Decomposing TAG Algorithms Using Simple Algebraizations

Alexander Koller
Dept. of Linguistics
University of Potsdam, Germany
koller@ling.uni-potsdam.de

Marco Kuhlmann
Dept. of Linguistics and Philology
Uppsala University, Sweden
marco.kuhlmann@lingfil.uu.se

Abstract

We review a number of different 'algebraic' perspectives on TAG and STAG in the framework of interpreted regular tree grammars (IRTGs). We then use this framework to derive a new parsing algorithm for TAGs, based on two algebras that describe strings and derived trees. Our algorithm is extremely modular, and can easily be adapted to the synchronous case.

1 Introduction

Much of the early and recent literature on tree-adjoining grammars (TAG) is concerned with working out the formal relationships between TAG and other grammar formalisms. A common approach in this line of research has been to conceive the way in which TAG generates a string or a derived tree from a grammar as a two-step process: first a derivation tree is generated, then this derivation tree is mapped into a term over some algebra and evaluated there. Under this view, one can take different perspectives on how the labour of generating a string or derived tree should be divided between the mapping process and the algebra. In a way that we will make precise, linear context-free rewriting systems (LCFRSs, Weir (1988)) push much of the work into the algebra; Shieber (2006)'s analysis of synchronous TAG as bimorphisms puts the burden mostly on the mapping procedure; and a line of research using context-free tree languages (CFTLs), of which Maletti (2010) is a recent representative, strikes a balance between the other two approaches.

This research has done much to clarify the formal connections between TAG and other formalisms in terms of *generative capacity*. It has not been particularly productive with respect to finding new *algorithms* for parsing, training, and (in the synchronous case) decoding. This is regrettable because standard parsing algorithms for TAG (Vijay-Shanker and Joshi, 1985; Shieber et al., 1995) are complicated, require relatively involved correctness proofs, and are hard to teach. A similar criticism applies to parsing algorithms for LCFRSs (Burden and Ljunglöf, 2005). So far, no new parsing algorithms have arisen from Shieber's work on bimorphisms, or from the CFTL-based view. Indeed, Maletti (2010) leaves the development of such algorithms as an open problem.

This paper makes two contributions. First, we show how a number of the formal perspectives on TAG mentioned above can be recast in a uniform way as *interpreted regular tree grammars* (IRTGs, Koller and Kuhlmann (2011)). IRTGs capture the fundamental idea of generating strings, derived trees, or other objects from a regular tree language, and allow us to make the intuitive differences in how different perspectives divide the labour over the various modules formally precise.

Second, we introduce two new algebras. One captures TAG string languages; the other describes TAG derived tree languages. We show that both of these algebras are *regularly decomposable*, which means that the very modular algorithms that are available for parsing, training, and decoding of IRTGs can be applied to TAG. As an immediate consequence we obtain algorithms for these problems (for both TAG and STAG) that consist of small modules, each of which is simpler to understand, prove correct, and teach than a monolithic parser. As long as the grammar is binary, this comes at no cost in asymptotic parsing complexity.

Proceedings of the 11th International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+11), pages 135–143, Paris, September 2012.

The paper is structured as follows. In Section 2, we introduce some formal foundations and review IRTGs. In Section 3, we recast three existing perspectives on TAG as IRTGs. In Sections 4 and 5, we present algebras for derived trees and strings in TAG; we apply them to parsing in Section 6. Section 7 concludes and discusses future work.

2 Interpreted Regular Tree Grammars

We start by introducing some basic concepts.

2.1 Foundations

A *signature* is a finite set Σ of function symbols f, each of which has been assigned a non-negative integer called its *rank*. We write $f|_n$ to indicate that f has rank n. For the following, let Σ be a signature.

A *tree* over Σ takes the form $t = f(t_1, \ldots, t_n)$, where $f|_n \in \Sigma$ and $t_1, \ldots, t_n$ are trees over Σ. We write T_Σ for the set of all trees over Σ. The *nodes* of a tree can be identified by paths $\pi \in \mathbb{N}^*$ from the root: The root has the address ε, and the ith child of the node with the address π has the address πi. We write $t(\pi)$ for the symbol at path π in the tree t, and $t \downarrow \pi$ for the subtree of t at π.

A *context* over Σ is a tree $C \in T_{\Sigma \cup \{\bullet\}}$ which contains a single leaf labeled with the *hole* $\bullet$. $C[t]$ is the tree in T_Σ which is obtained by replacing the hole in C by some tree $t \in T_\Sigma$. We write C_Σ for the set of all contexts over Σ.

A *Σ-algebra* $\mathcal{A}$ consists of a non-empty set A called the *domain* and, for each function symbol $f|_n \in \Sigma$, a total function $f^\mathcal{A} : A^n \to A$, the *operation* associated with f. Symbols of arity 0 are also called *constants*. The trees in T_Σ are called the *terms* of this algebra. We can *evaluate* a term $t \in T_\Sigma$ to an object $[\![t]\!]_\mathcal{A} \in A$ by executing the operations:

$$[\![f(t_1, \ldots, t_n)]\!]_\mathcal{A} = f^\mathcal{A}([\![t_1]\!]_\mathcal{A}, \ldots, [\![t_n]\!]_\mathcal{A}) .$$

One algebra that we will use throughout the paper is the *string algebra* A^* over some alphabet A. Its elements are the strings over A; it has one binary operation "$\cdot$", which concatenates its two arguments, and one constant for each symbol in A which evaluates to itself. Another important algebra is the *term algebra* T_Σ over some ranked signature Σ. The domain of the term algebra is T_Σ, and for each symbol $f|_n \in \Sigma$, we have $f^{T_\Sigma}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$, i.e. every term evaluates to itself.

