly generated context-free grammar. Remember that JastAdd is an attribute grammar system that supports circular remote attribute grammars. It is written in Java and uses demand evaluation, whereas APS is written in Scala and uses visit sequence evaluation. Note that for APS we used ScalaBison [3] which is a recursive ascent-descent whereas JastAdd uses JJTree/JavaCC for parsing. However, the time duration in Figure 6.4 is only for attribute evaluation and does not include the parsing step. The exponential na-
Figure 6.2: Graph of time duration in log scaling (see Figure 6.1)

Figure 6.3: Graph of memory usage in log scaling (see Figure 6.1)

Dynamic scheduling memory is flattened at the end because of JVM max heap size and it forces repeated garbage collection calls.
ture of the demand evaluation in JastAdd becomes evident as number of the AST nodes gets larger. Also, skipping the monotonicity validation checks in APS which happens when it sets the value of a circular attribute instance helps the time complexity by a lot. This is because each monotonicity check operation for a circular attribute instance value that is of type of a set takes \(O(n \times k)\) where \(n\) is the number of items in a set and \(k\) is the monotonicity check being applied to each element of the set. We can conclude that APS without a monotonicity check is faster than JastAdd but with a monotonicity check is much slower than JastAdd.

<table>
<thead>
<tr>
<th>Input size in MB.</th>
<th>Nodes</th>
<th>non-terminals</th>
<th>terminals</th>
<th>JastAdd</th>
<th>APS (without monotonicity) check</th>
<th>APS (with monotonicity) check</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>time in sec.</td>
<td>memory in MB.</td>
<td>time in sec.</td>
</tr>
<tr>
<td>2</td>
<td>17045</td>
<td>256</td>
<td>16789</td>
<td>0.8</td>
<td>78</td>
<td>0.6</td>
</tr>
<tr>
<td>5</td>
<td>67244</td>
<td>512</td>
<td>66732</td>
<td>2.8</td>
<td>137</td>
<td>2.3</td>
</tr>
<tr>
<td>11</td>
<td>265762</td>
<td>1024</td>
<td>264738</td>
<td>15</td>
<td>479</td>
<td>9.3</td>
</tr>
<tr>
<td>21</td>
<td>1024167</td>
<td>2048</td>
<td>1022119</td>
<td>127</td>
<td>2793</td>
<td>72</td>
</tr>
<tr>
<td>69</td>
<td>3596625</td>
<td>4096</td>
<td>3592529</td>
<td>1193</td>
<td>11073</td>
<td>246</td>
</tr>
<tr>
<td>289</td>
<td>9075268</td>
<td>8192</td>
<td>9067076</td>
<td>JVM crashed after 1670 seconds</td>
<td>861</td>
<td>14362</td>
</tr>
</tbody>
</table>

Figure 6.4: Benchmarks of running Follow example
Figure 6.5: Graph of time duration in log scaling (see Figure 6.4)

Figure 6.6: Graph of memory usage in log scaling (see Figure 6.4)
Chapter 7

Conclusion

This thesis overviewed related works including other attribute grammar systems and declarative tools specifically designed for program analysis, attribute grammars, and various extensions to the original Knuth paper such as remote and circular remote extensions. Thereafter, it discussed in detail subjects such as attribute evaluation methods such as demand evaluation and synth functions, and most importantly static schedule evaluation methods such as visit sequences. Finally, it discussed statically scheduling circular remote attribute grammars and its implementation into APS. More specifically, it discussed prerequisite changes to APS modules such as fiber cycle breaking, canonical types, and canonical signatures that made it possible for the static scheduler to be a drop-in replacement, as well as the new static scheduler and Scala code generation to support circular remote attribute grammars. Finally, it tried to compare the implementation in terms of run-time to similar attribute grammar systems such as JastAdd.

One concession in the implementation that was done in order to speed up the development and deliver an MVP implementation was no support for conditional cycles. In fact, there is an “if guard” in the code that blocks the static scheduler from going further if it finds that attribute grammar contains a conditional cycle. Supporting conditional cycles is just a matter of implementation complexity because the chunk dependency graph and strongly
connected graph component algorithm have to take conditions into account. Hence, this feature was left as future work.

Before starting to work on this research topic, we were not even sure statically scheduling circular remote attribute grammars was even possible and our many failed attempts are described in the implementation chapter. However, this thesis showed that statically scheduling circular remote attribute grammars is possible and can be done efficiently compared to demand evaluation, and to do that it built upon previous works of Kastens [17] for $l$-ordered attribute grammars and the fact that we do not need to create runtime dependency graph to evaluate certain attribute grammars, as well as Boyland [6, 5, 8] and Hedin [21] for the definitions of remote and circular remote attribute grammars respectively and fiber approximation which makes it possible to define the semantics of remote attribute grammars as classical attribute grammars with finite number of attribute instances.
• $X_i$: non-terminal used at index $i$ of production $p$
• $(X, i)$: instantiated node or a non-terminal expanded in derivation $i$
• $(X:j, i)$: instantiated node with occurrence identifier $j$
• $(X, i).a$: attribute instance
• $l$: local attribute
• $v_i$: attribute occurrence
• $r_i$: attribute grammar semantic rule
• $\hat{v}_i$: attribute instance
• $\hat{r}_i$: instantiated attribute grammar rule
• $\mathcal{D}$: CFG derivation
• $\hat{\mathcal{D}}$: modified derivation with index of transition step included
• $\mathcal{T}$: derivation tree
• $\mathcal{V}(p)$: visit sequence for a production $p$
• $SDG_X$: summary dependency graph for non-terminal $X$
• $DG_p$: dependency graph for production $p$
• $DG_p^*$: augmented Dependency graph for production $p$
• phylum: non-terminal tree node used in APS attribute system

Figure 7.1: Legend
Bibliography


CURRICULUM VITAE

Seyedamirhossein Hesamian

Education

• M.S., University of Wisconsin-Milwaukee, May 2017
  Major: Computer Science
  Thesis: Analysis of BCNS and Newhope Key-exchange Protocols
  Supervised by Dr. Guangwu Xu

• B.S. (with Honors), University of Wisconsin-Milwaukee, May 2015
  Major: Computer Science

Teaching Experience

• University of Wisconsin-Milwaukee
  – CS-315: Computer Organization and Assembly Language
  – CS-351: Data Structures and Algorithms
  – CS 655: Compiler Implementation Laboratory