On the Specification and Use of Recursive Adaptable Grammars

by Luke Bessant

Advised by:
Adrian Johnstone, Elizabeth Scott, Reuben Rowe

April 11, 2022

Abstract

John Shutt, in his 1993 Master's thesis, proposed a new Turing-powerful formalism for specifying formal languages and their semantics, which he called "Recursive Adaptable Grammars" (RAGs). The purpose of RAGs was to serve as an alternative to existing adaptable grammar formalisms, which would retain the relative simplicity of context-free grammars. To compliment his discussion on the recursive adaptable grammar formalism, Shutt outlined the need for further research into parsing techniques to implement his ideas. This dissertation is an investigation into the specification of recursive adaptable grammars as defined by Shutt, and introduces a technique for implementing tools to recognise strings belonging to the language defined by a recursive adaptable grammar.

This dissertation first gives a discussion of the formalism presented by Shutt, with analysis of its definitions from the perspective of the author. A portfolio of example RAG specifications is used to assist the discussion on the formalism, with justifications of the design decisions made in building each. Another contribution if this thesis are implementations of extensions to an existing tool for exercising recursive adaptable grammars. A subset of the example RAG specifications help to illustrate this implementation's features. Finally, to help build a case for the RAG formalism, a chapter of this dissertation is dedicated to larger examples which simulate a structural operational semantics system.

Contents

Chapter 1

Introduction

Standard approaches to defining programming languages involve the use of a context-free grammar to specify the syntax of the language, complemented with techniques for defining the semantics of the language on another layer. Such approaches have the disadvantage of being overly convoluted. To study a language defined in this way, one must make connections between the disjointed definitions to understand its inner-workings.

During the late 1960s and early 1970s, advances were made in developing formalisms which allow the specification of syntax and semantics in the same place. Most notably, the conception of attribute grammars by Knuth in 1968 [3], which allow us to communicate context-sensitive information throughout a grammar using attributes. One is able to specify semantic equations which apply to the context-free-style rules to manipulate these attributes. Other advances include adaptable grammars, providing a means for manipulating the form of rules within the grammar through the use of semantic actions which incorporate context-sensitivity.

In 1993, John Shutt proposed his own technique for specifying formal languages and their semantics - recursive adaptable grammars [1]. A key feature of RAGs is the combination of syntactic and semantic specification within a single formalism. Namely, the domain of expressions used for both are the same. The purpose of RAGs is to serve as an alternative to existing adaptable grammar formalisms. Ideally, RAGs would retain the relative simplicity of context-free grammar definitions whilst being Turing-powerful by their expressivity. The simplicity stems from a clean combination of meta-syntactic expressions and the associated semantics in the rewrite rules of a grammar, and the presence of a single derivation step relation which handles both syntactic and semantic reductions. Shutt also noted that a drawback of his formalism is in its difficulty to couple with existing parser technology. Since the conception of the formalism, no RAG-based parsing procedure was ever proposed as a follow-up. RAG-based parsing is inherently costly in its nature, as RAG specifications can be heavily non-deterministic and often relies on unification of semantic values synthesised by derivations. This thesis serves as an investigation into the practicality of such a procedure, which can support all of the features of the RAG formalism.

Our research continues the work first started by Shutt in his thesis. With the aim of promoting this powerful language definition technique, we hope to show that it can be used to specify real-world languages. As such, we have devoted a chapter to showing the implementation of common language features such as operations over the integers and Booleans, as well as `if` statements and the maintenance of an environment of name-value pairs. We present these features as a structural operational semantics, which we encode in RAG rewrite rules. We also propose a solution to the parsing problem in the shape of a tool for implementing the formalism. The solution takes a breadth-first approach to parsing where we maintain a set of potential derivations of a given input string by applying rules greedily. During this process, we incrementally prune the set of those elements which we know can never derive our string. As well as an introduction to our implementation, this dissertation contains a critique of the RAG formalism as defined by Shutt. We hope to first introduce the formalism to the reader before moving on to cover our work on realising it in our tool.

1.1 Contributions

In this thesis, we make the following contributions:

- An introduction to the recursive adaptable grammar formalism from our own perspective, to complement that given by Shutt.
- A portfolio of example grammar specifications for a selection of languages which incrementally involve the features of the RAG formalism. We hope that these display the power of the formalism.
- Presentation of the tool we have built to implement the formalism. We discuss the details of the implementation and give example derivations generated by the tool. We also provide a history of the development of the tool.
- A case study in the use of recursive adaptable grammars to implement a real-world language. We detail the specification of a structural operational semantics language for both small-step and big-step semantics using common programming language features.

1.2 Outline

This work begins with a discussion of the background theory in chapter 2; an introduction to historical techniques for language definition and a section detailing universal algebra, a basis for the RAG formalism. Chapter 3 then discusses work in our field of research which relates to the RAG formalism, such as examples of attribute grammars. An overview of the RAG formalism is given in chapter 4, as well as a set of example language specifications using RAGs. We introduce the algorithm we have developed for implementing the RAG formalism in chapter 5, which will take the reader through our implementation of the major RAG features mentioned in chapter 4. We will also provide in chapter 6 a history of the development of the implementation. Our case study of the use of the RAG formalism to specify a structural operational semantics language is given in chapter 7, including our conclusions from this work. We summarise the work carried out during this thesis project in chapter 8, the conclusion, and propose future work on the RAG formalism.

Chapter 2

Background Theory

This chapter includes a discussion of concepts used within this thesis in the subject of formal language theory. The topics covered should give the reader a base level of understanding of language specification systems, which is necessary to understand the motivation, use, and technical details of recursive adaptable grammars. We start by considering the fundamental aspects of any formal language, and move on to a discussion of techniques for defining the structure and meaning of such languages.

2.1 Syntax and Semantics

When building languages, we often begin by thinking about what its expressions should look like, and what their *meaning* should be. An expression such as `1 + 2` has an obvious meaning to us through life experience. When dealing with formal languages, without some specification of this string and its semantics, this is merely a sequence of three symbols from an alphabet with no meaningful relation to its natural language counterpart.

The concrete structure of symbols which make up a language is known as its *syntax*. This constitutes the set of all sentences in the language. Sentences are sequences of symbols from the usually finite alphabet belonging to a language. Syntax merely defines the arrangement of symbols which combine to build sentences in a language, and not their meaning.

As mentioned by Kenneth Slonneger in *Formal Syntax and Semantics of Programming Languages* [4], "Syntax must be specified prior to semantics since meaning can be given only to correctly formed expressions in a language". In other words, one should reason about what expressions should look like in a language before a full consideration of how their meaning should be computed. This "meaning" Slonneger refers to usually requires more complicated computation to derive than syntax. We often compute the semantics of a sentence by first computing the semantics of its subcomponents and using them to build a bigger picture.

Slonneger also states that "for programming languages, semantics describes the behaviour that a computer follows when executing a program in the language". This is the essence of the concept of semantics, where we want to communicate to a computer what it should do when it sees a phrase in our language. Of course this can only be done by specifying the behaviour in terms of systematic evaluation steps. Formalisms for doing so, such as attribute grammars, will be discussed in later sections. In general, a sentence can have multiple meanings which collectively form the semantics of that sentence.

2.2 Grammars

We have mentioned that we want to be able to specify infinite sets of syntactic values. We need some simple means of doing so, with which we can construct expressions that describe infinitely many syntactic terms. We can make use of the notion of a generative grammar to solve this problem. Following those laid out in *Formal Syntax and Semantics of Programming Languages* [4], and *Compilers: Principles, Techniques and Tools* [8], we use the below definition for grammars:

Definition 1. A *grammar* is a 4-tuple (Σ, N, P, S) such that Σ is a finite set of *terminal* symbols; N is a set of *non-terminal* symbols; P is a finite set of production rules; S is the *start symbol* such that $S \in N$.

The concept of a *non-terminal* introduced in Definition 1 refers to the stand-in symbols which we use to describe possibly infinite sets of syntactic values. In *Introduction to the Theory of Computation* [9], Sipser refers to these as variables, and they are also referred to as *syntactic variables* in *Compilers: Principles, Techniques & Tools* [8]. This terminology captures the concept that we intend these symbols to be substituted using the production rules of the grammar, ultimately resulting in a piece of syntax when no more remain. The more common term *non-terminals* is used in this thesis. The notion of symbols which describe a set of syntactic values is referred to in this thesis as *meta-syntax*. That is, a symbol which does not appear in the syntax of a language, but affects how the syntax is structured. The purpose of the single *start symbol* defined by a grammar is outlined in §2.2.1.

As noted in *Compilers: Principles, Techniques & Tools* [8], the *terminals* are the elementary symbols of the language defined by the grammar. In other words, they are the elements we see when we observe the set which defines a language - unlike the non-terminals.

To demonstrate the use of the set of production rules, we give an example grammar in Ex. 1. The structure of each individual rule takes the form $\alpha \rightarrow \beta$ where α is a non-terminal in N, and β is a string of terminals and non-terminals. In general, each non-terminal is associated with at least one such production rule. We define the language $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$ to illustrate the basic structure of a grammar.

Example 1. Consider the following grammar.

$$\begin{array}{lcl} S & \rightarrow & \texttt{a}\,S\,\texttt{b} \\ S & \rightarrow & \epsilon \end{array}$$

This follows the notation given in *Compilers: Principles, Techniques & Tools* [8], however other notations often employ ::= in place of $\rightarrow$, such as that used by Slonneger [4]. The grammar defined in Ex. 1 has $\{S\}$ as its set of non-terminals, and $\{\texttt{a}, \texttt{b}\}$ as its set of terminals. S, being the only non-terminal, is also the start symbol. Our grammar consists of two production rules. The symbol ϵ is a piece of meta-syntax, an implicit member of N, which represents the empty string.

The language defined by a grammar can be computed by taking its start non-terminal and considering every possible substitution of non-terminals using the rewrite rules. This concept is referred to as *generation*. The language generated by a non-terminal γ is all the syntactic values which can be reached by application of all possible rewrite rules starting from γ. Examples of such applications are given in §2.2.1. By the grammar defined in Ex. 1, S generates the set $\{\epsilon, \texttt{ab}, \texttt{aabb}, ...\}$. More concisely, the set $\{\texttt{a}^n\texttt{b}^n \mid n \geq 0\}$.

2.2.1 Derivations

Henceforth we keep to the following typographic conventions:

1. Given a grammar $G = (N, \Sigma, P, S)$, its four constituent parts are referred to as N_G, Σ_G, P_G and S_G.

2. Non-terminals are given as single upper-case letters such as A, B, C.

3. Terminals are given as single lower-case letters in typewriter font, such as $\texttt{a}, \texttt{b}, \texttt{c}$.

4. Greek letters such as $\alpha, \beta, \gamma, \delta$ are sequences of terminals and non-terminals

Now we introduce the concept of a *derivation*. The following definition closely follows that given in *The Theory of Parsing, Translation, and Compiling* [10].

Definition 2. For a grammar G, a non-terminal $A \in N_G$, and $\alpha, \beta, \gamma \in (N_G \cup \Sigma_G)^*$, a *rewrite step*, denoted by $\Rightarrow$, is a relation of the form $\alpha A \beta \Rightarrow \alpha\gamma\beta$ iff $(A \rightarrow \gamma) \in P_G$.

A sequence of rewrite steps which ends with a member of a language constitutes a derivation sequence for that member. As mentioned in *Compilers: Principles, Techniques & Tools* [8], the terminal strings that can be derived from the start symbol form the language defined by the grammar. As such, a proof that a syntactic value belongs to a language requires a derivation sequence which begins with the grammar's start symbol, and ends with that syntactic value.

We use the below definition, following that used in *The Theory of Parsing, Translation, and Compiling* [10], to introduce the concept of a sentential form.

Definition 3. Given a grammar G, we define a kind of string called a *sentential form* recursively as follows:

1. α is a sentential form
2. If $\beta A \gamma$ is a sentential form, and $(A \to \delta) \in P_G$, then $\beta\delta\gamma$ is also a sentential form.

Namely, a *sentential form* can be thought of as the result of a step in a derivation. As well as this, the start symbol is also a sentential form in a derivation. Each derivation step consitutes a transition from one sential form to another by rewriting a single non-terminal using a production rule. The following derivation sequence derives the string `aaabbb` from start symbol S using the grammar defined in Ex. 1.

$$\begin{aligned} S &\Rightarrow \texttt{a}\,S\,\texttt{b} \\ &\Rightarrow \texttt{a}\,\texttt{a}\,S\,\texttt{b}\,\texttt{b} \\ &\Rightarrow \texttt{a}\,\texttt{a}\,\texttt{a}\,S\,\texttt{b}\,\texttt{b}\,\texttt{b} \\ &\Rightarrow \texttt{a}\,\texttt{a}\,\texttt{a}\,\texttt{b}\,\texttt{b}\,\texttt{b} \end{aligned}$$

In this case, the first three steps use the rule $S \to \texttt{a}\,S\,\texttt{b}$, and the final step applies the rule $S \to \epsilon$. By this process we have derived the sentence `aaabbb`. Each step in this derivation constructs a distinct sentential form.

2.3 Classes of Formal Languages

Noam Chomsky in, *Three Models for the Description of Language* [15], defined four types of grammars - each of which generates a different class of languages. In ascending order, the power of each type of grammar decreases, but the means for computing their products and doing derivations with them becomes less complicated. More restrictions are added to the grammar types successively which allow us to more easily reason about them. The classes of languages form a hierarchy, with outer-classes being able to define languages belonging to its inner-classes. For example, type-1 grammars can define languages which can be defined by type-2 and type-3 grammars, but not those exclusive to type-0 grammars.

2.3.1 Type-0 Grammars

Type-0 grammars define a set of languages referred to as the *recursively enumerable* languages, which are the most powerful of the language types.

Each rule consists of a string of syntactic and meta-syntactic values on each side, the only restriction being that the left-hand side of the rule must be non-empty. For a type-0 grammar G, we have the rule form $\alpha \rightarrow \beta$ such that $\alpha, \beta \in (N_G \cup \Sigma_G)^*$. Any expression on the left-hand side of a rule can be replaced by any expression on the right-hand side.

2.3.2 Type-1 Grammars

With type-1 grammars, a restriction is added so that only one non-terminal on the left-hand side of a rule is replaced on the right-hand side of the rule. We can still have an expression of multiple terminal and non-terminal values on the left-hand side, so long as only one of the non-terminals is substituted on the right-hand side. Thus for a type-1 grammar G we define rules of the form $\gamma\,\alpha\,\delta \rightarrow \gamma\,\beta\,\delta$ with $\alpha \in N_G$, and $\beta, \gamma, \delta \in (N_G \cup \Sigma_G)^*$. The languages generated by type-1 grammars are known as the *context-sensitive* languages. This is because we can rewrite a non-terminal with respect to its *context* - the string of symbols which surround it.

2.3.3 Type-2 Grammars

To further simplify, the restriction that the left-hand side of a rewrite rule can only include a single non-terminal value is introduced. As such, the languages generated by type-2 grammars are known as *context-free*. In a sentential form, a non-terminal is rewritten without considering the syntactic and meta-syntactic values which surround it. For a context-free grammar G, the rules for G take the form $\alpha \rightarrow \beta$ such that $\alpha \in N_G$ and $\beta \in (N_G \cup \Sigma_G)^*$. The grammar defined in Ex. 1 is a context-free grammar, defining a context-free language. However by the language hierarchy, it can also be defined by use of a type-0 grammar (and consequently a type-1 grammar). Type-2 grammars are perhaps the most popular class of grammar when defining programming languages, as we have polynomial-time parsing algorithms for them.

2.3.4 Type-3 Grammars

Building on type-2 grammars, another restriction is introduced to produce type-3 grammars. This restriction is that the right-hand side of a rule can only be a single terminal value, or a terminal followed by a non-terminal. Thus the rules for a type-3 grammar G take the form $\alpha \rightarrow \mathtt{a}$ or $\alpha \rightarrow \mathtt{a}\,\beta$ with $\alpha, \beta \in N_G$ and $\mathtt{a} \in \Sigma_G$. The languages such grammars generate are known as the *right-regular* languages. Instead of the latter rule form, we can instead enforce the form $\alpha \rightarrow \beta\,\mathtt{a}$. This system would then generate the *left-regular* languages. We cannot, however, use both rule forms. These regular language classes are the simplest classes in the classical Chomsky hierarchy [15].

It is a key observation that the grammar types described in this section only allow us to define the *syntax* we associate with languages. Each of the meta-syntactic values, the non-terminals, are rewritable to sentences - strings of terminal symbols. However the semantics of the sentences produced are not taken into consideration, and we have no way to define them in the grammar rules. To be able to specify the meaning of strings in a language, we need a more powerful formalism.

2.4 Attribute Grammars

We can in fact extend the simple context-free grammar model with features which allow recognition of context-sensitive languages. The idea is that we write a set of context-free rules which we augment with values, rules and conditions which define the semantics of the rules. The grammar rules which govern the syntax of a language are accompanied by these semantic actions, which can be thought of as being in a different domain. In other words, we still have an underlying context-free grammar, but we decorate it with semantic equations. Such grammars are known as *attribute grammars*.

The concept can be illustrated using an example language which a context-free grammar cannot specify. The language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ for $n \geq 0$ is impossible to specify with a CFG. However we can specify the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^m$.

Example 2. Consider the following grammar.

$$\begin{array}{lcl} S & \rightarrow & A\,C \\ A & \rightarrow & \mathtt{a}\,A\,\mathtt{b} \\ A & \rightarrow & \lambda \\ C & \rightarrow & \mathtt{c}\,C \\ C & \rightarrow & \lambda \end{array}$$

The rules defined in Ex. 2 extend those defined in Ex. 1 by including new non-terminals A and C. The terminal symbols are $\mathtt{a}, \mathtt{b}$ and $\mathtt{c}$. The language defined by non-terminal A is $\mathtt{a}^n\mathtt{b}^n$ for $n \geq 0$. This sub-language is allowed by the fact that terminals `a` and `b` appear in the same, recursive rule. In the single rule for S, A is appended by a C, which is rewritable to zero or more `c`'s. This grammar does in fact generate all strings where the number of `a`'s, `b`'s and `c`'s is equal - but not *only* those strings. The following derivation uses our grammar to recognise the string `aabbcc`.

$$\begin{array}{ll} S & \Rightarrow A\,C \\ & \Rightarrow \mathtt{a}\,A\,\mathtt{b}\,C \\ & \Rightarrow \mathtt{a\,a}\,A\,\mathtt{b\,b}\,C \\ & \Rightarrow \mathtt{a\,a\,b\,b}\,C \\ & \Rightarrow \mathtt{a\,a\,b\,b\,c}\,C \\ & \Rightarrow \mathtt{a\,a\,b\,b\,c\,c}\,C \\ & \Rightarrow \mathtt{a\,a\,b\,b\,c\,c} \end{array}$$

A similar derivation can be made for any string in the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$, where we rewrite A to a string in $\mathtt{a}^n\mathtt{b}^n$, and then rewrite C to a matching number of c's. However, we are also able to recognise strings not in $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ such as aabb with the following derivation.

$$\begin{aligned} S &\Rightarrow A\,C \\ &\Rightarrow \mathtt{a}\,A\,\mathtt{b}\,C \\ &\Rightarrow \mathtt{a}\,\mathtt{a}\,A\,\mathtt{b}\,\mathtt{b}\,C \\ &\Rightarrow \mathtt{a}\,\mathtt{a}\,\mathtt{b}\,\mathtt{b}\,C \\ &\Rightarrow \mathtt{a}\,\mathtt{a}\,\mathtt{b}\,\mathtt{b} \end{aligned}$$

The result is that we have written a context-free grammar for the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^m$ for $n, m \geq 0$. The semantic rules which are introduced by attribute grammars allow us to solve this problem.

We now formally define the concept of an attribute grammar.

Definition 4. An *attribute grammar* is a context-free grammar (N, Σ, P, S) augmented with semantic actions and attributes. For each $A \in N$ there are two disjoint sets $I(A)$ and $S(A)$ such that $I(A) \cup S(A)$ is the finite set of attributes associated with non-terminal A. Each attribute is assigned to based on semantic actions associated with each $p \in P$.

We write a simple attribute grammar which, during a derivation, remembers how many a's, b's and c's are generated, by using synthesised attribute *count* occuring on A and C.

Example 3. Consider the following attribute grammar.

$$\begin{array}{lcll} S & \rightarrow & A\,C & S.Eq_{abc} = A_1.count == C_1.count; \\ A & \rightarrow & \mathtt{a}\,A\,\mathtt{b} & A.count = A_1.count + 1; \\ A & \rightarrow & \lambda & A.count = 0; \\ C & \rightarrow & \mathtt{c}\,C & C.count = C_1.count + 1; \\ C & \rightarrow & \lambda & C.count == 0; \end{array}$$

In Ex. 3, each rule from Ex. 2 has been augmented with semantic actions. Note that references to a non-terminal X in the semantic actions denote the X on the left-hand side of the rule. Each reference to a non-terminal on the right-hand side of a rule is enumerated using a subscript. Consider the rules for non-terminal A. Each rule rewriting to λ assigns 0 to an attribute called *count*, as no symbols are generated. The rule rewriting to a A b sets the *count* attribute of the left-hand side A to the value of the sub-A's *count* attribute, plus one. Thus the *count* attribute of an A is exactly the number of a's and b's it generates. The first rule for C works similarly. We also have the attribute Eq_{abc} occuring on non-terminal S. This is an attribute of type Boolean, which is *true* when the number of a's, b's and c's is equal. In a parser, this could be used to cull derivations which do not satisfy this condition.

The conditions in the rules for C using attribute *count* have enforced that the derivation produces two $\texttt{c}$'s. A derivation of some string not in the language $\texttt{a}^n\texttt{b}^n\texttt{c}^n$ would result in false conditions, and would therefore not be a valid derivation using the attribute grammar. In this way, the semantic actions as well as the syntactic rules determine the language a grammar defines. The attribute grammar defined in Ex. 3 still specifies the language $\texttt{a}^n\texttt{b}^n\texttt{c}^m$ for $n, m \geq 0$. But the extra information we have computed as attribute Eq_{abc} can, in a parser, help to enforce that the valid strings are only those where $\texttt{a}^n\texttt{b}^n\texttt{c}^n$ for $n \geq 0$.

The imporance of *inherited* and *synthesised* attributes is key in the field of formal languages, and appears in other formalisms such as Recursive Adaptable Grammars, the topic of this thesis. Practically, the format of semantic actions differs with varying implementations. However, in each case we can specify semantic rules, conditions and assignments which allow us to introduce context-sensitivity. We can also specify the static and dynamic semantics of languages for popular examples such as building interpreters for simple programming languages.

2.5 Structural Operational Semantics

We have shown that by using attribute grammars we can specify the syntax and semantics of a language by joining CFG rules and semantic actions. Another useful concept is an *operational* description of the dynamic semantics of language. In programming language terms, what are the steps a program takes in its execution to produce a desired result? To answer this question, we can specify the *operational semantics* of a language.

Using the notions of transitions and configurations, we can describe the behaviour of a program in terms of individual computation steps. By configurations, we refer to groups of semantic entities, such as syntactic terms or environments, which together represent state. Transitions occur between these states, making modifications to semantic entities. Such steps are transitions between the states of a program in execution. In general we start from a state corresponding to a piece of syntax, and transition to a final state where computation ends, known as a *terminal state*. Usually this terminal state is a value of some sort, such as an integer. The structural operational semantics of a program defines a *transition system*, which can be thought of as a directed graph of configurations. The following definition of a transition system is given by H. Huttel in *Transitions and Trees* [5].

Definition 5. A *transition system* is a triple $(\Gamma, \rightarrow, T)$ where Γ is a set of configurations, $\rightarrow$ is the *transition relation* on $(\Gamma \times \Gamma)$, and $T \subseteq \Gamma$ is a set of terminal configurations.

The terminal configurations cannot transition to any other configuration. In effect, the terminal configurations are normal forms. Thus a term is either normal, or can have a transition applied to it. Pairs in the transition relation $\rightarrow$ are defined in a structural operational semantics specification as a finite set of inference rules. All inference rules in this thesis take the following form.

$$\frac{C_1\, C_2 \dots C_n}{\langle \alpha_1, \alpha_2, ..., \alpha_n \rangle \to \langle \beta_1, \beta_2, ..., \beta_n \rangle} \qquad (R_1)$$

R_1 defines a transition from configuration $\langle \alpha_1, \alpha_2, ..., \alpha_n \rangle$ to $\langle \beta_1, \beta_2, ..., \beta_n \rangle$, with side-conditions $C_1, C_2, ..., C_n$. The conditions above the line are known as the *premise* of the rule, whereas the relation below the line is known as the *conclusion*. In this thesis, the α_1 and β_1 are usually syntactic terms such as "`and(True, False)`", and the remaining $\alpha_2, ..., \alpha_n$ and $\beta_2, ..., \beta_n$ are further semantic entities which help to define an abstract state.

Such a rule should be read in a "round-the-clock" way. With configuration $\langle \alpha_1, \alpha_2, ..., \alpha_n \rangle$, when each side-condition C_i above the line is satisfied, we can reduce to to $\langle \beta_1, \beta_2, ..., \beta_n \rangle$.

The conditions $C_1, C_2, ...C_n$ are either *transitions* or *pattern-matches*. A transition $\gamma \to \gamma'$ says that the configuration γ must be able to reduce to the configuration γ'. For our purposes this is generally a transition on a sub-term, whose result we assign to a term variable. A pattern-match $\delta \triangleright \delta'$ tests that the configuration δ is able to match to configuration δ'. Matching a configuration with a closed term to one with an open term will usually result in a set of bindings over the term variables specified in the open term.

For the purpose of this thesis, the syntactic values in an operational semantics configurations are parenthesised terms. Such terms represent a tree structure corresponding to abstract syntax.

We will illustrate this concept by defining the operational semantics for simple Boolean expressions.

Example 4. Consider the following SOS rules.

$$\frac{}{\langle \texttt{or(True,True)} \rangle \to \langle \texttt{True} \rangle} \qquad (1)$$

$$\frac{}{\langle \texttt{or(True,False)} \rangle \to \langle \texttt{True} \rangle} \qquad (2)$$

$$\frac{}{\langle \texttt{or(False,True)} \rangle \to \langle \texttt{True} \rangle} \qquad (3)$$

$$\frac{}{\langle \texttt{or(False,False)} \rangle \to \langle \texttt{True} \rangle} \qquad (4)$$

$$\frac{\langle S_1 \rangle \to \langle V_1 \rangle \quad \langle S_2 \rangle \to \langle V_2 \rangle \quad \langle \texttt{or(}V_1, V_2\texttt{)} \rangle \to \langle V_3 \rangle}{\langle \texttt{or(}S_1\texttt{,}S_2\texttt{)} \rangle \to \langle V_3 \rangle} \qquad (5)$$

Rules 1 to 4 encode the truth table for the *OR* operation into a structural operational semantics specification. This is done by defining the resulting configuration in each possible case using literal Boolean values. Rules which have no premise, such as the first 4 rules, are known as *axioms*.

Rule 5 handles the case where we have an `or` term whose operands are not simply Boolean values. In the premise, we define V_1 and V_2 as the result of a transition on S_1 and S_2 respectively. The final condition in the premise defines V_3 as the result of a transition from a new `or` term, using V_1 and V_2 as its operands. The conclusion defines a transition from the original term to a configuration consisting of only V_3.

If you think about the transition relation as a graph, then the transitions in the premise can be thought of as creating a separate graph whose final configuration is used in the original "parent" graph. In programming terms, they can also be thought of as a recursive call to an evaluation process.

The following transition sequence uses the rules defined in Ex. 4 to describe the behaviour of a term `or(or(True,False),False)`.

$$\begin{aligned}&\langle \texttt{or(or(True,False),False)}\rangle\\&\quad\to\langle\texttt{True}\rangle\end{aligned}$$

Correctly, we reduce to a terminal configuration containing simply the Boolean value `True`. Observe that there is only a single transition in this sequence which takes us straight to the Boolean result of the expression. We have written our rules in such a way that this is always the case, and we lose the detail of sub-steps of the computation. Rules which work in this way are known as *big step* rules. An operational semantics specification which uses only big step rules is known as a *big step operational semantics*. The conclusion of each big step rule always transitions to a terminal configuration, which is usually a value.

We can write rules of a different style to capture the detail of sub-operations - a *small step* style. With small step semantics, we describe a computation in terms of the sub-computations which allow us to reach a terminal result. [5] mentions that in a big-step semantics a single transition $\gamma \to \gamma'$ describes the entire computation that starts in configuration γ, and γ' is always a terminal configuration. In a small-step semantics a single transition $\gamma \to \gamma'$ describes a single step of a larger computation, and γ' need not be a terminal configuration.

Example 5. Consider the following SOS rules.

$$\frac{}{\langle\texttt{or(True,True)}\rangle \to \langle\texttt{True}\rangle} \tag{1}$$

$$\frac{}{\langle\texttt{or(True,False)}\rangle \to \langle\texttt{True}\rangle} \tag{2}$$

$$\frac{}{\langle\texttt{or(False,True)}\rangle \to \langle\texttt{True}\rangle} \tag{3}$$

$$\frac{}{\langle\texttt{or(False,False)}\rangle \to \langle\texttt{True}\rangle} \tag{4}$$

$$\frac{\langle S_2\rangle \to \langle V_1\rangle}{\langle\texttt{or(True,}S_2\texttt{)}\rangle \to \langle\texttt{or(True,}V_2\texttt{)}\rangle} \tag{5}$$

$$\frac{\langle S_2\rangle \to \langle V_1\rangle}{\langle\texttt{or(False,}S_2\texttt{)}\rangle \to \langle\texttt{or(False,}V_2\texttt{)}\rangle} \tag{6}$$

$$\frac{\langle S_1\rangle \to \langle V_1\rangle}{\langle\texttt{or(}S_1\texttt{,}S_2\texttt{)}\rangle \to \langle\texttt{or(}V_1\texttt{,}S_2\texttt{)}\rangle} \tag{7}$$

The axioms 1 to 4 are identical to those defined in Ex. 4. Rules 5 and 6 both handle the case where only the left-operand of an `or` term is a Boolean value. In these cases, we simply use the result of a transition on the right-operand to rebuild a new `or` term in the conclusion of each rule. Rule 7 recognises `or` terms whose left-operand is not a Boolean value. Similar to rules 5 and 6, we transition on this operand and use the result to build a new `or` term.

With the small step rules defined in Ex. 5 we can describe the semantics of `or(or(True,False),False)` by single computation steps.

$$\begin{aligned}&\langle \texttt{or(or(True,False),False)}\rangle\\ &\quad\to \langle \texttt{or(True,False)}\rangle\\ &\quad\to \langle \texttt{True}\rangle\end{aligned}$$

It is useful to observe individual steps in a computation which are defined by a small step operational semantics. However, the simplicity with which one can write big step rules to describe a program makes them the more favourable option in many cases.

As previously mentioned, we can also include semantic entities in configurations which assist in computation. Consider the example where we can assign values to identifiers, and can use those identifiers to access the values they are associated with in later expressions. We need rules for accessing and assigning to some map entity which will hold the assignments to identifiers. We will call this map entity σ.

Example 6. Consider the following SOS rules.

$$\frac{\langle S_1, \sigma_1\rangle \to \langle V_1, \sigma_2\rangle}{\langle \texttt{sequence(}S_1\texttt{,}S_2\texttt{)}, \sigma_1\rangle \to \langle \texttt{sequence(}V_1\texttt{,}S_2\texttt{)}, \sigma_2\rangle} \tag{1}$$

$$\frac{}{\langle \texttt{sequence(done,}S_2\texttt{)}, \sigma_1\rangle \to \langle S_2, \sigma_1\rangle} \tag{2}$$

$$\frac{set(\sigma_1, ID, V_1) \rhd \sigma_2}{\langle \texttt{set(}ID\texttt{,}V_1\texttt{)}, \sigma_1\rangle \to \langle \texttt{done}, \sigma_2\rangle} \tag{3}$$

$$\frac{get(\sigma_1, ID) \rhd V_1}{\langle \texttt{get(}ID\texttt{)}, \sigma_1\rangle \to \langle V_1, \sigma_1\rangle} \tag{4}$$

Rules which allow us to evaluate a sequence of operations are included as rules 1 and 2. The first completes a transition on the left-operand of an `sequence` term and uses the result of that transition in a new `sequence` term. Rule 2 handles the case where the left-operand of the `sequence` term is a value `done`. This new `done` term (sometimes also called `skip`) signifies the conclusion of an operation which doesn't produce a syntactic result. As with the Boolean values in previous examples, a configuration whose syntax is simply `done` is a terminal configuration.

Rule 3 allows us to assign to an identifier in the map semantic entity. Some side operation *set* is used to create a new map entity, by adding to the original entity our new binding of ID to V_1. This new map entity appears in the right-hand side configuration of the conclusion, along with the value `done`.

With rule 4 we can retrieve values associated with identifiers in our map. A side operation *get* is used to access the value associated with ID in the map σ_1. This value is matched to the term variable V_1. The conclusion includes this value as the syntax of a new configuration, as well as the original map σ_1.

We can now share values across a transition sequence by way of assignment. A transition sequence for the term `sequence(set(x,True),get(x))` is given. The semantic entity σ is given as a set of bindings.

$$\begin{aligned}
&\langle \texttt{sequence(set(x,True),get(x))}, \{\}\rangle \\
&\quad \rightarrow \langle \texttt{sequence(done,get(x))}, \{\texttt{x} \coloneqq \texttt{True}\}\rangle \\
&\quad \rightarrow \langle \texttt{get(x)}, \{\texttt{x} \coloneqq \texttt{True}\}\rangle \\
&\quad \rightarrow \langle \texttt{True}, \{\texttt{x} \coloneqq \texttt{True}\}\rangle
\end{aligned}$$

More complicated examples such as control flow and function evaluation can follow from the use of semantic entities, to describe the behaviour of programming languages in particular.

2.6 Universal Algebra

The study of universal algebras is concerned with defining algebraic structures consisting of *sorts* - which can be thought of as types, and operations over those sorts. The Boolean values can be defined as a sort, as well as the natural numbers, the reals, and so forth. The case of Boolean values and operations will be used to illustrate the idea of an algebraic structure. The first step in defining an algebra is to define concept of a *signature*. That is, a formal representation of the constants and operations which an algebra consists of. The below definition of signatures is given in [19].

Definition 6. A *signature* Σ consists of:

1. A non-empty set S, the elements of which we call *sorts*.
2. An $S^* \times S$-indexed family

$$\langle \Sigma_{w,s} \mid w \in S^*, s \in S \rangle$$

of sets, where for the empty word $\lambda \in S^*$, and any sort $s \in S$, each elemend

$$c \in \Sigma_{\lambda,s}$$

is called a *constant symbol* or *name* of sort s; and for each non-empty word $w = s(1)...s(n) \in S^+$ and any sort $s \in S$, each element

$$\sigma \in \Sigma_{w,s}$$

is called an *operation* or *function symbol* or *name* of *type* (w, s); sometimes we term w the *domain type*, s the *co-domain type* and n the *arity* of σ.
Thus we can define Σ to be the pair

$$(S, \langle \Sigma_{w,s} \mid w \in S^*, s \in S \rangle)$$

For emphasis, in certain circumstances we refer to Σ as an *S-sorted signature*.

In our example, we can define a signature for the Boolean sort consisting of the operations; not, or, and. We define Σ^B with the sort $S = \{bool\}$ and a family of operations consisting of:

$$\Sigma_{\lambda,bool} = \{true, false\},\ \Sigma_{bool,bool} = \{not\},\ \Sigma_{bool\ bool,bool} = \{or, and\}$$

We have the constant symbols *true*, *false* which are nullary operators in Σ^B - they have arity zero and simply produce a Boolean value. We also have the unary operation *not* which takes a single Boolean value and produces a new one, and binary operations *or* and *and* which take two Boolean values and produce a new Boolean value. Hence we can define Σ^B as the pair:

$$\Sigma^B = (\{bool\}, \{\{true, false\}, \{not\}, \{or, and\}\})$$

As mentioned in [19], in small familiar examples such as this, we can instead write a signature in the form:

$$\Sigma = (s_1, s_2, ...; c_1, c_2, ...; \sigma_1\sigma_2, ...)$$

Where the s_i are the sorts in signature Σ, the c_i are the constants, and σ_i are the operations. We thus have:

$$\Sigma^B = (bool; true, false; not, or, and)$$

Clearly the definition of this signature does not define the results of expressions over its Boolean operations. However we can build an algebra by specifying each operation as a function of sequences of Boolean values to Boolean values. We use the below definition of algebras given in [19] to introduce the concept.

Definition 7. Let $\Sigma = (S, \langle \Sigma_{w,s} \mid w \in S^*, s \in S \rangle)$ be a signature. A Σ-*algebra* A consists of:

1. An S-indexed family $\langle A_s \mid s \in S \rangle$ of non-empty sets, where for each sort $s \in S$ the set A_s is called the *carrier* of sort s.

2. An $S^* \times S$-indexed family

$$\langle \Sigma^A_{w,s} \mid w \in S^*, s \in S \rangle$$

of sets of constants and sets of functions: for each sort $s \in S$

$$\langle \Sigma^A_{\lambda,s} = \{c_A \mid c \in \Sigma_{\lambda,s}\},$$

where $c_A \in A_s$ is termed a *constant* of sort $s \in S$ which interprets the constant symbol $c \in \Sigma_{\lambda,s}$ in the algebra. For each non-empty word $w = s(1)...s(n) \in S^+$ and each sort $s \in S$

$$\Sigma^A_{w,s} = \{\sigma_A \mid \sigma \in \Sigma_{w,s}\}$$

where $\sigma : A^w \to A_s$ is termed an *operation function* with domain

$$A^w = A_{s(1)} \times ... \times A_{s(n)},$$

codomain A_s and arity n which interprets the functional symbol σ in the algebra.
Thus we define A to be the pair

$$(\langle A_s \mid s \in S \rangle, \langle \Sigma^A_{w,s} \mid w \in S^*, s \in S \rangle)$$

When no ambiguity arises we may use A to denote both an algebra and its S-indexed family of carrier-sets.

For the running example of the simple Boolean operation, we can define the below functions to represent the operations; *not*, *or*, *and*; with signatures *not* : *bool* → *bool*, *or* : *bool* × *bool* → *bool*, *and* : *bool* × *bool* → *bool* as follows:

$$not(a) = \begin{cases} true, & \text{when } a = false \\ false, & \text{when } a = true \end{cases}$$

$$or(a, b) = \begin{cases} true, & \text{when } a = true \text{ or } b = true \\ false, & \text{otherwise} \end{cases}$$

$$and(a, b) = \begin{cases} true, & \text{when } a = true \text{ and } b = true \\ false, & \text{otherwise} \end{cases}$$

Thus we have defined the algebra $A = (bool; true, false; not, or, and)$, where *not*, *or*, *and* refer to the functions defined above.

We can also consider building an algebra of operations over both the Boolean values and the natural numbers. This would constitute a *many-sorted* algebra (specifically in this case, a *two-sorted* algebra). Consider the sets $A_{nat} = \mathbb{N}$ and $A_{bool} = \{true, false\}$ which make up the carrier of algebra AN over the Booleans and naturals. We begin with constants $0, true, false$ - the nullary operators of AN. Our new algebra can consist of operations over Booleans, naturals, or both sorts.

Firstly, we can define a function $equals : nat \times nat \rightarrow bool$ as follows:

$$equals(a, b) = \begin{cases} true, & \text{when } a = b \\ false, & \text{otherwise} \end{cases}$$

And then introduce operations over the naturals; $suc : nat \rightarrow nat$, $add : nat \times nat \rightarrow nat$, $mul : nat \times nat \rightarrow nat$ by the definitions:

$$suc(a) = a + 1,\ add(a, b) = a + b,\ mul(a, b) = a * b$$

Using these new operations, together with operations identical to those defined for A, we can define the two-sorted algebra AN as:

$$AN = (nat, bool; 0, true, false; suc, add, mul, not, or, and, equals).$$

The concept of a universal algebra is used in RAGs as the fundamental model for syntax, meta-syntax and semantics, as will be illustrated in Chapter 4. Namely, Shutt uses a hierarchy of algebras to define the different levels of expressions which together form a recursive adaptable grammar.

Chapter 3

Related Work

This chapter presents a series of work related to the recursive adaptable grammar formalism, on which this thesis is based. Each of the formalisms given in this chapter were part of the research carried out during the writing of this thesis, and contextualise the field of research in which the RAG formalism lies.

3.1 Definite Clause Grammars

Definite Clause Grammars (DCGs) were first introduced by Pereira and Warren in *Definite Clause Grammars for Language Analysis–A Survey of the Formalism and a Comparison with Augmented Transition Networks* [11], and employ the concept of clauses in logic programming. A definite clause consists of a head and a body, with the head being a single goal and the body being a sequence of zero or more goals. Such clauses take the form:

$$P \text{ :- } Q, R, S$$

Where P is the head of the clause, and the goals Q, R, S make up the body. Such clauses can be read as "when Q, R, S are true, P is also true". These clauses can also contain variables. The following example [11],

$$\mathit{Employed}(X) \text{ :- } \mathit{Employs}(Y, X)$$

can be read as "For some X and Y, X is employed if Y employs X". In a DCG, context-free rules are expressed as these kinds of logical statements. Then the problem of parsing a string in a given language is translated into the problem of satisfying the "start goal". We use the following example grammar, given in [11] on which our DCG rules will be modelled.

Example 7. Consider the following CFG rules.

$$\begin{array}{lll}
\textit{Sentence} & \rightarrow & \textit{Noun\_Phrase Verb\_Phrase} \\
\textit{Noun\_Phrase} & \rightarrow & \textit{Determiner Noun Rel\_Clause} \\
\textit{Noun\_Phrase} & \rightarrow & \textit{Name} \\
\textit{Verb\_Phrase} & \rightarrow & \textit{Trans\_Verb Noun\_Phrase} \\
\textit{Verb\_Phrase} & \rightarrow & \textit{Intrans\_Verb} \\
\textit{Rel\_Clause} & \rightarrow & \texttt{that}\ \textit{Verb\_Phrase} \\
\textit{Rel\_Clause} & \rightarrow & \epsilon \\
\textit{Determiner} & \rightarrow & \texttt{every} \\
\textit{Noun} & \rightarrow & \texttt{man} \\
\textit{Name} & \rightarrow & \texttt{mary} \\
\textit{Trans\_Verb} & \rightarrow & \texttt{loves} \\
\textit{Intrans\_Verb} & \rightarrow & \texttt{lives}
\end{array}$$

Using this grammar we can build simple sentences such as "`mary loves every man`". Each of these rules can also be expressed as a definite logic clause. For example, the first rule can be transformed into;

$$\textit{Sentence}(\textit{S0}, S) \text{ :- } \textit{Noun\_Phrase}(\textit{S0}, \textit{S1})\ \textit{Verb\_Phrase}(\textit{S1}, S)\ .$$

which is read as "if there is a noun phrase from $S0$ to $S1$ and a verb phrase from $S2$ to S, then there is a sentence from $S0$ to S" [11]. The effect of the arguments to each non-terminal is that we can identify the start and end points of a string generated by that non-terminal. Within a DCG, each rule is based on a definite clause of this form, perhaps with more information included as additional arguments. The above clause has the DCG form;

$$\textit{Sentence} \rightarrow \textit{Noun\_Phrase Verb\_Phrase}\ .$$

with the two arguments given in the definite clause representation now being implicit in each non-terminal. Thus a non-terminal of arity n in a DCG translates into a logical predicate of place $n + 2$. As will later be illustrated with RAGs, these arguments can be used to construct and carry context-dependent information. To illustrate, the non-terminals given in Ex. 8 each consist of a single explicit argument, used to build a tree representation of a sentence.

Example 8. Consider the following DCG rules.

$\mathit{Sentence}(\texttt{s(}NP,\ VP\texttt{)})\ \rightarrow\ \mathit{Noun\_Phrase}(NP),\ \mathit{Verb\_Phrase}(VP)\ .$

$\mathit{Noun\_Phrase}(\texttt{np(}Det, Noun, Rel\texttt{)})\ \rightarrow\ \mathit{Determiner}(Det),\ \mathit{Noun}(Noun),\ \mathit{Rel\_Clause}(Rel)\ .$

$\mathit{Noun\_Phrase}(\texttt{np(}Name\texttt{)})\ \rightarrow\ \mathit{Name}(Name)\ .$

$\mathit{Verb\_Phrase}(\texttt{vp(}TV, NP\texttt{)})\ \rightarrow\ \mathit{Trans\_Verb}(TV),\ \mathit{Noun\_Phrase}(NP)\ .$

$\mathit{Verb\_Phrase}(\texttt{vp(}IP\texttt{)})\ \rightarrow\ \mathit{Intrans\_Verb}(IP)\ .$

$\mathit{Rel\_Clause}(\texttt{rel(that,}\ VP\texttt{)})\ \rightarrow\ [\texttt{that}],\ \mathit{Verb\_Phrase}(VP)\ .$

$\mathit{Rel\_Clause}(\texttt{rel(}nil\texttt{)})\ \rightarrow []\ .$

$\mathit{Determiner}(\texttt{det(}W\texttt{)})\ \rightarrow [W],\ \{\mathit{Is\_Determiner}(W)\}\ .$

$\mathit{Noun}(\texttt{n(}W\texttt{)})\ \rightarrow [W],\ \{\mathit{Is\_Noun}(W)\}\ .$

$\mathit{Name}(\texttt{name(}W\texttt{)})\ \rightarrow [W],\ \{\mathit{Is\_Name}(W)\}\ .$

$\mathit{Trans\_Verb}(\texttt{tv(}W\texttt{)})\ \rightarrow [W],\ \{\mathit{Is\_Trans}(W)\}\ .$

$\mathit{Intrans\_Verb}(\texttt{iv(}W\texttt{)})\ \rightarrow [W],\ \{\mathit{Is\_Intrans}(W)\}\ .$

With the example associated dictionary:

$\mathit{Is\_Determiner}(\texttt{every})$

$\mathit{Is\_Noun}(\texttt{man})$

$\mathit{Is\_Name}(\texttt{mary})$

$\mathit{Is\_Trans}(\texttt{loves})$

$\mathit{Is\_Intrans}(\texttt{lives})$

In [11] the dictionary is referred to as "the rules defining those non-terminals which correspond to word classes (or parts of speech)". This simply allows us to write a general rule for non-terminals such as *Determiner* complemented by simple clauses of the form $\mathit{Is\_Determiner}(W)$. These define a set of predicates on terminal syntax. These predicates can then be used as side conditions for the grammar rules.