If some function $f^\mathcal{A}$ is partial, then $\mathcal{A}$ is a *partial* algebra; in this case there may be terms that do not have a value.

A *(tree) homomorphism* is a total function $h\colon T_\Sigma \to T_\Delta$ that expands symbols of Σ into trees over Δ while following the structure of the input tree. Formally, h is specified by pairs $(f, h(f))$, where $f \in \Sigma$ is a symbol with some rank n, and $h(f) \in T_{\Delta \cup \{x_1, \ldots, x_n\}}$ is a term with variables. The value of a term $t = f(t_1, \ldots, t_n) \in T_\Sigma$ under h is

$$h(f(t_1, \ldots, t_n)) = h(f)[h(t_1)/x_1, \ldots, h(t_n)/x_n] .$$

2.2 Regular Tree Languages

Sets of trees can be specified by *regular tree grammars (RTGs)* (Gécseg and Steinby, 1997; Comon et al., 2008). Formally, an RTG is a construct $\mathcal{G} = (N, \Sigma, P, S)$ where N and Σ are signatures of nonterminal and terminal symbols, $S \in N$ is a start symbol, and P is a finite set of production rules of the form $A \to t$, where $A \in N$ and $t \in T_{N \cup \Sigma}$. Every nonterminal has rank 0. The language generated by $\mathcal{G}$ is the set $L(\mathcal{G}) \subseteq T_\Sigma$ of all trees with only terminal symbols that can be obtained by repeatedly applying the production rules, starting from S.

The class of languages that can be generated by regular tree grammars are called *regular tree languages (RTLs)*. They share many of the closure properties that are familiar from regular string languages. In particular, if L_1 and L_2 are regular and h is a homomorphism, then $L_1 \cap L_2$ and $h^{-1}(L)$ are also regular. If h is linear, then $h(L_1)$ is regular as well.

2.3 Interpreted RTGs

Koller and Kuhlmann (2011) extend RTGs to *interpreted regular tree grammars (IRTGs)*, which specify relations between arbitrary algebras. In this paper, we will focus on relations between strings and (derived) trees, but IRTGs can be used also to describe languages of and relations between other kinds of algebras.

Formally, an IRTG $\mathbb{G}$ is a tuple $(\mathcal{G}, I_1, \ldots, I_k)$ consisting of an RTG $\mathcal{G}$ and $k \geq 1$ *interpretations* $I_1, \ldots, I_k$. Each interpretation I_i is a pair $(h_i, \mathcal{A}_i)$ of an algebra $\mathcal{A}_i$ and a tree homomorphism h_i that maps the trees generated by $\mathcal{G}$ to terms over $\mathcal{A}_i$ (see Fig. 1). The *language* $L(\mathbb{G})$ is then defined as

$$L(\mathbb{G}) = \{([\![h_1(t)]\!]_{\mathcal{A}_1}, \ldots, [\![h_k(t)]\!]_{\mathcal{A}_k}) \mid t \in L(\mathcal{G})\} .$$

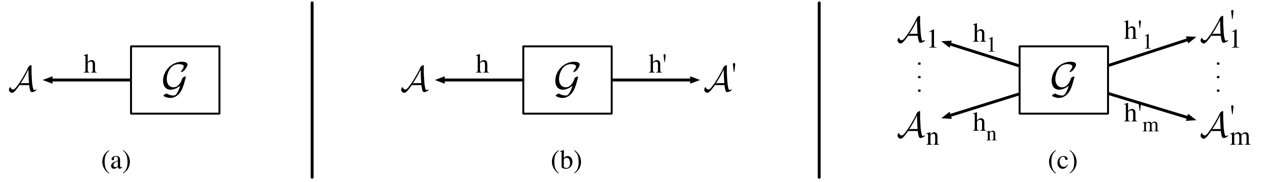

Figure 1: Schematic view of some IRTGs. (a) Monolingual grammar, $k = 1$, $L(\mathbb{G}) \subseteq \mathcal{A}$; (b) synchronous grammar, $k = 2$, $L(\mathbb{G}) \subseteq \mathcal{A} \times \mathcal{A}'$; (c) generalized synchronous grammar with n "input" and m "output" interpretations, $k = m + n$, $L(\mathbb{G}) \subseteq \mathcal{A}_1 \times \ldots \times \mathcal{A}_n \times \mathcal{A}'_1 \times \ldots \mathcal{A}'_m$.

The trees in $L(\mathcal{G})$ correspond to derivation trees in TAG; the elements of $L(\mathbb{G})$ are the objects described by the grammar. For instance, all context-free string languages can be described using $k = 1$ and the string algebra mentioned above (see Fig. 1a). String relations defined by synchronous CFGs or STSGs are exactly those described by IRTGs with $k = 2$ and two such algebras (Fig. 1b).

When parsing IRTGs, we are given input objects on a number of interpretations, and look for those derivation trees $t \in L(\mathcal{G})$ that are consistent with these input objects. Consider the case where we have an input object $a \in \mathcal{A}_1$ for a single interpretation; we are looking for the trees $t \in L(\mathcal{G})$ such that $[\![h_1(t)]\!]_{\mathcal{A}_1} = a$. Many important algebras (including the string and term algebras) are *regularly decomposable*: for each $a \in \mathcal{A}_1$, there is an RTG $D(a)$ – the *decomposition grammar* of a – such that $L(D(a))$ is the set of all terms over $\mathcal{A}_1$ that evaluate to a. Then the set of parses is $L(\mathcal{G}) \cap h_1^{-1}(L(D(a)))$. Using the closure properties of RTLs, this can be computed with a variety of generic algorithms, including bottom-up and top-down algorithms for intersection.

