USING FEATURE LOGIC PROGRAMMING TO DESCRIBE TREE TRANSDUCERS AND TREE ADJOINING GRAMMARS

WILLIAM C. Rounded

1. Introduction

This draft paper contains a new proposal for modeling tree transductions and tree adjoining grammars using feature logic. Originally the intent was to have a declarative logic which was less complex than monadic second order logic, in which both transductions and adjunction sequences could be expressed. I feel that the feature logic descriptions are in fact somewhat easier than their monadic second-order counterparts, just because feature logic is not such a concise description logic. The price paid, though, is undecidability. We employ a proof-theoretic formalism called disjunctive feature logic programming, which because it can be regarded as a fixed-point logic in which equalities between structures can be expressed, has an undecidable satisfiability problem.

This project has been carried out previously. One work is a PhD thesis and subsequent monograph by Bill Keller [5]. His work involves extending Kasper-Rounds logic by allowing regular expressions as feature modalities, thus introducing a fixed point formalism. Keller models tree adjoining grammars using these ideas. Another paper, by Vijay-Shanker and Joshi [12], explicitly extends TAGs so they work with feature structures (FTAGs). For this they use an extension of Kasper-Rounds logic with lambda-terms. The present work keeps the usual (typed) feature logic, but describes the derivation process differently, using disjunctive feature logic programming.

The key tool we use, besides feature logic programming, is the concept of a three-dimensional tree, due to Rogers [8]. We translate such trees into ordinary feature structures using a simple-minded encoding. We obtain a rather nice picture of adjunctions and transductions, which lets us compare them in a common setting. Rogers has already used these geometric notions to characterize tree-adjoining grammars, and our work roughly parallels his. Modelling the transducers, however, seems to be new.

Disjunctive feature logic programming, pace undecidability, is well-grounded theoretically. It is stated in a resolution-based style, and is packaged with a complete resolution proof system for feature logic. This whole system is derived from a general clausal logic programming paradigm for coherent algebraic directed-complete partial orders appearing in Rounds and Zhang [9]. Properties such as compactness and completeness follow immediately from this work. The main contribution to feature logic is a full proof theory for systems involving nonlogical axioms presented as implicational constraints. A side benefit is that at least in some cases we can consider infinite trees and feature structures as models of the logic.

The paper is organized as follows: Section 2 is a review of typed feature structures and feature logic, with a few words about bounded-complete partial orders, the only kind of poset we need here. Section 3 then introduces clausal theories and models, together with a resolution proof system. Again this is background. Section 4 introduces disjunctive feature logic programming and the relevant theoretical foundations, from Rounds and Zhang [9]. Section 5 contains definitions and examples of tree transducers and tree adjoining

---

1Thanks to Stuart Shieber for inviting me to Harvard in December 2005 to discuss ideas leading to this paper. Thanks also to the members of my University of Michigan seminar EECS 598-2, Winter 2007, for listening to sometimes incoherent assertions.
grammars, and Section 6 gives an informal description of the modelling of these formalisms. The beginning of this section describes the feature structure encoding of 3-dimensional trees, and gives a definition of the tree yield of a 3D feature structure.

Section 7 contains the mathematical details of the representation of transductions and TAG derivations in feature logic. The common idea is that models of a logic program are in all cases derivation trees for the respective grammatical systems. We begin with an easily understood formalism – regular tree grammars - in order to set the stage. Then we model top-down tree transductions, and finally model linear context-free tree grammars, which encode TAGS rather easily. Section 8 has a brief proof of undecidability.

2. Feature Structures and Feature Logic

The first two definitions are not really crucial to most of the paper, but appear first logically.

Definition 2.1. A bounded-complete partial order (bcpo) is a partially ordered set \((D, \sqsubseteq)\) in which any finite set of elements bounded from above has a least upper bound, and which has least upper bounds for any directed set. The latter is a subset \(X\) of \(D\) such that for any \(a, b \in X\), there is \(c \in X\) with \(a \sqsubseteq c\) and \(b \sqsubseteq c\). We require also a bottom element \(\bot \in D\).

Definition 2.2. A finite element of a bcpo is an element \(a \in D\) such that if \(X \subseteq D\) is directed and \(a \sqsubseteq \bigvee X\) then there is \(x \in X\) with \(a \sqsubseteq x\). A bcpo is algebraic if every element is the least upper bound of all the finite elements beneath it.

Definition 2.3. Let \((\Sigma, \sqsubseteq)\) be a finite bcpo of “types” and \(L\) a finite set of “feature names”. An appropriateness specification for \((\Sigma, L)\) is a relation \(Ap\) between \(\Sigma\) and the set of subsets of \(L\). This relation is required to satisfy the condition that \((\sigma, B) \in Ap\) and \(\sigma \sqsubseteq \sigma'\) implies \((\sigma', B) \in Ap\).