We mentioned that each DCG $n-ary$ non-terminal represents a predicate in a definite clause of place $n+2$. The first rule of this context-free grammar can simply be translated to the following definite clause.

$\mathit{Sentence}(\texttt{s(}NP,\ VP\texttt{)}, S0, S)\ \text{:-}\ \mathit{Noun\_Phrase}(NP, S0, S1),\ \mathit{Verb\_Phrase}(VP, S1, S)\ .$

For this example, the argument given to a non-terminal is referred to in [11] as its "interpretation", similar to a derivation tree. By the second rule, sentence with interpretation $\texttt{np(}Det, Noun, Rel\texttt{)}$ may consist of a determinder with interpretation *Det*, a noun with interpretation *Noun*, and a relative clause with interpretation *Rel*. This interpretation is a kind of synthesised attribute. The effect of this attribute's use is that the final interpretation of a sentence is a representation of the parse tree for that sentence. The notion of parameterized nonterminals bears resemblance to what we refer to in RAGs as non-nullary operators, defined in Chapter 4 - however RAG parameters are generally used as inherited attributes.

The right-hand side of a rule in a DCG may consist of a sequence of clauses similar to procedure calls, enclosed within {}. These allow side-conditions to be placed within a rule, which must be satisfied for the rule to be applicable. For example the rule,

$$Name(\texttt{name(}W\texttt{)}) \rightarrow [W],\ \{Is\_Name(W)\}\ .$$

contains the condition $Is\_Name(W)$ enforcing that for some W there is a corresponding dictionary entry for Is_Name. This bears resemblance to the semantic equations of an attribute grammar. Similar clauses can be specified to implement context sensitivity into a DCG, and there can be multiple of these clauses, all of which must be satisfied to make a rule applicable.

Given that each DCG rule has a translation to a definite clause, a logic language such as Prolog [16] can be used to parse a sentence. In Prolog this constitutes a top-down left-to-right proof procedure from the starting goal to search for a sentence.

3.2 Extended Affix Grammars

Extended Affix Grammars (EAGs) were introduced by Seutter in the publication *Informal introduction to the Extended Affix Grammar formalism and its compiler* [12]. They provide a means for describing context-sensitive languages by use of *syntax rules* and *meta-rules*. Appropriately named, syntax rules are used to describe the syntactic structure of a language akin to the rules of a context-free grammar. However, the non-terminals in an EAG can be decorated with parameters called *affixes* which can introduce context-sensitivity. [12] defines affixes as representing values that are dynamically evaluated and propagated during a parse process.

Affixes consist of an *affix expression* and an optional *affix flow*. The former is a simple expression consisting of values and/or identifiers which are evalated upon rule application. The latter indicates whether an affix value is inherited or derived (synthesised). This of course bears close resemblance to the attribute grammar definitions given in this chapter. An affix value is inherited if the *flow symbol* > precedes the affix expression, and is synthesised if it follows the expression.

The following example EAG presented in [12] illustrates the concept of syntax rules decorated with affix expressions.

Example 9. Consider the following EAG rules.

$$Duplicate(id + \text{“}+\text{”} + id + \text{“}=\text{”} + id + id + \text{“}\backslash n\text{”} >) \rightarrow Identifier(id)\ .$$

$$Identifier(l + ls >) \rightarrow$$

$$\{abcedfghijklmnopqrstuvwxyz\}(l >),$$

$$\{abcedfghijklmnopqrstuvwxyz0123456789\}{*}!\ (ls >),$$

$$\{\ \backslash t\backslash n\}{*}!$$

The syntactic forms generated by this grammar are simple identifiers starting with an alphabetic symbol, followed by zero or more alphanumeric symbols. This example uses two shorthand forms; {...} which stands for one of the set of symbols contained, and {...}∗! which stands for zero or more of each symbol. Such forms are referred to in [12] as *hyper sets*.

The parameters given to elements on either side of the rules define the affix expressions of those elements. In this example, the arity of each non-terminal is one. The notation "+" is used for concatenation of affix values in an affix expression. In the single rule for *Identifier*, the concatenation of the affix value derived as l is concatenated with the affix value derived as ls to form an identifier matching the syntax generated. For an identifier such as `foo`, *Identifier* synthesises the affix value `foo + foo = foofoo`. Each affix value in this example grammar is synthesised. As with DCGs, the concept of parameterised non-terminals in EAGs is similar to the use of non-nullary operators in RAGs - introduced later in this thesis.

Meta-rules in EAGs are used to create restrictions on affix values. These rules are specified as a CFG known as the *meta-grammar* of an EAG. As an alteration of the previous example, the following example makes use of a meta-rule `prefix` to generate only those identifiers beginning with `a`, `b` or `c`.

Example 10. Consider the following EAG rules.

$\mathit{prefix} :: \text{“}a\text{”} \;;\; \text{“}b\text{”} \;;\; \text{“}c\text{”} \;.$

$\mathit{Duplicate}(id + \text{“} + \text{”} + id + \text{“} = \text{”} + id + id + \text{“}\backslash n\text{”} >) \rightarrow \mathit{Identifier}(id)\;.$

$Identifier(\mathit{prefix} + ls >) \rightarrow$

$\quad\{abcedfghijklmnopqrstuvwxyz\}(\mathit{prefix} >),$

$\quad\{abcedfghijklmnopqrstuvwxyz0123456789\}{*}!\;(ls >),$

$\quad\{\;\;\backslash t\backslash n\}{*}!$

The affix value *prefix* synthesised by the single rule for *Identifier* must now comply with the values defined in the meta-rule *prefix* for a string to be recognised. This is one example of the introduction of context-sensitivity by use of meta-rules and affix values.

A theme re-visited when discussing RAGs in Chapter 4, EAGs can also contain rules which direct a parse without producing syntax. Such rules are referred to in [12] as *predicates*. The evaluation of a predicate may succeed or fail, in the former case no syntax is generated. In the latter case, the rule containing the predicate is not applicable. [12] introduces two built-in predicates, "*equal*" which only succeeds if two affix expressions are equal, and "*not equal*" covering the opposite case. Given the rules defined in Ex. 10, the following syntax rule can only generate two identifiers which are not equal.

$$\mathit{NonMatching} \rightarrow \mathit{Identifier}(l_1),\;\; \mathit{Identifier}(l_2),\; \mathit{not\ equal}(l_1, l_2)$$

Later in this thesis we introduce the RAG notion of a query which generates no syntactic value, but is used to direct a parse in a similar way. The grammar writer is able to build predicate rules in terms of existing predicates. Example applications include checking whether some value belongs to a list, or if some variable name has previously been declared.

The features of EAGs that support context-sensitivity are very similar to those incorporated in the RAG formalism given in the next chapter. In particular, RAGs contain the notion of a query, which can be used to implement something akin to EAG predicates, and used to direct a parse in a similar way. The details the strategies for parsing EAGs are left to [12], which presents a tool which implements the formalism.

3.3 Silver: an Extensible Attribute Grammar System

Silver is an attribute grammar specification language introduced by Van Wyk et al [17]. This domain-specific language allows the construction of *extensible* languages, where syntactic and semantic extensions can be built to compliment a core language. The modularity of such extensions allows one to implement new language features regardless of the existence of others.

The notion of *forwarding* built into Silver allows extension constructs to translate into semantically equivalent constructs in the host language. For instance, one may introduce *for* loops as a syntactic construct, where the semantics is that of an initialisation expression together with a while loop. This example is given in Ex. 11, an altered version of a specification presented in [17]. Since each Silver specification is a program, examples of its use are given in teletype.

Example 11. Consider the following Silver specification.

```
nonterminal Prog, Dcls, Dcl, Stmt, Expr, Type ;
synthesized attribute c :: String ;
attribute c occurs on Prog, Dcls, Dcl, Stmt, Expr, Type ;

abstract production program p::Prog ::= f::Dcls
{ p.c = "#include <stdio.h> \n" ++ f.c ;
p.errors := f.errors ;
f.env = [ :: Binding ] ; }

abstract production while w::Stmt ::= cond::Expr body::Stmt
{ w.c = "while ( " ++ cond.c ++ ") \n" ++ body.c ;
w.errors := cond.errors ++ body.errors ; }

abstract production for
f::Stmt ::= init::Stmt cond::Expr inc::Stmt body::Stmt
{ forwards to stmt_seq (init, while(cond, stmt_seq(body,inc))) ; }

abstract production stmt_seq s::Stmt ::= s1::Stmt s2::Stmt
```

```
{ s.c = s1.c ++ s2.c ; }

abstract production logical_and e::Expr ::= l::Expr r::Expr
{ e.c = "(" ++ l.c ++ " && " ++ r.c ++ ")";
e.errors := ... ;
e.typerep = booleanType(); }

abstract production logical_not e::Expr ::= ce::Expr
{ e.c = "( ! " ++ ce.c ")";
e.errors := ... ;
e.typerep = booleanType(); }

abstract production logical_or e::Expr ::= l::Expr r::Expr
{ e.typerep = booleanType();
e.errors := ... ;
forwards to logical_not ( logical_and (logical_not(l),
logical_not(r)));
}

abstract production func_call e::Expr ::= f::Id_t arg::Expr
{ e.c = f.lexeme ++ "(" ++ arg.c ++ ")" ;
e.errors := arg.errors; }
```

Specifications in core Silver generally begin with a declaration of attributes with the keywords `synthesized` and `inherited`, as well as non-terminals using the `nonterminal` keyword. The attribute `c` has type `String` and applies to the non-terminals `Prog`, `Stmt` and `Expr`. Observe that each production is declared as an `abstract production`. Such productions only define the semantic constructs of a language, and not its syntax. Each production has a semantic block given within {...} in which semantic equations using attributes are defined. The matching syntactic rules are given in Ex. 12. Each production in Silver consists of an identifier such as `logical_or` a non-terminal name which the rule is applied to, and the non-terminal names occuring on the right-hand side of the rule. Each can be paired with an identifier such as `e::Expr`.

There are two occurances of forwarding in this example specification. Namely, for the productions `for` and `logical_or`. The former declares that the abstract construct for a `for` loop should be that of a sequence consisting of an initialisation statement and a semantically equivalent while loop. The latter defines the abstract syntax of the boolean `or` expression to copy that of of an equivalent expression using `not` and `and`. Resultingly, use of the `c` attribute on each of these constructs *forwards* to that of its matching core language expression.

The following specification defines the concrete syntax corresponding to the rules given in Ex. 11.

Example 12. Consider the following Silver specification.

```
start nonterminal Prog_c ;
nonterminal FunDcls_c, FunDcl_c, Dcls_c, Dcl_c,
Stmts_c, Stmt_c, Type_c, Expr_c ;

terminal IntLit_t /[0-9]+/ ;
terminal Id_t /[a-zA-Z][a-zA-Z0-9\_]*/ ;
terminal NotOp '!' precedence = 12;
terminal AndOp '&&' precedence = 10, association = none ;
terminal OrOp '||' precedence = 8, association = none ;

synthesized attribute ast_Stmt_c :: Stmt occurs on Stmt_c ;
synthesized attribute ast_Stmts_c :: Stmt occurs on Stmts_c ;
synthesized attribute ast_Expr_c :: Expr occurs on Expr_c ;

concrete production program_c p::Prog_c ::= fds::FunDcls_c
{ p.ast_Prog_c = program(fds.ast_FunDcls_c) ; }

concrete production stmts_cons_c
ss::Stmts_c ::= s::Stmt_c stail::Stmts_c
{ ss.ast_Stmts_c = stmt_seq(s.ast_Stmt_c, stail.ast_Stmts_c) ; }

concrete production stmts_one_c ss::Stmts_c ::= s::Stmt_c
{ ss.ast_Stmts_c = s.ast_Stmt_c ; }

concrete production while_c
w::Stmt_c ::= 'while' '(' c::Expr_c ')' s::Stmt_c
{ w.ast_Stmt_c = while(c.ast_Expr_c, s.ast_Stmt_c) ; }

concrete production for_c
w::Stmt_c ::= 'for' '(' s1::Stmt_c ';' e::Expr_c ';' s2::Stmt_c ')' s3::Stmt_c
{ w.ast_Stmt_c = for(s1.ast_Stmt_c, e.ast_Expr_c, s2.ast_Stmt.c s3.ast_Stmt_c) ; }

concrete production assign_c
a::Stmt_c ::= id::Id_t '=' e::Expr_c ';'
{ a.ast_Stmt_c = assign(id,e.ast_Expr_c) ; }

concrete production logical_and_c
e::Expr_c ::= l::Expr_c '&&' r::Expr_c
{ a.ast_Expr_c = logical_and(l.ast_Expr_C, r.ast_Expr_c) ; }

concrete production logical_or_c
e::Expr_c ::= l::Expr_c '||' r::Expr_c
{ a.ast_Expr_c = logical_or(l.ast_Expr_C, r.ast_Expr_c) ; }
```

```
concrete production idref_c e::Expr_c ::= id::Id_t
{ e.ast_Expr_c = idref(id); }
```

Ex. 12 defines a list of terminals with the `terminal` keyword. Each consists of an identifier and a regular expression. One may also define a precedence and association for each terminal. These are given for the Boolean operators. The types of attributes prefixed by `ast_` refer to the abstract syntax constructs defined in Ex. 11. When defining attributes of these types in the semantic block of each rule, an abstract tree is constructed using the identifiers of the `abstract` productions. Such attributes are known as higher-order attributes. Together the specification in Ex. 11 and Ex. 12 define part of a simple language (referred to as SimpleC in [17]).

3.4 Van Wijngaarden Grammars

Another formalism with which we can define context-sensitive language is with Van Wijngaarden (VW) grammars [13]. A VW grammar consists of a composition of two grammars, hence they are also referred to as *two-level* grammars. These two levels are known as the *metarules* and the *hyperrules.*

The set of metarules is defined as a context-free grammar, however the non-terminals are replaced with *metanotions* and the terminals with *protonotions.* [13] defines protonotions as sequences of syntactic marks, and metanotions as sequences of big syntactic marks. However these can simply be thought of as sequences of nonterminals and terminals. In the metarules, individual productions are separated by a semi-colon, and whitespace is used to separate components of a single production.

Secondly, hyperrules consist of *hypernotions*, sequences of metanotions and protonotions. To distinguish between metarules and hyperrules, the former uses :: as a production symbol, and the latter uses : similarly. The hypernotions which make up a single production are separated by commas.

We now demonstrate how VW grammars can be used to generate context-sensitive languages. The following VW grammar given in [13] generates the context-sensitive language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 1\}$.

Example 13. Consider the following Van Wijngaarden grammar.

$$N :: i\ N\ ;\ i\ .$$
$$A :: \mathtt{a}\ ;\ \mathtt{b}\ ;\ \mathtt{c}\ .$$

$$S : aN,\ bN,\ cN\ .$$
$$AiN : A\ symbol,\ AN\ .$$
$$Ai : A\ symbol\ .$$

This grammar consists of two metarules and three hyperrules with metanotions N, A and hypernotions $AiN, Ai, A, AN, aN, bN, cN$. Note that the keyword *symbol* is an indicator that the result of the non-terminal preceding it is to be treated as a terminal symbol. The i's act as a kind of counter, indicating how many of each letter to generate. As noted in [13], the metarules generate a possibly infinite set of hyperrules. In this case, for instance, the rule "AiN : A *symbol*, AN ." generates the infinite set of hyperrules below.

$$\texttt{a}ii : \texttt{a}\ symbol,\ \texttt{a}i\ .$$
$$\texttt{a}iii : \texttt{a}\ symbol,\ \texttt{a}ii\ .$$
...
$$\texttt{b}ii : \texttt{b}\ symbol,\ \texttt{b}i\ .$$
$$\texttt{b}iii : \texttt{b}\ symbol,\ \texttt{b}ii\ .$$
...
$$\texttt{c}ii : \texttt{c}\ symbol,\ \texttt{c}i\ .$$
$$\texttt{c}iii : \texttt{c}\ symbol,\ \texttt{c}ii\ .$$
...

By first replacing the A consistently with the metanotions it can generate, and then replacing the N similarly. The iN is essentially a pattern-match, where the remaining i's in a sequence are matched to N. Thus one less i appears on the right-hand side of each rule than the left. This causes an eventual base case on the number of each letter generated. Given that the hyperrule for S consists of a sequence $\texttt{a}N$, $\texttt{b}N$, $\texttt{c}N$, an equal number of `a`'s, `b`'s and `c`'s is generated. This is aided by the constraint that each instance of a metanotion on the right-hand side of a hyperrule must be rewritten consistently, referred to in [13] as the *Uniform Replacement Rule*.

The notion of metarules bears some similarity to rules of the same name in the Extended Affix Grammar formalism. In both cases, the metarules are essentially context-free grammars. However in the EAG formalism, metarules only appear in Affixes (the semantic domain), whereas metanotions and protonotions defined in a VW grammar can be used to define syntactic terms.

3.5 Christiansen Grammars

The Christiansen grammar formalism is a formalism in the class of adaptable (or, adaptive) grammars. The formalism was defined by Christiansen in *The Syntax and Semantics of Extensible Languages* with the name *generative grammars* [14], however as in Shutt's thesis [1], to resolve confusion with Chomsky's generative grammars [15], we refer to these grammars as *Christiansen grammars*.

Christiansen describes the formalism as a generalisation of attribute grammars. To quote [14]; the attribute machinery is made responsible for the generation and management of new syntactic rules. The definition of context-sensitive languages can be achieved by making use of attributes which carry generated rules - effectively modifying the set of rules usable to rewrite a non-terminal based on context-sensitive information.

To illustrate the concept, we give an example definition of a context-sensitive language. The grammar defined in Ex. 14 is similar to a grammar defined in [14], but extended to generate the language $\{www \mid w \in [\mathtt{a} - \mathtt{z}]^*\}$ - a language used as an example context-sensitive language thoughout the thesis.

Example 14. Consider the following Christiansen grammar.

$$\langle \mathit{Program} \downarrow G\rangle \rightarrow \langle \mathit{Decl} \downarrow G \uparrow w\rangle\ \langle \mathit{Body} \downarrow \{w\text{-}\mathit{rule}\}\rangle\ \langle \mathit{Body} \downarrow \{w\text{-}\mathit{rule}\}\rangle$$
$$\mathit{where}\ w\text{-}\mathit{rule} = \langle \mathit{Body} \downarrow G'\rangle \rightarrow w$$
$$\langle \mathit{Decl} \downarrow G \uparrow ch \cdot w\rangle \rightarrow \langle \mathit{Char} \downarrow G \uparrow ch\rangle\ \langle \mathit{Decl} \downarrow G \uparrow w\rangle$$
$$\langle \mathit{Decl} \downarrow G \uparrow \langle\rangle\rangle \rightarrow \langle \mathit{Nothing}\rangle$$
$$\langle \mathit{Char} \downarrow G \uparrow \mathtt{a}\rangle \rightarrow \mathtt{a}$$
$$\ldots$$
$$\langle \mathit{Char} \downarrow G \uparrow \mathtt{z}\rangle \rightarrow \mathtt{z}$$

Occuring on each non-terminal is an inherited attribute $\downarrow G$. By convention, the first attribute position of a non-terminal is known as the *language attribute*. According to [14], this attribute variable propogates the a grammar between non-terminals. In the single rule for *Program*, the language attribute $\downarrow G$ is copied to *Decl*. Thus the original grammar structure is passed to *Decl*.

Also in the single rule for *Program*, the *Body* non-terminal receives $\{w\text{-}\mathit{rule}\}$ as its inherited language attribute, which is specified as the grammar $\langle \mathit{body} \downarrow G'\rangle \rightarrow w$ - where w is the attribute variable synthesised by the rewriting of *Decl*. The attribute G' distinguishes the language attribute of this single-rule grammar from the original grammar. By this construction, the only usable rule in the language attribute of *Body* is defined by *w-rule*. Thus the value bound to w is the only string generatable by *Body*. The effect of this is that the string *Decl* rewrites to is copied twice in the first rule of the grammar - enforcing the language $\{www \mid w \in [\mathtt{a} - \mathtt{z}]^*\}$.

The rules for *Decl* synthesise an attribute value which matches the string generated by the initial *Decl*. In the second rule, attribute position $\uparrow ch \cdot w$ computes the concatenation of the attribute *ch* generated by rewriting *Char*, and the value synthesised as w by rewriting *Decl*. $\langle \mathit{Nothing}\rangle$ is a meta-symbol used to generate the empty string - we use λ for this purpose throughout the thesis.

It seems that the right-hand side of the rule for *Body* depends on the attribute w synthesised by rewriting *Decl* in the first rule. As we will see with RAGs, this single rule actually represents an infinite set of *instance* rules. As defined in [14];

Definition 8. An *instance* of a generative grammar is the structure, if any, that results from a substitution of its attribute expression by their respective values assuming a consistent assignment of values to attributes throughout the rule.

Thus the purpose of context-sensitive equations which seem to generate rules, such as for *w-rule* in Ex. 14, is to reduce a possibly infinite set of rules to a single attribute based representation. Similarly to RAGs, the derivation relation is a relation over the possibly infinite set of instance rules, not the original attribute-decorated rules we write when defining a language using Christiansen grammars.

Chapter 4

Overview of Recursive Adaptable Grammars

4.1 Introduction

With recursive adaptable grammars (RAGs) we can define recursive enumerable languages in a style familiar to those with background knowledge of context-free grammars. Context-dependant information is computed by means of a simple derivation step relation, defined later in this chapter.

This chapter will first explore the definitions for the recursive adaptable grammar formalism defined by Shutt. Following this, we discuss RAG syntax generation and expression evaluation. A portfolio of example specifications is given at the end of this chapter with an accompanying derivation example.

4.2 Contributions

Shutt in his 1993 thesis "Recursive Adaptable Grammars" [1] proposed parsing techniques for his formalism as a future research topic. A contribution of this thesis is to introduce such a technique. The high-level mechanics of the strategy will be explored later in this thesis.

Shutt provided few examples of RAG definitions, with no example derivations. Therefore this thesis also provides a portfolio of RAG examples for simple languages which illustrate useful, practical idioms for building RAGs. These examples display the uses of the features of RAGs defined in §4.4 and of the formalism generally. Each RAG example will be accompanied by example derivations to assist the reader's understanding.

4.3 Recursive Adaptable Grammars

The recursive adaptable grammar formalism was built to incorporate adaptability whilst also maintaining a rule style familiar to those with a background of context-free grammar specification. Shutt's goal was to construct a Turing-powerful formalism which encapsulates all computation in a simple derivation step relation. With recursive adaptable grammars we can define recursive enumerable languages.

Context-free grammars only allow us to reason about the *syntax* of a language, and not its semantics. To denote the “meaning” of each string we need another formalism, such as attribute grammars. In this way, the semantics are a complement to the underlying CFG, as opposed to being part of the CFG formalism itself.

Contrastingly, the RAG formalism allows us to express semantic values associated with meta-syntactic terms. The strategy of rewriting non-terminals in context-free generation is extended in RAGs to the rewriting of pairs to allow the specification of semantics. Each rewritable pair consists of a syntactic or meta-syntactic value as its left-component, and a semantic value as its right-component. The formal nature of each of the components of a pair is defined in §4.4. A pair can have potentially infinitely many semantic values as its right-component. To specify all of them, variables are used to stand in for those values when we write RAGs.

4.3.1 Syntax, Meta-Syntax, and Semantics

It is clear in the context-free grammar formalism that the syntax and meta-syntax specified by a grammar are two disjoint sets. A terminal cannot be used as a non-terminal, and vice-versa. And as mentioned in Chapter 2, without an augmenting formalism such as attribute grammars, one cannot specify semantics with a context-free grammar.

Contrastingly, the RAG formalism relies on the domains of syntactic, meta-syntactic and semantic values being identical. This class is that of the *answers*, known as the *answer algebra*. We can use any answer as syntax, meta-syntax or a semantic value. A potent example of this is that the semantic value of some derivation can be used as meta-syntax to generate strings.

4.3.2 Algebras

Shutt defines a hierarchy of four algebras as the basis for all RAG terms. Each algebra has a fixed set of operators - those which are embedded in the RAG formalism. We are also able to define our own operators and use them in specifications. Each of the algebras with their fixed operators are described by the following points:

- *Terminal algebra* - Terms constructed using only operators in this domain are called *terminals*, and generally have the same role as terminals in context-free grammars.
- *Answer algebra* - This algebra makes use of binary concatenation and the constant empty string operator λ as its fixed operators. In this dissertation, concatenation is specified using juxtaposition. The terminal algebra is a subset of the answer algebra. This algebra generally

consists of those operators one might define as *non-terminals* - those which can generate terms other than themselves. The terms in this algebra are called *answers*.

- *Query algebra* - With this algebra, we extend the answer algebra by introducing the binary *query operator*. The formal details and importance of queries will be elaborated on later in this chapter. Terms in this algebra are called *queries*.
- *Configuration algebra* - The configuration algebra introduces the binary *pairing* operator and the unary *inverse* operator. The significance of the inverse operator will be detailed later in this chapter. Instances of the pairing operator are called *pairs*. These are the fundamental artefacts which are rewritten during derivation steps. Terms of this algebra are called *configurations*.

All RAG terms fall into these algebra classes. The class associated with a term is determined by its highest-level operator. For example, a term which contains the query operator, but not the inverse nor pairing operator, is a query. By nature of the hierarchy, every answer is also a query and a configuration, every query is also a configuration, and so on.

4.3.3 Conventions

Throughout this thesis, RAG specifications follow the following conventions.

- Terminal answers are given in teletype font. For example, `a`, `b`, `c`.
- Answers which are not terminals are given in italics, and start with an upper case letter. For example, *S*, *OR*.
- Pairs are represented using angled brackets. For example $\langle A, \mathtt{a} \rangle$ is a pair whose left-component is the answer A and whose right-component is a terminal `a`.
- Queries are written as operators of the form $(a \,?\, b)$ where a and b are the operands of the query.
- Inverses are written using the operator '!'. The inverse of of a query $A \,?\, B$ is $!(A \,?\, B)$.

4.3.4 Example Specification

The simple example of a single Boolean operation language illustrates the fundamental rewriting strategy employed by the RAG formalism, and how to compute semantics. Firstly, consider how one might write a set of context-free rules to define syntax for Boolean expressions, such as the *OR* operation, in Backus–Naur form.

Example 15. A context-free grammar for *OR* denoted as the operator `|`.

$$OR \rightarrow \texttt{T} \vee BOOL \tag{1}$$

$$OR \rightarrow \texttt{F} \vee BOOL \tag{2}$$

$$BOOL \rightarrow \texttt{T} \tag{3}$$

$$BOOL \rightarrow \texttt{F} \tag{4}$$

The terminals are $\{\texttt{T}, \vee, \texttt{F}\}$, and the non-terminals are $\{OR, BOOL\}$, with the start symbol being OR. With a left-most top-down strategy we can derive $\texttt{F}\vee\texttt{T}$ with the sequence $OR \Rightarrow \texttt{F}\vee BOOL \Rightarrow \texttt{F}\vee\texttt{T}$. Of course, we cannot compute the semantics of any of the strings we can generate using Ex. 15. We will use a similar structure of rules in a RAG specification which not only generates the same syntax as our CFG, but also illustrates how the dynamic semantics of *OR* can be computed.

Example 16. Consider the following RAG.

$$\langle OR,\, \texttt{T}\rangle \rightarrow \texttt{T} \vee \langle BOOL,\, v_1\rangle \tag{1}$$

$$\langle OR,\, v_1\rangle \rightarrow \texttt{F} \vee \langle BOOL,\, v_1\rangle \tag{2}$$

$$\langle BOOL,\, \texttt{T}\rangle \rightarrow \texttt{T} \tag{3}$$

$$\langle BOOL,\, \texttt{F}\rangle \rightarrow \texttt{F} \tag{4}$$

The nullary terminal constants are $\{\texttt{T}, \texttt{F}, \vee\}$, the nullary non-terminal answers are $\{OR, BOOL\}$, and the start symbol is OR. The only non-nullary operator is concatenation.

Each rule specifies how to rewrite pairs whose left-component matches that of the rule's left-hand side pair (referred to as the *head*) - the pair on the left-hand side of a rule. The right-component of a pair can be thought of the semantic value associated with the string the pair generates. On the right-hand side of rules, we usually have the right-component of a pair as a variable. Such variables range over the answer algebra. Formally, a variable can be substituted for any answer when it comes to derivations. However, generally only those which are able to match to the right-component of the head of a rule allow the pair to be rewritten. Thus the rules in a RAG specification are schemas for *concrete* rules, whose variables have all been substituted for answers.

Rules (3) and (4) for *BOOL* recognise the Boolean constants $\texttt{T}, \texttt{F}$ as the syntax of their right-hand sides respectively. In each case, the Boolean value generated by the rule's right-hand side is also synthesised as the semantic result of the rule. This is denoted by use of $\texttt{T}, \texttt{F}$ as the right-components of the pairs on the left-hand side of rules (3) and (4) respectively. In other words, when generating a Boolean value, the semantics of that value is itself.

Rule (1) for *OR* generates expressions whose left-operand is the Boolean value `T`. The right-operand can be any Boolean expression, since the variable

v_1 is used to specify the semantics of the pair $\langle BOOL, v_1\rangle$. Therefore either rule (3) or (4) can be used to rewrite this pair. When it comes to derivation, this variable must be substituted for `T` or `F`, and the corresponding rule is used to rewrite the pair. The semantic result of this rule is `T`, since any *OR* expression with `T` as an operand is always `T`.

Rule (2) for *OR* generates *OR* expressions whose left-operand is the Boolean value `F`. In the same way as rule (1), a pair containing a variable as its right-component is used to specify any Boolean value. The only Boolean constant explicitly specified in this rule is `F`. Whichever Boolean value this pair rewrites to in a derivation, v_1 will be substituted for that value by rules (3) and (4). Therefore the semantic result of this rule, v_1, is substituted for the second Boolean constant in the *OR* operation. This satisfies the truth table for *OR*.

The following is a derivation of the syntax $\mathtt{F} \vee \mathtt{T}$ from start symbol *OR*.

$$\langle OR, \mathtt{T}\rangle \Rightarrow \mathtt{F} \vee \langle BOOL, \mathtt{T}\rangle \Rightarrow \mathtt{F} \vee \mathtt{T}$$

The semantic value of this string is the Boolean constant `T`. More complicated examples of RAGs are given later in this chapter with accompanying example derivations.

4.3.5 Queries

Shutt outlined his desire to handle recursive computations, and stated that such these should be computed as part of the formal derivation process. The introduction of the query operator was his way of handling this feature. The computation associated with a query occurs as a part of the RAG derivation process.

A query operation $(a\,?\,b)$ represents the set of semantic values computed by a derivation of b with start symbol a. Proposition 4.4 of Shutt's thesis [1] states the property of RAG rewriting that enables this.

$$\text{For all answers } a, b, c,\ (a\,?\,b) \overset{*}{\Rightarrow} c \text{ if and only if } \langle a,\ c\rangle \overset{*}{\Rightarrow} b$$

From an implementation perspective, a query can be thought of as spawning a recursive parsing process. Namely, we begin a new sub-derivation and search for a derivation of the query's right-operand, using the left-operand as the start symbol. As a result, we end up with a set of semantic values associated with this sub-derivation. The query can be rewritten to any of these values.

Note that since the query operator belongs in the query algebra, its operands can be queries. However, to satisfy the relation above, each subquery must first be rewritten to an answer before the outer query itself can be. The formal mechanics of query rewriting are defined in §4.4.3.

4.4 RAG Definitions

Each of the definitions listed in this chapter are given with respect to Shutt's technical 1999 report "Recursive Adaptable Grammars" [2], written as a later amendment to his 1993 thesis.

4.4.1 Algebras

The following definitions refer to the fixed operators used in the RAG formalism; the *query*, *pairing* and *inverse* operators. Recall that the *query operator* of arity two is denoted $x\,?\,y$, the *pairing operator* of arity two, denoted $\langle x, y\rangle$, and the *inverse operator* of arity one, denoted $!x$. Each of these operators is associated with signatures Σ_{qry}, Σ_{pair}, and Σ_{inv} respectively. A signature is essentially a set of operators together with a function mapping each operator to an arity.

Definition 9. A vocabulary is a pair $V = (\Sigma, Z)$, where Σ is a signature; Z is a finite alphabet; Σ is disjoint from the string signature over Z; Σ does not contain a query, pairing, or inverse operator. Z is called the *terminal alphabet*, and Σ the *non-terminal signature.*

The string signature Σ_{str} refers to a signature containing both the empty string λ, and binary concatenation. Σ being disjoint from the string signature means that terms over V are simply the symbols in Z and operators in Σ, which by default does not contain λ or concatenation.

Definition 10. Given a vocabulary $V = (\Sigma, Z)$, the string algebra Z^* is called the *terminal algebra* over V, denoted T_V. Elements of T_V are called *terminals.*

The terminal algebra is the lowest-level algebra considered in Shutt's hierarchy, and does not introduce any operators which are not in the vocabulary V. Its terms are the strings in Z^* and operations in Σ.

A specification in is a pair $\Omega = (\Sigma, \mathcal{E})$ consisting of a signature Σ and a set $\mathcal{E}$ of equations over Σ. Terms in an Ω-algebra use operators belonging to Σ and must satisfy the equations in $\mathcal{E}$.

Definition 11. The *string specification* is $\Omega_{str} = (\Sigma_{str}, \mathcal{E}_{str})$, where Σ_{str} consists of the empty string and binary concatenation, denoted respectively by λ and juxtposition; and $\mathcal{E}_{str}$ consists of equations $x\lambda = x$, $\lambda x = x$, and $x(yz) = (xy)z$, in variables x, y, z.

The empty string λ and binary concatenation are introduced by Σ_{str} and must satisfy the equations $x\lambda = x$, $\lambda x = x$, and $x(yz) = (xy)z$, in variables x, y, z, which belong to $\mathcal{E}_{str}$.

Definition 12. Given a vocabulary $V = (\Sigma, Z)$, the *answer algebra* over V, denoted A_V, is the set of all terms in T_V together with all terms over Ω_{str}. Elements of A_V are called *answers.* Elements of $A_V - T_V$ are called *non-terminals.*

By including the string specification Ω_{str}, binary concatenation and the empty string λ are defined as operators in the answer algebra which satisfy the equations in Ω_{str}. Thus, a term made up of a concatenation of answers is also an answer. Any nullary operator is an answer, and any n-ary operator $\alpha(a_1, ..., a_n)$ with $a_1, ..., a_n$ as answers is also an answer.

Definition 13. Given a vocabulary $V = (\Sigma, Z)$, the *query algebra* over V, denoted Q_V, is the set of all terms in A_V $\cup$ all terms over $\Sigma \cup \Sigma_{str} \cup \Sigma_{qry}$. Elements of Q_V are called *queries.*

Definition 13 introduces the query operator, denoted $x\,?\,y$ for answers x, y, discussed further in following sections and used extensively in the example RAGs.

Definition 14. Given a vocabulary $V = (\Sigma, Z)$, the *configuration algebra*, denoted C_V, is the set of all terms in Q_V together with terms over $\Sigma \cup \Sigma_{str} \cup \Sigma_{qry} \cup \Sigma_{pair} \cup \Sigma_{inv}$, satisfying the following constraints.

$$\forall a \in A_V,\ a =!a$$
$$\forall c \in C_V,\ c =!!c$$

Elements of C_V are called *configurations.*

The relation $a = !a$ only holds when a is an answer, and, for all configurations the relation $v = !!v$ holds. The role of the inverse operator ! is explained later with respect to the derivation step relation. The definitions of the preceding algebra build the hierarchy $T_V \subseteq A_V \subseteq Q_V \subseteq C_V$.

4.4.2 Grammars

We now define the structure of rules in a RAG specification, as well as the concrete rules such specifications represent. Similar to a grammar in Backus–Naur form, rewrite rules are represented by the operator $\rightarrow$. The left-hand side of the arrow is the *head* pair, and the right-hand side is the configuration which that pair is rewritable to.

Definition 15. An *unbound rule* over vocabulary V is a structure of the form

$$\langle v_0, e_0 \rangle \rightarrow t_0\, \langle e_1, v_1 \rangle\, t_1\, \langle e_2, v_2 \rangle\, t_2 \ldots \langle e_n, v_n \rangle\, t_n$$

where $n \geq 0$, the t_k are answers over V, the v_k are distinct variables, and the e_k are terms in $Q_V(v_0, ..., v_n)$. The pair $\langle v_0, e_0 \rangle$ on the left-hand side of a rule is called the *head.* The domain of all unbound rules over V is denoted $\mathcal{R}_V$.

Note that the variables on the right-hand sides of pairs are *distinct.* However, a variable can be used as, or part of, the left-component of a pair following its definition. The unbound rules are schemas for sets of rules where the variables have been substituted for answers. Such rules are known as the *bound* rules, and are the rules used in a derivation. Theoretically a variable can be substituted for any answer. However, some substitutions may result in a pair which is not rewritable by a concrete rule.

Definition 16. A *bound rule* over vocabulary V is a structure of the form

$$\langle a_0, q_0 \rangle \to t_0 \, \langle q_1, a_1 \rangle \, t_1 \, \langle q_2, a_2 \rangle \, t_2 \, ... \, \langle q_n, a_n \rangle \, t_n$$

where $n \geq 0$, and the t_k, a_k are terms in A_V, and the q_k are terms in Q_V.

Each $a_1, ..., a_n$ which appears as the right-component of a pair is an answer, which is the semantic result of a string generated by that pair, computed by the semantics of the rule used to rewrite it. The semantic result of a rule is the right-component of its head pair.

Definition 17. For an unbound rule r over vocabulary V, an *instance* of r is a bound rule obtained by substituting the variables in r with answers in A_V. The set of all instances of unbound rule r over V is denoted $\beta_v(r)$.

$\beta_v(r)$ is usually an infinite set, as there are infinitely many answers which we can use to substitute the variables in r.

Definition 18. A *recursive adaptable grammar* $G = \langle \Sigma, Z, \rho, s \rangle$ is a four-tuple, where $V = \langle \Sigma, Z \rangle$ is a vocabulary, $\rho : A_V \to \mathcal{P}(\mathcal{R}_V)$ is the *rule function*, and $s \in A_V$ is the *start symbol* of G. The rule function of G is denoted ρ_G, and the start symbol s_G.

The subscript G may be used in place of subscript V wherever the latter occurs. Answer algebra A_G, terminal algebra T_G, etc. For a RAG G, the rule function for all terminals in T_G is defined as follows.

$$\forall t \in T_G, \quad \rho(t) = \{\langle v_0, t \rangle \to t\}$$

Namely, each terminal rewrites to itself, and synthesizes its own syntax as its semantic value. This is the only rule that a terminal can have. The rule function defines the set of *unbound* rules associated with an answer. $\rho(A)$ is the set of unbound rules for an answer A.

Definition 19. Suppose r and r' are unbound rules. The *concatenation* $r\,r'$ is constructed as follows. Let the rule $r = (\langle v_0, e_0 \rangle \to \pi)$ and $r' = (\langle v_0, e'_0 \rangle \to \pi')$. Assume without loss of generality that v_0 is the only variable-name shared by r and r'. Then, $r\,r' = (\langle v_0, e_0 \, e'_0 \rangle \to \pi \, \pi')$.

For answers a, b, we have $\rho(ab) = \{r_a r_b \,|\, (r_a \in \rho(a)) \wedge (r_b \in \rho(b))\}$.

In RAG specifications defined later in this document, the rule function is written in shorthand notation. For a RAG G, answer A, terms $e_1, e_2 \in Q_G(v_0, ..., v_n)$, and configurations c_1, c_2, the mapping

$$\rho(A) = \begin{Bmatrix} \langle v_0, e_1 \rangle \to c_1 \\ \langle v_0, e_2 \rangle \to c_2 \end{Bmatrix}$$

can instead be written in the following form.

$$\langle A, e_1 \rangle \to c_1$$
$$\langle A, e_2 \rangle \to c_2$$

4.4.3 Derivations

The purpose of a derivation in RAGs is to relate a syntactic string and its associated semantic value. Unlike with context-free grammars, derivation is not a simple case of replacing a non-terminal with the body of a rule associated with it. Recall that all bound rules contain a term in Q_G as the right-component of their head. For a pair to be rewritable, both components of that pair must match those of some bound rule's head pair. This often means that rewrite steps must be applied to the left or right-component of a pair to prepare that pair for rewriting with an instantiated rule.

The following definitions build the foundations for constructing derivations with recursive adaptable grammars.

Definition 20. For a RAG G and answer $a \in A_G$, $\beta_G(a)$ denotes the following set of bound rules:

$$\beta_G(a) = \{\langle a, q\rangle \to c \mid (\langle a, q\rangle \to c) \in \beta_G(r) \, \text{for some} \, r \in \rho_G(a)\}$$

The union of $\beta_G(a)$ for all answers $a \in A_G$ is denoted $\beta(G)$.

The set of bound rules for an answer a is the set of all rules whose head pair's left-component is also a. This is similar to saying that the rule set for a non-terminal in a context-free grammar is the set of rules whose left-hand side is that non-terminal. A pair $\langle a, c\rangle$ can only be rewritten using a rule r if $r \in \beta_G(a)$.

Definition 21. The *derivation step relation* for a RAG G is the minimal binary relation $\underset{G}{\Longrightarrow}$ over C_G satisfying the following axioms. Throughout the axioms, a, q, and c, with or without subscripts or primes, are assumed to be universally quantified over A_G, Q_G and C_G, respectively.

Axiom 1. If $(\langle a, q\rangle \to c) \in \beta(G)$ then $\langle a, !q\rangle \underset{G}{\Longrightarrow} c$.

Axiom 2. Suppose $\sigma \neq !$ is an operator of arity n over C_G, and for some $1 \leq k \leq n$, $c_k \underset{G}{\Longrightarrow} c'_k$. Let $c'_j = c_j$ for all $j \neq k$. Then $\sigma(c_1, ..., c_n) \underset{G}{\Longrightarrow} \sigma(c'_1, ..., c'_n)$.

Axiom 3. If $c_1 \underset{G}{\Longrightarrow} c_2$ then $!c_2 \underset{G}{\Longrightarrow} !c_1$.

Axiom 4. $(a_1?!(\langle a_1, a_2\rangle)) \underset{G}{\Longrightarrow} a_2$.

The first axiom describes the application of instanced rules in a derivation step. Note that the query q in the right-component of the left-hand side pair is written under inverse in the derivation step.

Axiom 2 defines the conditions for embedded rewriting, denoting that if we have a derivation step $x \underset{G}{\Longrightarrow} y$ then we also have $pxq \underset{G}{\Longrightarrow} pyq$.

Axiom 3 defines the behaviour of the inverse operator, primarily used in conjunction with Axiom 4 for query rewriting.

Axiom 4 allows the rewriting of queries to answers, and vice-versa under the inverse operator.

Definition 22. The *derivation relation* for a RAG G is the reflexive transitive closure of $\underset{G}{\Longrightarrow}$, denoted $\overset{*}{\underset{G}{\Longrightarrow}}$. The transitive closure of $\underset{G}{\Longrightarrow}$ is denoted $\overset{+}{\underset{G}{\Longrightarrow}}$. The *language generated* by an answer $a \in A_G$ is $L_G(a) = \{x \in A_G \,|\, \langle a, y\rangle \overset{*}{\underset{G}{\Longrightarrow}} x$ for some $y \in A_G\}$. The language accepted by G is $L_G(s_G)$. The (in general, multivalued) partial function *computed* by an anwer $a \in A_G$ is the function $f_a(x) = y$ iff $\langle a, y\rangle \overset{*}{\underset{G}{\Longrightarrow}} x$. The partial function computed by G is $f_G = f_{s_G}$.

4.5 Elementary Derivation with RAGs

Given the definitions in §4.4 we can begin to illustrate how to build RAGs to specify languages. The first simple examples show how to build languages without use of the query operator. That is, simply rewriting pairs whose left-component is an answer. This is to aid understanding of the elementary mechanics of pair rewriting.

We can first explore a language Shutt used in his original thesis, defined by the set $\{www \,|\, w \in [a - c]^*\}$. In other words, a concatenation of three identical strings over the language $\{a, b, c\}^*$. This language is not context-free, however we can easily specify this language using RAGs.

Example 17. Consider the following RAG.

$$\langle S,\, v_3\rangle \to \langle W,\, v_1\rangle\, \langle v_1,\, v_2\rangle\, \langle v_1,\, v_3\rangle \tag{1}$$

$$\langle W,\, \lambda\rangle \to \lambda \tag{2}$$

$$\langle W,\, \mathtt{a}\, v_1\rangle \to \mathtt{a}\, \langle W,\, v_1\rangle \tag{3}$$

$$\langle W,\, \mathtt{b}\, v_1\rangle \to \mathtt{b}\, \langle W,\, v_1\rangle \tag{4}$$

$$\langle W,\, \mathtt{c}\, v_1\rangle \to \mathtt{c}\, \langle W,\, v_1\rangle \tag{5}$$

The terminal constants are $\{\lambda, \mathtt{a}, \mathtt{b}, \mathtt{c}\}$, and the non-terminal constants are $\{S, W\}$, and the start symbol is S. There are no non-nullary operators other than concatenation.

The answer W accepts the language $[a - c]^*$, and associates each string with a semantic value matching its syntax. By use of W in rule (1), the answer S generates the language $\{www \,|\, w \in [a - c]^*\}$. The variable v_1 in this rule is used as the left-component of the second and third pairs in the body. The effect of this is that these pairs generate syntax matching the semantic value computed by rewriting the first pair, $\langle W, v_1\rangle$.

Rule (1) has v_3 as its semantic result. By observation of the rules for W, we see that v_1 in the first pair must be bound to a terminal in the language $[a - c]^*$ for these rules to be applicable. Therefore, by use of the implicit rule $\forall t \in T_G, \langle t, t\rangle \to t$, we know that both v_2 and v_3 must also be the same terminal, for each pair to generate the string associated with v_1. In other words, to generate a string www, with $w \in [a-c]^*$, v_1, v_2, v_3 must all be bound to w.

Using the specification defined in Ex. 18, we derive the string aaa with the following derivation steps.

$$\begin{aligned}
&\langle S, \mathtt{a}\rangle \\
&\Rightarrow \langle W, \mathtt{a}\rangle\langle \mathtt{a}, \mathtt{a}\rangle\langle \mathtt{a}, \mathtt{a}\rangle \\
&\Rightarrow \mathtt{a}\langle W, \lambda\rangle\langle \mathtt{a}, \mathtt{a}\rangle\langle \mathtt{a}, \mathtt{a}\rangle \\
&\Rightarrow \mathtt{a}\langle \mathtt{a}, \mathtt{a}\rangle\langle \mathtt{a}, \mathtt{a}\rangle \\
&\Rightarrow \mathtt{aa}\langle \mathtt{a}, \mathtt{a}\rangle \\
&\Rightarrow \mathtt{aaa}
\end{aligned}$$

The semantic value associated with the string aaa is a. Note that by Axiom 1 of the derivation step relation, the right-component of each pair in a derivation should be under inverse. However, since $\forall a \in A_G, a = !a$, the ! can be omitted for each pair in this particular derivation, as each right-component is an answer.