Under the IRTG perspective, the distinction between "monolingual" and synchronous grammars boils down to the choice of $k = 1$ (Fig. 1a) vs. $k > 1$ (Fig. 1b,c). The parsing algorithm generalizes easily to the synchronous case, and supports both the synchronous parsing of multiple input interpretations and the decoding into multiple output interpretations. See Koller and Kuhlmann (2011) for details.

3 Perspectives on TAG

There is an extensive literature on relating TAG to other grammar formalisms. In this section, we provide a uniform view on some of the most important such analyses by recasting them as IRTGs.

Derivation Trees. The fundamental insight that enables us to convert TAGs into IRTGs is that the set of derivation trees of a TAG G forms a regular tree language (Vijay-Shanker et al., 1987). In the formulation of Schmitz and Le Roux (2008), we can obtain an RTG $\mathcal{G}$ describing the derivation trees by using a nonterminal set $\{N_S, N_A \mid N \text{ nonterminal of } G\}$; the start symbol is S_S. $\mathcal{G}$ is defined over a signature whose symbols are the names of the elementary trees in G. The production rules encode the way in which these elementary trees can be combined using substitution (by expanding a nonterminal of the form N_S) and adjunction (by expanding a nonterminal of the form N_A). In this way, the derivation trees of the example grammar from Fig. 2a are described by an RTG $\mathcal{G}_0 = (N_0, \Sigma_0, P_0, S_0)$ with productions

$$\begin{aligned} S_S &\to \alpha_1(NP_S, S_A, VP_A) \\ NP_S &\to \alpha_2(NP_A) \\ VP_A &\to \beta_1(VP_A) \\ S_A, VP_A, NP_A &\to \text{nop} \end{aligned}$$

Notice that every node at which an adjunction may take place is represented by a nonterminal symbol, which must be expanded by a production rule. If no adjunction takes place, we expand it with a rule of the form $N \to \text{nop}$, which is available for every nonterminal N (see Fig. 2b).

LCFRS. The view of TAG as a linear context-free rewriting system (LCFRS; Weir (1988)) can be seen as an IRTG as follows. Consider a Σ_0-algebra $\mathcal{A}_L$ whose values are the derived trees of the TAG grammar G. $\mathcal{A}_L$ interprets each symbol in the derivation tree as a complex tree-building operation which spells out the derived trees. For instance, $\alpha_1^{\mathcal{A}_L}$ is a function (on three arguments, because α_1 has rank 3) which takes an initial tree t_1 and two auxiliary trees t_2 and t_3 as arguments, and returns the tree which results

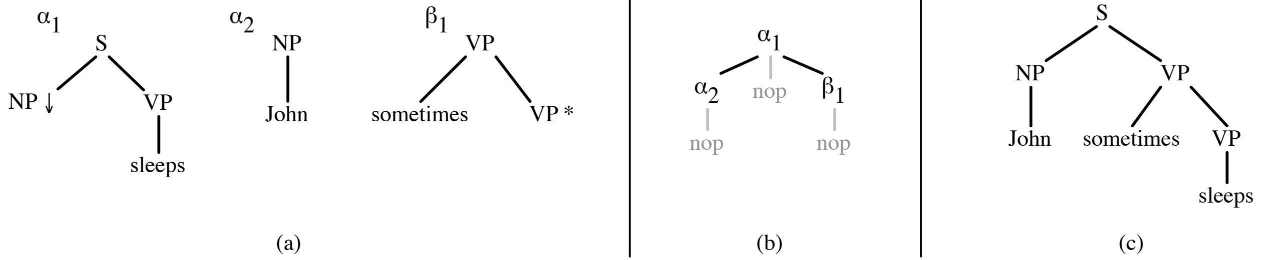

Figure 2: A TAG grammar (a), together with a derivation tree (b) and a derived tree (c). The "nop" nodes in the derivation tree indicate that no adjunction took place where one was possible, and are only needed for technical reasons.

from substituting and adjoining t_1, t_2, and t_3 into α_1 at appropriate places. Using such functions, one can directly interpret the tree $\alpha_1(\alpha_2(\text{nop}), \text{nop}, \beta_1(\text{nop}))$ as the derived tree in Fig. 2c. Therefore, for the IRTG $\mathbb{G}_L = (\mathscr{G}_0, (\text{id}, \mathcal{A}_L))$, where id is the identity homomorphism on T_{Σ_0}, $L(\mathbb{G}_L)$ is exactly the set of derived trees of G. One could instead obtain an IRTG for describing the string language of G by using an interpretation into a Σ_0-algebra of strings and string tuples in which the elementary trees evaluate to appropriate generalized concatenation operations.

STAG as Bimorphisms. Shieber (2004) proposes a different perspective on the generative process of synchronous tree substitution grammar (STSG). He builds upon earlier work on bimorphisms and represents an STSG as an IRTG $(\mathscr{G}_0, (h_1, T_{\Delta_1}), (h_2, T_{\Delta_2}))$, where T_{Δ_1} and T_{Δ_2} are appropriate term algebras. In this construction, the homomorphisms must carry some of the load: In an STSG whose left-hand side contains the trees α_1 and α_2 from Fig. 2a, we would have $h_1(\alpha_1) = S(x_1, VP(\mathsf{sleeps}))$. Shieber (2006) later extended this approach to STAG by replacing the tree homomorphisms with embedded push-down transducers, a more powerful tree rewriting device.