In fact, we can extend this RAG to generate strings using all letters of the English alphabet without needing to write rules following the structure of rules (3), (4) and (5) for each letter. By use of a *typed variable*, the set of unbound rules for an answer can be extended to cover both finite and infinite cases. For this case, we can build a single rule which specifies 26 rules using a typed variable.

Example 18. Consider the following RAG.

$$Z ::= [\mathtt{a} - \mathtt{z}]$$

$$\langle S, v_4\rangle \to \langle W, v_1\rangle\ \langle v_1, v_2\rangle\ \langle v_1, v_3\rangle \tag{1}$$

$$\langle W, \lambda\rangle \to \lambda \tag{2}$$

$$\langle W, v_1\ v_2\rangle \to \langle C, v_1\rangle\ \langle W, v_2\rangle \tag{3}$$

$$z : Z \quad \langle C, z\rangle \to \langle z, v_1\rangle \tag{4}$$

The context-free rule $Z ::= [\mathtt{a} - \mathtt{z}]$ generates the language $\{\mathtt{a}, \mathtt{b}, ..., \mathtt{z}\}$, external to the RAG formalism.

The terminal constants are $\{\lambda, \mathtt{a}, \mathtt{b}, ..., \mathtt{z}\}$, and the non-terminal constants are $\{S, W, C\}$, and the start symbol is S.

Rule (4) specifies a set of 26 bound rules, each substituting z by a letter of the alphabet $\{\mathtt{a}, ..., \mathtt{z}\}$, since z is defined as ranging over the alphabet Z. In effect, with this single rule we have defined the bound rules:

$$\langle C, \mathtt{a}\rangle \to \langle \mathtt{a}, v_1\rangle$$

$$\langle C, \mathtt{b}\rangle \to \langle \mathtt{b}, v_1\rangle$$

$$...$$

$$\langle C, \mathtt{z}\rangle \to \langle \mathtt{z}, v_1\rangle$$

Thus the answer C generates the language $[\mathtt{a}-\mathtt{z}]$.

Rule (3) for the answer W generates a single letter in Z by use of non-terminal C, and concatenates it with a string in $[\mathtt{a}-\mathtt{z}]^*$ by recursive use of W. As in Ex. 17, each rule for W generates a semantic result matching the syntax it generates.

Using the specification defined in Ex. 18, we derive the string xyxyxy with the following derivation steps.

$$\begin{array}{lr}
\langle S, \mathtt{xy}\rangle & \\
\Rightarrow \langle W, \mathtt{xy}\rangle\langle \mathtt{xy}, \mathtt{xy}\rangle\langle \mathtt{xy}, \mathtt{xy}\rangle & 1\\
\Rightarrow \langle C, \mathtt{x}\rangle\langle W, \mathtt{y}\rangle\langle \mathtt{xy}, \mathtt{xy}\rangle\langle \mathtt{xy}, \mathtt{xy}\rangle & 2\\
\Rightarrow \langle \mathtt{x}, \mathtt{x}\rangle\langle W, \mathtt{y}\rangle\langle \mathtt{xy}, \mathtt{xy}\rangle\langle \mathtt{xy}, \mathtt{xy}\rangle & 3\\
\Rightarrow \mathtt{x}\langle W, \mathtt{y}\rangle\langle \mathtt{xy}, \mathtt{xy}\rangle\langle \mathtt{xy}, \mathtt{xy}\rangle & 4\\
\Rightarrow \mathtt{x}\langle C, \mathtt{y}\rangle\langle W, \lambda\rangle\langle \mathtt{xy}, \mathtt{xy}\rangle\langle \mathtt{xy}, \mathtt{xy}\rangle & 5\\
\Rightarrow \mathtt{x}\langle \mathtt{y}, \mathtt{y}\rangle\langle W, \lambda\rangle\langle \mathtt{xy}, \mathtt{xy}\rangle\langle \mathtt{xy}, \mathtt{xy}\rangle & 6\\
\Rightarrow \mathtt{xy}\langle W, \lambda\rangle\langle \mathtt{xy}, \mathtt{xy}\rangle\langle \mathtt{xy}, \mathtt{xy}\rangle & 7\\
\Rightarrow \mathtt{xy}\langle \mathtt{xy}, \mathtt{xy}\rangle\langle \mathtt{xy}, \mathtt{xy}\rangle & 8\\
\Rightarrow \mathtt{xyxy}\langle \mathtt{xy}, \mathtt{xy}\rangle & 9\\
\Rightarrow \mathtt{xyxyxy} & 10
\end{array}$$

The semantic value associated with the string xyxyxy is xy.

4.6 Query Evaluation

Recall that a query $(x?y)$ rewrites to the semantic value associated with the derivation of syntax y from answer x. Provided below is a simple example of how queries can be used to generate syntax.

Example 19. Consider the following RAG.

$$\langle S, \lambda\rangle \to \langle A, v_1\rangle\, \langle (B\ ?\ v_1), v_2\rangle\, \langle (C\ ?\ v_1), v_3\rangle \tag{1}$$

$$\langle A, \lambda\rangle \to \lambda \tag{2}$$

$$\langle A, \mathtt{a}\, v_1\rangle \to \mathtt{a}\, \langle A, v_1\rangle \tag{3}$$

$$\langle B, \lambda\rangle \to \lambda \tag{4}$$

$$\langle B, \mathtt{b}\, v_1\rangle \to \mathtt{a}\, \langle B, v_1\rangle \tag{5}$$

$$\langle C, \lambda\rangle \to \lambda \tag{6}$$

$$\langle C, \mathtt{c}\, v_1\rangle \to \mathtt{a}\, \langle C, v_1\rangle \tag{7}$$

The terminal constants are $\{\lambda, \mathtt{a}, \mathtt{b}, \mathtt{c}\}$, and the non-terminal constants are $\{S, A, B, C\}$, and the start symbol is S.

The non-terminal S generates the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ for $n \geq 0$.

The answers A, B, C all generate strings in the language $\mathtt{a}^n$ for $n \geq 0$, but associate these strings with the semantic value $\mathtt{a}^n, \mathtt{b}^n, \mathtt{c}^n$ respectively.

In rule (1), the a's are generated by the pair $\langle A, v_1\rangle$ in rule (1), synthesising the same string of a's as the semantic value v_1 is bound to.

The pair $\langle (B \,?\, v_1), v_2\rangle$ generates a string of b's using a query, where v_1 is substituted for a string of a's. Since non-terminal B generates $\mathtt{a}^n$ but synthesises $\mathtt{b}^n$, the query will derive $\mathtt{b}^n$. Then the implicit rule $\forall t \in T_G, \langle t, t\rangle \to t$ is used to rewrite the resulting pair, binding v_2 to $\mathtt{b}^n$. This effect is also used to generate $\mathtt{c}^n$ by use of the pair $\langle (C \,?\, v_1), v_3\rangle$.

Using the specification defined in Ex. 19, we derive the string abc with the following derivation steps.

$$\begin{array}{ll}
\langle S, \lambda\rangle & \\
\Rightarrow \langle A, \mathtt{a}\rangle\langle (B \,?\, \mathtt{a}), \mathtt{b}\rangle\langle (C \,?\, \mathtt{a}), \mathtt{c}\rangle & 1 \\
\Rightarrow \mathtt{a}\langle A, \lambda\rangle\langle (B \,?\, \mathtt{a}), \mathtt{b}\rangle\langle (C \,?\, \mathtt{a}), \mathtt{c}\rangle & 2 \\
\Rightarrow \mathtt{a}\langle (B \,?\, \mathtt{a}), \mathtt{b}\rangle\langle (C \,?\, \mathtt{a}), \mathtt{c}\rangle & 3 \\
\Rightarrow \mathtt{a}\langle (B \,?\, !(\mathtt{a}\langle B, \lambda\rangle)), \mathtt{b}\rangle\langle (C \,?\, \mathtt{a}), \mathtt{c}\rangle & 4 \\
\Rightarrow \mathtt{a}\langle (B \,?\, !(\langle B, \mathtt{b}\rangle)), \mathtt{b}\rangle\langle (C \,?\, \mathtt{a}), \mathtt{c}\rangle & 5 \\
\Rightarrow \mathtt{a}\langle \mathtt{b}, \mathtt{b}\rangle\langle (C \,?\, \mathtt{a}), \mathtt{c}\rangle & 6 \\
\Rightarrow \mathtt{ab}\langle (C \,?\, \mathtt{a}), \mathtt{c}\rangle & 7 \\
\Rightarrow \mathtt{ab}\langle (C \,?\, !(\mathtt{a}\langle C, \lambda\rangle)), \mathtt{c}\rangle & 8 \\
\Rightarrow \mathtt{ab}\langle (C \,?\, !(\langle C, \mathtt{c}\rangle)), \mathtt{c}\rangle & 9 \\
\Rightarrow \mathtt{ab}\langle \mathtt{c}, \mathtt{c}\rangle & 10 \\
\Rightarrow \mathtt{abc} & 11
\end{array}$$

The semantic value associated with the string abc is λ.

This derivation contains two sub-derivation sequences for queries $(B \,?\, \mathtt{a})$ and $(C \,?\, \mathtt{a})$. These queries are rewritten in steps 4 to 6 and 8 to 10 respectively. They rewrite to b and c by application of the derivation step relation. Consider the former, $(B \,?\, \mathtt{a})$.

Applying axiom 1 of the derivation step relation to rule (4) of the RAG defined in Ex. 19, we know we have a concrete step $\mathtt{a}\,\langle B, \lambda\rangle \Rightarrow \mathtt{a}$. We can take the inverse of this derivation, $!\mathtt{a} \Rightarrow !(\mathtt{a}\langle B, \lambda\rangle)$. Since for all answers x, $!x = x$, we also have the step $\mathtt{a} \Rightarrow !(\mathtt{a}\langle B, \lambda\rangle)$.

In addition, $!(\mathtt{a}\,\langle B, \lambda\rangle) \Rightarrow !(\langle B, \mathtt{b}\rangle)$ by application of rule (5) under inverse. Putting these steps together, we have a sequence $\mathtt{a} \Rightarrow !(\mathtt{a}\,\langle B, \lambda\rangle) \Rightarrow !(\langle B, \mathtt{b}\rangle)$. Using axiom 2 of the derivation step relation, if we embed these steps to rewrite the right-hand operator of our query, we get $(B \,?\, \mathtt{a}) \Rightarrow (B \,?\, !(\mathtt{a}\,\langle B, \lambda\rangle)) \Rightarrow (B \,?\, !(\langle B, \mathtt{b}\rangle))$. Then using axiom 4 of the derivation step relation, we append a new step; $(B \,?\, \mathtt{a}) \Rightarrow (B?!(\mathtt{a}\,\langle B, \lambda\rangle)) \Rightarrow (B \,?\, !(\langle B, \mathtt{b}\rangle)) \Rightarrow \mathtt{b}$. Again using axiom 2, these steps are embeded in the configuration surrounding the original query $\mathtt{a}\,\langle B, \lambda\rangle \Rightarrow \mathtt{a}$, as steps 4 to 6.

Consequently, we have derived an answer from our query. This sub-derivation sequence appears in the full sequence of steps given above as the left-component of a pair. The query $(C\,?\,\texttt{a})$ is rewritten to $\texttt{c}$ similarly.

This example has shown how queries which appear as the left-component of a pair on the right-hand side of a rule are handled in a derivation sequence. Such queries are expressions about the syntax a RAG generates. However, we can also include queries in the semantic results of rules.

Example 20. Consider the following RAG.

$$\langle S, (I \; ? \; v_1)\rangle \rightarrow \langle N, v_1\rangle \tag{1}$$

$$\langle N, \texttt{0}\rangle \rightarrow \texttt{0} \tag{2}$$

$$\langle N, \texttt{s}\, v_1\rangle \rightarrow \texttt{s}\, \langle N, v_1\rangle \tag{3}$$

$$\langle I, \texttt{s}\, v_1\rangle \rightarrow \langle N, v_1\rangle \tag{4}$$

The terminal constants for this RAG are $\{\texttt{0}, \texttt{s}\}$, and the non-terminal constants are $\{S, I, N\}$, and the start symbol is S.

The non-terminals S, N, I all generate the language of natural numbers in Peano-form, $\texttt{s}^*\texttt{0}$.

For each string in the language of N, the semantic value of the string is synthesised by rules (2) and (3), and is identical to the string.

Rule (4) for the answer I synthesises the successor of the Peano number it generates, by prepending an $\texttt{s}$ to the generated string.

Recall that derivations using the RAG defined in Ex. 19 required queries to be rewritten to answers, since the queries were elements of the left-component of pairs on the right-hand side of a rule. In the case of Ex. 20, we instead have a query in the semantic result of rule (1). Therefore, when in a derivation sequence we have a pair $\langle S, \alpha\rangle$ with $\alpha \in A_G$ we must first derive $\langle S, !(I\,?\,\alpha)\rangle$ from α itself before we can apply rule (1).

Using the specification given in Ex. 20, we derive the string $\texttt{ss0}$ with the following derivation steps.

$$\begin{array}{ll}
\langle S, \texttt{sss0}\rangle & \\
\Rightarrow \langle S, !((I\,?\,!(\langle I, \texttt{sss0}\rangle)))\rangle & 1 \\
\Rightarrow \langle S, !((I\,?\,!(\langle N, \texttt{ss0}\rangle)))\rangle & 2 \\
\Rightarrow \langle S, !((I\,?\,!(\texttt{s}\langle N, \texttt{s0}\rangle)))\rangle & 3 \\
\Rightarrow \langle S, !((I\,?\,!(\texttt{ss}\langle N, \texttt{0}\rangle)))\rangle & 4 \\
\Rightarrow \langle S, !((I\,?\,\texttt{ss0}))\rangle & 5 \\
\Rightarrow \langle N, \texttt{ss0}\rangle & 6 \\
\Rightarrow \texttt{s}\langle N, \texttt{s0}\rangle & 7 \\
\Rightarrow \texttt{ss}\langle N, \texttt{0}\rangle & 8 \\
\Rightarrow \texttt{ss0} & 9
\end{array}$$

The semantic value associated with the string $\texttt{ss0}$ is $\texttt{sss0}$.

To reiterate, when a query appears as the right-component of a pair in the body of a rule, we derive an answer from the query in derivation sequences. Contrastingly, when a query is in the semantic result of a rewrite rule, we must derive the query from an answer in order to apply the rule in question.

4.7 Further RAG Specifications

As mentioned in the discussion of the contributions of this thesis, this section gives a list of RAG examples with accompanying derivations to aid understanding.

4.7.1 Specifications for $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$

Ex. 19 specified a RAG for the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ using queries. We can also define this language using a similar RAG, but making use of non-nullary operators.

Example 21. Consider the following RAG.

$$\langle S,\lambda\rangle \to \langle A,v_1\rangle\,\langle B(v_1),v_2\rangle\,\langle C(v_1),v_3\rangle \tag{1}$$

$$\langle A,\lambda\rangle \to \lambda \tag{2}$$

$$\langle A,\mathtt{a}\,v_1\rangle \to \mathtt{a}\,\langle A,v_1\rangle \tag{3}$$

$$\langle B(\lambda),\lambda\rangle \to \lambda \tag{4}$$

$$\langle B(\mathtt{a}\,v_1),\lambda\rangle \to \mathtt{b}\,\langle B(v_1),v_2\rangle \tag{5}$$

$$\langle C(\lambda),\lambda\rangle \to \lambda \tag{6}$$

$$\langle C(\mathtt{a}\,v_1),\lambda\rangle \to \mathtt{c}\,\langle C(v_1),v_2\rangle \tag{7}$$

The terminal constants for this RAG are $\{\lambda,\mathtt{a},\mathtt{b},\mathtt{c}\}$, the non-terminal constants are $\{S,A\}$, and $\{B,C\}$ are unary operators. The start symbol is S.

The answer A generates syntax in $\mathtt{a}^n$ for $n \geq 0$ and synthesises an identical semantic result. For $n \geq 0$, $B(\mathtt{a}^n)$ generates the syntax $\mathtt{b}^n$, and $C(\mathtt{a}^n)$ generates $\mathtt{c}^n$. Rules (5) and (7) can also be used to rewrite a pair whose left-component is $B(\mathtt{a}\alpha)$ and $C(\mathtt{a}\alpha)$ respectively, where α is a terminal other than $\mathtt{a}^n$. However, after this the derivation becomes stuck, as the pairs in both rules would not be rewritable.

When the argument to a B is λ, then that answer generates λ by rule (4). On the other hand when the argument is a string in $\mathtt{a}^+$, rule (5) recognises the leading $\mathtt{a}$ and generates a $\mathtt{b}$. The remainder of the string of $\mathtt{a}'s$ is used as the parameter to another B answer. Thus the answer $B(\mathtt{a}^k)$ generates $\mathtt{b}^k$ for all $k \geq 0$. This same pattern is applied for the answer C by rules (6) and (7), where $C(\mathtt{a}^k)$ generates $\mathtt{c}^k$.

Consequently, the answer S generates the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ by use of operators A, B, C.

Using the specification defined in Ex. 21, we derive the string aabbcc with the following derivation steps.

$$
\begin{array}{ll}
\langle S, \lambda\rangle & \\
\Rightarrow \langle A, \mathtt{aa}\rangle\langle B(\mathtt{aa}), \lambda\rangle\langle C(\mathtt{aa}), \lambda\rangle & 1 \\
\Rightarrow \mathtt{a}\langle A, \mathtt{a}\rangle\langle B(\mathtt{aa}), \lambda\rangle\langle C(\mathtt{aa}), \lambda\rangle & 2 \\
\Rightarrow \mathtt{aa}\langle A, \lambda\rangle\langle B(\mathtt{aa}), \lambda\rangle\langle C(\mathtt{aa}), \lambda\rangle & 3 \\
\Rightarrow \mathtt{aa}\langle B(\mathtt{aa}), \lambda\rangle\langle C(\mathtt{aa}), \lambda\rangle & 4 \\
\Rightarrow \mathtt{aab}\langle B(\mathtt{a}), \lambda\rangle\langle C(\mathtt{aa}), \lambda\rangle & 5 \\
\Rightarrow \mathtt{aabb}\langle B(\lambda), \lambda\rangle\langle C(\mathtt{aa}), \lambda\rangle & 6 \\
\Rightarrow \mathtt{aabb}\langle C(\mathtt{aa}), \lambda\rangle & 7 \\
\Rightarrow \mathtt{aabbc}\langle C(\mathtt{a}), \lambda\rangle & 8 \\
\Rightarrow \mathtt{aabbcc}\langle C(\lambda), \lambda\rangle & 9 \\
\Rightarrow \mathtt{aabbcc} & 10
\end{array}
$$

The semantic value associated with the string aabbcc is λ.

Contrasting the RAG specification given in Ex. 19, this example does the counting of a's in the arguments to answers. In Ex. 19, the a's were counted as the syntax of a query to B or C to rewrite to the b's or c's.

In fact, the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ can be specified even more simply, by using a RAG without queries or non-nullary operators.

Example 22. Consider the following RAG.

$$\langle S, \lambda\rangle \rightarrow \langle A, v_1\rangle\ \langle v_1, v_2\rangle \tag{1}$$

$$\langle A, \lambda\rangle \rightarrow \lambda \tag{2}$$

$$\langle A, \mathtt{b}\, v_1\, \mathtt{c}\rangle \rightarrow \mathtt{a}\ \langle A, v_1\rangle \tag{3}$$

The terminal constants for this RAG are $\{\lambda, \mathtt{a}, \mathtt{b}, \mathtt{c}\}$, and the non-terminal constants are $\{S, A\}$, and the start symbol is S.

Non-terminal A generates syntax $\mathtt{a}^n$ for $n \geq 0$, and synthesises as its semantic result the language $\mathtt{b}^n\mathtt{c}^n$. The semantic result of the rules for A is used as the left-component of the second right-hand side pair in rule (1). Consequently, the concatenation of $\langle A, v_1\rangle$ and $\langle v_1, v_2\rangle$ in rule (1) for S constructs the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$.

Using the specification defined in Ex. 22, we derive the string aabbcc with the following derivation steps.

$$
\begin{array}{ll}
\langle S, \lambda\rangle & \\
\Rightarrow \langle A, \mathtt{bbcc}\rangle\langle \mathtt{bbcc}, \mathtt{bbcc}\rangle & 1 \\
\Rightarrow \mathtt{a}\langle A, \mathtt{bc}\rangle\langle \mathtt{bbcc}, \mathtt{bbcc}\rangle & 2 \\
\Rightarrow \mathtt{aa}\langle A, \lambda\rangle\langle \mathtt{bbcc}, \mathtt{bbcc}\rangle & 3 \\
\Rightarrow \mathtt{aa}\langle \mathtt{bbcc}, \mathtt{bbcc}\rangle & 4 \\
\Rightarrow \mathtt{aabbcc} & 5
\end{array}
$$

The semantic value associated with the string `aabbcc` is λ. Unfortunately, this same RAG style cannot be extended to support the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n\mathtt{d}^n$.

4.7.2 Peano Arithmetic

Computing the dynamic semantics of arithmetic operations over the numerals is an interesting problem to solve using a RAG. For the simplicity of the following examples, instead of using the arabic numerals we will use numbers in Peano-form, $0 \to 0, 1 \to s0, 2 \to ss0$ etc. We can do addition over these numerals by removing a leading s from the right-hand operand and pre-pending one to the left-hand operand. For example, $s0 + ss0 = ss0 + s0 = sss0 + 0 = sss0$.

Example 23. Consider the following RAG.

$$\langle S, (R\,?\,v_1 \,\mathtt{+}\, v_2)\rangle \to \langle N, v_1\rangle \,\mathtt{+}\, \langle S, v_2\rangle \tag{1}$$

$$\langle S, v_1\rangle \to \langle N, v_1\rangle \tag{2}$$

$$\langle N, \mathtt{s}\, v_1\rangle \to \mathtt{s}\,\langle N, v_1\rangle \tag{3}$$

$$\langle N, \mathtt{0}\rangle \to \mathtt{0} \tag{4}$$

$$\langle R, v_1\rangle \to \langle N, v_1\rangle \,\mathtt{+}\, \mathtt{0} \tag{5}$$

$$\langle R, (R\,?\,(I\,?\,v_1) \,\mathtt{+}\, (D\,?\,v_2))\rangle \to \langle N, v_1\rangle \,\mathtt{+}\, \langle N, v_2\rangle \tag{6}$$

$$\langle I, \mathtt{s}\, v_1\rangle \to \langle N, v_1\rangle \tag{7}$$

$$\langle D, v_1\rangle \to \mathtt{s}\,\langle N, v_1\rangle \tag{8}$$

The terminal constants for this RAG are $\{\mathtt{0}, \mathtt{s}, \mathtt{+}\}$, and the non-terminal constants are $\{S, N, R, I, D\}$, and the start symbol is A.

The answer N generates the set of natural numbers in Peano-form, $\{\mathtt{0}, \mathtt{s}\,\mathtt{0}, \mathtt{s}\,\mathtt{s}\,\mathtt{0}, ...\}$. Rules (2) and (3) for N synthesise each Peano number generated as their semantic values.

I generates the same set of numbers. However, rule (7) synthesises the Peano number computed by incrementing each generated number by 1 as its semantic result.

D generates a subset of the Peano naturals by removing `0`. Rule (8) for D synthesises as its semantics the Peano number computed by decrementing each number generated.

Non-terminal R generates the language of binary addition expressions over the Peano naturals. The semantic result of rules (5) and (6) for R is the expression computed by removing the leading `s`'s from the right-operand, and pre-pending them to the left-operand. A recursive query on R is used to do so, with the base-case being specified by rule (5). Such recursion in the semantic result of a rule is somewhat akin to computing with recursive functions in programming.

The answer S generates the language of addition over the Peano naturals. Rule (1) for S synthesises a query on R to compute the result of the expression generated. Rule (2) handles the case where the input is a single Peano numeral.

Using the specification defined in Ex. 23, we derive the string $\mathsf{s\,s\,0 + s\,s\,0}$ with the following derivation steps.

$$
\begin{array}{ll}
\langle S, \mathsf{s\,0}\rangle & \\
\Rightarrow \langle S, !((R\,?\,!(\langle R, \mathsf{s\,0}\rangle))))\rangle & 1 \\
\Rightarrow \langle S, !((R\,?\,!(\langle R, !((R\,?\,!(\langle R, \mathsf{s\,0}\rangle))))\rangle))))\rangle & 2 \\
\Rightarrow \langle S, !((R\,?\,!(\langle R, !((R\,?\,!(\langle N, \mathsf{s\,0}\rangle + \mathsf{0}))))\rangle))))\rangle & 3 \\
\Rightarrow \langle S, !((R\,?\,!(\langle R, !((R\,?\,!(\mathsf{s}\,\langle N, \mathsf{0}\rangle + \mathsf{0}))))\rangle))))\rangle & 4 \\
\Rightarrow \langle S, !((R\,?\,!(\langle R, !((R\,?\,\mathsf{s\,0} + \mathsf{0}))\rangle))))\rangle & 5 \\
\Rightarrow \langle S, !((R\,?\,!(\langle R, !((R\,?\,(I\,?\,!(\langle I, \mathsf{s\,0}\rangle)) + \mathsf{0}))\rangle))))\rangle & 6 \\
\Rightarrow \langle S, !((R\,?\,!(\langle R, !((R\,?\,(I\,?\,!(\langle N, \mathsf{0}\rangle)) + \mathsf{0}))\rangle))))\rangle & 7 \\
\Rightarrow \langle S, !((R\,?\,!(\langle R, !((R\,?\,(I\,?\,\mathsf{0}) + \mathsf{0}))\rangle))))\rangle & 8 \\
\Rightarrow \langle S, !((R\,?\,!(\langle R, !((R\,?\,(I\,?\,\mathsf{0}) + (D\,?\,!(\langle D, \mathsf{0}\rangle)))))\rangle))))\rangle & 9 \\
\Rightarrow \langle S, !((R\,?\,!(\langle R, !((R\,?\,(I\,?\,\mathsf{0}) + (D\,?\,!(\mathsf{s}\,\langle N, \mathsf{0}\rangle)))))\rangle))))\rangle & 10 \\
\Rightarrow \langle S, !((R\,?\,!(\langle R, !((R\,?\,(I\,?\,\mathsf{0}) + (D\,?\,\mathsf{s\,0})))\rangle))))\rangle & 11 \\
\Rightarrow \langle S, !((R\,?\,!(\langle N, \mathsf{0}\rangle + \langle N, \mathsf{s\,0}\rangle))))\rangle & 12 \\
\Rightarrow \langle S, !((R\,?\,!(\mathsf{0} + \langle N, \mathsf{s\,0}\rangle))))\rangle & 13 \\
\Rightarrow \langle S, !((R\,?\,!(\mathsf{0} + \mathsf{s}\,\langle N, \mathsf{0}\rangle))))\rangle & 14 \\
\Rightarrow \langle S, !((R\,?\,\mathsf{0} + \mathsf{s\,0}))\rangle & 15 \\
\Rightarrow \langle N, 0\rangle + \langle S, \mathsf{s\,0}\rangle & 16 \\
\Rightarrow \mathsf{0} + \langle S, \mathsf{s\,0}\rangle & 17 \\
\Rightarrow \mathsf{0} + \langle N, \mathsf{s\,0}\rangle & 18 \\
\Rightarrow \mathsf{0} + \mathsf{s}\,\langle N, \mathsf{0}\rangle & 19 \\
\Rightarrow \mathsf{0} + \mathsf{s\,0} & 20
\end{array}
$$

The semantic value associated with the string $\mathsf{0 + s\,0}$ is $\mathsf{s\,0}$.

Step 1 involves the use of axiom 4 of the derivation step relation; we know we have a step $(R\,?\,!(\langle R, \mathsf{s\,0}\rangle)) \Rightarrow \mathsf{s\,0}$. This step can be reversed using axiom 3 with both sides under inverse to get $!\mathsf{s\,0} \Rightarrow !(R\,?\,!(\langle R, \mathsf{s\,0}\rangle))$, and consequently $\mathsf{s\,0} \Rightarrow !(R\,?\,!(\langle R, \mathsf{s\,0}\rangle))$, since $!a = a$ for all answers a. Using axiom 2, this step is embedded as the right component of the pair $\langle S, ...\rangle$.

With step 2, we again use axioms 4 and 3 of the derivation step relation to construct the single step $!(\langle R, \mathsf{s\,0}\rangle) \Rightarrow !((R\,?\,!(\langle R, \mathsf{s\,0}\rangle)))$. The result of this step is embeded in place of $!(\langle R, \mathsf{s\,0}\rangle)$ in the new configuration.

In steps 3 to 5 we make use of axiom 1 and rules (5), (3) and (4) respectively. Firstly, by rule (5) we know we have a step $\langle R, \mathsf{s\,0}\rangle \Rightarrow \langle N, \mathsf{s\,0}\rangle + \mathsf{0}$. Then by rule (3) we have the derivation step $\langle N, \mathsf{s\,0}\rangle \Rightarrow \mathsf{s}\,\langle N, \mathsf{0}\rangle$. We can embed the latter step into the former using axiom 2 to get the sequence

$\langle R, \mathtt{s}\,\mathtt{0}\rangle \Rightarrow \langle N, \mathtt{s}\,\mathtt{0}\rangle \mathbin{\mathtt{+}} \mathtt{0} \Rightarrow \mathtt{s}\,\langle N, \mathtt{0}\rangle \mathbin{\mathtt{+}} \mathtt{0}$. Then rule (4) is used to embed the step $\langle N, \mathtt{0}\rangle \Rightarrow \mathtt{0}$ into this sequence to get $\langle R, \mathtt{s}\,\mathtt{0}\rangle \Rightarrow \langle N, \mathtt{s}\,\mathtt{0}\rangle \mathbin{\mathtt{+}} \mathtt{0} \Rightarrow \mathtt{s}\,\langle N, \mathtt{0}\rangle \mathbin{\mathtt{+}} \mathtt{0} \Rightarrow \mathtt{s}\,\mathtt{0} \mathbin{\mathtt{+}} \mathtt{0}$. Using axiom 2 again, this sequence is embedded in place of $\langle R, \mathtt{s}\,\mathtt{0}\rangle$ to construct steps 3 to 5.

Steps 6 to 8 make use of the axioms and rules (7) and (4) to construct the sequence $\mathtt{s}\,\mathtt{0} \Rightarrow (I\,?\,!(\langle I, \mathtt{s}\,\mathtt{0}\rangle)) \Rightarrow (I\,?\,!(\langle N, \mathtt{0}\rangle)) \Rightarrow (I\,?\,\mathtt{0})$ which is embedded in our larger derivation sequence. In a similar way using rules (5) and (3), steps 9 to 11 derive the query $(D\,?\,\mathtt{s}\,\mathtt{0})$ from $\mathtt{0}$.

Step 12 uses axiom 1 with rule (6) to construct the derivation step $\langle R, !((R\,?\,(I\,?\,\mathtt{0}) \mathbin{\mathtt{+}} (D\,?\,\mathtt{s}\,\mathtt{0})))\rangle \Rightarrow \langle N, \mathtt{0}\rangle \mathbin{\mathtt{+}} \langle N, \mathtt{s}\,\mathtt{0}\rangle$. Using axiom 2 this step is embedded into the main sequence.

Steps 13 to 15 simply use rules (3) and (4) of the RAG specification to rewrite the pairs whose left-component is the answer N.

Then, by rule (1) and axiom 1 we have the step $\langle S, !((R\,?\,\mathtt{0} \mathbin{\mathtt{+}} \mathtt{s}\,\mathtt{0}))\rangle \Rightarrow \langle N, \mathtt{0}\rangle \mathbin{\mathtt{+}} \langle S, \mathtt{s}\,\mathtt{0}\rangle$ which can be embedded into the main sequence using axiom 2. Finally, we can simply derive the string $\mathtt{0} \mathbin{\mathtt{+}} \mathtt{s}\,\mathtt{0}$ using axiom 1 to do simple rule applications with steps 17 to 20.

As shown by this example derivation, large sequences can be reasoned with by considering the small steps which are embedded within them. Axiom 2 of the derivation step relation allows us to think about the small sub-rewrites which make up a single step in a complicated derivation. The notion of reducing steps over large terms to smaller sub-steps which are simply embedded in those terms makes understanding derivation sequences much simpler.

Example 24. We can also construct a smaller and simpler RAG to solve the same problem with only two non-terminal operators. Consider the following RAG.

$$\langle S, v_1\, v_2\, \mathtt{0}\rangle \to \langle E, v_1\rangle\, \mathtt{0} \mathbin{\mathtt{+}} \langle E, v_2\rangle\, \mathtt{0} \tag{1}$$

$$\langle E, \lambda\rangle \to \lambda \tag{2}$$

$$\langle E, \mathtt{s}\, v_1\rangle \to \mathtt{s}\,\langle E, v_1\rangle \tag{3}$$

The terminal constants for this RAG are $\{\mathtt{0}, \mathtt{s}, \mathtt{+}\}$, and the non-terminal constants are $\{S, E\}$, and the start symbol is S.

The answer E generates a possibly empty sequence of $\mathtt{s}$'s, and rules (2) and (3) for E synthesise each string generated as their semantic values.

Non-terminal S generates an addition expression over the Peano naturals and synthesises the result of the expression.

Using the specification defined in Ex. 24, we derive the string $\mathtt{s}\,\mathtt{s}\,\mathtt{0} \mathbin{\mathtt{+}} \mathtt{s}\,\mathtt{s}\,\mathtt{0}$ with the following derivation steps.

$$\begin{array}{ll}
\langle \texttt{s}\,, \texttt{s s s s 0}\rangle & \\
\Rightarrow \langle E, \texttt{s s}\rangle \texttt{0} + \langle E, \texttt{s s}\rangle \texttt{0} & 1 \\
\Rightarrow \texttt{s}\,\langle E, \texttt{s}\rangle\ \texttt{0} + \langle E, \texttt{s s}\rangle\ \texttt{0} & 2 \\
\Rightarrow \texttt{s s}\,\langle E, \lambda\rangle\ \texttt{0} + \langle E, \texttt{s s}\rangle\ \texttt{0} & 3 \\
\Rightarrow \texttt{s s 0} + \langle E, \texttt{s s}\rangle\ \texttt{0} & 4 \\
\Rightarrow \texttt{s s 0} + \texttt{s}\,\langle E, \texttt{s}\rangle\ \texttt{0} & 5 \\
\Rightarrow \texttt{s s 0} + \texttt{s s}\,\langle E, \lambda\rangle\ \texttt{0} & 6 \\
\Rightarrow \texttt{s s 0} + \texttt{s s 0} &
\end{array}$$

The semantic value associated with the string $\texttt{s s 0 + s s 0}$ is $\texttt{s s s s 0}$.

This derivation is much simpler to understand for the string $\texttt{s s 0 + s s 0}$, when compared to the derivation using the RAG given in Ex. 23. There are no queries to rewrite, and the computation of semantics simply results from semantic values which are answers. From an implementation perspective, this may also result in a much smaller use of computational resources - no recursive functions are required for derivations.

4.7.3 Infix-to-Postfix

We can also use a RAG to convert expressions in infix format to a postfix format. The syntax recognised by such a RAG would be an infix expression, and the semantic value of each such syntactic value would be its postfix equivalent. The specification in Ex. 25 does so with expressions over $\texttt{a}, \texttt{b}$ and binary operations $\texttt{+}, \texttt{*}$.

Example 25. Consider the following RAG.

$$\langle S, v_1\, v_2\, \texttt{+}\rangle \to \langle E, v_1\rangle\, \texttt{+}\, \langle S, v_2\rangle \tag{1}$$

$$\langle S, v_1\rangle \to \langle E, v_1\rangle \tag{2}$$

$$\langle E, v_1\, v_2\, \texttt{*}\,\rangle \to \langle T, v_1\rangle\, \texttt{*}\, \langle E, v_2\rangle \tag{3}$$

$$\langle E, v_1\rangle \to \langle T, v_1\rangle \tag{4}$$

$$\langle T, \texttt{a}\rangle \to \texttt{a} \tag{5}$$

$$\langle T, \texttt{b}\rangle \to \texttt{b} \tag{6}$$

$$\langle T, v_1\rangle \to \texttt{(}\ \langle S, v_1\rangle\ \texttt{)} \tag{7}$$

The terminal constants for this RAG are $\{\texttt{a}, \texttt{b}, \texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}\}$, the non-terminal constants are $\{S, E, T\}$, and the start symbol is S.

Non-terminal S generates the language of expressions with operations $\texttt{+}, \texttt{*}$ with operands $\texttt{a}, \texttt{b}$, and parenthesised expressions. Rules (1) and (2) for S synthesise each generated expression's postfix equivalent as their semantic values.

The answer E generates expressions with $\texttt{*}$ and operands $\texttt{a}, \texttt{b}$, as well as parenthesised expressions with $\texttt{+}, \texttt{*}$ by use of T.

Answer T generates `a`, `b`, and parenthesised expressions using `+`, `*` with operands `a` and `b`.

Using the specification defined in Ex. 25, we derive the string `(a+a*b)+b` with the following derivation steps.

$$
\begin{array}{ll}
\langle S, \texttt{aab**+b+}\rangle & \\
\Rightarrow \langle E, \texttt{aab*+}\rangle \,\texttt{+}\, \langle S, \texttt{b}\rangle & 1 \\
\Rightarrow \langle T, \texttt{aab*+}\rangle \,\texttt{+}\, \langle S, \texttt{b}\rangle & 2 \\
\Rightarrow \texttt{(}\langle S, \texttt{aab*+}\rangle\texttt{)} \,\texttt{+}\, \langle S, \texttt{b}\rangle & 3 \\
\Rightarrow \texttt{(}\langle E, \texttt{a}\rangle \,\texttt{+}\, \langle S, \texttt{ab*}\rangle\texttt{)} \,\texttt{+}\, \langle S, \texttt{b}\rangle & 4 \\
\Rightarrow \texttt{(}\langle T, \texttt{a}\rangle \,\texttt{+}\, \langle S, \texttt{ab*}\rangle\texttt{)} \,\texttt{+}\, \langle S, \texttt{b}\rangle & 5 \\
\Rightarrow \texttt{(a+}\langle S, \texttt{ab*}\rangle\texttt{)} \,\texttt{+}\, \langle S, \texttt{b}\rangle & 6 \\
\Rightarrow \texttt{(a+}\langle E, \texttt{ab*}\rangle\texttt{)} \,\texttt{+}\, \langle S, \texttt{b}\rangle & 7 \\
\Rightarrow \texttt{(a+}\langle T, \texttt{a}\rangle \,\texttt{*}\, \langle E, \texttt{b}\rangle\texttt{)} \,\texttt{+}\, \langle S, \texttt{b}\rangle & 8 \\
\Rightarrow \texttt{(a+a*}\langle E, \texttt{b}\rangle\texttt{)} \,\texttt{+}\, \langle S, \texttt{b}\rangle & 9 \\
\Rightarrow \texttt{(a+a*}\langle T, \texttt{b}\rangle\texttt{)} \,\texttt{+}\, \langle S, \texttt{b}\rangle & 10 \\
\Rightarrow \texttt{(a+a*b)+}\langle S, \texttt{b}\rangle & 11 \\
\Rightarrow \texttt{(a+a*b)+}\langle E, \texttt{b}\rangle & 12 \\
\Rightarrow \texttt{(a+a*b)+}\langle T, \texttt{b}\rangle & 13 \\
\Rightarrow \texttt{(a+a*b)+b} &
\end{array}
$$

The semantic value associated with the string `(a+a*b)+b` is `aab**+b+`.

4.7.4 Boolean Expression Evaluation

Computing the dynamic semantics for expressions is usually a simple problem to solve using RAGs. One such simple case is for the Boolean expressions. Ex. 26 defines a RAG which generates Boolean expressions with the operations `&`, `|`, `¬`, associating each string with its Boolean result. The Booleans *True*, *False* are specified as `T`, `F` respectively.

Example 26. Consider the following RAG.

$$\langle S, v_1\rangle \to \langle E, v_1\rangle \tag{1}$$

$$\langle S, (\texttt{T} ? v_1)\rangle \to \langle E, v_1\rangle \,\texttt{|}\, \langle S, v_2\rangle \tag{2}$$

$$\langle S, (\texttt{T} ? v_2)\rangle \to \langle E, v_1\rangle \,\texttt{|}\, \langle S, v_2\rangle \tag{3}$$

$$\langle S, ((\texttt{F} ? v_1) ? v_2)\rangle \to \langle E, v_1\rangle \,\texttt{|}\, \langle S, v_2\rangle \tag{4}$$

$$\langle E, v_1\rangle \to \langle D, v_1\rangle \tag{5}$$

$$\langle E, (\texttt{F} ? v_1)\rangle \to \langle D, v_1\rangle \,\texttt{\&}\, \langle E, v_2\rangle \tag{6}$$

$$\langle E, (\texttt{F} ? v_2)\rangle \to \langle D, v_1\rangle \,\texttt{\&}\, \langle E, v_2\rangle \tag{7}$$

$$\langle E, ((\texttt{T} ? v_1) ? v_2)\rangle \to \langle D, v_1\rangle \,\texttt{\&}\, \langle E, v_2\rangle \tag{8}$$

$$\langle D, v_1 \rangle \rightarrow \langle F, v_1 \rangle \tag{9}$$

$$\langle D, (N \mathbin{?} v_1) \rangle \rightarrow \neg\, \langle F, v_1 \rangle \tag{10}$$

$$\langle F, \texttt{F} \rangle \rightarrow \texttt{F} \tag{11}$$

$$\langle F, \texttt{T} \rangle \rightarrow \texttt{T} \tag{12}$$

$$\langle F, v_1 \rangle \rightarrow \texttt{(}\, \langle S, v_1 \rangle \,\texttt{)} \tag{13}$$

$$\langle N, \texttt{F} \rangle \rightarrow \texttt{T} \tag{14}$$

$$\langle N, \texttt{T} \rangle \rightarrow \texttt{F} \tag{15}$$

The terminal constants for this RAG are $\{\texttt{T}, \texttt{F}, \texttt{|}, \texttt{\&}, \texttt{(}, \texttt{)}\}$, and the non-terminal constants are $\{S, E, D, F, N\}$, and the start symbol is S.

Non-terminal F generates `T`, `F`, and parenthesised expressions using operations ¬, |, & and operands `T`, `F`. In each rule for F, the Boolean result of each expression generated is synthesised.

The answer D generates expressions using F both unmodified, and under ¬. The Boolean result of the expression generated is synthesised.

Answer N generates a Boolean value, and synthesises its negation as the semantic value for each.

Non-terminal E generates expressions with the & operation as the top-level operator. S generates expressions with the | operation as the top-level operator. In both cases, the result of the Boolean expression is synthesised as a semantic value.

Using the specification defined in Ex. 26, we derive the string `(T|F)&T` with the following derivation steps.

$$\begin{array}{lr}
\langle S, \mathtt{T}\rangle & \\
\Rightarrow \langle E, \mathtt{T}\rangle & 1 \\
\Rightarrow \langle E, !((\mathtt{T}\ ?\ !(\langle \mathtt{T}, \mathtt{T}\rangle))))\rangle & 2 \\
\Rightarrow \langle E, !((\mathtt{T}\ ?\ \mathtt{T}))\rangle & 3 \\
\Rightarrow \langle E, !(((\mathtt{T}\ ?\ !(\langle \mathtt{T}, \mathtt{T}\rangle))\ ?\ \mathtt{T}))\rangle & 4 \\
\Rightarrow \langle E, !(((\mathtt{T}\ ?\ \mathtt{T})\ ?\ \mathtt{T}))\rangle & 5 \\
\Rightarrow \langle D, \mathtt{T}\rangle\ \mathtt{\&}\ \langle E, \mathtt{T}\rangle & 6 \\
\Rightarrow \langle F, \mathtt{T}\rangle\ \mathtt{\&}\ \langle E, \mathtt{T}\rangle & 7 \\
\Rightarrow \mathtt{(}\ \langle S, \mathtt{T}\rangle\ \mathtt{)}\ \mathtt{\&}\ \langle E, \mathtt{T}\rangle & 8 \\
\Rightarrow \mathtt{(}\ \langle S, !((\mathtt{T}\ ?\ !(\langle \mathtt{T}, \mathtt{T}\rangle))))\rangle\ \mathtt{)}\ \mathtt{\&}\ \langle E, \mathtt{T}\rangle & 9 \\
\Rightarrow \mathtt{(}\ \langle S, !((\mathtt{T}\ ?\ \mathtt{T}))\rangle\ \mathtt{)}\ \mathtt{\&}\ \langle E, \mathtt{T}\rangle & 10 \\
\Rightarrow \mathtt{(}\ \langle E, \mathtt{T}\rangle\ \mathtt{|}\ \langle S, \mathtt{F}\rangle\ \mathtt{)}\ \mathtt{\&}\ \langle E, \mathtt{T}\rangle & 11 \\
\Rightarrow \mathtt{(}\ \langle D, \mathtt{T}\rangle\ \mathtt{|}\ \langle S, \mathtt{F}\rangle\ \mathtt{)}\ \mathtt{\&}\ \langle E, \mathtt{T}\rangle & 12 \\
\Rightarrow \mathtt{(}\ \langle F, \mathtt{T}\rangle\ \mathtt{|}\ \langle S, \mathtt{F}\rangle\ \mathtt{)}\ \mathtt{\&}\ \langle E, \mathtt{T}\rangle & 13 \\
\Rightarrow \mathtt{(\ T\ |}\ \langle S, \mathtt{F}\rangle\ \mathtt{)}\ \mathtt{\&}\ \langle E, \mathtt{T}\rangle & 14 \\
\Rightarrow \mathtt{(\ T\ |}\ \langle E, \mathtt{F}\rangle\ \mathtt{)}\ \mathtt{\&}\ \langle E, \mathtt{T}\rangle & 15 \\
\Rightarrow \mathtt{(\ T\ |}\ \langle D, \mathtt{F}\rangle\ \mathtt{)}\ \mathtt{\&}\ \langle E, \mathtt{T}\rangle & 16 \\
\Rightarrow \mathtt{(\ T\ |}\ \langle F, \mathtt{F}\rangle\ \mathtt{)}\ \mathtt{\&}\ \langle E, \mathtt{T}\rangle & 17 \\
\Rightarrow \mathtt{(\ T\ |\ F\ )}\ \mathtt{\&}\ \langle E, \mathtt{T}\rangle & 18 \\
\Rightarrow \mathtt{(\ T\ |\ F\ )}\ \mathtt{\&}\ \langle D, \mathtt{T}\rangle & 19 \\
\Rightarrow \mathtt{(\ T\ |\ F\ )}\ \mathtt{\&}\ \langle F, \mathtt{T}\rangle & 20 \\
\Rightarrow \mathtt{(\ T\ |\ F\ )\ \&\ T} & 21
\end{array}$$

The semantic value associated with the string $\mathtt{(\ T\ |\ F\ )\ \&\ T}$ is $\mathtt{T}$.

Step 1 makes simple use of axiom 1 with RAG rule (1) to construct the step $\langle S, \mathtt{T}\rangle \Rightarrow \langle E, \mathtt{T}\rangle$.

Step 2 reverses the step $\mathtt{T}\ ?\ !(\langle \mathtt{T}, \mathtt{T}\rangle) \Rightarrow \mathtt{T}$ using axioms 3 and 4 to construct the step $\mathtt{T} \Rightarrow !(\mathtt{T}\ ?\ !(\langle \mathtt{T}, \mathtt{T}\rangle))$. This step is then embedded in the ongoing derivation sequence.

Since for all terminals t there is the implicit rule $\langle t, t\rangle \to t$, step 3 constructs the step $\langle \mathtt{T}, \mathtt{T}\rangle \Rightarrow \mathtt{T}$ and embeds it into the current derivation sequence.

Similarly to step 2, step 4 constructs the step $\mathtt{T} \Rightarrow !(\mathtt{T}\ ?\ !(\langle \mathtt{T}, \mathtt{T}\rangle))$ and embeds it into the derivation sequence using axiom 2. And then as in step 3, the rule $\langle \mathtt{T}, \mathtt{T}\rangle \Rightarrow \mathtt{T}$ is constructed and embedded as step 5.

To construct step 6, rule (8) is used with axiom 1 to construct the derivation step $\langle E, !(((\mathtt{T}\ ?\ \mathtt{T})\ ?\ \mathtt{T}))\rangle \Rightarrow \langle D, \mathtt{T}\rangle\ \mathtt{\&}\ \langle E, \mathtt{T}\rangle$.

Steps 7 and 8 make use of axiom 1 with rules (9) and (13) respectively to construct the derivation steps $\langle D, \mathtt{T}\rangle \Rightarrow \langle F, \mathtt{T}\rangle \Rightarrow \mathtt{(}\ \langle S, \mathtt{T}\rangle\ \mathtt{)}$. This sequence is embedded in the ongoing larger derivation sequence.

As with steps 2 and 4, we then use axioms 3 and 4 to create the step $\mathtt{T} \Rightarrow !(\mathtt{T}\ ?\ !(\langle \mathtt{T}, \mathtt{T}\rangle))$ which is embedded into the derivation sequence as step 9.

As we have seen in previous steps, the rule $\langle \texttt{T}, \texttt{T} \rangle \Rightarrow \texttt{T}$ in step 10. Following this, the remainder of the derivation sequence makes simple use of the rewrite rules for the RAG given in Ex. 26 to derive the string `(T|F)&T`.

4.7.5 String Equality

Testing for equality of strings is a fundamental requirement for semantic checks such as variable declaration before use and type compatibility of assignment. We can use RAG rules to implement such checks. Using non-nullary operators, we can test the equality of two terminal strings. We build a RAG which uses a non-nullary answer as its start symbol, whose argument is the string to test against. When a string in $[\texttt{a..z}]^*$ is generated from this answer, the RAG synthesises `T` if it matches the argument, and `F` otherwise.

Example 27. Consider the following RAG.

$$Z ::= \texttt{a..z}$$

$$\langle \mathit{Eq}(\lambda), \texttt{T} \rangle \rightarrow \lambda \tag{1}$$

$$\langle \mathit{Eq}(\lambda), \texttt{F} \rangle \rightarrow \langle \mathit{Letter}, v_1 \rangle \, \langle \mathit{Star}(\mathit{Letter}), v_2 \rangle \tag{2}$$

$$z : Z,\ t : Z^* \quad \langle \mathit{Eq}(z\,t), \texttt{F} \rangle \rightarrow \lambda \tag{3}$$

$$z : Z,\ t : Z^* \quad \langle \mathit{Eq}(z\,t), v_1 \rangle \rightarrow \langle z, v_1 \rangle \, \langle \mathit{Eq}(t), v_1 \rangle \tag{4}$$

$$\langle \mathit{Star}(a), \lambda \rangle \rightarrow \lambda \tag{5}$$

$$\langle \mathit{Star}(a), v_1\, v_2 \rangle \rightarrow \langle a, v_1 \rangle \, \langle \mathit{Star}(a), v_2 \rangle \tag{6}$$

$$z : Z \quad \langle \mathit{Letter}, \lambda \rangle \rightarrow \langle z, v_1 \rangle \tag{7}$$

The terminal constants for this RAG are $\{\lambda, \texttt{T}, \texttt{F}, \texttt{a}, ..., \texttt{z}\}$, and the non-terminal constants are $\{\mathit{Eq}, \mathit{Star}, \mathit{Letter}\}$, and the start symbol is Eq.

The answer *Letter* generates letters in the alphabet defined by Z and associates each with semantic value λ.

Unary operator *Star* takes an answer a as its parameter and generates a^n for $n \geq 0$ as syntax. For each string $\mathit{Star}(a)$ with $a \in A_G$ generates, it synthesises a matching semantic result.

Non-terminal *Eq* is a unary operator which takes an answer as its parameter. *Eq* generates a string $a \in Z^*$ and synthesises `T` if a matches the argument to *Eq*, and `F` otherwise. The pattern matching in rules (3) and (4) split each argument up into its first letter, assigned to z, and the remainder, assigned to t.

Using the specification defined in Ex. 27, we derive the string `a b` using start symbol $\mathit{Eq}(\texttt{a b})$ with the following derivation steps.

$$\begin{array}{ll}
\langle Eq(\texttt{a}\,\texttt{b}), \texttt{T}\rangle & \\
\Rightarrow \langle \texttt{a}, \texttt{a}\rangle\langle Eq(\texttt{b}), \texttt{T}\,\rangle & 1 \\
\Rightarrow \texttt{a}\,\langle Eq(\texttt{b}), \texttt{T}\,\rangle & 2 \\
\Rightarrow \texttt{a}\,\langle \texttt{b}, \texttt{b}\rangle\langle Eq(\lambda), \texttt{T}\,\rangle & 3 \\
\Rightarrow \texttt{a}\,\texttt{b}\,\langle Eq(\lambda), \texttt{T}\,\rangle & 4 \\
\Rightarrow \texttt{a}\,\texttt{b} & 5
\end{array}$$

The semantic value associated with the string $\texttt{a}\,\texttt{b}$ by the start symbol $Eq(\texttt{a}\,\texttt{b})$ is $\texttt{T}$, as $\texttt{a}\,\texttt{b} = \texttt{a}\,\texttt{b}$.

Now, a derivation which synthesises $\texttt{F}$. Using the specification defined in Ex. 27, we derive the string $\texttt{a}\,\texttt{b}\,\texttt{c}$ using start symbol $Eq(\texttt{a}\,\texttt{b})$ with the following derivation steps.

$$\begin{array}{ll}
\langle Eq(\texttt{a}\,\texttt{b}), \texttt{F}\rangle & \\
\Rightarrow \langle \texttt{a}, \texttt{a}\rangle\langle Eq(\texttt{b}), \texttt{F}\rangle & 1 \\
\Rightarrow \texttt{a}\,\langle Eq(\texttt{b}), \texttt{F}\rangle & 2 \\
\Rightarrow \texttt{a}\,\langle \texttt{b}, \texttt{b}\rangle\langle Eq(\lambda), \texttt{F}\rangle & 3 \\
\Rightarrow \texttt{a}\,\texttt{b}\,\langle Eq(\lambda), \texttt{F}\rangle & 4 \\
\Rightarrow \texttt{a}\,\texttt{b}\,\langle Letter, \lambda\rangle\langle Star(Letter), \lambda\rangle & 5 \\
\Rightarrow \texttt{a}\,\texttt{b}\,\langle \texttt{c}, \texttt{c}\rangle\langle Star(Letter), \lambda\rangle & 6 \\
\Rightarrow \texttt{a}\,\texttt{b}\,\texttt{c}\,\langle Star(Letter), \lambda\rangle & 7 \\
\Rightarrow \texttt{a}\,\texttt{b}\,\texttt{c} & 8
\end{array}$$

The semantic value associated with the string $\texttt{a}\,\texttt{b}\,\texttt{c}$ by the start symbol $Eq(\texttt{a}\,\texttt{b})$ is $\texttt{F}$, as $\texttt{a}\,\texttt{b}\,\texttt{c} \neq \texttt{a}\,\texttt{b}$.

Chapter 5

Parsing RAGs

5.1 Introduction

In this thesis we introduce a predictive parsing strategy for languages defined by RAG specifications. The aim is to determine whether a given input is a member of that language, and if so, determine the semantic value attributed to it. The process takes a breadth-first approach to parsing. A set of schemas is maintained representing all of the configurations that can be reached from the start symbol, and their semantic values with respect to the rewrite relation. Along the way, this set of schemas is culled - when we identify a schema that will never be able to recognise our input, we discard it. These schemas are rewritten using RAG rules until each consists of syntax that can be matched against the input.

The discussion of our algorithm will be example-driven, using a set of simple RAG specifications which help to explain the important concepts underpinning the tool's capability as a RAG language generator. In particular, we consider examples that illustrate how the breadth-first strategy of rewriting and input recognition works, and how queries are handled in this process.

5.2 Configurations with Unknowns

In general, an initial pair can be rewritten to an infinite number of configurations because there are infinitely many answers we can attribute as its right-component. To compensate, the tool uses schemas for building ground configurations that contain *unknowns* - artifacts of the derivations process, whose values are computed as the rewriting proceeds. These unknowns are expressions over the variables in a RAG specification, so they appear initially as the right-component of a pair and can be referenced in the remaining configuration.

To rewrite a pair $\langle \alpha, w \rangle$, the tool looks up all rules whose head pair's left-component matches α and applies all of them. Instances of the unknown, w, which appear elsewhere in each new configuration are substituted for the semantic value of the rewrite that was applied, which may also contain unknowns.

Ultimately, the tool must substitute all of the unknowns it contains so that we can construct a concrete derivation of the input. The semantic value associated with each configuration is also maintained, substituting the unknowns it contains in the same way.

These unknowns differ from the variables found in grammar rules as they are not constrained to substitute for an answer. The tool may substitute for a query, which may then be rewritable to an answer.

In the sections to follow, the notation to represent unknowns will be $w_1, w_2, w_3, ...$ to distinguish them from the variables $v_1, v_2, v_3, ...$ in RAG specifications.

5.3 Breadth-First Language Generation

The breadth-first approach parsing applied in the tool considers all possible grammar rules to rewrite a pair. We must keep track of both a configuration and its semantics, which we bind together in a tuple called a *result schema.*

Definition 23. A *result schema* is a pair (S, V) where S is a configuration term possibly containing unknowns, and V a query, also possibly containing unknowns.

The set of all result schemas maintained by the tool is called the *candidate set.* Initially, this consists of a single result schema whose configuration is the start pair. The tool then analyses the configuration of each result schema. Whenever one is found whose left-most element is a pair, $\langle \alpha, w_1 \rangle$, the tool searches for all grammar rules for α and applies each, producing a new result schema where w_1 has been substituted for the semantic result of the rule used. During this process, when the left-most component of each result schema is an answer, each of these answers is matched against the left-most symbol of a copy of the input string, which is progressively shortened. Result schemas whose leading symbol does not match successfully are discarded from the candidate set.

5.3.1 Triple-String Example

The first example used to present the rewriting strategy is simple enough to illustrate how the algorithm deals with pairs to derive syntax. This is a simplified version of a RAG Shutt presented in his thesis, which instead used the letters $\texttt{a} - \texttt{z}$.

Example 28. Consider the following RAG.

$$\langle S,\, v_3 \rangle \to \langle W,\, v_1 \rangle \, \langle v_1,\, v_2 \rangle \, \langle v_1,\, v_3 \rangle \tag{1}$$

$$\langle W,\, \lambda \rangle \to \lambda \tag{2}$$

$$\langle W,\ \mathtt{a}\, v_1\rangle \rightarrow \mathtt{a}\,\langle W,\ v_1\rangle \tag{3}$$

$$\langle W,\ \mathtt{b}\, v_1\rangle \rightarrow \mathtt{b}\,\langle W,\ v_1\rangle \tag{4}$$

$$\langle W,\ \mathtt{c}\, v_1\rangle \rightarrow \mathtt{c}\,\langle W,\ v_1\rangle \tag{5}$$

The terminal constants in this RAG are $\{\mathtt{a},\mathtt{b},\mathtt{c}\}$, the nullary non-terminal operators are $\{S, W\}$, and the start symbol is S. The only non-nullary operator is concatenation. The language defined by the rules for S is $\{www \mid w \in [\mathtt{a}-\mathtt{c}]^*\}$.

Ex. 28 defines a language that is not context-free. The rules for W rewrite to a string in the language $[\mathtt{a}-\mathtt{c}]*$, and whilst doing so, synthesize an identical semantic value. Using the variable v_1 as the left-component of the second and third pairs in rule 1, the semantic value synthesized by the rewriting of the first pair, whose left-component is W, is repeated as the syntax for the following pairs.

Example Derivation

To display the breadth-first language generation process, we choose S as our start symbol and try to derive the string $\mathtt{aaa}$. The tool begins with the following candidate set:

$$(\langle S, w_1\rangle,\ w_1) \tag{R_1}$$

The pair $\langle S, w_1\rangle$ needs to be rewritten. The tool searches for rules for the answer S and finds rule 1. Applying this rule, R_1 is discarded and a new result schema is created. The candidate set now consists of:

$$(\langle W, w_2\rangle\,\langle w_2, w_3\rangle\,\langle w_2, w_4\rangle,\ w_4) \tag{R_2}$$

The configuration $\langle S, w_1\rangle$ has been substituted for the right-hand side of rule 1, and the variables v_1, v_2, v_3 it contains are replaced by unknowns w_2, w_3, w_4. Consequently, w_1 has been substituted for the semantic result of the rule, in this case w_4, as derived from the variable v_3 in the head pair of rule 1. Now the pair $\langle W, w_2\rangle$ must be rewritten. The tool finds rules 2, 3, 4 and 5 and applies each.

$$(\langle \lambda, w_3\rangle\,\langle \lambda, w_4\rangle,\ w_4) \tag{R_3}$$

$$(\mathtt{a}\,\langle W, w_5\rangle\,\langle \mathtt{a}\, w_5, w_3\rangle\,\langle \mathtt{a}\, w_5, w_4\rangle,\ w_4) \tag{R_4}$$

$$(\mathtt{b}\,\langle W, w_5\rangle\,\langle \mathtt{b}\, w_5, w_3\rangle\,\langle \mathtt{b}\, w_5, w_4\rangle,\ w_4) \tag{R_5}$$

$$(\mathtt{c}\,\langle W, w_5\rangle\,\langle \mathtt{c}\, w_5, w_3\rangle\,\langle \mathtt{c}\, w_5, w_4\rangle,\ w_4) \tag{R_6}$$

The pair $\langle \lambda, w_3\rangle$ in R_3 must be rewritten. Since λ is a terminal and $\forall t \in T_G$, $\langle t, t\rangle \rightarrow t$, we can rewrite this pair to the empty string. The unknown w_3 is therefore substituted for λ in the configuration. Our candidate set is:

$$(\langle\langle\lambda, w_4\rangle,\, w_4) \tag{R_7}$$

$$(\mathtt{a}\,\langle W, w_5\rangle\,\langle \mathtt{a}\,w_5, w_3\rangle\,\langle \mathtt{a}\,w_5, w_4\rangle,\, w_4) \tag{R_4}$$

$$(\mathtt{b}\,\langle W, w_5\rangle\,\langle \mathtt{b}\,w_5, w_3\rangle\,\langle \mathtt{b}\,w_5, w_4\rangle,\, w_4) \tag{R_5}$$

$$(\mathtt{c}\,\langle W, w_5\rangle\,\langle \mathtt{c}\,w_5, w_3\rangle\,\langle \mathtt{c}\,w_5, w_4\rangle,\, w_4) \tag{R_6}$$

Again we have a single pair $\langle\lambda, w_4\rangle$ which must be rewritten in the same way using the rule $\langle\lambda, \lambda\rangle \to \lambda$ - w_4 is substituted for λ.

$$(\lambda,\, w_4) \tag{R_8}$$

$$(\mathtt{a}\,\langle W, w_5\rangle\,\langle \mathtt{a}\,w_5, w_3\rangle\,\langle \mathtt{a}\,w_5, w_4\rangle,\, w_4) \tag{R_4}$$

$$(\mathtt{b}\,\langle W, w_5\rangle\,\langle \mathtt{b}\,w_5, w_3\rangle\,\langle \mathtt{b}\,w_5, w_4\rangle,\, w_4) \tag{R_5}$$

$$(\mathtt{c}\,\langle W, w_5\rangle\,\langle \mathtt{c}\,w_5, w_3\rangle\,\langle \mathtt{c}\,w_5, w_4\rangle,\, w_4) \tag{R_6}$$

Now each configuration's left-most element is an answer. Therefore the tool must look at the first symbol in the input syntax, and discard any non-matching configurations from the candidate set. The input is `aaa`, so each configuration is matched against `a`. Only R_4 can match an `a`, therefore each other result schema is discarded. The candidate result set now consists of:

$$(\mathtt{a}\,\langle W, w_5\rangle\,\langle \mathtt{a}\,w_5, w_3\rangle\,\langle \mathtt{a}\,w_5, w_4\rangle,\, w_4) \tag{R_4}$$

The pair $\langle W, w_5\rangle$ must now be rewritten using rules 2, 3, 4 and 5. In each case, the unknown w_5 is substituted for the semantic value of the rule used.

$$(\mathtt{a}\,\langle \mathtt{a}, w_3\rangle\,\langle \mathtt{a}, w_4\rangle,\, w_4) \tag{R_9}$$

$$(\mathtt{a}\,\mathtt{a}\,\langle W, w_6\rangle\,\langle \mathtt{a}\,\mathtt{a}\,w_6, w_3\rangle\,\langle \mathtt{a}\,\mathtt{a}\,w_6, w_4\rangle,\, w_4) \tag{R_{10}}$$

$$(\mathtt{a}\,\mathtt{b}\,\langle W, w_6\rangle\,\langle \mathtt{a}\,\mathtt{b}\,w_6, w_3\rangle\,\langle \mathtt{a}\,\mathtt{b}\,w_6, w_4\rangle,\, w_4) \tag{R_{11}}$$

$$(\mathtt{a}\,\mathtt{c}\,\langle W, w_6\rangle\,\langle \mathtt{a}\,\mathtt{c}\,w_6, w_3\rangle\,\langle \mathtt{a}\,\mathtt{c}\,w_6, w_4\rangle,\, w_4) \tag{R_{12}}$$

The pair $\langle \mathtt{a},\, w_3\rangle$ in R_9 must be rewritten. The implicit rule $\langle \mathtt{a},\, \mathtt{a}\rangle \to \mathtt{a}$ is used, and w_3 is substituted for `a`.

$$(\mathtt{a}\,\mathtt{a}\,\langle \mathtt{a}, w_4\rangle,\, w_4) \tag{R_{13}}$$

$$(\mathtt{a}\,\mathtt{a}\,\langle W, w_6\rangle\,\langle \mathtt{a}\,\mathtt{a}\,w_6, w_3\rangle\,\langle \mathtt{a}\,\mathtt{a}\,w_6, w_4\rangle,\, w_4) \tag{R_{10}}$$

$$(\mathtt{a}\,\mathtt{b}\,\langle W, w_6\rangle\,\langle \mathtt{a}\,\mathtt{b}\,w_6, w_3\rangle\,\langle \mathtt{a}\,\mathtt{b}\,w_6, w_4\rangle,\, w_4) \tag{R_{11}}$$

$$(\mathtt{a}\,\mathtt{c}\,\langle W, w_6\rangle\,\langle \mathtt{a}\,\mathtt{c}\,w_6, w_3\rangle\,\langle \mathtt{a}\,\mathtt{c}\,w_6, w_4\rangle,\, w_4) \tag{R_{12}}$$

Now each configuration begins with two answers, we are ready to match against the input syntax again. From aaa we match aa against each configuration. We see that only R_{13} and R_{10} can match aa, therefore R_{11} and R_{12} are discarded. The canddiate set of result schemas is now:

$$(\mathtt{a}\,\mathtt{a}\,\langle \mathtt{a}, w_4\rangle,\, w_4) \tag{R_{13}}$$

$$(\mathtt{a}\,\mathtt{a}\,\langle W, w_6\rangle\,\langle \mathtt{a}\,\mathtt{a}\,w_6, w_3\rangle\,\langle \mathtt{a}\,\mathtt{a}\,w_6, w_4\rangle,\, w_4) \tag{R_{10}}$$

Two pairs must be rewritten, $\langle \mathtt{a}, w_4\rangle$ in R_{13}, and $\langle W, w_6\rangle$ in R_{10}. The former is rewritten to the answer a using the implicit rule for terminals, and the latter is rewritten using rules 2, 3, 4 and 5.

$$(\mathtt{a}\,\mathtt{a}\,\mathtt{a},\, \mathtt{a}) \tag{R_{13}}$$

$$(\mathtt{a}\,\mathtt{a}\,\mathtt{a}\,\langle W, w_7\rangle\,\langle \mathtt{a}\,\mathtt{a}\,\mathtt{a}\,w_7, w_3\rangle\,\langle \mathtt{a}\,\mathtt{a}\,\mathtt{a}\,w_7, w_4\rangle,\, w_4) \tag{R_{14}}$$

$$(\mathtt{a}\,\mathtt{a}\,\mathtt{b}\,\langle W, w_7\rangle\,\langle \mathtt{a}\,\mathtt{a}\,\mathtt{b}\,w_7, w_3\rangle\,\langle \mathtt{a}\,\mathtt{a}\,\mathtt{b}\,w_7, w_4\rangle,\, w_4) \tag{R_{15}}$$

$$(\mathtt{a}\,\mathtt{a}\,\mathtt{c}\,\langle W, w_7\rangle\,\langle \mathtt{a}\,\mathtt{a}\,\mathtt{c}\,w_7, w_3\rangle\,\langle \mathtt{a}\,\mathtt{a}\,\mathtt{c}\,w_7, w_4\rangle,\, w_4) \tag{R_{16}}$$

Each configuration begins with three answers, so the tool can now match each against the input syntax again. We need to match each configuration against aaa. Only R_{13} and R_{14} have a prefix which can match aaa, so the others are discarded. The candidate set of result schemas is now:

$$(\mathtt{a}\,\mathtt{a}\,\mathtt{a},\, \mathtt{a}) \tag{R_{13}}$$

$$(\mathtt{a}\,\mathtt{a}\,\mathtt{a}\,\langle W, w_7\rangle\,\langle \mathtt{a}\,\mathtt{a}\,\mathtt{a}\,w_7, w_3\rangle\,\langle \mathtt{a}\,\mathtt{a}\,\mathtt{a}\,w_7, w_4\rangle,\, w_4) \tag{R_{14}}$$

After matching for aaa, the remaining input is now empty. While the pair $\langle W, w_7\rangle$ in R_{14} is nullable, the pairs $\langle \mathtt{a}\,\mathtt{a}\,\mathtt{a}\,w_7, w_3\rangle$ and $\langle \mathtt{a}\,\mathtt{a}\,\mathtt{a}\,w_7, w_4\rangle$ are not. Therefore R_{14} cannot match the remaining empty input syntax. For this reason, we can discard R_{14}. The final candidate set of result schemas is:

$$(\mathtt{a}\,\mathtt{a}\,\mathtt{a},\, \mathtt{a}) \tag{R_{13}}$$

All of the input syntax has been read by the tool and we have one remaining result schema, R_{14}. The set of semantic results associated with this derivation is the singleton set $\{\mathtt{a}\}$. The tool has identified the input syntax aaa and its corresponding semantic value a.

5.3.2 Triple-ABC Example

We specify another RAG example to display the breadth-first parsing process. This example also uses a variable as the left-component of a pair to illustrate how the algorithm handles adaptability.

Example 29. Consider the following RAG.

$$\langle S,\, \lambda\rangle \rightarrow \langle A,\, v_1\rangle\, \langle v_1,\, v_2\rangle \tag{1}$$

$$\langle A,\, \mathtt{b}\, v_1\, \mathtt{c}\rangle \rightarrow \mathtt{a}\, \langle A,\, v_1\rangle \tag{2}$$

$$\langle A,\, \lambda\rangle \rightarrow \lambda \tag{3}$$

The RAG specification in Ex. 29 defines the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ for $n \geq 0$. The terminal constants in this RAG are $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, the nullary non-terminal constants are $\{S, A\}$, and the start symbol is S. The only non-nullary operator is concatenation.

As with the triple-string example RAG defined in Ex. 28, the variable which specifies the semantics of a pair rewrite is used as the left-component of another pair. Variable v_1 defines the language of the second pair in rule 1. The pair $\langle A,\, v_1\rangle$ rewrites to $\mathtt{a}^n$ by use of rules 2 and 3, and in doing so, synthesises $\mathtt{b}^n\, \mathtt{c}^n$ as its semantics. The right-hand side of rule 1 therefore defines the syntax $\mathtt{a}^n\, \mathtt{b}^n\, \mathtt{c}^n$, synthesising λ.

Example Derivation

We will use the tool to try to derive the input syntax `aabbcc` with start symbol S. The derivation starts with a singleton candidate set consisting of the initial result schema, R_1:

$$(\langle S,\, w_1\rangle,\, w_1) \tag{R_1}$$

The pair $\langle S,\, w_1\rangle$ must be rewritten using rule 1, substituting w_1 for λ, the semantic value of the rule:

$$(\langle A,\, w_2\rangle\, \langle w_2,\, w_3\rangle,\, \lambda) \tag{R_2}$$

Now the pair $\langle A,\, w_2\rangle$ must be rewritten using rules 2 and 3. R_2 is rewritten to R_3 and R_4. In each case, w_2 is substituted for the semantic result of the rule used - $\mathtt{a}\, w_2$ and λ. Our candidate set of result schemas becomes:

$$(\mathtt{a}\, \langle A,\, w_4\rangle\, \langle \mathtt{a}\, w_4,\, w_3\rangle,\, \lambda) \tag{R_3}$$

$$(\langle \lambda,\, w_3\rangle,\, \lambda) \tag{R_4}$$

The pair $\langle \lambda,\, w_3\rangle$ must be rewritten using $\langle \lambda, \lambda\rangle \rightarrow \lambda$, with w_3 being substituted for λ. R_4 is rewritten to R_5 using this rule, and our candidate set now consists of the following result schemas:

$$(\mathtt{a}\, \langle A,\, w_4\rangle\, \langle \mathtt{b}\, w_4\, \mathtt{c},\, w_3\rangle,\, \lambda) \tag{R_3}$$

$$(\lambda,\, \lambda) \tag{R_5}$$

We can now look at the first symbol in our input syntax, `aabbcc`, `a`. We see that only R_3 has a prefix of `a`, so R_5 is discarded.

$$(\mathtt{a}\,\langle A,\, w_4\rangle\,\langle \mathtt{b}\,w_4\,\mathtt{c},\, w_3\rangle,\, \lambda) \qquad (R_3)$$

We have handled the leading a, so must rewrite the pair $\langle A,\, w_4\rangle$ using rules 2 and 3, substituting w_4 for the semantic result of the rules used:

$$(\mathtt{a\,a}\,\langle A,\, w_5\rangle\,\langle \mathtt{b\,b}\,w_5\,\mathtt{c\,c},\, w_3\rangle,\, \lambda) \qquad (R_6)$$

$$(\mathtt{a}\,\langle \mathtt{b\,c},\, w_3\rangle,\, \lambda) \qquad (R_7)$$

The pair $\langle \mathtt{b\,c},\, w_3\rangle$ must be rewritten to an answer using the implicit rule for all terminals. w_3 is substituted for b c. In this way, result schema R_7 is rewritten to R_8:

$$(\mathtt{a\,a}\,\langle A,\, w_5\rangle\,\langle \mathtt{b\,b}\,w_5\,\mathtt{c\,c},\, w_3\rangle,\, \lambda) \qquad (R_6)$$

$$(\mathtt{a\,b\,c},\, \lambda) \qquad (R_8)$$

Now we can look at the second symbol of our input syntax. We check whether each configuration can match the prefix aa. The configuration for R_6 can, but the configuration for R_8 cannot. R_8 is discarded, and our candidate set becomes:

$$(\mathtt{a\,a}\,\langle A,\, w_5\rangle\,\langle \mathtt{b\,b}\,w_5\,\mathtt{c\,c},\, w_3\rangle,\, \lambda) \qquad (R_6)$$

Since the prefix aa has been handled, we must rewrite the pair $\langle A,\, w_5\rangle$ using rules 2 and 3 to two new result schemas:

$$(\mathtt{a\,a\,a}\,\langle A,\, w_6\rangle\,\langle \mathtt{b\,b\,b}\,w_6\,\mathtt{c\,c\,c},\, w_3\rangle,\, \lambda) \qquad (R_9)$$

$$(\mathtt{a\,a}\,\langle \mathtt{b\,b\,c\,c},\, w_3\rangle,\, \lambda) \qquad (R_{10})$$

We have a pair $\langle \mathtt{b\,b\,c\,c},\, w_3\rangle$ in the configuration for R_{10} which can be rewritten using the rule for terminals. with w_3 being substituted for b b c. The result schema R_{10} is thus rewritten to R_{11}.

$$(\mathtt{a\,a\,a}\,\langle A,\, w_6\rangle\,\langle \mathtt{b\,b\,b}\,w_6\,\mathtt{c\,c\,c},\, w_3\rangle,\, \lambda) \qquad (R_9)$$

$$(\mathtt{a\,a\,b\,b\,c\,c},\, \lambda) \qquad (R_{11})$$

Now we are able to match a prefix of three letters from our input syntax. We match each result schema against aab. Only R_{11} can match this prefix, so R_9 is discarded. The candidate set is now:

$$(\mathtt{a\,a\,b\,b\,c\,c},\, \lambda) \qquad (R_{11})$$

Result schema R_{11} consists of no pairs, so cannot be rewritten. We instead match it against the entirety of the input syntax. We notice that the configuration for R_{11} is able to match the input syntax aabbcc, therefore this result schema is the product of a correct derivation of this input. The set of results for the derivation of aabbcc is the singleton set $\{\lambda\}$.

5.4 Typed Variables

As mentioned in section §4.5, we can specify *typed* variables to extend the set of bound rules associated with a non-terminal answer.

5.4.1 Triple-String Example

To display how the algorithm handles typed variables, we re-visit the triple-string RAG example defined in §5.3.1, Ex. 29. The language defined by the start symbol S in that example was $\{www \mid w \in [\mathtt{a} - \mathtt{c}]^*\}$, with a rule for each symbol $\mathtt{a}, \mathtt{b}, \mathtt{c}$. Using a typed variable, we can now specify with a single rule the language $[\mathtt{a} - \mathtt{z}]$ instead.

Example 30. Consider the following RAG.

$$Z ::= \mathtt{a..z}$$

$$\langle S, v_4\rangle \to \langle A, v_1\rangle\,\langle v_1, v_2\rangle\,\langle v_1, v_3\rangle \tag{1}$$

$$\langle A, \lambda\rangle \to \lambda \tag{2}$$

$$\langle A, v_1\, v_2\rangle \to \langle C, v_1\rangle\,\langle A, v_2\rangle \tag{3}$$

$$z : Z \quad \langle C, z\rangle \to \langle z, v_1\rangle \tag{4}$$

The terminal constants in this RAG are $[\mathtt{a}-\mathtt{z}]$, the nullary non-terminals are $\{S, A, C\}$, and the start symbol is S. The language generated by the start symbol S is $\{www \mid w \in [\mathtt{a} - \mathtt{z}]^*\}$

Rule 4 makes use of z, a variable whose language is defined outside of the RAG rule, but given as the context-free rule Z::=$\mathtt{a..z}$. This denotes that the z, when we rewrite using rule 4, can take any value in the language $[\mathtt{a} - \mathtt{z}]$.

Example Derivation

With S as our start symbol, using the RAG specification defined in Ex. 30, we try to derive the input syntax `xyxyxy`. We begin with a single starting result schema in our candidate set:

$$(\langle S,\, w_1\rangle,\, w_1) \tag{R_1}$$

The pair $\langle S,\, w_1\rangle$ must be rewritten using rule 1. R_1 is discarded and we create one new result schema:

$$(\langle A,\, w_2\rangle\,\langle w_2,\, w_3\rangle\,\langle w_2,\, w_4\rangle,\, w_4) \tag{R_2}$$

Now the pair $\langle A,\, w_2\rangle$ must be rewritten. We have rules 2 and 3 for the answer A, and so create two new result schemas in place of R_2:

$$(\langle \lambda,\, w_3\rangle\,\langle \lambda,\, w_4\rangle,\, w_4) \tag{R_3}$$

$$(\langle C,\, w_5\rangle\, \langle W,\, w_6\rangle\, \langle w_5\, w_6,\, w_3\rangle\, \langle w_5\, w_6,\, w_4\rangle,\, w_4) \tag{R_4}$$

Both configurations have pairs which need to be rewritten. We rewrite the pair $\langle \lambda,\, w_3\rangle$ in R_3 using $\langle \lambda, \lambda\rangle \to \lambda$, and the pair $\langle C,\, w_5\rangle$ in R_4 using rule 4. Our candidate set of result schemas becomes:

$$(\langle \lambda,\, w_4\rangle,\, w_4) \tag{R_5}$$

$$(\langle z,\, w_7\rangle\, \langle A,\, w_6\rangle\, \langle w_7\, w_6,\, w_3\rangle\, \langle w_7\, w_6,\, w_4\rangle,\, w_4) \tag{R_6}$$

The pair $\langle \lambda,\, w_4\rangle$ in R_5 must be rewritten. We have a pair $\langle z,\, w_7\rangle$ in the configuration for R_6, however since its left-component is a typed variable, we wait until the other candidate result schemas are ready to match against an input symbol before, substituting the z. Our candidate set is now:

$$(\lambda,\, \lambda) \tag{R_7}$$

$$(\langle z,\, w_7\rangle\, \langle A,\, w_6\rangle\, \langle w_7\, w_6,\, w_3\rangle\, \langle w_7\, w_6,\, w_4\rangle,\, w_4) \tag{R_6}$$

Result schema R_7 is empty, and the left-most element of R_6 is a pair whose left-component is a typed variable. At this point, we read the first symbol `x` from the input syntax `xyxyxy` and determine whether `x` belongs to the set of possible values z ranges over. The language for z is $[\mathtt{a}-\mathtt{z}]$, and $\mathtt{x} \in [\mathtt{a}-\mathtt{z}]$, so we can rewrite the pair $\langle z,\, w_7\rangle$ to `x`. The configuration for R_7 can not match `x`, and is discarded. Our candidate set becomes:

$$(\mathtt{x}\, \langle A,\, w_6\rangle\, \langle \mathtt{x}\, w_6,\, w_3\rangle\, \langle \mathtt{x}\, w_6,\, w_4\rangle,\, w_4) \tag{R_8}$$

The pair $\langle A,\, w_6\rangle$ must now be rewritten using rules 2 and 3:

$$(\mathtt{x}\, \langle \mathtt{x},\, w_3\rangle\, \langle \mathtt{x},\, w_4\rangle,\, w_4) \tag{R_9}$$

$$(\mathtt{x}\, \langle C,\, w_8\rangle\, \langle A,\, w_9\rangle\, \langle \mathtt{x}\, w_8\, w_9,\, w_3\rangle\, \langle \mathtt{x}\, w_8\, w_9,\, w_4\rangle,\, w_4) \tag{R_{10}}$$

Now the pairs $\langle \mathtt{x},\, w_3\rangle$ in R_9 and $\langle C,\, w_8\rangle$ in R_{10} must be rewritten:

$$(\mathtt{x}\,\mathtt{x}\, \langle \mathtt{x},\, w_4\rangle,\, w_4) \tag{R_{11}}$$

$$(\mathtt{x}\, \langle z,\, w_{10}\rangle\, \langle A,\, w_9\rangle\, \langle \mathtt{x}\, w_{10}\, w_9,\, w_3\rangle\, \langle \mathtt{x}\, w_{10}\, w_9,\, w_4\rangle,\, w_4) \tag{R_{12}}$$

Result schema R_{11} has a pair $\langle \mathtt{x},\, w_4\rangle$ which must be rewritten, and R_{12} has the pair $\langle z,\, w_{10}\rangle$ which we delay the rewriting of until we read some more input. The candidate set of result schemas is now:

$$(\mathtt{x}\,\mathtt{x}\,\mathtt{x},\, \mathtt{x}) \tag{R_{13}}$$

$$(\mathtt{x}\, \langle z,\, w_{10}\rangle\, \langle A,\, w_9\rangle\, \langle \mathtt{x}\, w_{10}\, w_9,\, w_3\rangle\, \langle \mathtt{x}\, w_{10}\, w_9,\, w_4\rangle,\, w_4) \tag{R_{12}}$$

We can now read the second symbol from the input syntax. We match the input prefix `xy` against each configurations. R_{13} cannot match `xy` as it begins with `x x`. For R_{12}, the tool determines that z can be substituted for `y`, since $\mathtt{y} \in [\mathtt{a} - \mathtt{z}]$. In one step, the pair $\langle z,\, w_{10}\rangle$ is rewritten to `y`.

$$(\mathtt{x\,y}\,\langle A,\, w_9\rangle\,\langle \mathtt{x\,y}\, w_9,\, w_3\rangle\,\langle \mathtt{x\,y}\, w_9,\, w_4\rangle,\, w_4) \qquad (R_{12})$$

The pair $\langle A,\, w_9\rangle$ must be rewritten using rules 2 and 3 in two new result schemas:

$$(\mathtt{x\,y}\,\langle \mathtt{x\,y},\, w_3\rangle\,\langle \mathtt{x\,y},\, w_4\rangle,\, w_4) \qquad (R_{14})$$

$$(\mathtt{x\,y}\,\langle C, w_{11}\rangle\,\langle A,\, w_{12}\rangle\,\langle \mathtt{x\,y}\, w_{11}\, w_{12},\, w_3\rangle\,\langle \mathtt{x\,y}\, w_{11}\, w_{12},\, w_4\rangle,\, w_4) \qquad (R_{15})$$

Now the pair $\langle \mathtt{x\,y},\, w_3\rangle$ in R_{14} must be rewritten using the rule for terminals, and the pair $\langle C, w_{11}\rangle$ must be rewritten using rule 4:

$$(\mathtt{x\,y\,x\,y}\,\langle \mathtt{x\,y},\, w_4\rangle,\, w_4) \qquad (R_{16})$$

$$(\mathtt{x\,y}\,\langle z, w_{13}\rangle\,\langle A,\, w_{12}\rangle\,\langle \mathtt{x\,y}\, w_{13}\, w_{12},\, w_3\rangle\,\langle \mathtt{x\,y}\, w_{13}\, w_{12},\, w_4\rangle,\, w_4) \qquad (R_{15})$$

We can now read the third symbol from our input syntax `xyxyxy`, and match each result schema to the prefix `xyx`. R_{16} matches this prefix. For R_{15}, we can substitute z for `x` and rewrite the pair $\langle z, w_{13}\rangle$ to `x`. Both result schemas have a successful match. Our candidate set is maintained as:

$$(\mathtt{x\,y\,x\,y}\,\langle \mathtt{x\,y},\, w_4\rangle,\, w_4) \qquad (R_{16})$$

$$(\mathtt{x\,y\,x}\,\langle A,\, w_{12}\rangle\,\langle \mathtt{x\,y\,x}\, w_{12},\, w_3\rangle\,\langle \mathtt{x\,y\,x}\, w_{12},\, w_4\rangle,\, w_4) \qquad (R_{17})$$

The pair $\langle \mathtt{x\,y},\, w_4\rangle$ in R_{16} must be rewritten using rule for terminals, and $\langle A,\, w_{12}\rangle$ in R_{17} must be rewritten using rules 2 and 3:

$$(\mathtt{x\,y\,x\,y\,x\,y},\, \mathtt{x\,y}) \qquad (R_{17})$$

$$(\mathtt{x\,y\,x}\,\langle \mathtt{x\,y\,x},\, w_3\rangle\,\langle \mathtt{x\,y\,x},\, w_4\rangle,\, w_4) \qquad (R_{18})$$

$$(\mathtt{x\,y\,x}\,\langle A,\, w_{12}\rangle\,\langle \mathtt{x\,y\,x}\, w_{12},\, w_3\rangle\,\langle \mathtt{x\,y\,x}\, w_{12},\, w_4\rangle,\, w_4) \qquad (R_{19})$$

Now the pairs $\langle \mathtt{x\,y\,x},\, w_3\rangle$ in R_{18} and $\langle A,\, w_{12}\rangle$ must be rewritten. Our candidate set of result schemas is now:

$$(\mathtt{x\,y\,x\,y\,x\,y},\, \mathtt{x\,y}) \qquad (R_{17})$$

$$(\mathtt{x\,y\,x\,x\,y\,x}\,\langle \mathtt{x\,y\,x},\, w_4\rangle,\, w_4) \qquad (R_{20})$$

$$(\mathtt{x\,y\,x}\,\langle C,\, w_{13}\rangle\,\langle A,\, w_{14}\rangle\,\langle \mathtt{x\,y\,x}\, w_{13}\, w_{14},\, w_3\rangle\,\langle \mathtt{x\,y\,x}\, w_{13}\, w_{14},\, w_4\rangle,\, w_4) \qquad (R_{21})$$

$\langle \mathtt{x\,y\,x},\, w_4\rangle$ in R_{20} must be rewritten as a terminal, and $\langle C,\, w_{13}\rangle$ must be rewritten using rule 4. The candidate set becomes:

$$(\mathtt{x\,y\,x\,y\,x\,y},\, \mathtt{x\,y}) \qquad (R_{17})$$

$$(\mathtt{x\,y\,x\,x\,y\,x\,x\,y\,x},\ \mathtt{x\,y\,x}) \tag{R_{22}}$$

$$(\mathtt{x\,y\,x}\,\langle z,\ w_{15}\rangle\,\langle A,\ w_{14}\rangle\,\langle \mathtt{x\,y\,x}\,w_{15}\,w_{14},\ w_3\rangle\,\langle \mathtt{x\,y\,x}\,w_{15}\,w_{14},\ w_4\rangle,\ w_4) \tag{R_{23}}$$

We can now match each result schema against the prefix of size four from the input syntax, `xyxy`. The configuration for R_{16} matches this prefix, but the configuration for R_{22} does not. For R_{24}, we substitute z for `y` and rewrite $\langle z,\ w_{15}\rangle$ to `y` in one step.

$$(\mathtt{x\,y\,x\,y\,x\,y},\ \mathtt{x\,y}) \tag{R_{17}}$$

$$(\mathtt{x\,y\,x\,y}\,\langle A,\ w_{14}\rangle\,\langle \mathtt{x\,y\,x\,y}\,w_{14},\ w_3\rangle\,\langle \mathtt{x\,y\,x\,y}\,w_{14},\ w_4\rangle,\ w_4) \tag{R_{24}}$$

The pair $\langle A,\ w_{14}\rangle$ in R_{24} must be rewritten using rules 2 and 3:

$$(\mathtt{x\,y\,x\,y\,x\,y},\ \mathtt{x\,y}) \tag{R_{17}}$$

$$(\mathtt{x\,y\,x\,y}\,\langle \mathtt{x\,y\,x\,y}\,w_3\rangle\,\langle \mathtt{x\,y\,x\,y},\ w_4\rangle,\ w_4) \tag{R_{25}}$$

$$(\mathtt{x\,y\,x\,y}\,\langle C,\ w_{15}\rangle\,\langle A,\ w_{16}\rangle\,\langle \mathtt{x\,y\,x\,y}\,w_{15}\,w_{16},\ w_3\rangle\,\langle \mathtt{x\,y\,x\,y}\,w_{15}\,w_{16},\ w_4\rangle,\ w_4) \tag{R_{26}}$$

Now the pair $\langle \mathtt{x\,y\,x\,y}\,w_3\rangle$ in R_{25} must be rewritten using the rule for terminals, and $\langle C,\ w_{15}\rangle$ in R_{26} must be rewritten using rule 4:

$$(\mathtt{x\,y\,x\,y\,x\,y},\ \mathtt{x\,y}) \tag{R_{17}}$$

$$(\mathtt{x\,y\,x\,y\,x\,y\,x\,y}\,\langle \mathtt{x\,y\,x\,y},\ w_4\rangle,\ w_4) \tag{R_{27}}$$

$$(\mathtt{x\,y\,x\,y}\,\langle z,\ w_{17}\rangle\,\langle A,\ w_{16}\rangle\,\langle \mathtt{x\,y\,x\,y}\,w_{17}\,w_{16},\ w_3\rangle\,\langle \mathtt{x\,y\,x\,y}\,w_{17}\,w_{16},\ w_4\rangle,\ w_4) \tag{R_{28}}$$

We can now match a prefix of size five to the input syntax. We match each result schema to `xyxyx`. R_{17} successfully matches this prefix, as does R_{27}. For R_{28} we substitute z for `x` and rewrite the pair $\langle z,\ w_{17}\rangle$ to `x`. Our candidate set of result schemas becomes:

$$(\mathtt{x\,y\,x\,y\,x\,y},\ \mathtt{x\,y}) \tag{R_{17}}$$

$$(\mathtt{x\,y\,x\,y\,x\,y\,x\,y}\,\langle \mathtt{x\,y\,x\,y},\ w_4\rangle,\ w_4) \tag{R_{27}}$$

$$(\mathtt{x\,y\,x\,y\,x}\,\langle A,\ w_{16}\rangle\,\langle \mathtt{x\,y\,x\,y\,x}\,w_{16},\ w_3\rangle\,\langle \mathtt{x\,y\,x\,y\,x}\,w_{16},\ w_4\rangle,\ w_4) \tag{R_{29}}$$

Now the pair $\langle \mathtt{x\,y\,x\,y},\ w_4\rangle$ in R_{27} must be rewritten using the rule for terminals, and $\langle A,\ w_{16}\rangle$ in R_{29} must be rewritten using rules 2 and 3:

$$(\mathtt{x\,y\,x\,y\,x\,y},\ \mathtt{x\,y}) \tag{R_{17}}$$

$$(\mathtt{x\,y\,x\,y\,x\,y\,x\,y\,x\,y\,x\,y},\ \mathtt{x\,y\,x\,y}) \tag{R_{30}}$$

$$(\mathtt{x\,y\,x\,y\,x}\,\langle \mathtt{x\,y\,x\,y\,x},\ w_3\rangle\,\langle \mathtt{x\,y\,x\,y\,x},\ w_4\rangle,\ w_4) \tag{R_{31}}$$

$$(\mathtt{x\,y\,x\,y\,x}\,\langle C,\, w_{18}\rangle\,\langle A,\, w_{19}\rangle\,\langle \mathtt{x\,y\,x\,y\,x}\, w_{18}\, w_{19},\, w_3\rangle\,\langle \mathtt{x\,y\,x\,y\,x}\, w_{18}\, w_{19},\, w_4\rangle,\, w_4) \tag{R_{32}}$$

We must rewrite $\langle \mathtt{x\,y\,x\,y\,x},\, w_3\rangle$ in R_{31} to a terminal, and the pair $\langle C,\, w_{18}\rangle$ in R_{32} using rule 5. Our candidate set becomes:

$$(\mathtt{x\,y\,x\,y\,x\,y},\, \mathtt{x\,y}) \tag{R_{17}}$$

$$(\mathtt{x\,y\,x\,y\,x\,y\,x\,y\,x\,y\,x\,y},\, \mathtt{x\,y\,x\,y}) \tag{R_{30}}$$

$$(\mathtt{x\,y\,x\,y\,x\,x\,y\,x\,y\,x}\,\langle \mathtt{x\,y\,x\,y\,x},\, w_4\rangle,\, w_4) \tag{R_{33}}$$

$$(\mathtt{x\,y\,x\,y\,x}\,\langle z,\, w_{20}\rangle\,\langle A,\, w_{19}\rangle\,\langle \mathtt{x\,y\,x\,y\,x}\, w_{20}\, w_{19},\, w_3\rangle\,\langle \mathtt{x\,y\,x\,y\,x}\, w_{20}\, w_{19},\, w_4\rangle,\, w_4) \tag{R_{34}}$$

We now match each result schema against the input prefix xyxyxy. R_{17} matches successfully. R_{30} also matches successfully as it begins with $\mathtt{x\,y\,x\,y\,x\,y}$. R_{33} does not match. For R_{34}, we substitute z for y and rewrite the pair $\langle z,\, w_{20}\rangle$ to y. Our candidate set of result schemas becomes:

$$(\mathtt{x\,y\,x\,y\,x\,y},\, \mathtt{x\,y}) \tag{R_{17}}$$

$$(\mathtt{x\,y\,x\,y\,x\,y\,x\,y\,x\,y\,x\,y},\, \mathtt{x\,y\,x\,y}) \tag{R_{30}}$$

$$(\mathtt{x\,y\,x\,y\,x\,y}\,\langle A,\, w_{19}\rangle\,\langle \mathtt{x\,y\,x\,y\,x\,y}\, w_{19},\, w_3\rangle\,\langle \mathtt{x\,y\,x\,y\,x\,y}\, w_{19},\, w_4\rangle,\, w_4) \tag{R_{35}}$$

The remaining input is empty, so we remove those result schemas whose configuration does not exactly match xyxyxy. R_{17} matches this syntax. Only the configuration for R_{17} is an exact match, so each other result schema is discarded.

$$(\mathtt{x\,y\,x\,y\,x\,y},\, \mathtt{x\,y}) \tag{R_{17}}$$

This is our final candidate set , consisting of one result schema R_{17}. The result set for the derivation of xyxyxy is the singleton set $\{\mathtt{x\,y}\}$.