Context-Free Tree Languages. Finally, Mönnich (1998) noticed that the language of derived trees of a TAG grammar is always a (monadic) context-free tree language (CFTL). It has been known since the seminal paper of Engelfriet and Schmidt (1977) that every CFTL can be generated by evaluating the trees of a regular tree language in a specific tree algebra that Engelfriet and Schmidt call the "tree substitution algebra"; we will call it the *YIELD algebra* T^Y_Σ here to avoid confusion, after the name of the evaluation function. Using this insight, we can capture Mönnich's perspective by describing the derived trees of a TAG grammar with an IRTG $(\mathscr{G}_0, (h, T^Y_\Sigma))$, where h is a homomorphism that spells out the elementary trees. Essentially, TAG derived trees are generated by using a TSG of "building instructions" (spelled out by the homomorphisms in a way that is similar to Shieber's), and evaluating the derived trees of the TSG in the YIELD algebra. This idea was made explicit for TAG by Morawietz and Mönnich (2001) and applied to STAG by Maletti (2010).

Discussion. These three different perspectives on TAG as IRTGs are summarized in Fig. 3. The choice of perspective has a significant impact on the algorithms that are natural for it, and the challenges one faces in developing them. The LCFRS view pushes the work of constructing a derived tree almost entirely into the algebra, which is relatively complicated and not binary. This makes it tricky to define a uniform algorithm for computing $D(w)$ for an input string w. When an LCFRS is used to encode an STAG grammar, it is also inconvenient to define a parsing or decoding algorithm that only gets a left string as its input. Shieber's perspective pushes almost all of the work into the translation from derivation trees to derived trees. Parsing involves computing the pre-image of $D(w)$ under embedded push-down transducers, which is harder than for ordinary homomorphisms. The YIELD approach strikes a balance between these two extremes, in that the workload is more evenly balanced between the (ordinary) homomorphism and the algebra. To our knowledge, no parsing or decoding algorithms for strings based on this perspective have been worked out; Maletti (2010) leaves this as an open problem. In the remainder of this paper we will fill this gap.

	homomorphisms	algebras	parsing	synchronous
LCFRSs	identity	complex	+	(+)
bimorphisms	embedded pushdown	term	-	+
CFTLs	tree homomorphisms	YIELD	-	+
this paper	tree homomorphisms	YIELD (simplified)	+	+

Figure 3: Perspectives on TAG.

4 An Algebra for Derived Trees

We will first introduce an algebra for derived trees and show how it can be used to cast TAG as an IRTG. We will then introduce a suitable TAG string algebra in Section 5.

4.1 The Tree Algebra T_Δ^D

The intuition of the derived tree algebra is that adjunction can be modeled using two substitutions of trees into a context. Take an auxiliary tree as a context C with a hole in place of the foot node, and say we want to adjoin it at some node π in the tree t. This can be done by splitting t into a context application $C'[t']$, where C' is the context in t with root ε and hole π. The result of the adjunction is then simply $C'[C[t]]$. In the derivation tree algebra, we use a special symbol @, which inserts its second argument into the hole of its first argument. Thus, in this algebra the term $@(C', @(C, t))$ evaluates to the result of the adjunction. The @ operation is a special case of the composition symbols $c_{n,k}$ used in the YIELD algebra of Engelfriet and Schmidt (1977). A similar intuition underlies the idea of "lifting" in Morawietz and Mönnich (2001), and the work of Maletti (2010).

Let Δ be a ranked signature of node labels. We define the algebra T_Δ^D of all TAG derived trees over Δ as a partial algebra whose domain contains all trees over Δ and all contexts over Δ; that is, the domain is $T_\Delta \cup C_\Delta$. Every $f|_k \in \Delta$ is a k-place operation in T_Δ^D. It takes k arguments $t_1, \dots, t_k \in T_\Delta \cup C_\Delta$. It is defined if either $t_1, \dots, t_k$ are all trees, or at most one of them is a context. In either case, it returns the result $f(t_1, \dots, t_k)$. In addition, T_Δ^D has a constant $*$ that evaluates to the empty context $\bullet$. Finally, the binary operation @ substitutes an element of T_Δ^D into a context. It is defined for arguments C, t where t is a context and t is either a context or a tree. It returns $C[t]$; this is a tree if t is a tree, and a context otherwise.

4.2 An IRTG for TAG Derived Tree Languages

For any given TAG grammar G that uses some alphabet Δ of node labels in its derived trees, we can now construct an IRTG $\mathbb{G} = (\mathcal{G}, (h_t, T_\Delta^D))$ such that $L(\mathbb{G})$ consists of exactly the derived trees that are generated by G. We choose $\mathcal{G}$ to be a Σ-RTG that describes exactly the derivation trees of G, and we need to construct the homomorphism h_t that maps derivation trees into "building instructions" for derived trees, i.e. terms of T_Δ^D.