5.5 Non-nullary Operators

The tool can also handle rules for an answer whose parameters must be matched against for the rewrite rules to be used, where an answer with the same identifier appears as the left-component of a pair.

5.5.1 Triple-ABC Example

The use of non-nullary operators presents another way to define the triple-abc language given in §5.3.2, defined by Ex. 29. We can define unary operators B, C whose parameters match zero or more of a, and use them to define rules which rewrite to that many b's or c's respectively. Consider the RAG specification defined in 31.

Example 31. Consider the following RAG.

$$\langle S, \lambda\rangle \to \langle A, v_1\rangle \, \langle B(v_1), v_2\rangle \, \langle C(v_1), v_3\rangle \tag{1}$$

$$\langle A, \lambda\rangle \to \lambda \tag{2}$$

$$\langle A, \mathtt{a}\, v_1\rangle \to \mathtt{a}\, \langle A, v_1\rangle \tag{3}$$

$$\langle B(\lambda), \lambda\rangle \to \lambda \tag{4}$$

$$\langle B(\mathtt{a}\, v_1), \lambda\rangle \to \mathtt{b}\, \langle B(v_1), v_2\rangle \tag{5}$$

$$\langle C(\lambda), \lambda\rangle \to \lambda \tag{6}$$

$$\langle C(\mathtt{a}\, v_1), \lambda\rangle \to \mathtt{c}\, \langle C(v_1), v_2\rangle \tag{7}$$

The terminal constants in this RAG are $\mathtt{a}, \mathtt{b}, \mathtt{c}$, the nullary non-terminals are $\{S, A\}$, and the unary non-terminals are $\{B, C\}$. The start symbol of this RAG is S. The language defined by the start symbol S is $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ for $n \geq 0$.

Example Derivation

Using S as our starting answer and the RAG specification defined in Ex. 31, we try to derive the input `abc`. We start with a candidate set consisting of initial result schema R_1:

$$(\langle S,\, w_1\rangle,\, w_1) \tag{R_1}$$

The pair $\langle S,\, w_1\rangle$ must be rewritten using rule 1:

$$(\langle A,\, w_2\rangle \, \langle B(w_2),\, w_3\rangle \, \langle C(w_2),\, w_4\rangle,\, \lambda) \tag{R_2}$$

Now the pair $\langle A,\, w_2\rangle$ in R_2 must be rewritten using rules 2 and 3 to result schemas R_3 and R_4 respectively:

$$(\langle B(\lambda),\, w_3\rangle \, \langle C(\lambda),\, w_4\rangle,\, \lambda) \tag{R_3}$$

$$(\mathtt{a}\, \langle A,\, w_5\rangle \, \langle B(\mathtt{a}\, w_5),\, w_3\rangle \, \langle C(\mathtt{a}\, w_5),\, w_4\rangle,\, \lambda) \tag{R_4}$$

Result schema R_4 begins with an answer, so we ignore it until we need to read an input symbol. We need to rewrite the pair $\langle B(\lambda),\, w_3\rangle$ in R_3. Since $B(\lambda)$ is a unary operator, we must match the argument λ to the parameters defined in the rules for B. λ matches to the parameter for rule 4, λ. However, we cannot match λ to the parameter $\mathtt{a}\, v_1$ in rule 5. Therefore only rule 4 is used to rewrite the pair $\langle B(\lambda),\, w_3\rangle$.

$$(\langle C(\lambda),\, w_4\rangle,\, \lambda) \tag{R_5}$$

$$(\mathtt{a}\, \langle A,\, w_5\rangle \, \langle B(\mathtt{a}\, w_5),\, w_3\rangle \, \langle C(\mathtt{a}\, w_5),\, w_4\rangle,\, \lambda) \tag{R_4}$$

We need to rewrite the pair $\langle C(\lambda),\, w_4\rangle$ in R_5, so must match the argument λ to the parameters in the head of the rules for C. We can match

λ to the parameter λ for rule 6, but not to $\mathtt{a}\,v_1$ in the parameter rule 7. Therefore only rule 6 is used to rewrite the pair $\langle C(\lambda),\, w_4\rangle$.

$$(\lambda,\, \lambda) \tag{R_6}$$

$$(\mathtt{a}\,\langle A,\, w_5\rangle\,\langle B(\mathtt{a}\,w_5),\, w_3\rangle\,\langle C(\mathtt{a}\,w_5),\, w_4\rangle,\, \lambda) \tag{R_4}$$

We can now match each result schema to the input prefix of size one, a. We find that R_6 cannot match the prefix a, but R_4 can. Therefore we discard R_6 and out candidate set becomes:

$$(\mathtt{a}\,\langle A,\, w_5\rangle\,\langle B(\mathtt{a}\,w_5),\, w_3\rangle\,\langle C(\mathtt{a}\,w_5),\, w_4\rangle,\, \lambda) \tag{R_4}$$

The pair $\langle A,\, w_5\rangle$ must now be rewritten using rules 2 and 3, generating result schemas R_7 and R_8:

$$(\mathtt{a}\,\langle B(\mathtt{a}),\, w_3\rangle\,\langle C(\mathtt{a}),\, w_4\rangle,\, \lambda) \tag{R_7}$$

$$(\mathtt{a}\,\mathtt{a}\,\langle A,\, w_6\rangle\,\langle B(\mathtt{a}\,\mathtt{a}\,w_6),\, w_3\rangle\,\langle C(\mathtt{a}\,\mathtt{a}\,w_6),\, w_4\rangle,\, \lambda) \tag{R_8}$$

We can ignore R_8 for now until we can match each result schema to a prefix of size two. We need to rewrite the pair $\langle B(\mathtt{a}),\, w_3\rangle$ in R_7. The argment for $B(\mathtt{a})$ cannot match the parameter for the head of rule 4, λ. However, we can match a to the parameter for rule 5, $\mathtt{a}\,v_1$, by assigning the value λ to the corresponding unknown for v_1. We therefore rewrite the pair $\langle B(\mathtt{a}),\, w_3\rangle$ using only rule 5 to a new result schema R_9:

$$(\mathtt{a}\,\mathtt{b}\,\langle \lambda, w_6\rangle\,\langle C(\mathtt{a}),\, w_4\rangle,\, \lambda) \tag{R_9}$$

$$(\mathtt{a}\,\mathtt{a}\,\langle A,\, w_6\rangle\,\langle B(\mathtt{a}\,\mathtt{a}\,w_6),\, w_3\rangle\,\langle C(\mathtt{a}\,\mathtt{a}\,w_6),\, w_4\rangle,\, \lambda) \tag{R_8}$$

We can now match each result schema with an input prefix of size two, ab. Only R_9 can match this prefix, so R_8 is discarded.. Our candidate set becomes:

$$(\mathtt{a}\,\mathtt{b}\,\langle \lambda, w_6\rangle\,\langle C(\mathtt{a}),\, w_4\rangle,\, \lambda) \tag{R_9}$$

We must rewrite the pair $\langle \lambda, w_6\rangle$ using the implicit rule $\langle \lambda,\, \lambda\rangle \to \lambda$, creating result schema R_{10}:

$$(\mathtt{a}\,\mathtt{b}\,\langle C(\mathtt{a}),\, w_4\rangle,\, \lambda) \tag{R_{10}}$$

We must now rewrite the pair $\langle C(\mathtt{a}),\, w_4\rangle$. The argument for $C(\mathtt{a})$ cannot match the parameter λ for rule 6, however can match the parameter $\mathtt{a}\,v_1$ if we substitute v_1 for λ. Therefore we rewrite the pair $\langle C(\mathtt{a}),\, w_4\rangle$ using only rule 7:

$$(\mathtt{a}\,\mathtt{b}\,\mathtt{c}\,\langle \lambda,\, w_7\rangle,\, \lambda) \tag{R_{11}}$$

We now rewrite the pair $\langle \lambda, w_6\rangle$ using the implicit for terminals, creating result schema R_{12}:

$$(\mathtt{a}\,\mathtt{b}\,\mathtt{c},\, \lambda) \tag{R_{11}}$$

The single remaining result schema in our candidate set, R_{11} cannot be rewritten, and we have no remaining input to match against. Therefore we have a successful derivation of the input syntax abc using start symbol S, and have a result set $\{\lambda\}$.

5.6 Queries

A query can be thought of as a specification of an answer computed as the semantic value of a derivation sequence of some syntax from a starting answer. With respect to Proposition 4.4 of Shutt's thesis [1], to rewrite a query $(\alpha \mathrel{?} \gamma)$ to an answer, we want find a β such that $\langle \alpha, \beta \rangle \Rightarrow^* \gamma$ with $\alpha, \beta, \gamma \in A_G$.

5.6.1 Input Derivation as a Query

As mentioned in previous sections, the goal of our algorithm, given a RAG specification G, some input γ and a starting answer α, is to find a β such that $\langle \alpha, \beta \rangle \Rightarrow^* \gamma$ holds. By Proposition 4.4 of Shutt's thesis [1], we can instead write $(\alpha \mathrel{?} \gamma) \Rightarrow^* \beta$. The algorithm is written with this relation in mind; its input is a query, and its output a set of results derived from the rewriting of that query. Given $(\alpha \mathrel{?} \gamma)$, the tool starts with a result schema $(\langle \alpha, w_1 \rangle, w_1)$, tries to derive γ and find the correct variable substitutions to compute β such that $(\alpha \mathrel{?} \gamma) \Rightarrow^* \beta$.

When queries occur in any configuration in a derivation sequence, the tool recursively calls itself, and replaces the queries with the semantic values associated with the sub-derivation sequence.

5.6.2 Triple-ABC Example

The first case the tool must be able to handle is the presence of a query as the left-component of a pair. Consequently, the tool must first rewrite the query to an answer before it is able to rewrite the pair itself. Using a new RAG specification defined in Ex. 32, we can define again the $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ language first specified in §5.3.2 with Ex. 29, now using queries.

Example 32. Consider the following RAG.

$$\langle S, \lambda \rangle \to \langle A, v_1 \rangle \, \langle (B \mathrel{?} v_1), v_2 \rangle \, \langle (C \mathrel{?} v_1), v_3 \rangle \tag{1}$$

$$\langle A, \lambda \rangle \to \lambda \tag{2}$$

$$\langle A, \mathtt{a}\, v_1 \rangle \to \mathtt{a} \, \langle A, v_1 \rangle \tag{3}$$

$$\langle B, \lambda \rangle \to \lambda \tag{4}$$

$$\langle B, \mathtt{b}\, v_1 \rangle \to \mathtt{a} \, \langle B, v_1 \rangle \tag{5}$$

$$\langle C, \lambda \rangle \to \lambda \tag{6}$$

$$\langle C, \mathtt{c}\, v_1 \rangle \to \mathtt{a} \, \langle C, v_1 \rangle \tag{7}$$

The terminal constants in this RAG are $\mathtt{a}, \mathtt{b}, \mathtt{c}$, the nullary non-terminals are $\{S, A, B, C\}$, and the start symbol is S. The language defined by the start symbol S is $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ for $n \geq 0$.

Example Derivation

With S as our starting answer and using the RAG specification defined in Ex. 32, we try to derive the syntax `abc`. For each sub-derivation, we denote the unknowns each result schema contains with u_i to distinguish from the unknowns w_i. We start with a candidate set consisting of the initial result schema R_1:

$$(\langle S,\, w_1\rangle,\, w_1) \tag{R_1}$$

The pair $\langle S,\, w_1\rangle$ must be rewritten using rule 1 to a new result schema R_2:

$$(\langle A,\, w_2\rangle\, \langle (B\,?\,w_2),\, w_3\rangle\, \langle (C\,?\,w_2),\, w_4\rangle,\, \lambda) \tag{R_2}$$

Now the pair $\langle A,\, w_2\rangle$ must be rewritten using rules 2 and 3, creating result schemas R_3 and R_4 respectively:

$$(\langle (B\,?\,\lambda),\, w_3\rangle\, \langle (C\,?\,\lambda),\, w_4\rangle,\, \lambda) \tag{R_3}$$

$$(\mathtt{a}\,\langle A,\, w_5\rangle\, \langle (B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\, \langle (C\,?\,\mathtt{a}\,w_5),\, w_4\rangle,\, \lambda) \tag{R_4}$$

Now we can ignore R_4 until we read a symbol from the input syntax, as it begins with an answer we can match against. The pair $\langle (B\,?\,\lambda),\, w_3\rangle$ in R_3 must be rewritten. Since the left-component of the pair is a query, we must first rewrite it to an answer before we are able to use a rule to rewrite the pair. We therefore begin a new sub-derivation of the syntax λ with starting answer B.

$$(\langle B,\, u_1\rangle,\, u_1) \tag{SUB_R_1}$$

We must rewrite the pair $\langle B,\, u_1\rangle$ using rules 4 and 5 to two new result schemas:

$$(\lambda,\, \lambda) \tag{SUB_R_2}$$

$$(\mathtt{a}\,\langle B,\, u_2\rangle,\, \mathtt{b}\,u_2) \tag{SUB_R_3}$$

Now we are able to match against a prefix of size one of the input for this sub-derivation, λ. Only SUB_R_2 can match λ, therefore SUB_R_3 is discarded from the candidate set of result schemas.

$$(\lambda,\, \lambda) \tag{SUB_R_2}$$

We have no remaining input to consume for this sub-derivation, and our candidate set consists of one result schema, SUB_R_2. Our result set for this sub-derivation is the set $\{\lambda\}$. We rewrite the query $(B\,?\,\lambda)$ to λ in the configuration of R_3. Our candidate set becomes:

$$(\langle \lambda,\, w_3\rangle\, \langle (C\,?\,\lambda),\, w_4\rangle,\, \lambda) \tag{R_5}$$

$$(\mathtt{a}\,\langle A,\, w_5\rangle\,\langle(B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\,\langle(C\,?\,\mathtt{a}\,w_5),\, w_4\rangle,\, \lambda) \tag{R_4}$$

Now the right-component of the pair $\langle\lambda,\, w_3\rangle$ is an answer, so it can be rewritten using the implicit rule $\langle\lambda,\, \lambda\rangle \rightarrow \lambda$:

$$(\langle(C\,?\,\lambda),\, w_4\rangle,\, \lambda) \tag{R_5}$$

$$(\mathtt{a}\,\langle A,\, w_5\rangle\,\langle(B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\,\langle(C\,?\,\mathtt{a}\,w_5),\, w_4\rangle,\, \lambda) \tag{R_4}$$

Again there is a pair with a query, $\langle(C\,?\,\lambda),\, w_4\rangle$, which must be rewritten to an answer before the pair itself can be rewritten. Therefore another sub-derivation process is called with start symbol C and input syntax λ.

$$(\langle C,\, u_1\rangle,\, w_1) \tag{SUB_R_1}$$

The pair $\langle C,\, u_1\rangle$ must be rewritten using rules 6 and 7, creating result schemas SUB_R_2 and SUB_R_3 respectively:

$$(\lambda,\, \lambda) \tag{SUB_R_2}$$

$$(\mathtt{a}\,\langle C,\, u_2\rangle,\, \mathtt{c}\,u_2) \tag{SUB_R_3}$$

We can now match each result schema against an input prefix of size one, λ. Only SUB_R_2 can match λ, so SUB_R_3 is discarded. The candidate set for this sub-derivation becomes:

$$(\lambda,\, \lambda) \tag{SUB_R_2}$$

There is no remaining input to match against for this sub-derivation, so the result set it has computed is the singleton set $\{\lambda\}$. We replace the query $(C\,?\,\lambda)$ with λ in result schema R_5, creating new result schema R_6:

$$(\langle\lambda,\, w_4\rangle,\, \lambda) \tag{R_6}$$

$$(\mathtt{a}\,\langle A,\, w_5\rangle\,\langle(B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\,\langle(C\,?\,\mathtt{a}\,w_5),\, w_4\rangle,\, \lambda) \tag{R_4}$$

Now the pair $\langle\lambda,\, w_4\rangle$ can be rewritten using $\langle\lambda,\, \lambda\rangle \rightarrow \lambda$. The candidate set becomes:

$$(\lambda,\, \lambda) \tag{R_7}$$

$$(\mathtt{a}\,\langle A,\, w_5\rangle\,\langle(B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\,\langle(C\,?\,\mathtt{a}\,w_5),\, w_4\rangle,\, \lambda) \tag{R_4}$$

For this derivation, each result schema is now ready to be matched against an input prefix of size one. We look at the first symbol, `a` of our input `abc` and match it against the configuration for each result schema. Only R_4 can match a prefix of `a`. Result schema R_7 is discarded. Our candidate set of result schemas is now:

$$(\mathtt{a}\,\langle A,\, w_5\rangle\,\langle(B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\,\langle(C\,?\,\mathtt{a}\,w_5),\, w_4\rangle,\, \lambda) \tag{R_4}$$

The pair $\langle A,\, w_5\rangle$ must now be rewritten using rules 2 and 3, creating result schemas R_8 and R_9 in our candidate set:

$$(\mathtt{a}\,\langle(B\,?\,\mathtt{a}),\, w_3\rangle\,\langle(C\,?\,\mathtt{a}),\, w_4\rangle,\, \lambda) \tag{R_8}$$

$$(\mathtt{a}\,\mathtt{a}\,\langle A,\, w_6\rangle\,\langle(B\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\, w_3\rangle\,\langle(C\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\, w_4\rangle,\, \lambda) \tag{R_9}$$

We have the pair $\langle(B\,?\,\mathtt{a}),\, w_3\rangle$ in result schema R_8 which cannot be rewritten until the query in its left-component is rewritten to an answer. The tool calls another sub-derivation process with start symbol B and input syntax $\mathtt{a}$.

$$(\langle B,\, u_1\rangle,\, u_1) \tag{SUB_R_1}$$

The initial pair $\langle B,\, w_1\rangle$ in SUB_R_1 must be rewritten using rules 4 and 5 to two new result schemas. The candidate set becomes:

$$(\lambda,\, \lambda) \tag{SUB_R_2}$$

$$(\mathtt{a}\,\langle B,\, u_2\rangle,\, \mathtt{b}\,u_2) \tag{SUB_R_3}$$

The result schemas are ready to be matched against an input prefix of size one, $\mathtt{a}$. Only SUB_R_3 has a prefix matching $\mathtt{a}$, to SUB_R_2 is discarded from the candidate set.

$$(\mathtt{a}\,\langle B,\, u_2\rangle,\, \mathtt{b}\,u_2) \tag{SUB_R_3}$$

Now the pair $\langle B,\, w_2\rangle$ must be rewritten using rules 4 and 5 to two new result schemas. Our candidate set becomes:

$$(\mathtt{a},\, \mathtt{b}) \tag{SUB_R_4}$$

$$(\mathtt{a}\,\mathtt{a}\,\langle B,\, u_3\rangle,\, \mathtt{b}\,\mathtt{b}\,u_3) \tag{SUB_R_5}$$

Since there is no remaining input syntax to match against, SUB_R_5 is now discarded, since its configuration does not consist of answers alone, and it does not exactly match $\mathtt{a}$.

$$(\mathtt{a},\, \mathtt{b}) \tag{SUB_R_4}$$

The result set for this derivation is therefore $\{\mathtt{b}\}$. We replace the query $(C\,?\,\mathtt{a})$ with $\mathtt{a}$ in the configuration for R_8, creating new result schema R_{10}:

$$(\mathtt{a}\,\langle\mathtt{b},\, w_3\rangle\,\langle(C\,?\,\mathtt{a}),\, w_4\rangle,\, \lambda) \tag{R_{10}}$$

$$(\mathtt{a}\,\mathtt{a}\,\langle A,\, w_6\rangle\,\langle(B\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\, w_3\rangle\,\langle(C\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\, w_4\rangle,\, \lambda) \tag{R_9}$$

Now the pair $\langle\mathtt{b},\, w_3\rangle$ in R_{10} must be rewritten using the implicit rule $\langle\mathtt{a},\, \mathtt{a}\rangle \rightarrow \mathtt{a}$. The candidate set becomes:

$$(\mathtt{a}\,\mathtt{b}\,\langle(C\,?\,\mathtt{a}),\, w_4\rangle,\, \lambda) \tag{R_{11}}$$

$$(\mathsf{a}\,\mathsf{a}\,\langle A,\, w_6\rangle\,\langle (B\,?\,\mathsf{a}\,\mathsf{a}\,w_6),\, w_3\rangle\,\langle (C\,?\,\mathsf{a}\,\mathsf{a}\,w_6),\, w_4\rangle,\, \lambda) \qquad (R_9)$$

We can now match each result schema against an input prefix of size two, `ab`. The configuration for R_{11} matches `ab`, but the configuration for R_9 cannot. R_9 is discarded from the candidate set, which becomes:

$$(\mathsf{a}\,\mathsf{b}\,\langle (C\,?\,\mathsf{a}),\, w_4\rangle,\, \lambda) \qquad (R_{11})$$

The pair $\langle (C\,?\,\mathsf{a}),\, w_4\rangle$ must be rewritten, meaning we must first rewrite the query $(C\,?\,\mathsf{a})$ in its left-component. The tool calls a new recursive sub-derivation process with start symbol C and input syntax `a`:

$$(\langle C,\, u_1\rangle,\, u_1) \qquad (SUB\_R_1)$$

Firstly, the initial pair $\langle C,\, u_1\rangle$ must be rewritten using rules 6 and 7 to two new result schemas. The candidate set becomes:

$$(\lambda,\, \lambda) \qquad (SUB\_R_2)$$

$$(\mathsf{a}\,\langle C,\, u_2\rangle,\, \mathsf{c}\,u_2) \qquad (SUB\_R_3)$$

Now we can look at an input prefix of size one for this sub-derivation, `a` and match it against each result schema. We find that the configuration for SUB_R_2 cannot match this prefix, however the configuration for SUB_R_3 can. SUB_R_2 is discarded from the candidate set, which becomes:

$$(\mathsf{a}\,\langle C,\, u_2\rangle,\, \mathsf{c}\,u_2) \qquad (SUB\_R_3)$$

The pair $\langle C,\, w_2\rangle$ must be rewritten using rules 6 and 7:

$$(\mathsf{a},\, \mathsf{c}) \qquad (SUB\_R_4)$$

$$(\mathsf{a}\,\mathsf{a}\,\langle C,\, w_3\rangle,\, \mathsf{c}\,\mathsf{c}\,u_3) \qquad (SUB\_R_5)$$

There is no remaining input to match against for this sub-derivation. Therefore result schemas which do not exactly match `a` are discarded. Only the configuration for SUB_R_4 matches `a`, so SUB_R_5 is discarded at this point. The candidate set is now:

$$(\mathsf{a},\, \mathsf{c}) \qquad (SUB\_R_4)$$

We have our final result set for this sub-derivation, {`c`}. We now rewrite the pair $(C\,?\,\mathsf{a})$ in result schema R_{11} to `b`, creating new result schema R_{12}:

$$(\mathsf{a}\,\mathsf{b}\,\langle \mathsf{c},\, w_4\rangle,\, \lambda) \qquad (R_{12})$$

The pair $\langle \mathsf{c},\, w_4\rangle$ must be rewritten using the implicit rule $\langle \mathsf{c},\, \mathsf{c}\rangle \to \mathsf{c}$ in a new result schema. Our candidate set of result schemas becomes:

$$(\mathsf{a}\,\mathsf{b}\,\mathsf{c},\, \lambda) \qquad (R_{13})$$

We can now match R_{13} to the input prefix of size three, `abc`. We find that the configuration for R_{13} is able to match this prefix, so R_{13} is maintained in the candidate set. There is no more input left to match against, so the result set for this derivation is $\{\lambda\}$.

5.6.3 Peano Increment Example

The tool must also be able to handle cases where, after a derivation of some input syntax, the semantic result computed is a query. The query must be rewritten to an answer before we can return the result set containing the semantics of the derivation in question. Consider the RAG specification defined in Ex. 33.

Example 33. Consider the following RAG.

$$\langle S, (I \; ? \; v_1)\rangle \to \langle N, v_1\rangle \tag{1}$$

$$\langle N, \mathtt{0}\rangle \to \mathtt{0} \tag{2}$$

$$\langle N, \mathtt{s}\, v_1\rangle \to \mathtt{s}\, \langle N, v_1\rangle \tag{3}$$

$$\langle I, \mathtt{s}\, v_1\rangle \to \langle N, v_1\rangle \tag{4}$$

The terminal constants in this RAG are $\mathtt{s}, \mathtt{0}$, the nullary non-terminals are $\{S, N, I\}$, and the start symbol is S. The language defined by the start symbol S is the set of all natural numbers in Peano form, $\{\mathtt{0}, \mathtt{s0}, \mathtt{ss0}, ...\}$.

Example Derivation

Using the RAG specification in Ex. 33, we take S as our starting answer and search for a derivation for the string $\mathtt{0}$ and the associated semantic value. The algorithm begins with a single sentential form schema in its candidate set, R_1:

$$(\langle S,\ w_1\rangle,\ w_1) \tag{R_1}$$

The pair $\langle S, w_1\rangle$ must first be rewritten using rule 1. The candidate set becomes:

$$(\langle N,\ w_2\rangle,\ (I \,?\, w_2)) \tag{R_2}$$

Now the pair $\langle N, w_2\rangle$ must be rewritten using rules 2 and 3:

$$(\mathtt{0},\ (I \,?\, \mathtt{0})) \tag{R_3}$$

$$(\mathtt{s}\, \langle N, w_3\rangle,\ (I \,?\, \mathtt{s}\, w_3)) \tag{R_4}$$

We are ready to look at the input prefix of size one. We match each result schema's configuration against $\mathtt{0}$, and find that only the configuration for R_3 can match this symbol. R_4 is discarded, and the candidate set becomes:

$$(\mathtt{0},\ (I \,?\, \mathtt{0})) \tag{R_3}$$

There is no remaining input to match against, and we have a derivation of the syntax $\mathtt{0}$ from start symbol S. However, before the process can return a result set, the query $(I \,?\, \mathtt{0})$ must be rewritten to an answer if possible. The tool begins a new sub-derivation process with starting symbol I and input syntax `0`:

$$(\langle I,\, u_1\rangle,\, u_1) \qquad (SUB\_R_1)$$

The pair $\langle I,\, w_1\rangle$ in SUB_R_1 must first be rewritten by use of rule 4 to a new result schema:

$$(\langle N,\, u_2\rangle,\, \mathtt{s}\, u_2) \qquad (SUB\_R_2)$$

Now the pair $\langle N,\, u_2\rangle$ must be rewritten using rules 2 and 3, creating result schemas SUB_R_3 and SUB_R_4 respectively. The candidate set becomes:

$$(\mathtt{0},\, \mathtt{s}\,\mathtt{0}) \qquad (SUB\_R_3)$$

$$(\mathtt{s}\,\langle N,\, u_3\rangle,\, \mathtt{s}\,\mathtt{s}\, u_3) \qquad (SUB\_R_4)$$

Now for this sub-derivation we can match each result schema to the input prefix of size one, `0`. We find that SUB_R_3 can match the prefix `0`, but the configuration for SUB_R_4 cannot. Result schema SUB_R_4 is discarded from the candidate set, which is now:

$$(\mathtt{0},\, \mathtt{s}\,\mathtt{0}) \qquad (SUB\_R_3)$$

There is no more input remaining to consume, and we have a single result schema SUB_R_3 in the candidate set. Therefore the result set for the derivation of syntax `0` from start symbol B is the singleton set $\{\mathtt{s}\,\mathtt{0}\}$.

5.7 Formal Derivation Output

As discussed in §5.2, the process of deriving syntax involves leaving *unknowns*, gaps in configurations which are filled in later with values after further rewrite steps. This does not constitute a formal derivation sequence under the derivation step relation. Rewriting is instead done using unbound rules.

A goal of the tool is to output a formal derivation sequence by keeping track of the rewrite steps the tool uses to derive syntax, and replacing unknowns in all steps when their values are found by the algorithm.

5.7.1 Formal derivation of Result Schemas

The configuration component C of a result schema $(C,\, a)$ with unknowns $w_1, ..., w_n$ has a direct relation to a configuration in a formal derivation sequence. For all answers $a_1, ..., a_n$, we have that $\langle S,\, a[a_1/w_1, ..., a_n/w_n]\rangle \Rightarrow^* C[a_1/w_1, ..., a_n/w_n]$, where S is the start symbol of the RAG specification. In other words, after applying matching substitutions a_i to all the unknowns w_i in both the configuration C and its semantic value a, we have a formal derivation sequence $\langle S,\, a\rangle \Rightarrow^* C$. We can use this relation to build a formal derivation sequence alongside the derivation process of the tool. To do this, we must keep track of the derivation sequence for each configuration.

Definition 24. A *derivation schema* is a tuple (S, V, D) where S is a configuration possibly containing unknowns, and V a query, also possibly containing unknowns. D is a derivation sequence representing the derivation of a configuration containing unknowns, for each step.

We progressively build towards a formal derivation sequence by maintaining the rewrite steps used to reach its configuration. The unknowns it contains are substituted in the same way as the unknowns contained within the derivation schema's configuration, for each step in the sequence.

5.7.2 Rewriting Without Inverses

The strategy introduced for generating the language defined by a RAG specification in this chapter makes no use of the inverse operator, which is pivotal to query rewriting in a formal derivation.

The rewrite sequence $a \,?\, b \overset{*}{\Rightarrow} c$ relies on Axiom 5.23d of Shutt's thesis [1] to work which says that, for $a_1, a_2 \in A_G$, $(a_1 \,?\, !(\langle a_1, a_2\rangle)) \Rightarrow a_2$. We have to use the inverse operator when performing such rewrites.

By Proposition 4.4, we can instead interpret a query rewrite as $\langle a, c\rangle \Rightarrow^* b$, a derivation of syntax b, taking a as our start symbol, and computing result semantic result c. This correlates to the tool's derivation process, whose higher-level goal is also to find a derivation $\langle a, c\rangle \Rightarrow^* b$ if it exists. Therefore, we can handle queries by converting them to an answer via a recursive call to the derivation algorithm, which will return a set of semantic results if one or more derivations are found.

However, if we want to produce a formal derivation output with the tool, we must take these recursive calls to the derivation algorithm and convert their rewrite sequences such that they make use of the axioms which define Shutt's derivation step relation.

5.7.3 Triple-ABC Example

To illustrate how formal derivation sequences are generated in cases where we have queries as the left-component of pairs, we re-visit the RAG specification defined in §5.6.2, Ex. 32.

Example 34. Consider the following RAG.

$$\langle S, \lambda\rangle \to \langle A, v_1\rangle \, \langle (B \;?\; v_1), v_2\rangle \, \langle (C \;?\; v_1), v_3\rangle \tag{1}$$

$$\langle A, \lambda\rangle \to \lambda \tag{2}$$

$$\langle A, \mathtt{a}\, v_1\rangle \to \mathtt{a}\, \langle A, v_1\rangle \tag{3}$$

$$\langle B, \lambda\rangle \to \lambda \tag{4}$$

$$\langle B, \mathtt{b}\, v_1\rangle \to \mathtt{a}\, \langle B, v_1\rangle \tag{5}$$

$$\langle C, \lambda\rangle \to \lambda \tag{6}$$

$$\langle C, \mathtt{c}\, v_1\rangle \to \mathtt{a}\, \langle C, v_1\rangle \tag{7}$$

The terminal constants in this RAG are $\mathtt{a}, \mathtt{b}, \mathtt{c}$, the nullary non-terminals are $\{S, A, B, C\}$, and the start symbol is S. The language defined by the start symbol S is $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ for $n \geq 0$.

Example Derivation

With S as our starting answer and using the RAG specification defined in Ex. 34, we try to derive the syntax `abc`. We start with a candidate set consisting of the initial derivation schema R_1:

$$(\langle S,\, w_1\rangle,\, w_1,\, \langle S,\, w_1\rangle) \tag{R_1}$$

The derivation steps are current identical to the configuration $\langle S,\, w_1\rangle$. The pair $\langle S,\, w_1\rangle$ must be rewritten using rule 1 to a new result schema R_2:

$$\begin{aligned}&(\langle A,\, w_2\rangle\, \langle (B\,?\,w_2),\, w_3\rangle\, \langle (C\,?\,w_2),\, w_4\rangle,\, \lambda,\\ &\qquad \langle S,\, \lambda\rangle \Rightarrow \langle A,\, w_2\rangle\, \langle (B\,?\,w_2),\, w_3\rangle\, \langle (C\,?\,w_2),\, w_4\rangle\\ &)\end{aligned} \tag{R_2}$$

The unknown w_1 in the derivation steps has been substituted for λ, and the rewrite step just applied to the configuration has been appended. Now the pair $\langle A,\, w_2\rangle$ must be rewritten using rules 2 and 3:

$$\begin{aligned}&(\langle (B\,?\,\lambda),\, w_3\rangle\, \langle (C\,?\,\lambda),\, w_4\rangle,\, \lambda,\\ &\qquad \langle S,\, \lambda\rangle\\ &\qquad \Rightarrow \langle A,\, \lambda\rangle\, \langle (B\,?\,\lambda),\, w_3\rangle\, \langle (C\,?\,\lambda),\, w_4\rangle\\ &\qquad \Rightarrow \langle (B\,?\,\lambda),\, w_3\rangle\, \langle (C\,?\,\lambda),\, w_4\rangle\\ &)\end{aligned} \tag{R_3}$$

$$\begin{aligned}&(\mathtt{a}\,\langle A,\, w_5\rangle\, \langle (B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\, \langle (C\,?\,\mathtt{a}\,w_5),\, w_4\rangle,\, \lambda,\\ &\qquad \langle S,\, \lambda\rangle\\ &\qquad \Rightarrow \langle A,\, \mathtt{a}\,w_5\rangle\, \langle (B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\, \langle (C\,?\,\mathtt{a}\,w_5),\, w_4\rangle\\ &\qquad \Rightarrow \mathtt{a}\,\langle A,\, w_5\rangle\, \langle (B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\, \langle (C\,?\,\mathtt{a}\,w_5),\, w_4\rangle\\ &)\end{aligned} \tag{R_4}$$

In the derivation steps for both derivation schemas, the unknown w_2 has been substituted in all steps for the semantic value of the rule used to rewrite. The pair $\langle (B\,?\,\lambda),\, w_3\rangle$ in R_3 must be rewritten. We begin a new sub-derivation of the syntax λ with starting answer B.

$$(\langle B,\, w_1\rangle,\, w_1,\, \langle B,\, w_1\rangle) \tag{SUB_R_1}$$

We must rewrite the pair $\langle B,\, w_1\rangle$ using rules 4 and 5. Each step is applied to the derivation steps of the two new result schemas created, and the unknown in the pair rewritten is substituted:

$$(\lambda,\, \lambda,\, \langle B,\, \lambda\rangle \Rightarrow \lambda) \tag{SUB_R_2}$$

$$(\mathtt{a}\,\langle B,\, w_2\rangle,\, \mathtt{b}\,w_2,\, \langle B,\, \mathtt{b}\,w_2\rangle \Rightarrow \mathtt{a}\,\langle B,\, w_2\rangle) \qquad (SUB\_R_3)$$

Now we are able to match against a prefix of size one of the input for this sub-derivation, λ. Only SUB_R_2 can match λ, SUB_R_3 is discarded from the candidate, which becomes:

$$(\lambda,\, \lambda, \langle B,\, \lambda\rangle \Rightarrow \lambda) \qquad (SUB\_R_2)$$

There is no remaining input to consume for this sub-derivation. The candidate set consists of one result schema, SUB_R_2. The result set for this sub-derivation is $\{\lambda\}$. We rewrite the query $(B\,?\,\lambda)$ to λ in the configuration of R_3.

We have a sequence $\langle B,\, \lambda\rangle \Rightarrow \lambda$ for this sub-derivation. To apply this to the rewrite steps for derivation schema R_3, we must represent this sequence in formal terms. λ is an answer, so we know that $!(\lambda) = \lambda$. We need to embed the pair $\langle B,\, \lambda\rangle$ in a term which constitutes a rewrite step using Axiom 5.23d of Shutt's thesis [1]. We can do this by surrounding the pair with an inverse operator, and use this as the right-operand of the query we are rewriting, i.e. $(B\,?\,!(\langle B,\, \lambda\rangle))$. By embedding these modified steps to the derivation steps for R_5, we have part of a formal derivation sequence which rewrites a query.

$$\begin{aligned}
&(\langle \lambda,\, w_3\rangle\, \langle (C\,?\,\lambda),\, w_4\rangle,\, \lambda,\\
&\quad \langle S,\, \lambda\rangle\\
&\quad \Rightarrow \langle A,\, \lambda\rangle\, \langle (B\,?\,\lambda),\, w_3\rangle\, \langle (C\,?\,\lambda),\, w_4\rangle\\
&\quad \Rightarrow \langle (B\,?\,\lambda),\, w_3\rangle\, \langle (C\,?\,\lambda),\, w_4\rangle) \qquad (R_5)\\
&\quad \Rightarrow \langle (B\,?\,!(\langle B,\, \lambda\rangle)),\, w_3\rangle\, \langle (C\,?\,\lambda),\, w_4\rangle)\\
&\quad \Rightarrow \langle \lambda,\, w_3\rangle\, \langle (C\,?\,\lambda),\, w_4\rangle\\
&)
\end{aligned}$$

$$\begin{aligned}
&(\mathtt{a}\,\langle A,\, w_5\rangle\, \langle (B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\, \langle (C\,?\,\mathtt{a}\,w_5),\, w_4\rangle,\, \lambda,\\
&\quad \langle S,\, \lambda\rangle\\
&\quad \Rightarrow \langle A,\, \mathtt{a}\,w_5\rangle\, \langle (B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\, \langle (C\,?\,\mathtt{a}\,w_5),\, w_4\rangle \qquad (R_4)\\
&\quad \Rightarrow \mathtt{a}\,\langle A,\, w_5\rangle\, \langle (B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\, \langle (C\,?\,\mathtt{a}\,w_5),\, w_4\rangle\\
&)
\end{aligned}$$

Now the right-component of the pair $\langle \lambda,\, w_3\rangle$ is an answer, so it can be rewritten using the implicit rule $\langle \lambda,\, \lambda\rangle \to \lambda$. Each w_3 is substituted for λ. The candidate set becomes:

$$\begin{aligned}
&(\langle (C\,?\,\lambda),\, w_4\rangle,\, \lambda,\\
&\qquad \langle S,\, \lambda\rangle\\
&\qquad\quad \Rightarrow \langle A,\, \lambda\rangle\, \langle (B\,?\,\lambda),\, \lambda\rangle\, \langle (C\,?\,\lambda),\, w_4\rangle\\
&\qquad\quad \Rightarrow \langle (B\,?\,\lambda),\, \lambda\rangle\, \langle (C\,?\,\lambda),\, w_4\rangle)\\
&\qquad\quad \Rightarrow \langle (B\,?\,!(\langle B,\, \lambda\rangle)),\, \lambda\rangle\, \langle (C\,?\,\lambda),\, w_4\rangle)\\
&\qquad\quad \Rightarrow \langle \lambda,\, \lambda\rangle\, \langle (C\,?\,\lambda),\, w_4\rangle)\\
&\qquad\quad \Rightarrow \langle (C\,?\,\lambda),\, w_4\rangle\\
&)
\end{aligned} \tag{R_6}$$

$$\begin{aligned}
&(\mathtt{a}\,\langle A,\, w_5\rangle\, \langle (B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\, \langle (C\,?\,\mathtt{a}\,w_5),\, w_4\rangle,\, \lambda,\\
&\qquad \langle S,\, \lambda\rangle\\
&\qquad\quad \Rightarrow \langle A,\, \mathtt{a}\,w_5\rangle\, \langle (B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\, \langle (C\,?\,\mathtt{a}\,w_5),\, w_4\rangle\\
&\qquad\quad \Rightarrow \mathtt{a}\,\langle A,\, w_5\rangle\, \langle (B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\, \langle (C\,?\,\mathtt{a}\,w_5),\, w_4\rangle\\
&)
\end{aligned} \tag{R_4}$$

Again there is a pair with a query, $\langle (C\,?\,\lambda),\, w_4\rangle$, which must be rewritten to an answer before the pair itself can be rewritten. Therefore another sub-derivation process is called with start symbol C and input syntax λ.

$$(\langle C,\, w_1\rangle,\, w_1,\, \langle C,\, w_1\rangle) \tag{SUB_R_1}$$

The pair $\langle C,\, w_1\rangle$ must be rewritten using rules 6 and 7, creating result schemas SUB_R_2 and SUB_R_3 respectively:

$$(\lambda,\, \lambda,\, \langle C,\, \lambda\rangle \Rightarrow \lambda) \tag{SUB_R_2}$$

$$(\mathtt{a}\,\langle C,\, w_2\rangle,\, \mathtt{c}\,w_2,\, \langle C,\, \mathtt{c}\,w_2\rangle \Rightarrow \mathtt{a}\,\langle C,\, w_2\rangle) \tag{SUB_R_3}$$

We can now match each result schema against an input prefix of size one, λ. Only SUB_R_2 can match λ, so SUB_R_3 is discarded. The candidate set for this sub-derivation becomes:

$$(\lambda,\, \lambda,\, \langle C,\, \lambda\rangle \Rightarrow \lambda) \tag{SUB_R_2}$$

There is no remaining input to match against for this sub-derivation, so the result set it has computed is the singleton set $\{\lambda\}$. We replace the query $(C\,?\,\lambda)$ with λ in result schema R_5, creating new result schema R_6. We have the sub-derivation sequence $\langle C,\, \lambda\rangle \Rightarrow \lambda$. As with the previous query rewrite for R_6, we can rewrite this sequence as $(C\,?\,!(\langle C,\, \lambda\rangle)) \Rightarrow \lambda$ and embed each step in the derivation schema.

$$(\langle \lambda,\, w_4\rangle,\, \lambda) \tag{R_7}$$

$$
\begin{array}{l}
(\langle \lambda,\, w_4\rangle,\, \lambda,\\
\quad \langle S,\, \lambda\rangle\\
\quad \Rightarrow \langle A,\, \lambda\rangle\, \langle (B\,?\,\lambda),\, \lambda\rangle\, \langle (C\,?\,\lambda),\, w_4\rangle\\
\quad \Rightarrow \langle (B\,?\,\lambda),\, \lambda\rangle\, \langle (C\,?\,\lambda),\, w_4\rangle)\\
\quad \Rightarrow \langle (B\,?\,!(\langle B,\, \lambda\rangle)),\, \lambda\rangle\, \langle (C\,?\,\lambda),\, w_4\rangle)\\
\quad \Rightarrow \langle \lambda,\, \lambda\rangle\, \langle (C\,?\,\lambda),\, w_4\rangle)\\
\quad \Rightarrow \langle (C\,?\,\lambda),\, w_4\rangle)\\
\quad \Rightarrow \langle (C\,?\,!(\langle C,\, \lambda\rangle)),\, w_4\rangle)\\
\quad \Rightarrow \langle \lambda,\, w_4\rangle\\
)
\end{array}
\qquad (R_7)
$$

$$
\begin{array}{l}
(\mathtt{a}\,\langle A,\, w_5\rangle\, \langle (B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\, \langle (C\,?\,\mathtt{a}\,w_5),\, w_4\rangle,\, \lambda,\\
\quad \langle S,\, \lambda\rangle\\
\quad \Rightarrow \langle A,\, \mathtt{a}\,w_5\rangle\, \langle (B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\, \langle (C\,?\,\mathtt{a}\,w_5),\, w_4\rangle\\
\quad \Rightarrow \mathtt{a}\,\langle A,\, w_5\rangle\, \langle (B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\, \langle (C\,?\,\mathtt{a}\,w_5),\, w_4\rangle\\
)
\end{array}
\qquad (R_4)
$$

Now the pair $\langle \lambda,\, w_4\rangle$ can be rewritten using $\langle \lambda,\, \lambda\rangle \to \lambda$. The candidate set becomes:

$$
\begin{array}{l}
(\lambda,\, \lambda,\\
\quad \langle S,\, \lambda\rangle\\
\quad \Rightarrow \langle A,\, \lambda\rangle\, \langle (B\,?\,\lambda),\, \lambda\rangle\, \langle (C\,?\,\lambda),\, \lambda\rangle\\
\quad \Rightarrow \langle (B\,?\,\lambda),\, \lambda\rangle\, \langle (C\,?\,\lambda),\, \lambda\rangle)\\
\quad \Rightarrow \langle (B\,?\,!(\langle B,\, \lambda\rangle)),\, \lambda\rangle\, \langle (C\,?\,\lambda),\, \lambda\rangle)\\
\quad \Rightarrow \langle \lambda,\, \lambda\rangle\, \langle (C\,?\,\lambda),\, \lambda\rangle)\\
\quad \Rightarrow \langle (C\,?\,\lambda),\, \lambda\rangle)\\
\quad \Rightarrow \langle (C\,?\,!(\langle C,\, \lambda\rangle)),\, \lambda\rangle)\\
\quad \Rightarrow \langle \lambda,\, \lambda\rangle\\
)
\end{array}
\qquad (R_8)
$$

$$
\begin{array}{l}
(\mathtt{a}\,\langle A,\, w_5\rangle\, \langle (B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\, \langle (C\,?\,\mathtt{a}\,w_5),\, w_4\rangle,\, \lambda,\\
\quad \langle S,\, \lambda\rangle\\
\quad \Rightarrow \langle A,\, \mathtt{a}\,w_5\rangle\, \langle (B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\, \langle (C\,?\,\mathtt{a}\,w_5),\, w_4\rangle\\
\quad \Rightarrow \mathtt{a}\,\langle A,\, w_5\rangle\, \langle (B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\, \langle (C\,?\,\mathtt{a}\,w_5),\, w_4\rangle\\
)
\end{array}
\qquad (R_4)
$$

Each result schema is now ready to be matched against an input prefix of size one. We look at the first symbol, `a` of our input `abc` and match it against the configuration for each result schema. Only R_4 can match a prefix of `a`. Result schema R_7 is discarded. Our candidate set of result schemas is now:

$$
\begin{aligned}
&(\mathtt{a}\,\langle A,\, w_5\rangle\,\langle (B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\,\langle (C\,?\,\mathtt{a}\,w_5),\, w_4\rangle,\ \lambda,\\
&\quad \langle S,\, \lambda\rangle\\
&\quad\ \Rightarrow \langle A,\, \mathtt{a}\,w_5\rangle\,\langle (B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\,\langle (C\,?\,\mathtt{a}\,w_5),\, w_4\rangle\\
&\quad\ \Rightarrow \mathtt{a}\,\langle A,\, w_5\rangle\,\langle (B\,?\,\mathtt{a}\,w_5),\, w_3\rangle\,\langle (C\,?\,\mathtt{a}\,w_5),\, w_4\rangle\\
&)
\end{aligned}
\tag{R_4}
$$

The pair $\langle A,\, w_5\rangle$ must now be rewritten using rules 2 and 3, creating result schemas R_8 and R_9 in our candidate set:

$$
\begin{aligned}
&(\mathtt{a}\,\langle (B\,?\,\mathtt{a}),\, w_3\rangle\,\langle (C\,?\,\mathtt{a}),\, w_4\rangle,\ \lambda,\\
&\quad \langle S,\, \lambda\rangle\\
&\quad\ \Rightarrow \langle A,\, \mathtt{a}\rangle\,\langle (B\,?\,\mathtt{a}),\, w_3\rangle\,\langle (C\,?\,\mathtt{a}),\, w_4\rangle\\
&\quad\ \Rightarrow \mathtt{a}\,\langle A,\, \lambda\rangle\,\langle (B\,?\,\mathtt{a}),\, w_3\rangle\,\langle (C\,?\,\mathtt{a}),\, w_4\rangle)\\
&\quad\ \Rightarrow \mathtt{a}\,\langle (B\,?\,\mathtt{a}),\, w_3\rangle\,\langle (C\,?\,\mathtt{a}),\, w_4\rangle\\
&)
\end{aligned}
\tag{R_9}
$$

$$
\begin{aligned}
&(\mathtt{a}\,\mathtt{a}\,\langle A,\, w_6\rangle\,\langle (B\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\, w_3\rangle\,\langle (C\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\, w_4\rangle,\ \lambda,\\
&\quad \langle S,\, \lambda\rangle\\
&\quad\ \Rightarrow \langle A,\, \mathtt{a}\,\mathtt{a}\,w_6\rangle\,\langle (B\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\, w_3\rangle\,\langle (C\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\, w_4\rangle\\
&\quad\ \Rightarrow \mathtt{a}\,\langle A,\, \mathtt{a}\,w_6\rangle\,\langle (B\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\, w_3\rangle\,\langle (C\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\, w_4\rangle\\
&)
\end{aligned}
\tag{R_{10}}
$$

We have the pair $\langle (B\,?\,\mathtt{a}),\, w_3\rangle$ in result schema R_8 which cannot be rewritten until the query in its left-component is rewritten to an answer. The tool calls another sub-derivation process with start symbol B and input syntax $\mathtt{a}$.

$$(\langle B,\, w_1\rangle,\, w_1,\, \langle B,\, w_1\rangle) \tag{SUB_R_1}$$

The initial pair $\langle B,\, w_1\rangle$ in SUB_R_1 must be rewritten using rules 4 and 5 to two new result schemas. The candidate set becomes:

$$(\lambda,\, \lambda,\, \langle B,\, \lambda\rangle \Rightarrow \lambda) \tag{SUB_R_2}$$

$$(\mathtt{a}\,\langle B,\, w_2\rangle,\, \mathtt{b}\,w_2,\, \langle B,\, \mathtt{b}\,w_2\rangle \Rightarrow \mathtt{a}\,\langle B,\, w_2\rangle) \tag{SUB_R_3}$$

The result schemas are ready to be matched against an input prefix of size one, $\mathtt{a}$. Only SUB_R_3 has a prefix matching $\mathtt{a}$, to SUB_R_2 is discarded from the candidate set.

$$(\mathtt{a}\,\langle B,\, w_2\rangle,\, \mathtt{b}\,w_2,\, \langle B,\, \mathtt{b}\,w_2\rangle \Rightarrow \mathtt{a}\,\langle B,\, w_2\rangle) \tag{SUB_R_3}$$

Now the pair $\langle B,\, w_2\rangle$ must be rewritten using rules 4 and 5 to two new result schemas. Our candidate set becomes:

$$
\begin{array}{ll}
(\mathtt{a},\ \mathtt{b}, & \\
\quad \langle B,\ \mathtt{b}\rangle & \\
\quad \Rightarrow \mathtt{a}\,\langle B, \lambda\rangle & (SUB\_R_4) \\
\quad \Rightarrow \mathtt{a} & \\
) &
\end{array}
$$

$$
\begin{array}{ll}
(\mathtt{a}\,\mathtt{a}\,\langle B,\ w_3\rangle,\ \mathtt{b}\,\mathtt{b}\,w_3, & \\
\quad \langle B,\ \mathtt{b}\,\mathtt{b}\,w_3\rangle & \\
\quad \Rightarrow \mathtt{a}\,\langle B,\ \mathtt{b}\,w_3\rangle & (SUB\_R_5) \\
\quad \Rightarrow \mathtt{a}\,\mathtt{a}\,\langle B, w3\rangle & \\
) &
\end{array}
$$

Since there is no remaining input syntax to match against, SUB_R_5 is now discarded, since its configuration does not consist of answers alone, and it does not exactly match a.

$$
\begin{array}{ll}
(\mathtt{a},\ \mathtt{b}, & \\
\quad \langle B,\ \mathtt{b}\rangle & \\
\quad \Rightarrow \mathtt{a}\,\langle B, \lambda\rangle & (SUB\_R_4) \\
\quad \Rightarrow \mathtt{a} & \\
) &
\end{array}
$$

The result set for this derivation is therefore $\{\mathtt{b}\}$. We replace the query $(C\,?\,\mathtt{a})$ with a in the configuration for R_8, creating new result schema R_{10}.

We have the sub-derivation sequence $\langle B,\ \mathtt{b}\rangle \Rightarrow \mathtt{a}\,\langle B, \lambda\rangle \Rightarrow \mathtt{a}$. We reverse, the sequence, and embed the steps in an inverse operator, and use this as the right-operand of the original query. We also append the result of the sub-derivation, b. This results in the sequence $(B\,?\,\mathtt{a}) \Rightarrow (B\,?\,!(\mathtt{a}\,\langle B,\ \lambda\rangle)) \Rightarrow (B\,?\,!(\langle B,\ \mathtt{b}\rangle)) \Rightarrow \mathtt{b}$. These steps are embedded in place of the original query in R_9 in a new derivation schema R_{11}.

$$
\begin{array}{ll}
(\mathtt{a}\,\langle \mathtt{b},\ w_3\rangle\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle,\ \lambda, & \\
\quad \langle S,\ \lambda\rangle & \\
\quad \Rightarrow \langle A,\ \mathtt{a}\rangle\,\langle (B\,?\,\mathtt{a}),\ w_3\rangle\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle & \\
\quad \Rightarrow \mathtt{a}\,\langle A,\ \lambda\rangle\,\langle (B\,?\,\mathtt{a}),\ w_3\rangle\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle) & \\
\quad \Rightarrow \mathtt{a}\,\langle (B\,?\,\mathtt{a}),\ w_3\rangle\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle) & (R_{11}) \\
\quad \Rightarrow \mathtt{a}\,\langle (B\,?\,!(\mathtt{a}\,\langle B,\ \lambda\rangle)),\ w_3\rangle\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle) & \\
\quad \Rightarrow \mathtt{a}\,\langle (B\,?\,!(\langle B,\ \mathtt{b}\rangle)),\ w_3\rangle\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle) & \\
\quad \Rightarrow \mathtt{a}\,\langle \mathtt{b},\ w_3\rangle\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle & \\
) &
\end{array}
$$

$$\begin{array}{l}
(\mathtt{a}\,\mathtt{a}\,\langle A,\ w_6\rangle\,\langle (B\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\ w_3\rangle\,\langle (C\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\ w_4\rangle,\ \lambda, \\
\quad \langle S,\ \lambda\rangle \\
\quad \Rightarrow \langle A,\ \mathtt{a}\,\mathtt{a}\,w_6\rangle\,\langle (B\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\ w_3\rangle\,\langle (C\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\ w_4\rangle \\
\quad \Rightarrow \mathtt{a}\,\langle A,\ \mathtt{a}\,w_6\rangle\,\langle (B\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\ w_3\rangle\,\langle (C\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\ w_4\rangle \\
)
\end{array} \qquad (R_{10})$$

Now the pair $\langle \mathtt{b},\ w_3\rangle$ in R_{11} must be rewritten using the implicit rule $\langle \mathtt{a},\ \mathtt{a}\rangle \rightarrow \mathtt{a}$. The candidate set becomes:

$$\begin{array}{l}
(\mathtt{a}\,\mathtt{b}\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle,\ \lambda, \\
\quad \langle S,\ \lambda\rangle \\
\quad \Rightarrow \langle A,\ \mathtt{a}\rangle\,\langle (B\,?\,\mathtt{a}),\ \mathtt{b}\rangle\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle \\
\quad \Rightarrow \mathtt{a}\,\langle A,\ \lambda\rangle\,\langle (B\,?\,\mathtt{a}),\ \mathtt{b}\rangle\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle) \\
\quad \Rightarrow \mathtt{a}\,\langle (B\,?\,\mathtt{a}),\ \mathtt{b}\rangle\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle) \\
\quad \Rightarrow \mathtt{a}\,\langle (B\,?\,\mathtt{!}(\mathtt{a}\,\langle B,\ \lambda\rangle)),\ \mathtt{b}\rangle\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle) \\
\quad \Rightarrow \mathtt{a}\,\langle (B\,?\,\mathtt{!}(\langle B,\ \mathtt{b}\rangle)),\ \mathtt{b}\rangle\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle) \\
\quad \Rightarrow \mathtt{a}\,\langle \mathtt{b},\ \mathtt{b}\rangle\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle) \\
\quad \Rightarrow \mathtt{a}\,\mathtt{b}\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle \\
)
\end{array} \qquad (R_{12})$$

$$\begin{array}{l}
(\mathtt{a}\,\mathtt{a}\,\langle A,\ w_6\rangle\,\langle (B\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\ w_3\rangle\,\langle (C\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\ w_4\rangle,\ \lambda, \\
\quad \langle S,\ \lambda\rangle \\
\quad \Rightarrow \langle A,\ \mathtt{a}\,\mathtt{a}\,w_6\rangle\,\langle (B\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\ w_3\rangle\,\langle (C\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\ w_4\rangle \\
\quad \Rightarrow \mathtt{a}\,\langle A,\ \mathtt{a}\,w_6\rangle\,\langle (B\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\ w_3\rangle\,\langle (C\,?\,\mathtt{a}\,\mathtt{a}\,w_6),\ w_4\rangle \\
)
\end{array} \qquad (R_{10})$$

We can now match each result schema against an input prefix of size two, `ab`. The configuration for R_{12} matches `ab`, but the configuration for R_{10} cannot. R_{10} is discarded from the candidate set, which becomes:

$$\begin{array}{l}
(\mathtt{a}\,\mathtt{b}\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle,\ \lambda, \\
\quad \langle S,\ \lambda\rangle \\
\quad \Rightarrow \langle A,\ \mathtt{a}\rangle\,\langle (B\,?\,\mathtt{a}),\ \mathtt{b}\rangle\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle \\
\quad \Rightarrow \mathtt{a}\,\langle A,\ \lambda\rangle\,\langle (B\,?\,\mathtt{a}),\ \mathtt{b}\rangle\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle) \\
\quad \Rightarrow \mathtt{a}\,\langle (B\,?\,\mathtt{a}),\ \mathtt{b}\rangle\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle) \\
\quad \Rightarrow \mathtt{a}\,\langle (B\,?\,\mathtt{!}(\mathtt{a}\,\langle B,\ \lambda\rangle)),\ \mathtt{b}\rangle\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle) \\
\quad \Rightarrow \mathtt{a}\,\langle (B\,?\,\mathtt{!}(\langle B,\ \mathtt{b}\rangle)),\ \mathtt{b}\rangle\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle) \\
\quad \Rightarrow \mathtt{a}\,\langle \mathtt{b},\ \mathtt{b}\rangle\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle) \\
\quad \Rightarrow \mathtt{a}\,\mathtt{b}\,\langle (C\,?\,\mathtt{a}),\ w_4\rangle \\
)
\end{array} \qquad (R_{12})$$

The pair $\langle (C\,?\,\mathtt{a}),\ w_4\rangle$ must be rewritten, meaning we must first rewrite the query $(C\,?\,\mathtt{a})$ in its left-component. The tool calls a new recursive sub-derivation process with start symbol C and input syntax `a`:

$$(\langle C,\, w_1\rangle,\, w_1, \langle C,\, w_1\rangle) \qquad (SUB\_R_1)$$

Firstly, the initial pair $\langle C,\, w_1\rangle$ must be rewritten using rules 6 and 7 to two new result schemas. The candidate set becomes:

$$(\lambda,\, \lambda, \langle C,\, \lambda\rangle \Rightarrow \lambda) \qquad (SUB\_R_2)$$

$$(\mathtt{a}\,\langle C,\, w_2\rangle,\, \mathtt{c}\, w_2, \langle C,\, \mathtt{c}\, w_2\rangle \Rightarrow \mathtt{a}\,\langle C,\, w_2\rangle) \qquad (SUB\_R_3)$$

Now we can look at an input prefix of size one for this sub-derivation, a and match it against each result schema. We find that the configuration for SUB_R_2 cannot match this prefix, however the configuration for SUB_R_3 can. SUB_R_2 is discarded from the candidate set, which becomes:

$$(\mathtt{a}\,\langle C,\, w_2\rangle,\, \mathtt{c}\, w_2, \langle C,\, \mathtt{c}\, w_2\rangle \Rightarrow \mathtt{a}\,\langle C,\, w_2\rangle) \qquad (SUB\_R_3)$$

The pair $\langle C,\, w_2\rangle$ must be rewritten using rules 6 and 7:

$$\begin{array}{l}(\mathtt{a},\, \mathtt{c} \\ \quad \langle C,\, \mathtt{c}\rangle \\ \quad \Rightarrow \mathtt{a}\,\langle C,\, \lambda\rangle \\ \quad \Rightarrow \mathtt{a} \\ )\end{array} \qquad (SUB\_R_4)$$

$$\begin{array}{l}(\mathtt{a}\,\mathtt{a}\,\langle C,\, w_3\rangle,\, \mathtt{c}\,\mathtt{c}\, w_3, \\ \quad \langle C,\, \mathtt{c}\,\mathtt{c}\, w_3\rangle\rangle \\ \quad \Rightarrow \mathtt{a}\,\langle C,\, \mathtt{c}\, w_3\rangle \\ \quad \Rightarrow \mathtt{a}\,\mathtt{a}\,\langle C,\, w_3\rangle \\ )\end{array} \qquad (SUB\_R_5)$$

There is no remaining input to match against for this sub-derivation. Therefore result schemas which do not exactly match a are discarded. Only the configuration for SUB_R_4 matches a, so SUB_R_5 is discarded at this point. The candidate set is now:

$$\begin{array}{l}(\mathtt{a},\, \mathtt{c} \\ \quad \langle C,\, \mathtt{c}\rangle \\ \quad \Rightarrow \mathtt{a}\,\langle C,\, \lambda\rangle \\ \quad \Rightarrow \mathtt{a} \\ )\end{array} \qquad (SUB\_R_4)$$

We have our final result set for this sub-derivation, {c}. We now rewrite the pair $(C\,?\,\mathtt{a})$ in result schema R_{12} to b, creating new result schema R_{13}.

We have the rewrite steps $\langle C,\, \mathtt{c}\rangle \Rightarrow \mathtt{a}\,\langle C,\, \lambda\rangle \Rightarrow \mathtt{a}$ for this sub-derivation sequence. We reverse this sequence, embed its steps in an inverse operator used as the right-operand of the original query, and append the result c.

The steps become: $(C \,?\, \mathtt{a}) \Rightarrow (C\,?\,!(\mathtt{a}\,\langle C,\,\lambda\rangle)) \Rightarrow (C\,?\,!(\langle C,\,\mathtt{c}\rangle)) \Rightarrow \mathtt{c}$. New derivation steps are generated where the original query has been substituted for each step. The candidate set becomes:

$$
\begin{array}{ll}
(\mathtt{a\,b}\,\langle \mathtt{c},\, w_4\rangle,\, \lambda, & \\
\quad \langle S,\, \lambda\rangle & \\
\quad \Rightarrow \langle A,\, \mathtt{a}\rangle\,\langle (B\,?\,\mathtt{a}),\, w_3\rangle\,\langle (C\,?\,\mathtt{a}),\, w_4\rangle & \\
\quad \Rightarrow \mathtt{a}\,\langle A,\, \lambda\rangle\,\langle (B\,?\,\mathtt{a}),\, w_3\rangle\,\langle (C\,?\,\mathtt{a}),\, w_4\rangle) & \\
\quad \Rightarrow \mathtt{a}\,\langle (B\,?\,\mathtt{a}),\, w_3\rangle\,\langle (C\,?\,\mathtt{a}),\, w_4\rangle) & \\
\quad \Rightarrow \mathtt{a}\,\langle (B\,?\,!(\mathtt{a}\,\langle B,\, \lambda\rangle)),\, w_3\rangle\,\langle (C\,?\,\mathtt{a}),\, w_4\rangle) & \\
\quad \Rightarrow \mathtt{a}\,\langle (B\,?\,!(\langle B,\, \mathtt{b}\rangle)),\, w_3\rangle\,\langle (C\,?\,\mathtt{a}),\, w_4\rangle) & (R_{13}) \\
\quad \Rightarrow \mathtt{a}\,\langle \mathtt{b},\, w_3\rangle\,\langle (C\,?\,\mathtt{a}),\, w_4\rangle) & \\
\quad \Rightarrow \mathtt{a\,b}\,\langle (C\,?\,\mathtt{a}),\, w_4\rangle & \\
\quad \Rightarrow \mathtt{a\,b}\,\langle (C\,?\,!(\mathtt{a}\,\langle C,\, \lambda\rangle)),\, w_4\rangle & \\
\quad \Rightarrow \mathtt{a\,b}\,\langle (C\,?\,!(\langle C,\, \mathtt{c}\rangle)),\, w_4\rangle & \\
\quad \Rightarrow \mathtt{a\,b}\,\langle \mathtt{c},\, w_4\rangle & \\
) &
\end{array}
$$

The pair $\langle \mathtt{c},\, w_4\rangle$ must be rewritten using the implicit rule $\langle \mathtt{c},\, \mathtt{c}\rangle \to \mathtt{c}$ in a new result schema. Our candidate set of result schemas becomes:

$$(\mathtt{a\,b\,c},\, \lambda) \qquad (R_{14})$$

$$
\begin{array}{ll}
(\mathtt{a\,b\,c},\, \lambda, & \\
\quad \langle S,\, \lambda\rangle & \\
\quad \Rightarrow \langle A,\, \mathtt{a}\rangle\,\langle (B\,?\,\mathtt{a}),\, \mathtt{b}\rangle\,\langle (C\,?\,\mathtt{a}),\, \mathtt{c}\rangle & \\
\quad \Rightarrow \mathtt{a}\,\langle A,\, \lambda\rangle\,\langle (B\,?\,\mathtt{a}),\, \mathtt{b}\rangle\,\langle (C\,?\,\mathtt{a}),\, \mathtt{c}\rangle) & \\
\quad \Rightarrow \mathtt{a}\,\langle (B\,?\,\mathtt{a}),\, \mathtt{b}\rangle\,\langle (C\,?\,\mathtt{a}),\, \mathtt{c}\rangle) & \\
\quad \Rightarrow \mathtt{a}\,\langle (B\,?\,!(\mathtt{a}\,\langle B,\, \lambda\rangle)),\, \mathtt{b}\rangle\,\langle (C\,?\,\mathtt{a}),\, \mathtt{c}\rangle) & \\
\quad \Rightarrow \mathtt{a}\,\langle (B\,?\,!(\langle B,\, \mathtt{b}\rangle)),\, \mathtt{b}\rangle\,\langle (C\,?\,\mathtt{a}),\, \mathtt{c}\rangle) & \\
\quad \Rightarrow \mathtt{a}\,\langle \mathtt{b},\, \mathtt{b}\rangle\,\langle (C\,?\,\mathtt{a}),\, \mathtt{c}\rangle) & (R_{14}) \\
\quad \Rightarrow \mathtt{a\,b}\,\langle (C\,?\,\mathtt{a}),\, \mathtt{c}\rangle & \\
\quad \Rightarrow \mathtt{a\,b}\,\langle (C\,?\,!(\mathtt{a}\,\langle C,\, \lambda\rangle)),\, \mathtt{c}\rangle & \\
\quad \Rightarrow \mathtt{a\,b}\,\langle (C\,?\,!(\langle C,\, \mathtt{c}\rangle)),\, \mathtt{c}\rangle & \\
\quad \Rightarrow \mathtt{a\,b}\,\langle \mathtt{c},\, \mathtt{c}\rangle & \\
\quad \Rightarrow \mathtt{a\,b\,c} & \\
) &
\end{array}
$$

We have derived the syntax `abc` and built a formal derivation sequence in doing so. The set of semantic results for this derivation is the set $\{\lambda\}$.

5.7.4 Peano Increment Example

We re-visit the RAG specification defined in §5.6.3, Ex. 33, to illustrate how the tool generates a formal derivation sequence involving a case where we have a query as the semantic result of a rule.

Example 35. Consider the following RAG.

$$\langle S, (I \text{ ? } v_1)\rangle \to \langle N, v_1\rangle \tag{1}$$

$$\langle N, \mathtt{0}\rangle \to \mathtt{0} \tag{2}$$

$$\langle N, \mathtt{s}\, v_1\rangle \to \mathtt{s}\, \langle N, v_1\rangle \tag{3}$$

$$\langle I, \mathtt{s}\, v_1\rangle \to \langle N, v_1\rangle \tag{4}$$

The terminal constants in this RAG are $\mathtt{s}, \mathtt{0}$, the nullary non-terminals are $\{S, N, I\}$, and the start symbol is S. The language defined by the start symbol S is the set of all natural numbers in Peano form, $\{\mathtt{0}, \mathtt{s0}, \mathtt{ss0}, ...\}$.

Example Derivation

Using the RAG specification defined in Ex. 35, we take S as our starting answer and search for a derivation for the string $\mathtt{0}$ and the associated semantic value, and in doing so, we build a formal derivation sequence. The algorithm begins with a single derivation schema in its candidate set, R_1:

$$(\langle S,\, w_1\rangle,\, w_1, \langle S,\, w_1\rangle) \tag{R_1}$$

The pair $\langle S, w_1\rangle$ must first be rewritten using rule 1. We apply this step to the derivation steps in a new derivation schema R_2, substituting w_1 for the result of rule 1. The candidate set becomes:

$$(\langle N,\, w_2\rangle,\, (I\,?\,w_2), \langle S,\, (I\,?\,w_2)\rangle \Rightarrow \langle N,\, w_2\rangle) \tag{R_2}$$

Now the pair $\langle N, w_2\rangle$ must be rewritten using rules 2 and 3. In each new derivation schema, we substitute w_2 for the result of the rule used to derive its configuration. The rewrite step is appended in R_3 and R_4:

$$\begin{aligned} &(\mathtt{0},\, (I\,?\,\mathtt{0}),\\ &\qquad \langle S,\, (I\,?\,\mathtt{0})\rangle\\ &\qquad \Rightarrow \langle N,\, \mathtt{0}\rangle\\ &\qquad \Rightarrow \mathtt{0}\\ &) \end{aligned} \tag{R_3}$$

$$\begin{aligned} &(\mathtt{s}\,\langle N, w_3\rangle,\, (I\,?\,\mathtt{s}\,w_3),\\ &\qquad \langle S,\, (I\,?\,\mathtt{s}\,w_3)\rangle\\ &\qquad \Rightarrow \langle N,\, \mathtt{s}\,w_3\rangle\\ &\qquad \Rightarrow \mathtt{s}\,\langle N,\, w_3\rangle\\ &) \end{aligned} \tag{R_4}$$

We are ready to look at the input prefix of size one. We match each result schema's configuration against $\mathtt{0}$, and find that only the configuration for R_3 can match this symbol. R_4 is discarded, and the candidate set becomes:

$$\begin{aligned}&(\mathtt{0},\ (I\,?\,\mathtt{0}),\\ &\qquad \langle S,\ (I\,?\,\mathtt{0})\rangle\\ &\qquad \Rightarrow \langle N,\ \mathtt{0}\rangle\\ &\qquad \Rightarrow \mathtt{0}\\ &)\end{aligned} \tag{R_3}$$

There is no remaining input to match against for this sub-derivation, and we have a derivation of the syntax $\mathtt{0}$ from start symbol I. However, before the process can return a result set, the query $(I\,?\,\mathtt{0})$ must be rewritten to an answer if possible. The tool begins a new sub-derivation process with starting symbol I and input syntax $\mathtt{0}$:

$$(\langle I,\ u_1\rangle,\ u_1, \langle I,\ u_1\rangle) \tag{SUB_R_1}$$

The pair $\langle I,\ w_1\rangle$ in SUB_R_1 must first be rewritten by use of rule 4 to a new result schema:

$$(\langle N,\ u_2\rangle,\ \mathtt{s}\,u_2, \langle I,\ \mathtt{s}\,u_2\rangle \Rightarrow \langle N,\ u_2\rangle) \tag{SUB_R_2}$$

Now the pair $\langle N,\ u_2\rangle$ must be rewritten using rules 2 and 3, creating result schemas SUB_R_3 and SUB_R_4 respectively. The candidate set becomes:

$$\begin{aligned}&(\mathtt{0},\ \mathtt{s}\,\mathtt{0},\\ &\qquad \langle I, \mathtt{s0}\rangle\\ &\qquad \Rightarrow \langle N, \mathtt{0}\rangle\\ &\qquad \Rightarrow \mathtt{0}\\ &)\end{aligned} \tag{SUB_R_3}$$

$$\begin{aligned}&(\mathtt{s}\,\langle N,\ u_3\rangle,\ \mathtt{s}\,\mathtt{s}\,u_3\\ &\qquad \langle I,\ \mathtt{s}\,\mathtt{s}\,u_3\rangle\\ &\qquad \Rightarrow \langle N,\ \mathtt{s}\,u_3\rangle\\ &\qquad \Rightarrow \mathtt{s}\,\langle N,\ u_3\rangle\\ &)\end{aligned} \tag{SUB_R_4}$$

Now for this sub-derivation we can match each result schema to the input prefix of size one, $\mathtt{0}$. We find that SUB_R_3 can match the prefix $\mathtt{0}$, but the configuration for SUB_R_4 cannot. Result schema SUB_R_4 is discarded from the candidate set, which is now:

$$\begin{aligned}&(\mathtt{0},\ \mathtt{s}\,\mathtt{0},\\ &\qquad \langle I, \mathtt{s0}\rangle\\ &\qquad \Rightarrow \langle N, \mathtt{0}\rangle\\ &\qquad \Rightarrow \mathtt{0}\\ &)\end{aligned} \tag{SUB_R_3}$$

There is no remaining input to match against for this sub-derivation. We have one derivation schema, giving us a result set $\{\mathtt{s}\,\mathtt{0}\}$. We replace the query $(I\,?\,\mathtt{0})$ with $\mathtt{s}\,\mathtt{0}$ in a new derivation schema R_4. We have computed a formal sub-derivation sequence $\langle I, \mathtt{s0}\rangle \Rightarrow \langle N, \mathtt{0}\rangle \Rightarrow \mathtt{0}$.

We take this sequence and embed each step in an inverse operator, which we use as the right-operand to the original query: $(I\,?\,!(\langle I, \mathtt{s0}\rangle)) \Rightarrow (I\,?\,!(\langle N, \mathtt{0}\rangle)) \Rightarrow (I\,?\,\mathtt{0})$. We then prepend the result of the sub-derivation, $\mathtt{s}\,\mathtt{0}$ to this sequence. We get $\mathtt{s}\,\mathtt{0} \Rightarrow (I\,?\,!(\langle I, \mathtt{s0}\rangle)) \Rightarrow (I\,?\,!(\langle N, \mathtt{0}\rangle)) \Rightarrow (I\,?\,\mathtt{0})$. This sequence can now be applied to the derivation steps for R_4. By prepending the derivation sequence for R_3 with new steps in which we replace the query $(I\,?\,\mathtt{0})$ with each step of this new sequence, we get the following:

$$\begin{aligned}
&(\mathtt{0},\ \mathtt{s}\,\mathtt{0},\\
&\quad \langle S,\ \mathtt{s}\,\mathtt{0})\rangle\\
&\quad \Rightarrow \langle S,\ (I\,?\,!(\langle I,\ \mathtt{s}\,\mathtt{0}\rangle))\rangle\\
&\quad \Rightarrow \langle S,\ (I\,?\,!(\langle N,\ \mathtt{0}\rangle))\rangle\\
&\quad \Rightarrow \langle S,\ (I\,?\,\mathtt{0})\rangle\\
&\quad \Rightarrow \langle N,\ \mathtt{0}\rangle\\
&\quad \Rightarrow \mathtt{0}\\
&)
\end{aligned} \tag{R_4}$$

We have found a derivation for the syntax $\mathtt{0}$ from start symbol S, and computed the semantic result set $\{\mathtt{s}\,\mathtt{0}\}$. In doing so, a formal derivation sequence has been constructed.

5.8 The Left-Recursive Case

Recall the example RAG specification defined in §5.4.1, Ex. 30, for the language $\{www \mid w \in [\mathtt{a}-\mathtt{z}]^*\}$. A similar RAG specification was given by Shutt in Example 4.2 of his thesis [1]. In that example, Shutt made use of left-recursion to concatenate a pair which would rewrite to a letter, with a pair which rewrites to a sequence of letters. An identical example is given in the RAG definition in Ex. 36. This specification defined the same language as the RAG specification defines in §5.4.1, Ex. 30. However, left-recursion is now used in place of right-recursion for rule 3.

Example 36. Consider the following RAG.

$$Z ::= \mathtt{a..y}$$

$$\langle S, v_4\rangle \to \langle A, v_1\rangle\, \langle v_1, v_2\rangle\, \langle v_1, v_3\rangle \tag{1}$$

$$\langle A, \lambda\rangle \to \lambda \tag{2}$$

$$\langle A, v_1\, v_2\rangle \to \langle A, v_1\rangle\, \langle C, v_2\rangle \tag{3}$$

$$z : Z \quad \langle C, z\rangle \to \langle z, v_1\rangle \tag{4}$$

The terminal constants in this RAG are [a−z], the nullary non-terminals are $\{S, A, C\}$, and the start symbol is S. The language generated by the start symbol S is $\{www \mid w \in [\mathtt{a}-\mathtt{z}]^*\}$

Example Derivation

With S as our start symbol, using the RAG specification defined in Ex. 36, we try to derive the input syntax aaa. We begin with a single starting result schema in our candidate set:

$$(\langle S,\, w_1\rangle,\, w_1) \tag{R_1}$$

The pair $\langle S,\, w_1\rangle$ must be rewritten using rule 1. R_1 is discarded and we create one new result schema:

$$(\langle A,\, w_2\rangle\,\langle w_2,\, w_3\rangle\,\langle w_2,\, w_4\rangle,\, w_4) \tag{R_2}$$

Now the pair $\langle A,\, w_2\rangle$ must be rewritten. We have rules 2 and 3 for the answer A, and so create two new result schemas in place of R_2:

$$(\langle \lambda,\, w_3\rangle\,\langle \lambda,\, w_4\rangle,\, w_4) \tag{R_3}$$

$$(\langle A,\, w_5\rangle\,\langle C,\, w_6\rangle\,\langle w_5\, w_6,\, w_3\rangle\,\langle w_5\, w_6,\, w_4\rangle,\, w_4) \tag{R_4}$$

Now we must rewrite the pair $\langle \lambda,\, w_3\rangle$ to λ, substituting w_3 for λ. The pair $\langle A,\, w_5\rangle$ in R_4 must also be rewritten using rule 2 and 3.

$$(\langle \lambda,\, w_4\rangle,\, w_4) \tag{R_5}$$

$$(\langle C,\, w_7\rangle\,\langle C,\, w_6\rangle\,\langle w_7\, w_6,\, w_3\rangle\,\langle w_7\, w_6,\, w_4\rangle,\, w_4) \tag{R_6}$$

$$(\langle A,\, w_7\rangle\,\langle C,\, w_8\rangle\,\langle C,\, w_6\rangle\,\langle w_7\, w_8\, w_6,\, w_3\rangle\,\langle w_7\, w_8\, w_6,\, w_4\rangle,\, w_4) \tag{R_7}$$

The pair $\langle C,\, w_7\rangle$ in R_5 must be rewritten to λ, substituting w_4 for λ. $\langle C,\, w_7\rangle$ in R_6 must be rewritten using rule 4, substituting w_7 for a new unknown in R_9. And the pair $\langle A,\, w_7\rangle$ must be rewritten using rules 2 and 3.

$$(\lambda,\, \lambda) \tag{R_8}$$

$$(\langle z,\, w_8\rangle\,\langle C,\, w_6\rangle\,\langle w_8\, w_6,\, w_3\rangle\,\langle w_8\, w_6,\, w_4\rangle,\, w_4) \tag{R_9}$$

$$(\langle C,\, w_8\rangle\,\langle C,\, w_6\rangle\,\langle w_8\, w_6,\, w_3\rangle\,\langle w_8\, w_6,\, w_4\rangle,\, w_4) \tag{R_{10}}$$

$$(\langle A,\, w_9\rangle\,\langle C,\, w_{10}\rangle\,\langle C,\, w_8\rangle\,\langle C,\, w_6\rangle\,\langle w_9\, w_{10}\, w_8\, w_6,\, w_3\rangle\,\langle w_9\, w_{10}\, w_8\, w_6,\, w_4\rangle,\, w_4) \tag{R_7}$$

There will always be some result schema which consists of a configuration whose head pair's left-component is A. Therefore this derivation process will never be able to match against any input symbols. To do so, every configuration's left-most element must be an answer which has not been matched against. The derivation process is in a loop at this point, which it cannot break out of. Therefore, despite being a RAG specification we can use in formal derivations, this example is not supported by the tool due to its left-recursion issue.

Chapter 6

Implementation

6.1 The Initially Implemented Algorithm

The implementation of our tool was first started by Dr. Reuben Rowe, who developed it enough to handle small example specifications given in Shutt's thesis [1] - for example, $\{\mathtt{a}^n\mathtt{b}^n \,|\, n \in Z^*\}$ which we extended to $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ in 4.7.1, and $\{www \,|\, w \in [\mathtt{a}-\mathtt{z}]^*\}$ given in 4.5 of this thesis. Note that in the latter case, the original implementation used explicit rules for each character in the alphabet, instead of a single rule with a typed variable representing all of them. However the tool could not handle the following important cases: queries appearing in the semantic result of a rule (an example of this is given for the Peano arithmetic grammar in §4.7.2), and the use of typed variables (§4.5). This section gives an overview of the original state of the tool before this thesis project.

6.1.1 Sentential Form Schemas

The structures used in previous chapter, *result schemas* and *derivation schemas*, are higher-level representations of what the tool uses internally - a structure we call a *sentential form schema*. Recall that each sentential form will contain a set of unknowns used within its configuration. Instead of substituting unknowns for values as soon as they are discovered, the unknowns are instead maintained with bindings to values through the derivation process. The initial internal sentential form for a derivation using a grammar with starting symbol S will be:

$$([\langle S,\, w_1\rangle], [w_1], \{w_1\}, [\langle S,\, w_1\rangle])$$

The left-most component, $[\langle S,\, w_1\rangle]$, is a configuration consisting of a single element, the pair $\langle S,\, w_1\rangle$. The second component is a polynomial holding one element, the unknown w_1 - the semantic value of the sentential form schema. Thirdly, we have the set $\{w_1\}$, which denotes that this sentential form schema uses one unknown, w_1, which has no binding. Finally, we have an intermediate derivation sequence $[\langle S,\, w_1\rangle]$, consisting of one step. This final element will develop to a formal derivation sequence, if a derivation of our syntax exists from start symbol S.

6.1.2 Input Consumption

In the previous chapter, it was shown that the derivation of result schemas involved matching configurations against input symbols, and maintaining those input symbols in the result schemas if a successful match was made. In fact, the tool removes the symbols it matches at this point from the prefix of the configurations it handles if the match is successful. Sentential form schemas with non-empty configurations remaining in the candidate set after every input symbol has been matched against can be discarded - they can never match the input syntax at this point. Only those which are empty signify a successful derivation.

6.1.3 Internal Data Structures

Each of the following points describes the classes of RAG components used within the original tool, following an object-oriented approach to the implementation of our language parsing strategy.

Answers: The tool's internal form represents an answer as the combination of an identifier and a list of polynomials, the arguments of an answer. To distinguish between terminal and non-terminal answers, each answer also consists of a flag which is set when the answer is a terminal.

Polynomials: Internally, a polynomial is a list of elements belonging to the query algebra, along with the domain of unknowns. Namely, they take the structure $[\alpha_1, \alpha_2, ..., \alpha_n]$ with each α_i being an answer, a query, or a variable. The algebraic domain of a polynomial is determined by that of the polynomial terms it contains, and as such has no need for a flag, as the answers do.

Queries: The query operator is implemented as a class which consists of two polynomials as its internal attributes, the query *syntax* and query *meta-syntax*. Polynomials are used as these attributes because the formal definition of a query says that a query's operands can consist of sub-queries.

Pairs: Pairs are represented internally as a class consisting of a polynomial - the pair's left-component, and an unknown - the pair's right-component. Previous sections have described the need to use an unknown as the right-component of a pair. The left-components of pairs are constrained to be polynomials, to account for the fact that a pair must consist of either a query or an answer as its left-component for it to be rewritable.

Configurations: The configurations used by the tool are lists of configuration terms defined by a RAG specification, not including queries. The internal structure is a list $[\beta_1, \beta_2, ..., \beta_n]$ where each β_i is either a pair or an answer. Configurations are used as the right-component of rules, and the products of derivations over those rules in a sentential form schema. Elements of the internal representation of a configuration cannot be queries, since queries outside of pairs are meaningless and non-rewritable by the tool. Configurations are therefore distinguished from polynomials in the implementation.

Sentential form schemas: Each sentential form schema is a class of structure (C, R, V, D) where C is the configuration for that sentential form schema, R is a polynomial - the semantic value for derivation of C, V is a set of unknown mappings, and D is the derivation sequence for C. The unknowns in a sentential form schema must be bound to polynomials, thus the third component V of a sentential form schema is a mapping of unknowns to polynomials.

Unknowns: Each pair consists of an unknown as its right-component, representing pair variables in a RAG specification. The unknowns are a class containing a unique identifier used for referencing each unknown during the substitution process. Each unknown can also be assigned a type - a set of possible values.

Candidate set: The candidate set is a set of sentential form schemas. Sentential forms are added to and discarded from the candidate set during the process of deriving syntax. When the tool has finished its derivation, the semantic result of each sentential form schema in the candidate set is returned as part of a result set after being resolved of its unknowns and queries.

Rules: Rules are represented internally as a class containing a configuration - the rule's right-hand side, a polynomial - the rules's semantic result, and a set of unknowns representing the variables used within the rule specification. Each rule is associated with an answer by a mapping of a answers to sets of rules in the grammar class.

Grammar: A grammar is a class made up of an answer, being the start symbol for the grammar, and a mapping from answers to rule sets - defining internally the rule schema of a RAG specification. Each derivation process begins by taking a pair whose left-component is the grammar's start symbol, whose right-component is an unknown, and using this in an initial configuraton - a starting point for the derivation of some syntax.

6.1.4 Pseudo-Code

The pseudo-code for the inital parsing strategy is given in terms of procedures `PARSE`, `FORK` and `ADVANCE`. The `PARSE` procedure is the outer-loop of the tool. Given a RAG specification G, a starting answer $\alpha \in A_G$ and some input syntax β, we construct a query $(\alpha\,?\,\beta)$, which we use as the argument to the first call of the `PARSE` procedure. The return value will be a set of semantic results for that query.

The PARSE Procedure

The original `PARSE` procedure takes a query and computes its result given a rule set, by replacing the query with a pair so that its result can be computed by derivation of the query syntax from the query meta-syntax, as per Proposition 4.4 of Shutt's thesis [1]. The procedure maintains a set of candidate sentential form schemas through its execution. We continuously try to rewrite pairs using the `FORK` procedure, and consume input using the `ADVANCE` procedure.

Algorithm 1: Rewrite a query and return a set of semantic results associated with the input syntax.

```
1  PARSE($q$):
       Input  : A query $q$ with meta-syntax $q_m$ and syntax $q_s$
       Output: A set of parse results
2      $cs$ := new candidate set $\{([\langle q_m, w\rangle], w, \{w\}, [\langle q_m, v\rangle])\}$ with an
        initial sentential form schema, where $w$ is a free variable
3      while $q_s$ not empty and $cs$ not empty do
4          Remove candidates from $cs$ with non-empty configurations
5          while some candidate configuration has a pair as its
            left-most element do
6              FORK($cs$)
7              ADVANCE($cs$, $\lambda$, False)
8          end
9          if $q_s$ not empty then
10             $tok$ := remove left-most token from $q_s$
11             ADVANCE($cs$, $tok$, True)
12         end
13     end
14     while $q_s$ not empty and $cs$ not empty do
15         FORK($cs$)
16         ADVANCE($cs$, $\lambda$, False)
17     end
18     Remove sentential form schemas with non-empty configurations
        from $cs$
19     return the set of candidate semantic results
```

The FORK Procedure

The original `FORK` procedure takes a candidate set and identifies the sentential form schemas which have a pair as the left-most component of their configuration. These pairs are then rewritten.

Algorithm 2: Rewriting all pairs existing as the left-most element of the configuration of sentential form schemas.

```
1  FORK($cs$):
       Input : A candidate set of sentential form schemas, $cs$
2      $cs_{temp}$ := clone $cs$
3      forall sentential form schemas $c$ in $cs$ do
4          remove $c$ from $cs$
5          if the configuration for $c$ is empty OR left-most element of
             the configuration of $c$ is not a pair then
6              add $c$ to $cs_{temp}$
7          else if first element of the configuration for $c$ is a pair then
8              $lhc$ := the the left-hand component of the pair which is
                 the first element of the configuration of $c$
9              if $lhc$ is an unknown then
10                 error
11             else if $lhc$ is a query then
12                 $results$ := PARSE($lhc$)
13                 forall $result$ in $results$ do
14                     clone $c$, replace the first pair's left-component
                         with $result$, and add the resulting sentential form
                         to $c_{temp}$
15                 end
16             else if $lhc$ is an answer then
17                 $rules$ := look up the rule set for $lhc$
18                 forall $rule$ in $rules$ do
19                     clone $c$, apply $rule$ by replacing the original pair
                         with the rule's right-hand side, and add the
                         resulting sentential form to $c_{temp}$
20                 end
21     end
22     add all elements of $c_{temp}$ to $cs$
```

The ADVANCE Procedure

The original `ADVANCE` procedure takes a candidate set of sentential form schemas, a token from input, and a Boolean *discardNonMatching*. Each sentential form schema is matched against the input token, and those which do not match are discarded from the candidate set if the Boolean value *discardNonMatching* is `True`. This procedure alters the candidate set without producing a new one, and so has no return statements.