Each node in the derivation tree is labeled with the name α of an elementary tree (or nop); the subtrees below it describe trees that are combined with this elementary tree using substitution and adjunction. The purpose of $h_t(\alpha)$ is to spell out the way in which α does this combining. Substitution is modeled by simply leaving a variable in $h_t(\alpha)$ in the appropriate place; it will be filled with the initial tree when h_t is evaluated. Adjunction is modeled through the @ operator, as indicated above. Formally, let

$$A \to \alpha(B_S^1, \dots, B_S^k, B_A^{k+1}, \dots, B_A^n)$$

be the (unique) rule in $\mathcal{G}$ that contains the symbol α. Let i be a function that maps each substitution node π in α to the position of the nonterminal occurrence that corresponds to π in the right-hand side of this rule, i.e. to a number between 1 and k. Likewise, let i map each node at which an adjunction may take place to the position of the adjunction nonterminal, i.e. a number between $k+1$ and n. We define a function h_α for each α that maps nodes π of α to terms over T_Δ^D, and let $h_t(\alpha) = h_\alpha(\varepsilon)$. Then $h_\alpha(\pi) = x_{i(\pi)}$ if π is a substitution node; $h_\alpha(\pi) = a$ if π is a lexical leaf with label a; $h_\alpha(\pi) = *$ if π is a foot node, and

$$h_\alpha(\pi) = @(x_{i(\pi)}, f(h_\alpha(\pi 1), \dots, h_\alpha(\pi n))$$

if π is a non-leaf with label f. In this way, we can construct $h_t(\alpha)$ for each elementary tree α.

We illustrate this construction by converting the grammar of Fig. 2a into an IRTG $\mathbb{G} = (\mathcal{G}_0, (h_t, T^D_\Delta))$, where $\Delta = \{S_2, NP_1, VP_2, VP_1, \mathsf{John}_0, \mathsf{sometimes}_0, \mathsf{sleeps}_0\}$. The subscripts are needed to distinguish occurrences of the label same "VP" in Fig. 2c with different ranks. The homomorphism h_t looks as follows:

$$h_t(\alpha_1) = @(x_2, S_2(x_1, @(x_3, VP_1(\mathsf{sleeps}))))$$
$$h_t(\alpha_2) = @(x_1, NP_1(\mathsf{john}))$$
$$h_t(\beta_1) = @(x_1, VP_2(\mathsf{sometimes}, *))$$
$$h_t(\mathrm{nop}) = *$$

Notice how the ability of α_1 to allow adjunction at the S_2 and VP_2 nodes translates into uses of @, which perform the adjunctions by inserting the contexts (= auxiliary trees) that are passed in x_2 and x_3, respectively, in the right places. The NP substitution happens by simply inserting the tree (= initial tree) that is passed in x_1. The term $h_t(\beta_1)$ illustrates how the contexts that model the auxiliary trees are built by combining $*$ (which stands for the empty context containing just one hole) into larger structures. The term $h_t(\mathrm{nop})$ simply evaluates to the empty context; adjoining it anywhere leaves a tree unchanged.

When applied to the derivation tree in Fig. 2b, the homomorphism h_t returns the term $@(*, S_2(@(*, NP_1(\mathsf{john})), @(@(*, VP_2(\mathsf{sometimes}, *)), VP_1(\mathsf{sleeps}))))$. This term evaluates to the derived tree in Fig. 2c.

5 A String Algebra for TAG

Now consider how the basic ideas from Section 4 can be applied to obtain an algebra for TAG string languages. The domain of any such algebra must contain both strings (for the yields of initial trees) and pairs of strings (for the yields of auxiliary trees: one string to the left of the foot node and one string to the right). We have several options in defining the operations on such an algebra. One option is to use the same signature as for the derived tree algebra from Section 4. Unfortunately, this has the effect that the algebra contains operations of rank greater than two, which increases the parsing complexity.

5.1 The TAG String Algebra $\mathcal{A}^T$

We choose to instead build a binary string algebra. The string algebra $\mathcal{A}^T$ for TAG over the finite alphabet A is a partial algebra whose domain contains all strings and all pairs of strings over A; that is, the domain is $A^* \cup (A^* \times A^*)$. We write w for a string and $\overline{w} = (w_1, w_2)$ for a string pair.

Every element a of A is a constant of $\mathcal{A}^T$ with $a^{\mathcal{A}^T} = a$. There is also a constant $*$ with $*^{\mathcal{A}^T} = (\varepsilon, \varepsilon)$. $\mathcal{A}^T$ has a binary partial concatenation operation conc, which is defined if at least one of its two arguments is a string. When defined, it concatenates strings and string pairs as follows:

$$\mathsf{conc}^{\mathcal{A}^T}(w_1, w_2) = w_1 w_2$$
$$\mathsf{conc}^{\mathcal{A}^T}(w_1, \overline{w_2}) = (w_1 w_{21}, w_{22})$$
$$\mathsf{conc}^{\mathcal{A}^T}(\overline{w_1}, w_2) = (w_{11}, w_{12} w_2)$$

Finally, there is a binary partial wrapping operation wrap, which is defined if its first argument is a string pair. This operation wraps its first argument around the second, as follows:

$$\mathsf{wrap}^{\mathcal{A}^T}(\overline{w_1}, w_2) = w_{11} w_2 w_{12}$$
$$\mathsf{wrap}^{\mathcal{A}^T}(\overline{w_1}, \overline{w_2}) = (w_{11} w_{21}, w_{22} w_{12})$$

Notice that these operations closely mirror the operations for well-nested LCFRSs with fan-out 2 that were used in Gómez-Rodríguez et al. (2010).

5.2 An IRTG for TAG String Languages

We can use $\mathcal{A}^T$ to construct, for any given TAG grammar G over some alphabet A of terminal symbols, an IRTG $\mathbb{G} = (\mathcal{G}, (h_s, \mathcal{A}^T))$ such that $L(\mathbb{G})$ consists of exactly the strings that are generated by G. We again describe the derivation trees using a Σ-RTG $\mathcal{G}$. Say that $\mathcal{A}^T$ is a Δ-algebra. It remains to construct a homomorphism h_s from T_Σ to T_Δ.

This is most easily done by defining a second homomorphism h_{st} that maps from T^D_Δ to $\mathcal{A}^T$. h_{st} effectively reads off the yield of a tree or context, and is defined by mapping each operation symbol of T^D_Δ to a term over $\mathcal{A}^T$. In particular, it breaks tree-constructing symbols $f \in \Delta$ in T^D_Δ up into sequences of binary concatenation operations. Thus $h_s = h_{st} \circ h_t$ becomes a homomorphism from Σ into terms of $\mathcal{A}^T$.

$$h_{st}(f) = \mathsf{conc}(x_1, \ldots, \mathsf{conc}(x_{k-1}, x_k)) \quad \text{if } k \geq 2$$
$$h_{st}(f) = x_1 \quad \text{if } f|_1$$
$$h_{st}(f) = f \quad \text{if } f|_0$$
$$h_{st}(@) = \mathsf{wrap}(x_1, x_2)$$
$$h_{st}(*) = *$$

To describe the string language Fig. 2a, we can use the IRTG $(\mathcal{G}_0, (h_s, A^T))$, for the alphabet $A = \{\text{john}, \text{sometimes}, \text{sleeps}\}$. The homomorphism h_s comes out as follows:

$$h_s(\alpha_1) = \mathsf{wrap}(x_2, \mathsf{conc}(x_1, \mathsf{wrap}(x_3, \mathsf{sleeps})))$$
$$h_s(\alpha_2) = \mathsf{wrap}(x_1, \mathsf{john})$$
$$h_s(\beta_1) = \mathsf{wrap}(x_1, \mathsf{conc}(\mathsf{sometimes}, *))$$
$$h_s(\mathrm{nop}) = *$$

Each $h_s(\alpha)$ encodes the operation of the elementary tree α as a generalized concatenation function on strings and string pairs. Applying h_s to the derivation tree in Fig. 2b produces the term $\mathsf{wrap}(*, \mathsf{conc}(\mathsf{wrap}(*, \mathsf{john}), \mathsf{wrap}(\mathsf{wrap}(*, \mathsf{conc}(\mathsf{sometimes}, *)), \mathsf{sleeps})))$. This term evaluates in A^T to "John sometimes sleeps."

5.3 Synchronous Grammars

In summary, for any TAG grammar G, we can obtain an IRTG $(\mathcal{G}, (h_s, A^T))$ for the strings described by G, and an IRTG $(\mathcal{G}, (h_t, T_\Delta^D))$ for the derived trees. Both IRTGs use the same central RTG $\mathcal{G}$. This means that we can combine both views on TAG in a single IRTG with two interpretations, $\mathbb{G} = (\mathcal{G}, (h_s, A^T), (h_t, T_\Delta^D))$.

We can take this idea one step further in order to model synchronous TAG grammars. An STAG grammar can be seen as an RTG generating the derivation trees, plus two separate devices that build the left and right derived tree from a given derivation tree (Shieber, 2004; Shieber, 2006). Each "half" of the STAG grammar is simply an ordinary TAG grammar; they are synchronized with each other by requiring that in each STAG derivation, the individual TAG components must use the same derivation tree. As we have seen, an individual TAG grammar can be represented as an IRTG with two interpretations. We can therefore represent an STAG grammar as an IRTG with four interpretations, $\mathbb{G} = (\mathcal{G}, (h_s^1, A_1^T), (h_t^1, T_{\Delta_1}^D), (h_s^2, A_2^T), (h_t^2, T_{\Delta_2}^D))$. The language of $\mathbb{G}$ consists of four-tuples containing two derived trees and their two associated string yields – one each for the left and right-hand side of the STAG grammar. Notice that unlike in the LCFRS view on STAG, the four individual components are kept separate at all points, and decoding any combination of inputs into any combination of outputs is straightforward.

6 Decomposing the Parsing Algorithm

With these two algebras in place, all that remains to be done to define parsing and decoding algorithms for TAG and STAG is to show that A^T and T_Δ^D are regularly decomposable; then the generic algorithms for IRTG can do the rest.

6.1 Decomposition in the String Algebra

A term t that evaluates to some string or string pair w in the string algebra A^T describes how w can be built recursively from smaller parts using concatenation and wrapping. Just as in a CKY parser, these parts are either spans $[i, k]$ identifying the substring $w_i \dots w_{k-1}$, or span pairs $[i, j, k, l]$ identifying the pair $(w_i \dots w_{j-1}, w_k \dots w_{l-1})$ of substrings.

We can obtain a decomposition grammar $D(w)$ for w by using these spans and span pairs as nonterminals. The production rules of $D(w)$ spell out how larger parts can be built from smaller ones using concatenation and wrapping operations, as follows:

$$[i,k] \to \mathsf{conc}([i,j],[j,k])$$
$$[i,j,k,l] \to \mathsf{conc}([i,j'],[j',j,k,l])$$
$$[i,j,k,l] \to \mathsf{conc}([i,j,k,k'],[k',l])$$
$$[i,l] \to \mathsf{wrap}([i,j,k,l],[j,k])$$
$$[i,j,k,l] \to \mathsf{wrap}([i,i',l',l],[i',j,k,l'])$$
$$[i,i+1] \to w_i$$
$$[i,i,j,j] \to *$$

The start symbol of $D(w)$ is the span that corresponds to the entire string or string pair w. If w is a string of length n, it is $[1, n+1]$; for a string pair $\overline{w} = (w_1 \dots w_{m-1}, w_m \dots w_n) \in A^T$, the start symbol is $[1, m, m, n+1]$. Notice that the size of $D(w)$ is $O(n^6)$ because of the second wrapping rule, and the grammar can also be computed in time $O(n^6)$.