Algorithm 3: Algorithm to consume input and alter the candidate set of sentential forms accordingly by pruning non-matching candidates.

```
1 ADVANCE(cs, token, discardNonMatching):
      Input  : A candidate set of sentential form schemas cs, a token
               from input token, a Boolean value
               discardNonMatching
2     forall candidate sentential form schema c in cs do
3         front := the left-most element of the configuration of c
4         if front is an answer and matches tok then
5             remove tok from the left-most position of the
                configuration for cs
6         else if discardNonMatching then
7             remove c from cs
8     end
```

6.2 Contributions to the Existing Implementation

As mentioned in the previous section, the original tool was built to handle only the examples which Shutt included in his 1993 thesis on recursive adaptable grammars [1]. During this thesis project, research into the RAG formalism resulted in the necessary understanding to write new RAGs which solved problems such as addition of Peano numbers, string equality and the storing of an environment. Consequently, the tool had to be extended to support more features of the RAG formalism so that these new examples could be tested, and derivations could be made using them. On top of the RAG features, implementation of the formal derivation output system allows us to better reason about examples and analyse the languages which they produce. This section will detail the additions to the tool which were implemented as part of this thesis project.

6.2.1 A Front-end for RAG Specifications and Derivations

The previous section noted that in the orignal tool all RAG specifications had to be hardcoded in Java. A primary aim to make the tool easier to use was the implementation of a front-end which would allow one to write out RAG rules as they appear on paper. To do so, we used the attribute grammar sytem ART [18] to translate specifications in the format given below to the hardcoded specifications that one had to write for the original tool. The user is now able to write a RAG specification in a text file with the intuitive format given below and call the parser program with the RAG file and an input string to test. The screenshots in this section show use of some of the RAG examples given in this chapter with the tool's front-end.

Format of Input RAGs

Contrasting the examples seen so far in this thesis, each of the RAG specifications used as input to the tool in this section follow the format rules given below.

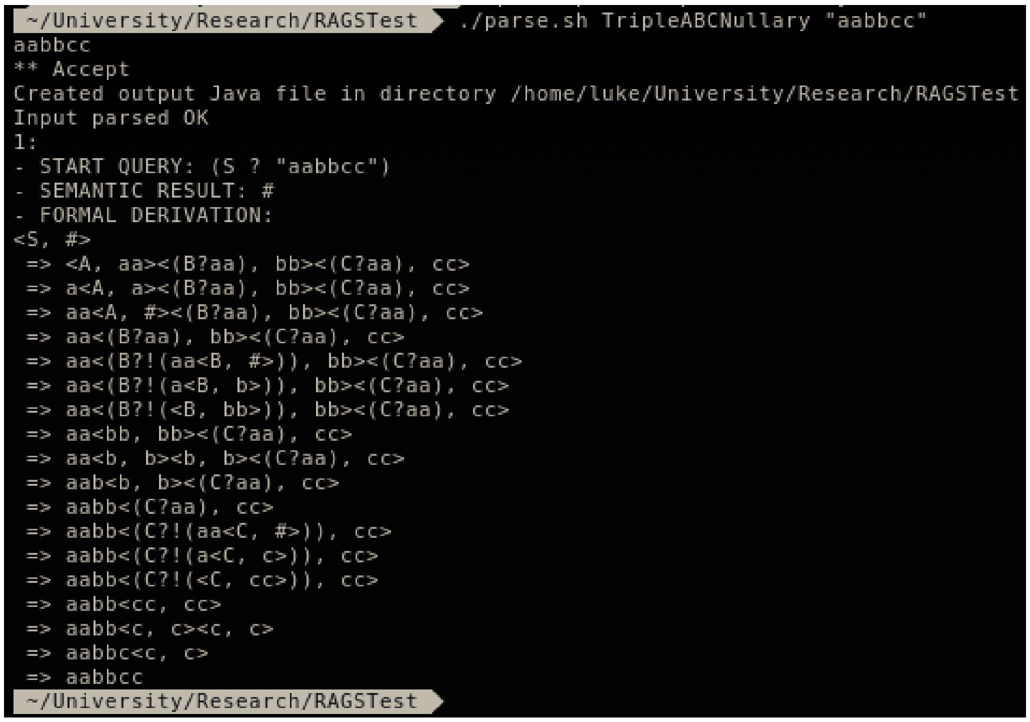
- Each specification is given with a name and a starting answer.
- Terminals are encapsulated in single quotes such as `'a'`, `'b'`, `'c'`.
- All variables begin with `&`. For example, `&v1`, `&v2`, ...
- Wherever λ would be used, instead the symbol `#` is written.
- Non-nullary operators have their operands encapsulated in square brackets, not parentheses. For example, `A[#]`, `C[abc]`.
- Typed variable definitions appear on the left-hand side of a rule and only use the pre-defined types `&WORD` and `&LETTER`.

Triple-ABC Using Queries

The following example is the representation of the RAG defined in Ex. 32 for use with the RAG tool.

```
Name: TripleABCNullary
Start: S

<S, #> -> <A, &v1> <(B?&v1), &v2> <(C?&v1), &v3>

<A, #> -> #
<A, 'a' &v1> -> 'a' <A, &v1>

<B, #> -> #
<B, 'b' &v1> -> 'a' <B, &v1>

<C, #> -> #
<C, 'c' &v1> -> 'a' <C, &v1>
```

The image Fig. 6.2.1 shows a screenshot of this example being used with example input `aabbcc` with the RAG tool. Note that, since every derivation process in the tool is turned into a query, the output is given with a start query on S with the syntax `aabbcc`. The computed semantic result of the string is given, as well as the formal derivation associated with the string.



```
 ~/University/Research/RAGSTest  ./parse.sh TripleABCNullary "aabbcc"
aabbcc
** Accept
Created output Java file in directory /home/luke/University/Research/RAGSTest
Input parsed OK
1:
- START QUERY: (S ? "aabbcc")
- SEMANTIC RESULT: #
- FORMAL DERIVATION:
<S, #>
 => <A, aa><(B?aa), bb><(C?aa), cc>
 => a<A, a><(B?aa), bb><(C?aa), cc>
 => aa<A, #><(B?aa), bb><(C?aa), cc>
 => aa<(B?aa), bb><(C?aa), cc>
 => aa<(B?!(aa<B, #>)), bb><(C?aa), cc>
 => aa<(B?!(a<B, b>)), bb><(C?aa), cc>
 => aa<(B?!(<B, bb>)), bb><(C?aa), cc>
 => aa<bb, bb><(C?aa), cc>
 => aa<b, b><b, b><(C?aa), cc>
 => aab<b, b><(C?aa), cc>
 => aabb<(C?aa), cc>
 => aabb<(C?!(aa<C, #>)), cc>
 => aabb<(C?!(a<C, c>)), cc>
 => aabb<(C?!(<C, cc>)), cc>
 => aabb<cc, cc>
 => aabb<c, c><c, c>
 => aabbc<c, c>
 => aabbcc
 ~/University/Research/RAGSTest
```

Figure 6.2.1: An execution of the RAG tool for the triple-ABC language using queries

The tool correctly finds the semantic value of the string `aabbcc`, being `#`. The query used to start the parsing process is also given, which has the start symbol of the grammar and the input string as its left and right components respectively.

6.2.2 Formal Derivation Output

Included in Fig. 6.2.1 is the formal derivation of the string `aabbcc` using the Triple-ABC grammar. The original tool did not produce such sequences, and only printed the semantic values of the strings it derived. Section 5.7 details how these sequences are obtained during the parsing process. Given this output, one is able to understand how particular strings are derived using a grammar and can work through the steps themselves to verify their own understanding. Despite the steps taken being unrepresentative of those the tool made in its breadth-first process, we believe that their inclusion is an important part of our work, and are confident that they will help the users of the tool understand the formalism. Currently, the output for invalid strings is simply an exception message indicating that the result set is empty. Namely, the set of semantic results used as output is empty due to there being no derivations made.

6.2.3 Work on the Existing Algorithm

We detail in this subsection the contributions made to the orginal parsing algorithm. Each of the major additions is first explained with the context of its usefulness. We then present the updated pseudo-code, representing the changes we made within the tool. Lines in each procedure of our pseudo-code are referred to in the explanations of our major changes.

Nested Queries

None of the example RAG specifications given in Shutt's thesis [1] included nested queries. As an example of a nested query, consider the query $(R\,?\,(I\,?\,v_1) + (D\,?\,v_2))$ given in the Peano arithmetic example within this thesis (see 4.7.2). Here, the inner queries must be resolved before the outer one can be resolved. Due to none of Shutt's examples specifying them, nested queries were not implemented in the original version of the tool, so an important extension was to allow these. The tool must also cope with the fact that a query can return a set of results - the `PARSE` procedure returns the entire resulting candidate set.

In our new `FORK` (see Algorithm 5) procedure, lines 13 to 17 detail our solution to this problem. Recall that in the original implementation, we only called the `PARSE` procedure on the outer query and iterated over its results to construct a new pair containing each query result in a new sentential form schema. To build on this, we now resolve any nested queries recursively using the PARSE procedure. This will give us a list of resolved queries, where each nested query has been replaced by a result in each resolved query. We can then call `PARSE` on all of these resolved queries, and keep track of all of the results of these. Following this, we handle all of the results in the same way as the original `FORK` procedure.

Typed Variables

Although Shutt's thesis included an example of using typed variables matching the specifcation in Fig. 18 of §4.5 in this thesis for the language $\{www \mid w \in Z^*\}$, the original implementation did not support this language feature. Instead, it simply implemented individual rules for each of the characters in the alphabet. In general, the typed variables we have used in our specifications can also be handled in this way. We do, however, believe that supporting typed variables is a convenient feature in our implementation. They allow us to greatly reduce the number of rules. In Fig. 18 of §4.5, we have reduced the number of rules by 25 by using a typed variable since we have just one rule to represent all 26 characters of the alphabet.

In our new `FORK` procedure (Algorithm 5), we handle the case where a pair has a typed variable as its left-component by simply ignoring it for the time being. This is because we do not currently know what the typed variable should be bound to, as we have not consumed enough tokens from the input stream. Wherever such a pair exists as the left-most component of a sentential form schema, we add that sentential form schema to the set of handled ones and move on. Our new `ADVANCE` (Algorithm 6) procedure picks up these pairs when it is time to consume an input token. Essentially, we first type check the typed variable against the input token we have. If they do not match, then we have a sentential form schema which can never match the rest of the input. Otherwise, if the typed variable allows a concatenation of characters in its alphabet we bind it to the first token, and produce a new pair with an identical typed variable at the front of our sentential form schema. A typed variable can also not allow a concatenation of characters, only representing a single one. For this, we simply bind the typed variable to the token. In all cases, we then remove the pair with our typed variable from the front of the sentential form we are looking at.

Queries in the Semantic Result of a Rule

None of the example specification Shutt wrote included queries as the semantic results of rules. We found this to be a useful feature for certain specifications. For example, in Fig. 23 of §4.7.2, the first rule synthesises a query $(R\,?\,v_1 + v_2)$ as its result, which will recursively adds the two Peano numbers bound to v_1 and v_2 together. The result of this query is therefore a single Peano number. Other examples are included in Fig. 26 of §4.7.4 where we use queries as a means for checking the results of evaluating sub-expressions. Note that in rules (2) and (3) of this specificaton, the left-component of the queries are both the terminal `T`. The effect of this is that if the value bound to v_1 in (2) or v_2 in (3) is not `T`, then the rule cannot be applied.

This was the simplest extension to implement. Our change to the existing codebase involves updating the original `PARSE` procedure. The amendments are highlighted in Algorithm 4. On line 19, we now resolve any unknowns in the semantic results of each remaining sentential form schema. Each sentential form schema carries a mapping of unknowns to values, so this step is trivial. We then look for any queries in these semantic results. For each found, we call the `PARSE` procedure on it and replace the query with each of the results in a new sentential form schema. At the end of this process, none of the semantic results returned by our procedure include any queries.

Updated Pseudo-Code

The pseudo-code for the updated parsing strategy is given in the same way as the original, in terms of procedures `PARSE`, `FORK` and `ADVANCE`. Recall that the `PARSE` procedure is the outer-loop of the tool. The return value will still be a set of semantic results for that query.

The PARSE Procedure

The updated `PARSE` procedure is very similar to the original. However an additional step is added on line 19, where we resolve unknowns and queries in the semantic results of candidate schemas. This is where the notion of queries appearing in the semantic result of a derivation is implemented. The original tool without this step would simply return an uncomputed query as part of the result.

Algorithm 4: Rewrite a query and return a set of semantic results associated with the input syntax.

```
1  PARSE($q$):
       Input  : A query $q$ with meta-syntax $q_m$ and syntax $q_s$
       Output: A set of parse results
2      $cs$ := new candidate set $\{([\langle q_m, w\rangle], w, \{w\}, [\langle q_m, v\rangle])\}$ with an
         initial sentential form schema, where $w$ is a free variable
3      while $q_s$ not empty and $cs$ not empty do
4          Remove candidates from $cs$ with non-empty configurations
5          while some candidate configuration has a pair as its
             left-most element do
6              FORK($cs$)
7              ADVANCE($cs$, $\lambda$, False)
8          end
9          if $q_s$ not empty then
10             $tok$ := remove left-most token from $q_s$
11             ADVANCE($cs$, $tok$, True)
12         end
13     end
14     while $q_s$ not empty and $cs$ not empty do
15         FORK($cs$)
16         ADVANCE($cs$, $\lambda$, False)
17     end
18     Remove sentential form schemas with non-empty configurations
         from $cs$
19     Resolve unknowns, and evaluate queries using PARSE that
         appear in the semantic results of all candidate sentential form
         schemas in $cs$
20     return the set of candidate semantic results
```

The FORK Procedure

The updated `FORK` procedure also takes a candidate set and identifies the sentential form schemas which have a pair as the left-most component of their configuration. Unlike the original `FORK` procedure, pairs whose left-component is a typed variable are maintained unaltered in the candidate set - see lines 10 and 11. This is because they are later evaluated in the

`ADVANCE` procedure when we consume a token from the input string. If the left-component is an untyped variable, we simply throw an error as in the original implementation.

We also amend the original algorithm's handling of queries as the left-component of pairs, allowing us to handle the case of nested queries. On lines 13 to 17 of the pseudocode below, we first resolve any inner queries. This will result in a set of new queries, since in theory each inner query may compute multiple semantic values. We then call `PARSE` on each of these new queries and keep track of all of their results. Then, as in the original algorithm, we handle all of the results by replacing *lhc* with the result of each query in a new sentential form schema.

Algorithm 5: Rewriting all pairs existing as the left-most element of the configuration of sentential form schemas.

```
FORK($cs$):
    Input : A candidate set of sentential form schemas, $cs$
    $cs_{temp}$ := clone $cs$
    forall sentential form schemas $c$ in $cs$ do
        remove $c$ from $cs$
        if the configuration for $c$ is empty OR left-most element of
          the configuration of $c$ is not a pair then
            add $c$ to $cs_{temp}$
        else if first element of the configuration for $c$ is a pair then
            $lhc$ := the the left-hand component of the pair which is
              the first element of the configuration of $c$
            if $lhc$ is an unknown then
                add $c$ to $cs_{temp}$ if unknown $lhc$ is typed
                otherwise, error
            else if $lhc$ is a query then
                $resolvedQueries$ := resolve any inner queries in $lhc$
                  using the PARSE procedure
                $results$ := []
                forall $resolvedQuery$ do
                    $results$ := $append(results, \text{PARSE}(resolvedQuery))$
                end
                forall $result$ in $results$ do
                    clone $c$, replace the first pair's left-component
                      with $result$, and add the resulting sentential form
                      to $c_{temp}$
                end
            else if $lhc$ is an answer then
                $rules$ := look up the rule set for $lhc$
                forall $rule$ in $rules$ do
                    clone $c$, apply $rule$ by replacing the original pair
                      with the rule's right-hand side, and add the
                      resulting sentential form to $c_{temp}$
                end
    end
    add all elements of $c_{temp}$ to $cs$
```

The ADVANCE Procedure

Our amended `ADVANCE` procedure takes a candidate set of sentential form schemas, a token from input, and a Boolean *discardNonMatching* in the same way as the original `ADVANCE` procedure.

The additions in the new procedure allow us to implement typed variables, discussed in previous chapters. Due to the workings of the new `FORK` procedure, if there is ever a pair as the left-most element of a configuration, then that pair's left component must be a typed variable. All other types of pairs are rewritten within FORK, however our amendments to that procedure simply maintain pairs with typed variables as their left-components.

On line 7 we extract the typed variable from the pair we are looking at. We then check that its type is compatible with the symbol at the front of the remainder of the input string. If it is not, we know that the candidate sentential form we have can never match the remainder of the input, so it is discarded. Otherwise, we check that the typed variable allows a concatenation of symbols in its alphabet, for example $z \in [\mathtt{a} - \mathtt{z}]^*$ as opposed to $z \in [\mathtt{a} - \mathtt{z}]$. If concatenation is allowed, we simply bind our typed variable to the current input token, and create an identical typed variable in a new pair. We set this new pair as the front of the configuration of the current sentential form schema. If the typed variable does not allow concatenation, we simply bind it to the current input token. We then remove the front pair from our candidate sentential form.

Algorithm 6: Algorithm to consume input and alter the candidate set of sentential forms accordingly by pruning non-matching candidates.

```
1  ADVANCE(cs, token, discardNonMatching):
       Input : A candidate set of sentential form schemas cs, a token
               from input token, a Boolean value
               discardNonMatching
2      forall candidate sentential form schema c in cs do
3          front := the left-most element of the configuration of c
4          if front is an answer and matches tok then
5              remove tok from the left-most position of the
                configuration for cs
6          else if front is a pair then
7              v := the typed unknown as the left-component of front
8              if the type of v does not match tok then
9                  remove c from cs
10             else if v can concatenate symbols in its alphabet then
11                 bind v to tok, add a new pair to the configuration for
                    c after front whose left-component is identical to v
12             else
13                 bind v to tok
14             remove front from the configuration for c
15         else if discardNonMatching then
16             remove c from cs
17     end
```

6.2.4 Repository of the Tool

The current state of the RAG tool can be found at the GitHub repository `https://github.com/reubenrowe/rags/`, for readers who wish to delve into the specific lines of code which implement the algorithnm described in this chapter and the previous.

Chapter 7

Case Study - Structural Operational Semantics

7.1 Introduction

In this chapter we introduce a substantial use-case for RAGs which illustrates their use. The examples will simulate the structural operational semantics for small programs by encoding SOS inference rules into RAG specifications. Each RAG specification in this chapter will model a SOS specification. The RAGs will recognise prefix terms, and compute as their semantic values as the terminal configurations you would receive by reducing identical terms using a set of SOS rules. The case study will consider both big-step and small-step SOS rules, with the goal of determining to what extent there is a consistent strategy for writing a RAG rules which simulate a set of given SOS rules.

7.2 Big-step Semantics

The aim of this section is to attempt to write RAG rules which simulate big-step SOS rules. With such an idiom, we can easily write a RAG rule for a given SOS rule of the big-step style. The first examples will deal with SOS rules for simple values and operations. Following this, an illustration of how one can simulate semantic entities used by SOS rules, such as stores.

7.2.1 Boolean Arithmetic

We begin with simple examples using operations over the Boolean values. Since there are only four permutations of Boolean operations, we can simply encode the truth table for a given operation into the rules themselves. This is first shown for the Boolean *OR* operation in this section.

The reduction rules in the SOS specification defined in Ex. 37 give a big-step semantics for the boolean *OR* operation.

Example 37. Consider the following SOS rules.

$$\frac{\langle S_1\rangle \to \langle V_1\rangle \quad \langle S_2\rangle \to \langle V_2\rangle \quad \langle \texttt{or\_\_}(V_1, V_2)\rangle \to \langle V_3\rangle}{\langle \texttt{or}(S_1, S_2)\rangle \to \langle V_3\rangle} \tag{SOS-1}$$

$$\frac{}{\langle \texttt{or\_\_}(\texttt{True}, \texttt{True})\rangle \to \langle \texttt{True}\rangle} \tag{SOS-2}$$

$$\frac{}{\langle \texttt{or\_\_}(\texttt{True}, \texttt{False})\rangle \to \langle \texttt{True}\rangle} \tag{SOS-3}$$

$$\frac{}{\langle \texttt{or\_\_}(\texttt{False}, \texttt{True})\rangle \to \langle \texttt{True}\rangle} \tag{SOS-4}$$

$$\frac{}{\langle \texttt{or\_\_}(\texttt{False}, \texttt{False})\rangle \to \langle \texttt{False}\rangle} \tag{SOS-5}$$

Ex. 37 specifies six axioms, SOS-2 to SOS-5 - rules which have no conditions above the line. SOS-2 through SOS-5 encode the truth table for the Boolean *OR* operation in reduction rules. The Boolean values `True` and `False` are terminals, meaning the configurations $\langle\texttt{True}\rangle$ and $\langle\texttt{False}\rangle$ cannot be reduced.

The reduction rule labelled SOS-1 computes the result of the *OR* operation on two sub-terms S_1 and S_2. The first and second conditions above the line denote a transition from each sub-term to a boolean value, and in doing so, bindings for two new term variables for each are created, V_1 and V_2. The transition $\langle \texttt{or\_\_}(V_1, V_2)\rangle \to \langle V_3\rangle$ uses the truth table rules SOS-2 - SOS-5 to compute the result of the *OR* operation on the Boolean values V_1, V_2 are bound to. The resulting Boolean value is associated with the term variable V_3, which is used as the final reduction state of the rule.

RAG Representation

Given the understanding of the SOS rules in Ex. 37, we can now re-imagine them in RAG format. In a similar style, we can encode the truth table for the *OR* operation into the rules and use it to evaluate expressions.

The RAG specification in Ex. 38 simulates the SOS rules defined in Ex. 37.

Example 38. Consider the following RAG specification.

$$\langle \mathit{Term},\ (\mathit{Or\_\_}\ ?\ v_1\ v_2)\rangle \to \texttt{or(}\ \langle \mathit{Term},\ v_1\rangle\ \texttt{,}\ \langle\ \mathit{Term}, v_2\rangle\ \texttt{)} \tag{RAG-1}$$

$$\langle \mathit{Term},\ \texttt{True}\rangle \to \texttt{True} \tag{RAG-2}$$

$$\langle \mathit{Term},\ \texttt{False}\rangle \to \texttt{False} \tag{RAG-3}$$

$$\langle \mathit{Or\_\_},\ \texttt{True}\rangle \to \texttt{True}\,\texttt{True} \tag{RAG-4}$$

$$\langle \mathit{Or\_\_},\ \texttt{True}\rangle \to \texttt{True}\,\texttt{False} \tag{RAG-5}$$

$$\langle \mathit{Or\_\_},\ \texttt{True}\rangle \to \texttt{False}\,\texttt{True} \tag{RAG-6}$$

$$\langle \mathit{Or\_\_},\ \texttt{False}\rangle \to \texttt{False}\,\texttt{False} \tag{RAG-7}$$

The start symbol for this RAG specification is *Term*. The nullary operators for this RAG are $\{Term, Or_{--}, \texttt{T}, \texttt{r}, \texttt{u}, \texttt{e}, \texttt{F}, \texttt{a}, \texttt{l}, \texttt{s}, \texttt{o}, \texttt{(}, \texttt{)}, \lambda\}$. The only non-nullary operator is concatenation.

Rule RAG-1 simulates the SOS reduction rule SOS-1, recognising an `or` term consisting of two sub-terms represented by the pairs $\langle Term, v_1\rangle$ and $\langle Term, v_2\rangle$. Each of these pairs recognises either another `or` term, or a Boolean value. The rules for *Term* all synthesise a Boolean value as their semantics, therefore the variables v_1 and v_2 will be bound to some Boolean value when the pairs they appear in are rewritten. The semantics of this rewrite rule is specified as a query on Or_{--} with the variables v_1 and v_2, which rewrites to the result of the *OR* operation on the Boolean values v_1 and v_2 are bound to.

The rules RAG-2 and RAG-3 for the RAG specification defined in Ex. 38 correspond to the SOS reduction rules SOS-6 and SOS-7 in Ex. 37. Our simulation strategy is to represent the big-step SOS relation $\langle\alpha\rangle \to \langle\beta\rangle$ with a RAG rule $\langle Term, \beta\rangle \to \alpha$, thus rules RAG-2 and RAG-3 recognise each Boolean value, and synthesize this value as the semantics of the rewrite rule.

Rules RAG-4, RAG-5, RAG-6 and RAG-7 together encode the truth table for the *OR* operation in RAG rules. These rules all generate two Boolean values, and synthesise the the truth value of their disjunction. This encoding allows a simulation of the transition $\langle \texttt{or}_{--}(V_1, V_2\rangle \to V_3$ used in SOS-1, represented in RAG-1.

RAG Derivation

Given below is a derivation sequence for the syntax `or(False, or(False, True))`, with starting answer *Term*, using the RAG specification defined in Ex. 38:

$$\begin{aligned}
&\langle Term, \texttt{True}\rangle\\
&\Rightarrow \langle Term, !((Or_{--}?!\,(\langle Or_{--}, \texttt{True}\rangle)))\rangle\\
&\Rightarrow \langle Term, !((Or_{--}?\,\texttt{FalseTrue}))\rangle\\
&\Rightarrow \texttt{or(}\langle Term, \texttt{False}\rangle, \langle Term, \texttt{True}\rangle\texttt{)}\\
&\Rightarrow \texttt{or(False,}\,\langle Term, \texttt{True}\rangle\texttt{)}\\
&\Rightarrow \texttt{or(False,}\,\langle Term, !((Or_{--}?!\,(\langle Or_{--}, \texttt{True}\rangle)))\rangle\texttt{)}\\
&\Rightarrow \texttt{or(False,}\,\langle Term, !((Or_{--}?\,\texttt{FalseTrue}))\rangle\texttt{)}\\
&\Rightarrow \texttt{or(False,or(}\langle Term, \texttt{False}\rangle, \langle Term, \texttt{True}\rangle\texttt{))}\\
&\Rightarrow \texttt{or(False,or(False,}\,\langle Term, \texttt{True}\rangle\texttt{))}\\
&\Rightarrow \texttt{or(False,or(False,True))}
\end{aligned}$$

The semantic value associated with the syntax `or(False, or(False, True))` is `True`.

Theorem 1. Consider the SOS rules in Ex. 37 and RAG rules in Ex. 38. Given a SOS reduction $\langle\theta\rangle \to \langle\theta'\rangle$, there is a corresponding RAG derivation sequence: $\langle Term, \theta'\rangle \Rightarrow^* \theta$.

Proof. For the SOS axioms $\langle \texttt{True} \rangle \to \langle \texttt{True} \rangle$, $\langle \texttt{False} \rangle \to \langle \texttt{False} \rangle$ we clearly have the RAG derivations $\langle \mathit{Term}, \texttt{True} \rangle \Rightarrow \texttt{True}$, $\langle \mathit{Term}, \texttt{False} \rangle \Rightarrow \texttt{False}$ by application of axiom 5.23b of Shutt's thesis [1] for rules RAG-2 and RAG-3.

Now consider the big-step SOS reduction step $\langle \texttt{or}(\alpha, \beta) \rangle \to \langle \gamma \rangle$, computing $\alpha \vee \beta = \gamma$ for sub-terms α, β. By application of rules RAG-1 and RAG-4 to RAG-7 with respect to axioms 5.23d and 5.23b of Shutt's thesis [1] we have a derivation sequence $\langle \mathit{Term}, \gamma \rangle \Rightarrow \langle \mathit{Term}, !(\mathit{Or}\_\_ ? !(\langle \mathit{Or}\_\_, \gamma \rangle)) \rangle \Rightarrow \langle \mathit{Term}, !(\mathit{Or}\_\_ ? \, \alpha \, \beta)) \rangle \Rightarrow \texttt{or(}\langle \mathit{Term}, \alpha' \rangle \texttt{,} \langle \mathit{Term}, \beta' \rangle \texttt{)}$, such that $\langle \mathit{Term}, \alpha' \rangle \Rightarrow^* \alpha$ and $\langle \mathit{Term}, \beta' \rangle \Rightarrow^* \beta$. □

The format of our RAG rules suggests a strategy for simulating big-step SOS rules with a RAG specification. The SOS rules are all represented by a corresponding rule for *Term* in the RAG, where the left-hand configuration α in an SOS reduction $\alpha \to \beta$ is the syntax of a RAG rule, and the reduction result β is the semantic value the rule synthesises. Therefore in this case, each rule for *Term* synthesises a Boolean value.

Further Examples

It's simple to extend the example given for the *OR* rules to other Boolean operations using the same pattern for simulating the big-step SOS rules.

$$\frac{\langle S_1 \rangle \to \langle V_1 \rangle \quad \langle S_2 \rangle \to \langle V_2 \rangle \quad \langle \texttt{and\_\_}(V_1, V_2) \rangle \to \langle V_3 \rangle}{\langle \texttt{and}(S_1, S_2) \rangle \to \langle V_3 \rangle} \tag{SOS-8}$$

$$\frac{}{\langle \texttt{and\_\_}(\texttt{True}, \texttt{True}) \rangle \to \langle \texttt{True} \rangle} \tag{SOS-9}$$

$$\frac{}{\langle \texttt{and\_\_}(\texttt{True}, \texttt{False}) \rangle \to \langle \texttt{False} \rangle} \tag{SOS-10}$$

$$\frac{}{\langle \texttt{and\_\_}(\texttt{False}, \texttt{True}) \rangle \to \langle \texttt{False} \rangle} \tag{SOS-11}$$

$$\frac{}{\langle \texttt{and\_\_}(\texttt{False}, \texttt{False}) \rangle \to \langle \texttt{False} \rangle} \tag{SOS-12}$$

We can write RAG rules for simulating the reduction rules SOS-8 - SOS-12 in the as we did for the for the *OR* reduction rules. We use a single RAG rewrite rule to generate the syntax of our `and` terms, complemented with truth table-style rules with which we can compute the *AND* operation. We extend the RAG specification defined in Ex. 38 with additional nullary operators $\{\mathit{And}\_\_, \texttt{n}, \texttt{d}\}$ and the following rules.

$$\langle \mathit{Term}, (\mathit{And}\_\_ ? \ v_1 \, v_2) \rangle \to \texttt{and(} \langle \mathit{Term}, v_1 \rangle \texttt{,} \langle \mathit{Term}, v_2 \rangle \texttt{)} \tag{RAG-8}$$

$$\langle \mathit{And}\_\_, \texttt{True} \rangle \to \texttt{True}\,\texttt{True} \tag{RAG-9}$$

$$\langle \mathit{And}\_\_, \texttt{False} \rangle \to \texttt{True}\,\texttt{False} \tag{RAG-10}$$

$$\langle \mathit{And}\_\_, \texttt{False} \rangle \to \texttt{False}\,\texttt{True} \tag{RAG-11}$$

$$\langle \mathit{And}\_\_, \texttt{False} \rangle \to \texttt{False}\,\texttt{False} \tag{RAG-12}$$

The structure of these rules is the same as those used for `or(.., ..)` terms. The rules RAG-9 - RAG-12 have been constructed to encode the truth table for the *AND* operation. Other binary operations over the Booleans written as big-step SOS rules can be simulated in exactly the same way.

Similarly, we can simulate the big-step SOS rules for the *NOT* operation. Consider the reduction rules SOS-13, SOS-14, SOS-15:

$$\frac{\langle S_1\rangle \to \langle V_1\rangle \quad \langle \texttt{not\_\_}(V_1)\rangle \to \langle V_2\rangle}{\langle \texttt{not}(S_1)\rangle \to \langle V_2\rangle} \tag{SOS-13}$$

$$\frac{}{\langle \texttt{not\_\_(True)}\rangle \to \langle \texttt{False}\rangle} \tag{SOS-14}$$

$$\frac{}{\langle \texttt{not\_\_(False)}\rangle \to \langle \texttt{True}\rangle} \tag{SOS-15}$$

Each `not` term consists of a single sub-term, which by use of a transition we reduce to a value, and V_2 is bound to this value. The reduction state on the right-hand side of this rule is $\langle V_2\rangle$ - namely, the value V_2 is bound to. We can simulate these SOS rules with the rewrite rules defined in the RAG specification below, whose operators are $\{\,Term, Not\_\_, \texttt{n}, \texttt{o}, \texttt{t}, \texttt{T}, \texttt{r}, \texttt{u}, \texttt{e}, \texttt{F}, \texttt{a}, \texttt{l}, \texttt{s}, \texttt{(}, \texttt{)}\,\}$.

$$\langle\, Term, (Not\_\_\ ?\ v_1)\rangle \to \texttt{not(}\,\langle\, Term, v_1\rangle\,\texttt{)} \tag{RAG-13}$$

$$\langle Not\_\_, \texttt{False}\rangle \to \texttt{True} \tag{RAG-14}$$

$$\langle Not\_\_, \texttt{True}\rangle \to \texttt{True} \tag{RAG-15}$$

Rule RAG-13 generates `not` terms which have a single sub-term, recognised by the pair $\langle\, Term,\, v_1\rangle$. The semantic result of RAG-13 is a query which rewrites to the Boolean result of negating the Boolean value bound to v_1, given the rules RAG-14 and RAG-15.

7.2.2 Integer Arithmetic Modulo 3

The pattern used to simulate big-step SOS rules for Boolean operations in a RAG specification can easily be carried over to other finite examples, for example, operations on the integers modulo 3. Consider the SOS specification in Ex. 39:

Example 39. Consider the following SOS rules.

$$\frac{\langle S_1\rangle \to \langle V_1\rangle \quad \langle S_2\rangle \to \langle V_2\rangle \quad \langle \texttt{add\_\_}(V_1, V_2)\rangle \to \langle V_3\rangle}{\langle \texttt{add}(S_1, S_2)\rangle \to \langle V_3\rangle} \tag{SOS-1}$$

$$\frac{}{\langle \texttt{0}\rangle \to \langle \texttt{0}\rangle} \tag{SOS-2}$$

$$\frac{}{\langle \texttt{1}\rangle \to \langle \texttt{1}\rangle} \tag{SOS-3}$$

$$\frac{}{\langle \texttt{2}\rangle \to \langle \texttt{2}\rangle} \tag{SOS-4}$$

$$\frac{}{\langle \texttt{add\_\_(0,0)}\rangle \to \langle \texttt{0}\rangle} \tag{SOS-5}$$

$$\frac{}{\langle \texttt{add\_\_(0,1)} \rangle \to \langle \texttt{1} \rangle} \qquad \text{(SOS-6)}$$

$$\frac{}{\langle \texttt{add\_\_(0,2)} \rangle \to \langle \texttt{2} \rangle} \qquad \text{(SOS-7)}$$

$$\frac{}{\langle \texttt{add\_\_(1,0)} \rangle \to \langle \texttt{1} \rangle} \qquad \text{(SOS-8)}$$

$$\frac{}{\langle \texttt{add\_\_(1,1)} \rangle \to \langle \texttt{2} \rangle} \qquad \text{(SOS-9)}$$

$$\frac{}{\langle \texttt{add\_\_(1,2)} \rangle \to \langle \texttt{0} \rangle} \qquad \text{(SOS-10)}$$

$$\frac{}{\langle \texttt{add\_\_(2,0)} \rangle \to \langle \texttt{2} \rangle} \qquad \text{(SOS-11)}$$

$$\frac{}{\langle \texttt{add\_\_(2,1)} \rangle \to \langle \texttt{0} \rangle} \qquad \text{(SOS-12)}$$

$$\frac{}{\langle \texttt{add\_\_(2,2)} \rangle \to \langle \texttt{1} \rangle} \qquad \text{(SOS-13)}$$

The SOS specification in Ex. 39 defines three axioms, SOS-2, SOS-3 and SOS-4, which say that the integer values `0`, `1` and `2` all reduce to themselves.

It also defines a reduction rule SOS-1, whose format is similar to that of the Boolean operations seen in previous sections. We have an `add` term, consisting of two sub-terms. We compute two values from those sub-terms by calling transitions on them, and then use those values in another transition using the rules for `add__` to compute the result. The result is the integer value which the original term is reduced to by application of this rule.

RAG Representation

To simulate the big-step SOS rules for integer addition given in Ex. 39, consider the RAG specification in Ex. 40 below:

Example 40. Consider the following RAG.

$$\langle \mathit{Term}, (\mathit{Add}\_\_ \ ?\ v_1\ v_2) \rangle \to \texttt{add(}\ \langle \mathit{Term}, v_1 \rangle\ \texttt{,}\ \langle \mathit{Term}, v_2 \rangle\ \texttt{)} \qquad \text{(RAG-1)}$$

$$\langle \mathit{Term}, \texttt{0} \rangle \to \texttt{0} \qquad \text{(RAG-2)}$$

$$\langle \mathit{Term}, \texttt{1} \rangle \to \texttt{1} \qquad \text{(RAG-3)}$$

$$\langle \mathit{Term}, \texttt{2} \rangle \to \texttt{2} \qquad \text{(RAG-4)}$$

$$\langle \mathit{Add}\_\_, \texttt{0} \rangle \to \texttt{0}\,\texttt{0} \qquad \text{(RAG-5)}$$

$$\langle \mathit{Add}\_\_, \texttt{1} \rangle \to \texttt{0}\,\texttt{1} \qquad \text{(RAG-6)}$$

$$\langle \mathit{Add}\_\_, \texttt{2} \rangle \to \texttt{0}\,\texttt{2} \qquad \text{(RAG-7)}$$

$$\langle \mathit{Add}\_\_, \texttt{1} \rangle \to \texttt{1}\,\texttt{0} \qquad \text{(RAG-8)}$$

$$\langle \mathit{Add}\_\_, \texttt{2} \rangle \to \texttt{1}\,\texttt{1} \qquad \text{(RAG-9)}$$

$$\langle \mathit{Add}\_\_, \texttt{0} \rangle \to \texttt{1}\,\texttt{2} \qquad \text{(RAG-10)}$$

$$\langle \mathit{Add}\_\_, \texttt{2} \rangle \to \texttt{2}\,\texttt{0} \qquad \text{(RAG-11)}$$

$$\langle \mathit{Add}\_\_, \texttt{0} \rangle \to \texttt{2}\,\texttt{1} \qquad \text{(RAG-12)}$$

$$\langle \mathit{Add}\_\_, \texttt{1} \rangle \to \texttt{2}\,\texttt{2} \qquad \text{(RAG-13)}$$

The nullary operators are $\{\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{a}, \mathtt{d}, \mathtt{(}, \mathtt{)}, \mathit{Term}, \mathit{Add}\_\_\}$. The start symbol of this RAG is *Term*.

The rule RAG-1 recognises an `add` term consisting of two sub-terms built by the pairs $\langle \mathit{Term}, v_1\rangle$ and $\langle \mathit{Term}, v_2\rangle$. The variables v_1 and v_2 can only be bound to integer values, since each rule for *Term* synthesises either `0`, `1` or `2`. RAG-1 has as its semantic value a query which rewrites to the result of addition on the integers v_1 and v_2 are bound to, by use of the rules RAG-5 to RAG-13.

Rules RAG-2, RAG-3 and RAG-4 simply define the integers `0`, `1` and `2` as valid terms - the rule for each synthesises the generated integer as its semantic value.

Similar to the Boolean truth-table RAG rules, the rewrite rules RAG-5 to RAG-13 generate all possible pairings of the numbers `0`, `1` and `2`, and synthesise the result of addition on the pair of integers a given rule recognises.

Example Derivation

Given below is a derivation sequence for the syntax `add(add(1,1),2)` using the RAG specification defined in Ex. 40:

$$
\begin{aligned}
&\langle \mathit{Term}, \mathtt{1}\rangle\\
&\Rightarrow \langle \mathit{Term}, !((\mathit{Add}\_\_?!(\langle \mathit{Add}\_\_, \mathtt{1}\rangle)))\rangle\\
&\Rightarrow \langle \mathit{Term}, !((\mathit{Add}\_\_?\mathtt{22}))\rangle\\
&\Rightarrow \mathtt{add}(\langle \mathit{Term}, \mathtt{2}\rangle, \langle \mathit{Term}, \mathtt{2}\rangle)\\
&\Rightarrow \mathtt{add}(\langle \mathit{Term}, !((\mathit{Add}\_\_?!(\langle \mathit{Add}\_\_, \mathtt{2}\rangle)))\rangle, \langle \mathit{Term}, \mathtt{2}\rangle)\\
&\Rightarrow \mathtt{add}(\langle \mathit{Term}, !((\mathit{Add}\_\_?\mathtt{11}))\rangle, \langle \mathit{Term}, \mathtt{2}\rangle)\\
&\Rightarrow \mathtt{add}(\mathtt{add}(\langle \mathit{Term}, \mathtt{1}\rangle, \langle \mathit{Term}, \mathtt{1}\rangle), \langle \mathit{Term}, \mathtt{2}\rangle)\\
&\Rightarrow \mathtt{add}(\mathtt{add}(\mathtt{1}, \langle \mathit{Term}, \mathtt{1}\rangle), \langle \mathit{Term}, \mathtt{2}\rangle)\\
&\Rightarrow \mathtt{add}(\mathtt{add}(\mathtt{1}, \mathtt{1}), \langle \mathit{Term}, \mathtt{2}\rangle)\\
&\Rightarrow \mathtt{add}(\mathtt{add}(\mathtt{1}, \mathtt{1}), \mathtt{2})
\end{aligned}
$$

We have computed that the semantic value associated with the syntax `add(add(1,1),2)` is the integer value `1`.

7.2.3 Control Flow and Semantic Entities

To handle examples such as control flow statements, where program state is modified, we implement SOS rules which support the use of semantic entities such as a store. These semantic entities accompany terms in SOS configurations.

Handling State

Some means of encapsulating multiple semantic entities is necessary when simulating SOS transitions between configurations consisting of more than simply a term space. The examples in this sub-section will include SOS transitions over configurations $\langle \theta, \sigma \rangle$ with a term space and a store space, named θ and σ respectively. To represent such configurations in RAG specifications, we introduce an answer $\mathit{State}_{--}[\theta, \sigma]$.

Consider the answer below:

$$\mathit{State}_{--}(\texttt{True}, \mathit{Map}_{--}(\texttt{x}, \texttt{False}, \mathit{Map}_{--}(\texttt{y}, \texttt{True}, \lambda)))$$

This is a representation of a SOS state $\langle \texttt{True}, \{\texttt{x} \mapsto \texttt{False}, \texttt{y} \mapsto \texttt{True}\}\rangle$, illustrated as a RAG answer operator. The store itself is also an answer operator; we have a construction $\mathit{Map}_{--}(\alpha, \beta, \gamma)$ where α is an answer, β is the answer which α is 'bound to' in the store, and γ is either λ or another Map_{--} operator.

Each RAG rewrite rule synthesises an answer of this form - holding the value a term has been reduced to, and the store which may have been altered by reductions of sub-terms.

String Equality

Store access requires the comparison of two identifiers to retrieve a value. The RAG rules first defined in §4.7.5 can be used for this purpose. The rules are listed below:

$$Z ::= \texttt{a..z}$$

$$\langle \mathit{Eq}_{--}(\lambda), \texttt{T}\rangle \to \lambda \tag{1}$$

$$\langle \mathit{Eq}_{--}(\lambda), \texttt{F}\rangle \to \langle \mathit{Letter}, v_1\rangle \, \langle \mathit{Star}(\mathit{Letter}), v_2\rangle \tag{2}$$

$$z : Z,\ t : Z^* \quad \langle \mathit{Eq}_{--}(z\, t), \texttt{F}\rangle \to \lambda \tag{3}$$

$$z : Z,\ t : Z^* \quad \langle \mathit{Eq}_{--}(z\, t), v_1\rangle \to \langle z, v_1\rangle \, \langle \mathit{Eq}_{--}(t), v_1\rangle \tag{4}$$

$$\langle \mathit{Star}(a), \lambda\rangle \to \lambda \tag{5}$$

$$\langle \mathit{Star}(a), v_1\, v_2\rangle \to \langle a, v_1\rangle \, \langle \mathit{Star}(a), v_2\rangle \tag{6}$$

$$z : Z \quad \langle \mathit{Letter}, \lambda\rangle \to \langle z, v_1\rangle \tag{7}$$

Using a query $(\mathit{Eq}_{--}(a)\,?\,b)$ we can test the equality of two nullary answers a, b. From a programming perspective, we can recursively visis Map_{--} operators and test the equality of each identifier with an answer we are looking for, since the Map_{--} operators are formatted like a tree.

Store Access

Now we have the means of keeping a store, and a means to test equality of strings, we look at how to access the store space.

Example 41. Consider the following SOS rules.

$$\frac{}{\langle \texttt{sequence}(\texttt{done}, S_1), \sigma\rangle \rightarrow \langle S_1, \sigma\rangle} \tag{SOS-1}$$

$$\frac{\langle S_1, \sigma_1\rangle \rightarrow \langle V_1, \sigma_2\rangle}{\langle \texttt{sequence}(S_1, S_2), \sigma_1\rangle \rightarrow \langle \texttt{sequence}(V_1, S_2), \sigma_2\rangle} \tag{SOS-2}$$

$$\frac{\texttt{get}_{\_\_}(\sigma_1, \mathit{ID}) \triangleright V_1}{\langle \texttt{get}(\mathit{ID}), \sigma_1\rangle \rightarrow \langle V_1, \sigma_1\rangle} \tag{SOS-3}$$

$$\frac{\texttt{put}_{\_\_}(\sigma_1, \mathit{ID}, V_1) \triangleright \sigma_2}{\langle \texttt{put}(\mathit{ID}, V_1), \sigma_1\rangle \rightarrow \langle \texttt{done}, \sigma_2\rangle} \tag{SOS-4}$$

The first thing to note is that we have rules SOS-1 and SOS-2 which specify a binary operator `sequence`. In effect, this operator combines two consecutive operations so that when one is complete (signified by `done`), the second is evaluated.

SOS-3 recognises a `get` term, whose argument is an identifier. The side operation `get__` is used to match the result of looking for ID in σ_1 to term variable V_1. If the search is successful, V_1 will be bound to the value which ID is bound to in σ_1.

SOS-6 recognises a `put` term, whose arguments are an identifier and a value. A side operation `put__` is used to match the resulting store to a new semantic entity σ_2. The rule reduces to syntax `done` to signify completion of this operation.