6.2 Decomposition in the Derived Tree Algebra

The parts from which a term over the derived tree algebra T_Δ^D builds some derived tree or context $\tau \in T_\Delta^D$ are the subtrees or the contexts within τ. Each subtree can be identified by its root node π in τ; each context can be identified by its root π and its hole π'.

Thus we can obtain a decomposition grammar $D(\tau)$ using a nonterminal A_π to indicate the subtree starting at π and a nonterminal $B_{\pi/\pi'}$ to indicate the context from π to π'. As above, the rules spell out the ways in which larger subtrees and contexts

can be constructed from smaller parts:

$$A_\pi \to f(A_{\pi 1}, \ldots, A_{\pi n}) \qquad t(\pi) = f|_n$$
$$A_\pi \to @(B_{\pi/\pi'}, A_{\pi'}) \qquad \pi' \text{ node in } t \downarrow \pi$$
$$B_{\pi/\pi'} \to f(A_{\pi 1}, \ldots, B_{\pi i/\pi'}, \ldots, A_{\pi n}) \quad \pi i \leq \pi', t(\pi) = f|_n$$
$$B_{\pi/\pi'} \to @(B_{\pi/\pi''}, B_{\pi''/\pi'}) \qquad \pi < \pi'' \leq \pi'$$
$$B_{\pi/\pi} \to *$$

Again, the start symbol is simply the representations of τ itself. If τ is a tree, it is A_ε; for a context with hole π, the start symbol is $B_{\varepsilon/\pi}$. If τ has n nodes, the grammar $D(\tau)$ has $O(n^3)$ rules because of the second rule for @.

6.3 Decomposing the TAG Parsing Algorithm

To illustrate the use of these decomposition grammars in the context of the parsing algorithm of Section 2, we parse the string $w =$ "John sometimes sleeps" using the IRTG $\mathbb{G} = (\mathcal{G}_0, (h_s, A^T))$ for the string perspective on the example grammar from Fig. 2 (cf. Section 5).

Step 1: Decomposition Grammar. First, the parser computes the decomposition grammar $D(w)$. As explained above, this grammar has the start symbol $[1,4]$ and rules given in Fig. 4 (among others). The complete grammar generates a set of 72 terms over A^T, each of which evaluates to w. The term that is important here is $\mathsf{wrap}(*, \mathsf{conc}(\mathsf{wrap}(*, \mathsf{john}), \mathsf{wrap}(\mathsf{wrap}(*, \mathsf{conc}(\mathsf{sometimes}, *)), \mathsf{sleeps})))$. Crucially, the TAG grammar is completely irrelevant at this point: $L(D(w))$ consists of *all* terms over A^T which evaluate to w, regardless of whether they correspond to grammatical derivation trees or not.

Step 2: Inverse Homomorphism. Next, we compute an RTG $\mathcal{G}'$ for $h_s^{-1}(L(D(w)))$. This grammar describes all trees over Σ that are mapped by h_s into terms that evaluate to w. Its nonterminal symbols are still spans and span pairs, but the terminal symbols are now names of elementary trees. The grammar has the start symbol $[1,4]$ and the following rules:

$$[1,4] \to \alpha_1([1,2],[1,1,4,4],[2,3,4,4])$$
$$[1,2] \to \alpha_2([1,1,2,2])$$
$$[2,3,4,4] \to \beta_1([2,2,4,4])$$
$$[1,1,4,4] \to \text{nop}$$
$$[1,1,2,2] \to \text{nop}$$
$$[2,2,4,4] \to \text{nop}$$

$$[1,2] \to \mathsf{john}$$
$$[2,3] \to \mathsf{sometimes}$$
$$[3,4] \to \mathsf{sleeps}$$
$$[1,1,2,2] \to *$$
$$[1,2] \to \mathsf{wrap}([1,1,2,2],[1,2])$$
$$[3,3,4,4] \to *$$
$$[2,3,4,4] \to \mathsf{conc}([2,3],[3,3,4,4])$$
$$[2,2,4,4] \to *$$
$$[2,3,4,4] \to \mathsf{wrap}([2,2,4,4],[2,3,4,4])$$
$$[2,4] \to \mathsf{wrap}([2,3,4,4],[3,4])$$
$$[1,4] \to \mathsf{conc}([1,2],[2,4])$$
$$[1,4] \to \mathsf{wrap}([1,1,4,4],[1,4])$$
$$[1,1,4,4] \to *$$

Figure 4: The decomposition grammar.

An algorithm that computes $\mathcal{G}'$ is given by Koller and Kuhlmann (2011). The basic idea is to simulate the backwards application to the rules of $D(w)$ on the right-hand sides of the rules of the homomorphism h_s. As an example, consider the rule

$$h_s(\alpha_1) = \mathsf{wrap}(x_2, \mathsf{conc}(x_1, \mathsf{wrap}(x_3, \mathsf{sleeps})))$$

If we instantiate the variables x_2, x_1, x_3 with the spans $[1,2]$, $[1,1,4,4]$ and $[2,3,4,4]$, respectively, then the backwards application of $D(w)$ yields $[1,4]$. This warrants the first production of $\mathcal{G}'$.

Step 3: Intersection. $\mathcal{G}'$ is an RTG that represents all derivation trees that are consistent with the input string; these are not necessarily grammatical according to $\mathbb{G}$. On the other hand, $\mathcal{G}_0$ generates exactly the grammatical derivation trees, including ones that do not describe the input string. To obtain an RTG for the derivation trees that are grammatical *and* match the input, we intersect $\mathcal{G}_0$ and $\mathcal{G}'$. This yields a grammar $\mathcal{G}''$ whose nonterminals are pairs of nonterminals from $\mathcal{G}_0$ and $\mathcal{G}'$ and the following rules:

$$S_{[1,4]} \to \alpha_1(NP_{[1,2]}, S_{[1,1,4,4]}, VP_{[2,3,4,4]})$$
$$NP_{[1,2]} \to \alpha_2(NP_{[1,1,2,2]})$$
$$VP_{[2,3,4,4]} \to \beta_1(VP_{[2,2,4,4]})$$
$$S_{[1,1,4,4]} \to \text{nop}$$
$$NP_{[1,1,2,2]} \to \text{nop}$$
$$VP_{[2,2,4,4]} \to \text{nop}$$

As expected, $L(\mathcal{G}'')$ contains a single tree, namely the derivation tree in Fig. 2b.

Discussion. $\mathscr{G}''$ is essentially a standard TAG parse chart for w. We have obtained it in three steps. Step 1 was an algebra-specific decomposition step; this was the only step in the parsing algorithm that was particular to TAG. Steps 2 and 3 then performed generic operations on RTGs, and are exactly the same whether we would parse with respect to TAG, context-free grammars, or a grammar formalism that describes objects in some other algebra. Thus this is a TAG parsing algorithm which decomposes into three parts, each of which is easier to understand, teach, and prove correct than a monolithic algorithm.

The runtime of the overall algorithm is $O(n^6)$ as long as both the algebra and the underlying RTG are binary. The string algebra was binary by design; furthermore, the RTG of every IRTG that encodes a TAG can be brought into a binary normal form (Gómez-Rodríguez et al., 2010). If we are parsing input on several interpretations simultaneously, e.g. in STAG parsing, binarization is not always possible (Huang et al., 2009), and the parsing algorithm takes exponential runtime. See also Koller and Kuhlmann (2011) for a discussion of binarization.

7 Conclusion

We have shown how a variety of formal perspectives on TAG can be uniformly understood in terms of IRTGs. By introducing two new, regularly decomposable algebras for strings and derived trees, we have shown how to obtain a modular parsing algorithm for TAG and STAG. This algorithm can be adapted to support synchronous parsing and decoding. For IRTGs with weighted RTGs, which are capable of capturing PTAG (Resnik, 1992) and synchronous PTAG, we can also perform Viterbi parsing and EM training on the parse chart.

The general advantage of the parsing algorithm presented here is that it decomposes into simple components. Recombining these yields an algorithm that is essentially identical to the standard CKY parser for TAG (Shieber et al., 1995). We can obtain other parsing algorithms by varying the way in which intersection and inverse homomorphisms are computed.

Acknowledgments. We thank Matthias Büchse, John Hale, Andreas Maletti, Heiko Vogler, and our reviewers for helpful comments, and Thutrang Nguyen for an initial implementation.

References

H. Burden and P. Ljunglöf. 2005. Parsing linear context-free rewriting systems. In *Proc. of the 9th IWPT*.

H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi. 2008. Tree automata techniques and applications. Available on `http://tata.gforge.inria.fr/`.

J. Engelfriet and E. Schmidt. 1977. IO and OI. I. *Journal of Computer and System Sciences*, 15(3):328–353.

F. Gécseg and M. Steinby. 1997. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 1–68. Springer.

C. Gómez-Rodríguez, M. Kuhlmann, and G. Satta. 2010. Efficient parsing of well-nested linear context-free rewriting systems. In *Proc. of HLT/NAACL*.

L. Huang, H. Zhang, D. Gildea, and K. Knight. 2009. Binarization of synchronous context-free grammars. *Computational Linguistics*, 35(4):559–595.

A. Koller and M. Kuhlmann. 2011. A generalized view on parsing and translation. In *Proc. of the 12th IWPT*.

A. Maletti. 2010. A tree transducer model for synchronous tree-adjoining grammars. In *Proc. of the 48th ACL*.

U. Mönnich. 1998. Adjunction as substitution. an algebraic formulation of regular, context-free, and tree-adjoining languages. In *Proc. of FG*.

F. Morawietz and U. Mönnich. 2001. A model-theoretic description of tree adjoining grammars. *Electronic Notes in Theoretical Computer Science*, 53.

P. Resnik. 1992. Probabilistic tree-adjoining grammar as a framework for statistical natural language processing. In *Proc. of COLING*.

S. Schmitz and J. Le Roux. 2008. Feature unification in TAG derivation trees. In *Proc. of the 9th TAG+ Workshop*.

S. Shieber, Y. Schabes, and F. Pereira. 1995. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36.

S. Shieber. 2004. Synchronous grammars as tree transducers. In *Proc. of the 7th TAG+ Workshop*.

S. Shieber. 2006. Unifying synchronous tree-adjoining grammars and tree transducers via bimorphisms. In *Proc. of the 11th EACL*.

K. Vijay-Shanker and A. Joshi. 1985. Some computational properties of Tree Adjoining Grammars. In *Proc. of the 23rd ACL*.

K. Vijay-Shanker, D. Weir, and A. Joshi. 1987. Characterizing structural descriptions produced by various grammatical formalisms. In *Proc. of the 25th ACL*.

D. Weir. 1988. *Characterizing Mildly Context-Sensitive Grammar Formalisms*. Ph.D. thesis, University of Pennsylvania, Philadelphia, USA.