Example 42. Consider the following RAG.

$$Z ::= \texttt{a..z}$$

$$\langle \mathit{Term}(\sigma), v_1\rangle \rightarrow \langle \mathit{Sequence}(\sigma), v_1\rangle \tag{RAG-1}$$

$$\langle \mathit{Term}(\sigma), v_1\rangle \rightarrow \langle \mathit{Boolean}(\sigma), v_1\rangle \tag{RAG-2}$$

$$\langle \mathit{Term}(\sigma), v_1\rangle \rightarrow \langle \mathit{Set}(\sigma), v_1\rangle \tag{RAG-3}$$

$$\langle \mathit{Term}(\sigma), v_1\rangle \rightarrow \langle \mathit{Get}(\sigma), v_1\rangle \tag{RAG-4}$$

$$\begin{aligned}&\langle \mathit{Sequence}(\sigma), v_2\rangle \rightarrow \\ &\quad \texttt{sequence(}\, \langle \mathit{Term}(\sigma), v_1\rangle \,\texttt{,}\, \langle \mathit{Term}((v_1 \texttt{ ? sigma})), v_2\rangle \,\texttt{)}\end{aligned} \tag{RAG-5}$$

$$\langle \mathit{Boolean}(\sigma),\ \mathit{State}_{\_\_}(\texttt{True}, \sigma)\rangle \rightarrow \texttt{True} \tag{RAG-6}$$

$$\langle \mathit{Boolean}(\sigma),\ \mathit{State}_{\_\_}(\texttt{False}, \sigma)\rangle \rightarrow \texttt{False} \tag{RAG-7}$$

$$\langle ID,\, v_1\rangle \to \langle Star(Echo),\, v_1\rangle \tag{RAG-8}$$

$$\langle Get(\sigma), State_{--}((Get_{--}(\sigma)\,?\,v_1),\sigma)\rangle \to \texttt{get(}\langle ID,\, v_1\rangle\texttt{)} \tag{RAG-9}$$

$$\begin{aligned}&\langle Put(\sigma),\; State_{--}(\texttt{done}, (Put_{--}(\sigma, (v_2\,?\,\texttt{theta}))\,?\,v_1))\rangle \to\\ &\quad \texttt{put(}\langle ID,\, v_1\rangle\texttt{,}\langle Term(\sigma), v_2\rangle\texttt{)}\end{aligned} \tag{RAG-10}$$

$$\begin{aligned}&\langle Put_{--}(Map_{--}(k,v,m),a),\; Map_{--}(k,a,m)\rangle \to\\ &\quad \langle ID, v_1\rangle\langle(isTrue_{--}\,?\,(Eq_{--}(k)\,?\,v_!)), v_2\rangle\end{aligned} \tag{RAG-11}$$

$$\begin{aligned}&\langle Put_{--}(Map_{--}(k,v,m),a),\; Map_{--}(k,a,(Put_{--}(m,a)\,?\,v_1)))\rangle \to\\ &\quad \langle ID, v_1\rangle\langle(isFalse_{--}\,?\,(Eq_{--}(k)\,?\,v_1)),\; v_2\rangle\end{aligned} \tag{RAG-12}$$

$$\langle Put_{--}(\lambda, a),\; Map_{--}(v_1, a, \lambda)\rangle \to \langle ID,\, v_1\rangle \tag{RAG-13}$$

$$\langle Get_{--}(Map_{--}(k,a,m)), a\rangle \to \langle ID, v_1\rangle\langle(isTrue_{--}\,?\,(Eq_{--}(k)\,?\,v_1)), v_2\rangle \tag{RAG-14}$$

$$\begin{aligned}&\langle Get_{--}(Map_{--}(k,a,m)), (Get_{--}(m)\,?\,v_1)\rangle \to\\ &\quad \langle ID, v_1\rangle\langle(isFalse_{--}\,?\,(Eq_{--}(k)\,?\,v_1)), v_2\rangle\end{aligned} \tag{RAG-15}$$

$$\langle Get_{--}(\lambda),\; \lambda\rangle \to \langle ID, v_1\rangle \tag{RAG-16}$$

$$\langle isTrue_{--}, \lambda\rangle \to \texttt{True} \tag{RAG-17}$$

$$\langle isFalse_{--}, \lambda\rangle \to \texttt{False} \tag{RAG-18}$$

$$\langle Eq_{--}(\lambda), \texttt{True}\rangle \to \lambda \tag{RAG-19}$$

$$\langle Eq_{--}(\lambda), \texttt{False}\rangle \to \langle Letter, v_1\rangle\,\langle Star(Letter), v_2\rangle \tag{RAG-20}$$

$$z : Z,\;\; t : Z^* \quad \langle Eq_{--}(z\,t), \texttt{False}\rangle \to \lambda \tag{RAG-21}$$

$$z : Z,\;\; t : Z^* \quad \langle Eq_{--}(z\,t), v_1\rangle \to \langle z, v_1\rangle\,\langle Eq_{--}(t), v_1\rangle \tag{RAG-22}$$

$$\langle Star(a), \lambda\rangle \to \lambda \tag{RAG-23}$$

$$\langle Star(a), v_1\, v_2\rangle \to \langle a, v_1\rangle\,\langle Star(a), v_2\rangle \tag{RAG-24}$$

$$z : Z \quad \langle Letter, \lambda\rangle \to \langle z, v_1\rangle \tag{RAG-25}$$

$$z : Z \quad \langle Echo, v_1\rangle \to \langle z, v_1\rangle \tag{RAG-26}$$

$$\langle State_{--}(\theta,\sigma), \theta\rangle \to \texttt{theta} \tag{RAG-27}$$

$$\langle State_{--}(\theta,\sigma), \sigma\rangle \to \texttt{sigma} \tag{RAG-28}$$

The nullary operators are $\{\mathtt{a}, ..., \mathtt{z}, \mathtt{(}, \mathtt{)}, \mathit{isTrue}_{--}, \mathit{isFalse}_{--}, \mathit{Letter}, \mathit{Echo}, \mathit{ID}\}$. The unary operators are $\{\mathit{Term}, \mathit{Sequence}, \mathit{Boolean}, \mathit{Get}, \mathit{Put}, \mathit{Eq}_{--}, \mathit{Star}\}$. The binary operators are $\{\mathit{Put}_{--}, \mathit{Get}_{--}, \mathit{State}_{--}\}$ and concatenation. The only ternary operator is Map_{--}. The start symbol of this RAG is *Term*.

Unary operators *Term*, *Sequence*, *Boolean*, *Get* and *Put* all take a σ as their arguments. Namely, the state operator State_{--} whose operands are altered by the RAG rules.

Each State_{--} answer has both a θ and a σ operand. The θ component describes a the representation of a SOS term such as $True$, ranging over the terminals. The σ component corresponds to the σ seen in SOS rules, being the *store space* of a state. This store space must be a Map_{--} operator. So that we can extract either answer from the State_{--} operator, rules RAG-27 and RAG-28 are used. For example, a query $(\mathit{State}_{--}(\theta, \sigma)\,?\,\mathtt{theta})$ would rewrite to θ. The strings `theta` and `sigma` are only used for this purpose.

RAG-5 represents the two *sequence* rules in the SOS specification, SOS-3 and SOS-4. A `sequence` term is recognised, consisting of two sub-terms. v_1 is always bound to a State_{--} answer, as all rules for *Term* synthesise such an operator. A query on this operator in the second pair uses rule RAG-28 to specify the sigma component of the State_{--} answer - being the store. This store is used as the store space of the second *Term* answer by specifying it as the argument. The state answer synthesised by this second pair rewrite is the semantic result of the RAG-5.

RAG-9 represents the SOS rule for `get` terms, SOS-5. A `get(`a`)` term is generated with a a string of letters by rule RAG-8. A query on meta-syntax $\mathit{Get}_{--}(\sigma)$ with syntax v_1, bound to an identifier, is used to rewrite to the value bound to the generated identifier in σ.

RAG-10 represents the SOS rule for `put` terms, SOS-6. The `put` terms are recognised with an identifier and a sub-term. The state synthesised has `done` as its θ component, and the map synthesised by binding the θ element of the sub-term to the identifier v_1 is bound to by use of a query.

The rules for operator Put_{--} take a Map_{--} operator and an answer bound to a as their arguments. Each of these rules generates an identifier, which is the identifier a will be 'bound' to in the new Map_{--} operator.

In rules RAG-11 and RAG-12 the queries on Eq_{--} test the equality of the current Map_{--} operator's left-most answer and the identifier generated by the first pair. The queries on isTrue_{--} and $\mathit{isFalse}_{--}$ will always rewrite to λ, so that only the identifier generated by the first pair is recognised by the rules. Note that RAG-12 includes a query on Put_{--} in its semantic result. This can be thought of as a recursive operation to search for an identifier later in the map, if the current identifier doesn't match.

The rules for Get_{--} take a Map_{--} operator and an answer, bound to a, as their arguments. Each generates an identifier. The rules check for a match between the identifier generated, and the left-most answer in the Map_{--} operator passed as an argument, by use of Eq_{--}. In the case of a match, the a component of Map_{--} is synthesised, being the value the identifier is bound to. Otherwise, by use of a query on Get_{--}, rule RAG-15 effectively searches further in the Map_{--} operator for the identifier generated.

If Statements

Storing state gives us a way to support SOS rules which implement control flow statements. Firstly, we will consider the 'if' statement. We extend the specification in Ex. 41 with the following big-step rules:

$$\frac{\langle S_1, \sigma_1\rangle \rightarrow \langle \texttt{True}, \sigma_2\rangle}{\langle \texttt{if}(S_1, S_2, S_3), \sigma_1\rangle \rightarrow \langle S_2, \sigma_2\rangle} \tag{SOS-7}$$

$$\frac{\langle S_1, \sigma_1\rangle \rightarrow \langle \texttt{False}, \sigma_2\rangle}{\langle \texttt{if}(S_1, S_2, S_3), \sigma_1\rangle \rightarrow \langle S_3, \sigma_2\rangle} \tag{SOS-8}$$

SOS-7 recognises the case where the condition of the statement transitions to `True`, and SOS-8 recognises a transition to `False`. The former reduces to the second sub-term, and the latter reduces to the third sub-term. Since the store may have been altered during the transition above the line in each rule, the new store σ_2 is used in the reduction state.

To correspond to these SOS rules, the RAG rules below extend the RAG specification defined by Ex. 42:

$$\langle \mathit{Term}(\sigma), v_1\rangle \rightarrow \langle \mathit{If}(\sigma), v_1\rangle \tag{RAG-29}$$

$$\begin{aligned}&\langle \mathit{If}(\sigma), v_2\rangle \rightarrow\\ &\quad \texttt{if(}\langle \mathit{Term}(\sigma), v_1\rangle\texttt{,}\langle \mathit{Term}((v_1 \,?\, \texttt{sigma})), v_2\rangle\texttt{,}\\ &\quad \langle \mathit{Term}((v_1 \,?\, \texttt{sigma})), v_3\rangle\texttt{)}\langle(\mathit{isTrue}_{--} \,?\, (v_1 \,?\, \texttt{theta})), v_4\rangle\end{aligned} \tag{RAG-30}$$

$$\begin{aligned}&\langle \mathit{If}(\sigma), v_3\rangle \rightarrow\\ &\quad \texttt{if(}\langle \mathit{Term}(\sigma), v_1\rangle\texttt{,}\langle \mathit{Term}((v_1 \,?\, \texttt{sigma})), v_2\rangle\texttt{,}\\ &\quad \langle \mathit{Term}((v_1 \,?\, \texttt{sigma})), v_3\rangle\texttt{)}\langle(\mathit{isFalse}_{--} \,?\, (v_1 \,?\, \texttt{theta})), v_4\rangle\end{aligned} \tag{RAG-31}$$

Rule RAG-30 corresponds to SOS-7. An `if` term is recognised, with three sub-terms. The first synthesises a $State_{--}$ operator σ. The store component of this is then specified as the argument to the *Term* answer in the second and third pairs by use of a query. The final pair can only rewrite to λ, so doesn't affect the syntax generated by this rule. The semantic result of this rule is the $State_{--}$ answer synthesised by rewriting the second sub-term.

RAG-31 corresponds to SOS-8. This rule handles the case where the first sub-term synthesises `False` as the θ component of its $State_{--}$ answer. The $State_{--}$ operator synthesised by rewriting the third pair is the semantic result of this rule.

To conclude, each rule for *If* synthesises the $State_{--}$ operator of the second sub-term if the first sub-term rewrites to `True`, and the $State_{--}$ operator of the third sub-term otherwise.

7.3 Small-Step Semantics

This section will now show that we can also represent small-step structural operational semantics with RAGs. A small-step semantics involves rules which evaluate a term in terms of single reduction steps on its operands. We will re-visit operations over the Boolean values and integers modulo 3 explored in the previous section, but now in the small-step format.

7.3.1 Boolean Operations

The Boolean operations *OR* and *NOT* are defined in this sub-section as a small step semantics. In each case, a RAG specification is given which simulates the reduction steps taken by the SOS specification in evaluating a term.

Small-Step Or

The style of the small-step rules for *OR* differs from that of the big-step rule. The rules consider all possible sub-terms explicitly and specifies the reduction of an `or` term in terms of its sub-terms.

Example 43. Consider the following SOS rules.

$$\frac{}{\langle \texttt{or(True, True)} \rangle \rightarrow \langle \texttt{True} \rangle} \tag{SOS-1}$$

$$\frac{}{\langle \texttt{or(True, False)} \rangle \rightarrow \langle \texttt{True} \rangle} \tag{SOS-2}$$

$$\frac{}{\langle \texttt{or(False, True)} \rangle \rightarrow \langle \texttt{True} \rangle} \tag{SOS-3}$$

$$\frac{}{\langle \texttt{or(False, False)} \rangle \rightarrow \langle \texttt{False} \rangle} \tag{SOS-4}$$

$$\frac{\langle S_2 \rangle \rightarrow \langle V_2 \rangle}{\langle \texttt{or(True,}S_2\texttt{)} \rangle \rightarrow \langle \texttt{or(True,}V_2\texttt{)} \rangle} \tag{SOS-5}$$

$$\frac{\langle S_2 \rangle \rightarrow \langle V_2 \rangle}{\langle \texttt{or(False,}S_2\texttt{)} \rangle \rightarrow \langle \texttt{or(False,}V_2\texttt{)} \rangle} \tag{SOS-6}$$

$$\frac{\langle S_1 \rangle \rightarrow \langle V_1 \rangle}{\langle \texttt{or(}S_1\texttt{,}S_2\texttt{)} \rangle \rightarrow \langle \texttt{or(}V_1\texttt{,}S_2\texttt{)} \rangle} \tag{SOS-7}$$

Rules SOS-1, SOS-2, SOS-3 and SOS-4 define the truth table for the *OR* operation by explicitly specifying its result for all permutations of its operands.

Rules SOS-5 and SOS-6 handle the case where we have an `or` term whose left-operand is a Boolean value, and whose right-operand is not. In such cases, we simply use a transition operation on the right-operand to evaluate the result of one step on that sub-term.

Rule SOS-7 handles the case where the left-operand is not a Boolean constant. We call a transition on the left-operand, reducing it by one step. The reduction state is the original `or` term with the left-operand being the result of that step.

We can write RAG rules which follow a similar structure. Note that in rule SOS-7, the sub-term S_2 appears unaltered in the reduction state. Therefore in the RAG specification we need rules that generate terms and also synthesise SOS terms as their semantic value, to replicate a sub-term. The answer associated with these rules is called *ReBuild*.

The small-step rules are applied repeatedly until a term reduces to a terminal value. Therefore our RAG specification needs a way to keep reducing until a single Boolean value is found. We call the answer whose rules have this purpose *Repeat*.

Example 44. Consider the following RAG.

$$\langle \mathit{Repeat}, v_1\rangle \rightarrow \langle \mathit{Boolean}, v_1\rangle \tag{RAG-1}$$

$$\langle \mathit{Repeat}, (\mathit{Repeat}\,?\,v_1)\rangle \rightarrow \langle \mathit{Transition}, v_1\rangle \tag{RAG-2}$$

$$\langle \mathit{ReBuild}, v_1\rangle \rightarrow \langle \mathit{Boolean}, v_1\rangle \tag{RAG-3}$$

$$\langle \mathit{ReBuild}, \texttt{or(}v_1\texttt{,}v_2\texttt{)}\rangle \rightarrow \texttt{or(}\langle \mathit{ReBuild}, v_1\rangle\texttt{,}\langle \mathit{ReBuild}, v_2\rangle\texttt{)} \tag{RAG-4}$$

$$\langle \mathit{Transition}, (\mathit{Or}_{\_\_}\,?\,v_1 v_2)\rangle \rightarrow \texttt{or(}\langle \mathit{Boolean}, v_1\rangle\texttt{,}\langle \mathit{Boolean}, v_2\rangle\texttt{)} \tag{RAG-5}$$

$$\langle \mathit{Transition}, \texttt{or(}v_1\texttt{,}v_2\texttt{)}\rangle \rightarrow \texttt{or(}\langle \mathit{Boolean}, v_1\rangle\texttt{,}\langle \mathit{Transition}, v_2\rangle\texttt{)} \tag{RAG-6}$$

$$\langle \mathit{Transition}, \texttt{or(}v_1\texttt{,}v_2\texttt{)}\rangle \rightarrow \texttt{or(}\langle \mathit{Transition}, v_1\rangle\texttt{,}\langle \mathit{ReBuild}, v_2\rangle\texttt{)} \tag{RAG-7}$$

$$\langle \mathit{Boolean}, \texttt{True}\rangle \rightarrow \texttt{True} \tag{RAG-8}$$

$$\langle \mathit{Boolean}, \texttt{False}\rangle \rightarrow \texttt{False} \tag{RAG-9}$$

$$\langle \mathit{Or}_{\_\_}, \texttt{True}\rangle \rightarrow \texttt{True}\,\texttt{True} \tag{RAG-10}$$

$$\langle \mathit{Or}_{\_\_}, \texttt{True}\rangle \rightarrow \texttt{True}\,\texttt{False} \tag{RAG-11}$$

$$\langle \mathit{Or}_{\_\_}, \texttt{True}\rangle \rightarrow \texttt{False}\,\texttt{True} \tag{RAG-12}$$

$$\langle \mathit{Or}_{\_\_}, \texttt{False}\rangle \rightarrow \texttt{False}\,\texttt{False} \tag{RAG-13}$$

The nullary terminal constants are $\{\texttt{T}, \texttt{r}, \texttt{u}, \texttt{e}, \texttt{F}, \texttt{a}, \texttt{l}, \texttt{s}, \texttt{o}, \texttt{(}, \texttt{,}, \texttt{)}\}$. The nullary non-terminal answers are $\{\mathit{Repeat}, \mathit{Transition}, \mathit{ReBuild}, \mathit{Boolean}\}$, and the start symbol is *Repeat*.

The rules for *Repeat* effectively define a recursive transition sequence. The base case for such recursion is when a single Boolean constant is generated. If some other term is generated, then the semantic value associated with it is used in a query on *Repeat* to keep the reduction steps going.

The rules for *Transition* recognise all possible `or` terms. In each rule, the semantic result is the result of applying one reduction step to the syntactic SOS term. Rule RAG-5 recognises an `or` with two sub-terms, both being Boolean values. A query is used to synthesise a Boolean constant. RAG-6 generates `or` terms whose left-operand is a Boolean, and whose right-operand is not a Boolean constant. RAG-7 generates `or` terms whose left-operand is not a Boolean constant.

Given below is a derivation sequence for the syntax `or(False,or(True,False))`, using the RAG specification defined in Ex. 44:

$$
\begin{aligned}
&\langle \mathit{Repeat}, \texttt{True}\rangle\\
&\Rightarrow \langle \mathit{Repeat}, !((\mathit{Repeat}?!(\langle \mathit{Repeat}, \texttt{True}\rangle))))\rangle\\
&\Rightarrow \langle \mathit{Repeat}, !((\mathit{Repeat}?!(\langle \mathit{Repeat}, !((\mathit{Repeat}?!(\langle \mathit{Repeat}, \texttt{True}\rangle))))\rangle))))\rangle\\
&\Rightarrow \langle \mathit{Repeat}, !((\mathit{Repeat}?!(\langle \mathit{Repeat}, !((\mathit{Repeat}?!(\langle \mathit{Boolean}, \texttt{True}\rangle))))\rangle))))\rangle\\
&\Rightarrow \langle \mathit{Repeat}, !((\mathit{Repeat}?!(\langle \mathit{Repeat}, !((\mathit{Repeat}?\texttt{True}))\rangle))))\rangle\\
&\Rightarrow \langle \mathit{Repeat}, !((\mathit{Repeat}?!(\langle \mathit{Transition}, \texttt{True}\rangle))))\rangle\\
&\Rightarrow \langle \mathit{Repeat}, !((\mathit{Repeat}?!(\langle \mathit{Transition}, !((\mathit{Or}_{\_\_}?!(\langle \mathit{Or}_{\_\_}, \texttt{True}\rangle))))\rangle))))\rangle\\
&\Rightarrow \langle \mathit{Repeat}, !((\mathit{Repeat}?!(\langle \mathit{Transition}, !((\mathit{Or}_{\_\_}?\texttt{FalseTrue}))\rangle))))\rangle\\
&\Rightarrow \langle \mathit{Repeat}, !((\mathit{Repeat}?!(\texttt{or(}\langle \mathit{Boolean}, \texttt{False}\rangle\texttt{,}\langle \mathit{Boolean}, \texttt{True}\rangle\texttt{)}))))\rangle\\
&\Rightarrow \langle \mathit{Repeat}, !((\mathit{Repeat}?!(\texttt{or(False,}\langle \mathit{Boolean}, \texttt{True}\rangle\texttt{)}))))\rangle\\
&\Rightarrow \langle \mathit{Repeat}, !((\mathit{Repeat}?\texttt{or(False,True)}))\rangle\\
&\Rightarrow \langle \mathit{Transition}, \texttt{or(False,True)}\rangle\\
&\Rightarrow \texttt{or(}\langle \mathit{Boolean}, \texttt{False}\rangle\texttt{,}\langle \mathit{Transition}, \texttt{True}\rangle\texttt{)}\\
&\Rightarrow \texttt{or(False,}\ \langle \mathit{Transition}, \texttt{True}\rangle\texttt{)}\\
&\Rightarrow \texttt{or(False,}\ \langle \mathit{Transition}, !((\mathit{Or}_{\_\_}?!(\langle \mathit{Or}_{\_\_}, \texttt{True}\rangle))))\rangle\texttt{)}\\
&\Rightarrow \texttt{or(False,}\ \langle \mathit{Transition}, !((\mathit{Or}_{\_\_}?\texttt{TrueFalse}))\rangle\texttt{)}\\
&\Rightarrow \texttt{or(False,or(}\langle \mathit{Boolean}, \texttt{True}\rangle\texttt{,}\langle \mathit{Boolean}, \texttt{False}\rangle\texttt{))}\\
&\Rightarrow \texttt{or(False,or(True,}\langle \mathit{Boolean}, \texttt{False}\rangle\texttt{))}\\
&\Rightarrow \texttt{or(False,or(True,False))}
\end{aligned}
$$

Correctly, the semantic value associated with the input syntax is the Boolean constant `True`.

Small-Step Not

We now consider the small-step rules for the *NOT* operation, since all binary Boolean operations take the pattern given for the *OR* rules.

Example 45. Consider the following SOS rules.

$$\frac{}{\langle \texttt{not(True)}\rangle \rightarrow \langle \texttt{False}\rangle} \qquad \text{(SOS-1)}$$

$$\frac{}{\langle \texttt{not(False)}\rangle \rightarrow \langle \texttt{True}\rangle} \qquad \text{(SOS-2)}$$

$$\frac{\langle S_1 \rangle \rightarrow \langle V_1 \rangle}{\langle \texttt{not(}S_1\texttt{)} \rangle \rightarrow \langle \texttt{not(}V_1\texttt{)} \rangle} \tag{SOS-3}$$

Rules SOS-1 and SOS-2 simply define the truth table for the *NOT* operation by specifying the result of the two possible cases where the operand is a Boolean.

SOS-3 handles the case where the operand is a term which is not a Boolean constant. In this case, we simply transition on the sole operand to perform one step of reduction on it, and then use the reduced sub-term in a new `not` term.

In the same way as for the *OR* operation, we can define the RAG rules for *NOT* as follows.

Example 46. Consider the following RAG.

$$\langle \mathit{Repeat}, v_1 \rangle \rightarrow \langle \mathit{Boolean}, v_1 \rangle \tag{RAG-1}$$

$$\langle \mathit{Repeat}, (\mathit{Repeat}\,?\,v_1) \rangle \rightarrow \langle \mathit{Transition}, v_1 \rangle \tag{RAG-2}$$

$$\langle \mathit{ReBuild}, v_1 \rangle \rightarrow \langle \mathit{Boolean}, v_1 \rangle \tag{RAG-3}$$

$$\langle \mathit{ReBuild}, \texttt{not(}v_1\texttt{)} \rangle \rightarrow \texttt{not(}\langle \mathit{ReBuild}, v_1 \rangle\texttt{)} \tag{RAG-4}$$

$$\langle \mathit{Transition}, (\mathit{Not}_{\_\_}\,?\,v_1 v_2) \rangle \rightarrow \texttt{not(}\langle \mathit{Boolean}, v_1 \rangle\texttt{)} \tag{RAG-5}$$

$$\langle \mathit{Transition}, \texttt{not(}v_1\texttt{)} \rangle \rightarrow \texttt{not(}\langle \mathit{Transition}, v_1 \rangle\texttt{)} \tag{RAG-6}$$

$$\langle \mathit{Boolean}, \texttt{True} \rangle \rightarrow \texttt{True} \tag{RAG-7}$$

$$\langle \mathit{Boolean}, \texttt{False} \rangle \rightarrow \texttt{False} \tag{RAG-8}$$

$$\langle \mathit{Not}_{\_\_}, \texttt{False} \rangle \rightarrow \texttt{True} \tag{RAG-9}$$

$$\langle \mathit{Not}_{\_\_}, \texttt{True} \rangle \rightarrow \texttt{False} \tag{RAG-10}$$

The nullary terminal constants are $\{\texttt{T}, \texttt{r}, \texttt{u}, \texttt{e}, \texttt{F}, \texttt{a}, \texttt{l}, \texttt{s}, \texttt{n}, \texttt{o}, \texttt{t}, \texttt{(}, \texttt{,}, \texttt{)}\}$. The nullary non-terminal answers are $\{\mathit{Repeat}, \mathit{Transition}, \mathit{ReBuild}, \mathit{Boolean}\}$, and the start symbol is *Repeat*.

Rule RAG-5 generates `not` terms whose operand is a Boolean value. In this case, the semantic result specifies the *NOT* operation on that value by use of a query on $\mathit{Not}_{\_\_}$.

RAG-6 generates `not` terms whose operand is not a Boolean value, but another term. The result of applying one step to that sub-term is packaged in a new `not` term, which is synthesised as the rule's result.

Given below is a derivation sequence for the syntax `not(not(True))`, using the RAG specification defined in Ex. 46:

$$\begin{aligned}
&\langle \mathit{Repeat}, \texttt{True}\rangle\\
&\Rightarrow \langle \mathit{Repeat}, !((\mathit{Repeat}?!(\langle \mathit{Repeat}, \texttt{True}\rangle)))\rangle\\
&\Rightarrow \langle \mathit{Repeat}, !((\mathit{Repeat}?!(\langle \mathit{Repeat}, !((\mathit{Repeat}?!(\langle \mathit{Repeat}, \texttt{True}\rangle)))\rangle)))\rangle\\
&\Rightarrow \langle \mathit{Repeat}, !((\mathit{Repeat}?!(\langle \mathit{Repeat}, !((\mathit{Repeat}?!(\langle \mathit{Boolean}, \texttt{True}\rangle)))\rangle)))\rangle\\
&\Rightarrow \langle \mathit{Repeat}, !((\mathit{Repeat}?!(\langle \mathit{Repeat}, !((\mathit{Repeat}?\texttt{True}))\rangle)))\rangle\\
&\Rightarrow \langle \mathit{Repeat}, !((\mathit{Repeat}?!(\langle \mathit{Transition}, \texttt{True}\rangle)))\rangle\\
&\Rightarrow \langle \mathit{Repeat}, !((\mathit{Repeat}?!(\langle \mathit{Transition}, !((\mathit{Not}\_\_?!(\langle \mathit{Not}\_\_, \texttt{True}\rangle)))\rangle)))\rangle\\
&\Rightarrow \langle \mathit{Repeat}, !((\mathit{Repeat}?!(\langle \mathit{Transition}, !((\mathit{Not}\_\_?\texttt{False}))\rangle)))\rangle\\
&\Rightarrow \langle \mathit{Repeat}, !((\mathit{Repeat}?!(\texttt{not(}\langle \mathit{Boolean}, \texttt{False}\rangle\texttt{)})))\rangle\\
&\Rightarrow \langle \mathit{Repeat}, !((\mathit{Repeat}?\texttt{not(False)}))\rangle\\
&\Rightarrow \langle \mathit{Transition}, \texttt{not(False)}\rangle\\
&\Rightarrow \texttt{not(}\langle \mathit{Transition}, \texttt{False}\rangle\texttt{)}\\
&\Rightarrow \texttt{not(}\langle \mathit{Transition}, !((\mathit{Not}\_\_?!(\langle \mathit{Not}\_\_, \texttt{False}\rangle)))\rangle\texttt{)}\\
&\Rightarrow \texttt{not(}\langle \mathit{Transition}, !((\mathit{Not}\_\_?\texttt{True}))\rangle\texttt{)}\\
&\Rightarrow \texttt{not(not(}\langle \mathit{Boolean}, \texttt{True}\rangle\texttt{))}\\
&\Rightarrow \texttt{not(not(True))}
\end{aligned}$$

The semantic value associated with the syntactic term `not(not(True))` is correctly `True`.

7.3.2 Integer arithmetic modulo 3

The small-step semantics for finite integer operations is similar to that of the Boolean operations.

Small-Step Addition

We again consider addition over the naturals modulo 3. We have a set of rules which compute the result of an operation whose operands are two values, a rule for when the left-operand is not a value, and a rule for when the right-operand is not a value.

Example 47. Consider the following SOS rules.

$$\frac{}{\langle \texttt{add(0, 0)}\rangle \to \langle \texttt{0}\rangle} \quad \text{(SOS-1)}$$

$$\frac{}{\langle \texttt{add(0, 1)}\rangle \to \langle \texttt{1}\rangle} \quad \text{(SOS-2)}$$

$$\frac{}{\langle \texttt{add(0, 2)}\rangle \to \langle \texttt{2}\rangle} \quad \text{(SOS-3)}$$

$$\frac{}{\langle \texttt{add(1, 0)}\rangle \to \langle \texttt{1}\rangle} \quad \text{(SOS-4)}$$

$$\frac{}{\langle \texttt{add(1, 1)}\rangle \to \langle \texttt{2}\rangle} \quad \text{(SOS-5)}$$

$$\frac{}{\langle \texttt{add(1, 2)}\rangle \to \langle \texttt{0}\rangle} \quad \text{(SOS-6)}$$

$$\frac{}{\langle \texttt{add(2, 0)} \rangle \rightarrow \langle 2 \rangle} \tag{SOS-7}$$

$$\frac{}{\langle \texttt{add(2, 1)} \rangle \rightarrow \langle 0 \rangle} \tag{SOS-8}$$

$$\frac{}{\langle \texttt{add(2, 2)} \rangle \rightarrow \langle 1 \rangle} \tag{SOS-9}$$

$$\frac{\langle S_2 \rangle \rightarrow \langle V_2 \rangle}{\langle \texttt{add(0,}S_2\texttt{)} \rangle \rightarrow \langle \texttt{add(0,}V_2\texttt{)} \rangle} \tag{SOS-10}$$

$$\frac{\langle S_2 \rangle \rightarrow \langle V_2 \rangle}{\langle \texttt{add(1,}S_2\texttt{)} \rangle \rightarrow \langle \texttt{add(1,}V_2\texttt{)} \rangle} \tag{SOS-11}$$

$$\frac{\langle S_2 \rangle \rightarrow \langle V_2 \rangle}{\langle \texttt{add(2,}S_2\texttt{)} \rangle \rightarrow \langle \texttt{add(2,}V_2\texttt{)} \rangle} \tag{SOS-12}$$

$$\frac{\langle S_1 \rangle \rightarrow \langle V_1 \rangle}{\langle \texttt{add(}S_1\texttt{,}S_2\texttt{)} \rangle \rightarrow \langle \texttt{add(}V_1\texttt{,}S_2\texttt{)} \rangle} \tag{SOS-13}$$

Rules SOS-1 to SOS-9 lay out the result of an add operation for each permutation of the possible operands.

Rules SOS-10, SOS-11 and SOS-12 handle the cases where the left-operand is an integer, but the right-operand is not. We apply one reduction step to the right-operand by way of a transition operation, then re-package the `add` term.

Rule SOS-13 handles the case where the left-operand is not a value. A reduction step is applied to that operand, and the result of the step is used in final reduction `add` term.

As with the SOS rules, the RAG rules for this operation follow a similar pattern to that of the Boolean operations.

Example 48. Consider the following RAG.

$$\langle \mathit{Repeat}, v_1 \rangle \rightarrow \langle \mathit{Integer}, v_1 \rangle \tag{RAG-1}$$

$$\langle \mathit{Repeat}, (\mathit{Repeat}\,?\,v_1) \rangle \rightarrow \langle \mathit{Transition}, v_1 \rangle \tag{RAG-2}$$

$$\langle \mathit{ReBuild}, v_1 \rangle \rightarrow \langle \mathit{Integer}, v_1 \rangle \tag{RAG-3}$$

$$\langle \mathit{ReBuild}, \texttt{add(}v_1\texttt{,}v_2\texttt{)} \rangle \rightarrow \texttt{add(}\langle \mathit{ReBuild}, v_1 \rangle\texttt{,}\mathit{ReBuild}, v_2 \rangle\texttt{)} \tag{RAG-4}$$

$$\langle \mathit{Transition}, (\mathit{Add}_{--}\,?\,v_1 v_2) \rangle \rightarrow \texttt{add(}\langle \mathit{Integer}, v_1 \rangle\texttt{,}\langle \mathit{Integer}, v_2 \rangle\texttt{)} \tag{RAG-5}$$

$$\langle \mathit{Transition}, \texttt{add(}v_1\texttt{,}v_2\texttt{)} \rangle \rightarrow \texttt{add(}\langle \mathit{Integer}, v_1 \rangle\texttt{,}\langle \mathit{Transition}, v_2 \rangle\texttt{)} \tag{RAG-6}$$

$$\langle \mathit{Transition}, \texttt{add(}v_1\texttt{,}v_2\texttt{)} \rangle \rightarrow \texttt{add(}\langle \mathit{Transition}, v_1 \rangle\texttt{,}\langle \mathit{ReBuild}, v_2 \rangle\texttt{)} \tag{RAG-7}$$

$$\langle \mathit{Add}_{--}, \texttt{0} \rangle \rightarrow \texttt{0 0} \tag{RAG-8}$$

$$\langle \mathit{Add}_{--}, \texttt{1} \rangle \rightarrow \texttt{0 1} \tag{RAG-9}$$

$$\langle \mathit{Add}_{\_\_}, \mathtt{2}\rangle \to \mathtt{0}\,\mathtt{2} \qquad \text{(RAG-10)}$$

$$\langle \mathit{Add}_{\_\_}, \mathtt{1}\rangle \to \mathtt{1}\,\mathtt{0} \qquad \text{(RAG-11)}$$

$$\langle \mathit{Add}_{\_\_}, \mathtt{2}\rangle \to \mathtt{1}\,\mathtt{1} \qquad \text{(RAG-12)}$$

$$\langle \mathit{Add}_{\_\_}, \mathtt{0}\rangle \to \mathtt{1}\,\mathtt{2} \qquad \text{(RAG-13)}$$

$$\langle \mathit{Add}_{\_\_}, \mathtt{2}\rangle \to \mathtt{2}\,\mathtt{0} \qquad \text{(RAG-14)}$$

$$\langle \mathit{Add}_{\_\_}, \mathtt{0}\rangle \to \mathtt{2}\,\mathtt{1} \qquad \text{(RAG-15)}$$

$$\langle \mathit{Add}_{\_\_}, \mathtt{1}\rangle \to \mathtt{2}\,\mathtt{2} \qquad \text{(RAG-16)}$$

The nullary terminal constants are $\{\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{a}, \mathtt{d}, \mathtt{(}, \mathtt{,}, \mathtt{)}\}$. The nullary non-terminal answers are $\{\mathit{Repeat}, \mathit{Transition}, \mathit{ReBuild}, \mathit{Integer}\}$, and the start symbol is *Repeat*.

We define all results of addition for the possible permutations of integers with rules RAG-8 to RAG-16.

RAG-5 generates an `add` term whose operands are both integers, and uses a query on $\mathit{Add}_{\_\_}$ to synthesise the result of their addition.

Rule RAG-6 Generates `add` terms whose left-operand is an integer, but whose right-operand is not. The result of applying a step to the right-operand is bound to v_1, which is used to create a new `add` term, the semantic result of the rule.

Rule RAG-7 recognises `add` terms whose left-operand is not an integer. The result of applying a single reduction step to that sub-term is used to synthesise a new `add` term.

Given below is a derivation sequence for the syntax `add(1,2)`, using the RAG specification defined in Ex. 46:

$$\begin{aligned}
&\langle \mathit{Sequence}, \mathtt{0}\rangle \\
&\Rightarrow \langle \mathit{Sequence}, !((\mathit{Sequence}?!(\langle \mathit{Sequence}, \mathtt{0}\rangle))))\rangle \\
&\Rightarrow \langle \mathit{Sequence}, !((\mathit{Sequence}?!(\langle \mathit{Integer}, \mathtt{0}\rangle))))\rangle \\
&\Rightarrow \langle \mathit{Sequence}, !((\mathit{Sequence}?\mathtt{0}))\rangle \\
&\Rightarrow \langle \mathit{Transition}, \mathtt{0}\rangle \\
&\Rightarrow \langle \mathit{Transition}, !((\mathit{Add}_{\_\_}?!(\langle \mathit{Add}_{\_\_}, \mathtt{0}\rangle))))\rangle \\
&\Rightarrow \langle \mathit{Transition}, !((\mathit{Add}_{\_\_}?\mathtt{12}))\rangle \\
&\Rightarrow \mathtt{add(}\langle \mathit{Integer}, \mathtt{1}\rangle\mathtt{,}\langle \mathit{Integer}, \mathtt{2}\rangle\mathtt{)} \\
&\Rightarrow \mathtt{add(1,}\langle \mathit{Integer}, \mathtt{2}\rangle\mathtt{)} \\
&\Rightarrow \mathtt{add(1,2)}
\end{aligned}$$

The semantic value computed for the syntactic term `add(1,2)` is `0`.

Further examples of integer operations, such as for multiplication and division, take an almost identical pattern of rules, as with the same operations in big-step style.

7.4 Findings

This chapter has illustrated that we can use recursive adaptable grammars to simulate complex systems such as operational semantics. The relation between a set of SOS rules and the RAG rules which model them has been mostly one-to-one throughout the examples given. However, RAG rules which handle the details of string matching and semantic entity handling were necessary. Such features are generally implicit in a SOS specification.

We have conjured an idiom for writing RAG rules which model SOS rules. Namely, we write a RAG rule which recognises the left-hand side of a SOS conclusion, and synthesises the right-hand side of the conclusion as its semantic value. Conditions in the premise can also be encoded into a RAG rule. For example, type checking by use of matching in SOS can be written into a RAG using answers which only generate values we want to recognise when used in a pair. Similarly, we have modelled transitions in a SOS premise by using an answer *Transition* which generates a SOS term and synthesises the application of a step to it which we would receive using a SOS rule. This was necessary for small-step semantics.

Chapter 8

Conclusion

The original aim of this research was to look for an effective means for parsing recursive adaptable grammars. By presenting a strategy with which we can derive a syntactic term and its semantics using a RAG, it can be concluded that the formalism can be used to solve practical problems under an implementation. The tool given in this thesis, and the strategy with which it parses RAGs, provides an example of how such an implementation can be built and used to understand the RAG formalism.

A primary contribution of this thesis was to present to the reader a portfolio of example RAG specifications to aid future research into the formalism. The examples given throughout this document are a good illustration of a set of idioms one can employ to build recursive adaptable grammars. A wide range of problems has been shown to be solvable by RAGs in a relatively simple manner, which should ultimately serve as a good case for the formalism. The lack of RAG examples provided by Shutt in his original thesis did not cover all of the features of his formalism. Consequently, the new specifications provided in this dissertation were built from curiosity of how exactly certain features could be used to produce languages in different ways. This thesis should therefore have given a good representation of each RAG feature.

The approach to the research presented in this thesis was to try to answer the question of whether we can build an implementation of the whole RAG formalism as a parser. Trying to solve this practical problem first required a good understanding of how the formalism works. Research was put into how to build RAGs to solve certain problems, resulting in the set of examples given. This required analysis of Shutt's original publications on the topic, by which we could understand the purpose of the more intricate parts of the formalism and how to apply them.

Following the research into Shutt's formalism, work was put into trying to implement it by building a tool to parse RAGs. Before this research project, there existed a tool which supported the basic example specifications given by Shutt. Since Shutt did not provide many example RAGs, some features of the formalisms were left unrepresented. Much of the time spent over this thesis project was in building up the portfolio of RAG examples, and trying to build the tool's capabilities up to support those examples. As a result, our understanding of the formalism increased as we tried to implement new features. We were also able to identify through this development some RAG features which Shutt did not hold in high regard in his thesis, such as typed variables. This research project has highlighted the use of these variables in several examples and has illustrated their usefulness. New light has been shed on a particular part of Shutt's formalism that he did not himself devote attention to.

Future Work

This subsection summarises our proposed future research on the formalism investigated in this thesis, and the implementation work carried out during our research project.

Comparison of RAGs with other Formalisms

Future research into the topic of recursive adaptable grammars should feature a critical comparison of this formalism with other similar formalisms. Such a comparison should consider the relative utility of RAGs against models such as attribute grammars or other adaptable grammar specification systems. This should also consider the ease of specifying languages with each formalism, as well as the practicality of building an efficient implementation that supports all of its features. A great case for RAGs would be made if they were found to be favourable against more commonly used formalisms. We hope to see RAGs being used as commonly as attribute grammar systems to implement general purpose programming languages. The aim should be to demonstrate whether or not one should consider writing a recursive adaptable grammar to specify a formal language, instead of using, for example, an attribute grammar.

A Greater Portfolio of Language Features

We propose that, in addition to the grammars we have presented in this thesis, a greater portfolio of example grammars is written to display the power of RAGs. We have implemented many useful features into our grammars, including control flow, binary expressions over the integers and Booleans, testing for string equality and the maintenance of an environment. Note that our implementation of operations over the integers and Booleans in Chapter 7 did not combine these types into one specification. Future case studies on RAGs could include the modelling of programming language features such as static type checking, type inference, and the simply-typed lambda calculus. As well as this, supporting operator overloading and polymorphism would provide a very interesting use of the RAG formalism. Theoretically RAGs can simulate any computational system, however the degree to which it is practical to model certain systems is a primary point of future research.

An investigation into the implementation of more advanced features in RAGs would help to judge the practicality of the formalism as a technique for specifying useful programming languages. For example, we believe that the specifying of recursive functions using RAGs would be a leap forward in the direction of implementing functional languages. One could also conceive of the implementation of classes and objects in RAGs. To make the case for RAGs as a good formalism for language design and implementation, we hope to push future research towards building specifications for general purpose programming languages.

Work on Implementation

As it stands our implementation works for all features of the formalism, however extensions could be made to make the tool more powerful. Importantly, the deployment of a more effective system with which a user can specify information about types associated with typed variables. On top of this, more development on making the tool easy to use and interpret is necessary. The development of the RAG tool introduced by this thesis was a key contribution of this research project, and a great deal of understanding of the RAG formalism was resultant from this work. However, we have noticed that the tool performs slowly when using the larger specifications given in this thesis. We believe that this is the result of the breadth-first greedy parsing strategy we have employed. A proposal for future work is to build a more efficient RAG-based parser, so that further case studies into more complex language features are not held back by the performance of our tool.

Further work on a RAG implementation could also include more powerful interfaces for working with RAG specifications. For example, one might implement a system where the tool takes a RAG specification, and presents the user with a terminal which allows strings to be entered consecutively such that the outputs are shown for every such input in a terminal-like format. On top of this, future work on the tool should provide more specific feedback when it is given strings which are not in the language of a RAG. This would aid understanding in what the user may have done wrong when writing a RAG. Finally, more complicated types for typed variables should be introduced to increase their usefulness. For the examples given in this thesis, only individual letters in the latin alphabet were required, as well as the Kleene-closure over them.

Bibliography

[1] John N. Shutt *Recursive Adaptable Grammars.* Worcester Polytechnic Institute, 1993.

[2] John N. Shutt *Recursive Adaptable Grammars - Technical Report.* Worcester Polytechnic Institute, 1999.

[3] Donald E. Knuth *Semantics of Context-Free Languages.* California Institute of Technology, 1968.

[4] Kenneth Slonneger, Barry L. Kurtz *Formal Syntax and Semantics of Programming Languages* 1995

[5] Hans Huttel *Transitions and Trees - An Introduction to Structural Operational Semantics* 2010

[6] Benjamin C. Pierce *Types and Programming Languages* 2002

[7] Dick Grune, Ceriel J.H. Jacobs *Parsing Techniques - A Practical Guide* 2008

[8] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman *Compilers: Principles, Techniques & Tools* 2008

[9] Michael Sipser *Introduction to the Theory of Computation* 2013

[10] Alfred V. Aho, Jeffrey D. Ullman *The Theory of Parsing, Translation, and Compiling* 1972

[11] Fernando C. N. Pereira, David H. D. Warren *Definite Clause Grammars for Language Analysis–A Survey of the Formalism and a Comparison with Augmented Transition Networks* 1980

[12] M. Seutter *Informal introduction to the Extended Affix Grammar formalism and its compiler* 1998

[13] G. Geoffroy *Van Wijngaarden grammars, metamorphism and K-ary malwares.* 2018

[14] H. Christiansen *The Syntax and Semantics of Extensible Languages* 1998

[15] N. Chomsky *Three Models for the Description of Language* 1956

[16] *SWI-Prolog Documentation https://www.swi-prolog.org/pldoc/index.html*

[17] *E. Van Wyk, D. Bodin, J. Gao, L. Krishnan* 2010

[18] *Software Language Engineering with ART https://art.csle.cs.rhul.ac.uk/*

[19] K. Meinke, J. V. Tucker *Universal Algebra* 1993

[20] Reuben Rowe, Luke Bessant *Recursive Adaptable Grammars - Tool Implementation Repository:* ‘`https://github.com/reubenrowe/rags/`’ 2021