The idea here is that types in \(\Sigma\) are linguistic categories. A pair \((S, \{l, r\}) \in Ap\) indicates that \(S\) is a binary symbol which can have a left and right daughter. It might also be possible for \(S\) to be unary; this could be modelled by \((S, \{c\}) \in Ap\).

We present a very concise version of typed feature logic.

Definition 2.4. Conjunctive formulas:

- true
- \(\sigma\) for \(\sigma \in \Sigma\)
- \(x : \phi\) for \(x \in L^*\) and \(\phi\) a conjunctive formula
- \(\pi \equiv \rho\) for \(\pi, \rho \in L^*\)
- \(\phi \land \psi\) for \(\phi, \psi\) conjunctive formulas.

Definition 2.5. A clause is a finite set (possibly empty) of conjunctive formulas.

We identify a clause with the disjunction of all the formulas in it. The empty clause is “inconsistent”. The following definition of typed feature structure is from Moshier [7].

Definition 2.6. A typed feature structure over \((\Sigma, L)\) is a triple \((P, \equiv, \tau)\), where:

1. \(P\) is a nonempty prefix-closed subset of \(L^*\);
2. \(\equiv\) is a right-invariant congruence relation on \(P\) in the sense that \(\pi l \in P\) and \(\pi \equiv \rho\) imply \(\rho l \in P\) and \(\pi l \equiv \rho l\);

\(^2\)In Carpenter [3], appropriateness also requires saying what type of value a feature can have. This stipulation does not seem to be necessary for our purposes.
Every clause has a finite set (possibly empty) of subsumption-minimal satisfiers.

Let \( D \) be the set of all feature structures, and let \( f, g \in D \). We say that \( f \sqsubseteq g \) if \( P_f \sqsubseteq P_g \), \( \equiv_f \equiv_g \), and \( \tau_f(\pi) \sqsubseteq \tau_g(\pi) \) for all \( \pi \in P_f \).

Definition 2.8 (Feature structure after a path). Let \( f = (P, \equiv, \tau) \) be a feature structure and \( \pi \in P \). By \( f/\pi \) (“\( f \) after \( \pi \)”) we mean the structure with paths \( \{ \rho | \pi \rho \in P \} \), with the induced equivalence relation \( \rho \equiv_{\pi} \rho' \) iff \( \pi \rho \equiv \pi \rho' \), and with \( \tau_\pi(\rho) = \tau(\pi \rho) \).

Definition 2.9. The set \([\phi]\) of satisfiers of a formula \( \phi \) is given as follows:

1. \([\bot] = D\), the set of all feature structures.
2. \([\sigma] = \{ f : \sigma \sqsubseteq f \}\);
3. \([l : \phi] = \{ f | f/l \in [\phi] \}\);
4. \([\pi \equiv \rho] = \{ f | f/\pi \equiv f/\rho \}\);
5. \([\phi \land \psi] = [\phi] \cap [\psi] \).

If \( f \in [\phi] \), we write \( f \models \phi \). For a clause \( K \), we write \( f \models K \) if for some \( \phi \in K \), \( f \models \phi \).

Proposition 2.10 (Persistence). If \( f \sqsubseteq g \) and \( f \in [\phi] \) then \( g \in [\phi] \).

Theorem 2.11. The set \((D, \sqsubseteq)\) is an algebraic bcpo. The finite elements are the feature structures \( f \) such that \( \equiv_f \) has finite index.

Proposition 2.12. Every satisfiable conjunctive formula has exactly one subsumption-minimal satisfier.

Proposition 2.13. Every clause has a finite set (possibly empty) of subsumption-minimal satisfiers.

Notation. To represent clauses, it is convenient to identify logically equivalent conjunctive formulas. We can do this simply by using the minimal satisfier itself, coded as a data structure using some notational representation. For example, the minimal satisfier of \( f : (a \land (g : b)) \) will be written as \( f : a[g : b] \). This is a term-like notation based on the \( \psi \)-terms of Ait-Kaci and Podelski [1]. Types appear as “function symbols” in this notation, and square brackets replace parentheses, a device suggestive of the familiar attribute-value matrices.

From now on we will think of a clause as a finite set of subsumption-minimal feature structures. This will make logic programming rules quite easy to state. In particular, if \( K \) is a clause, \( f \models K \) iff \( g \sqsubseteq f \) for some \( g \in K \).

3. Theories and Models

Definition 3.1. Let \( K, L \) be clauses. We say \( K \models L \) if every satisfier of \( K \) is a satisfier of \( L \). A theory is a set of clauses. A theory \( T \models L \) if whenever \( f \models K \) for all \( K \in T \), then \( f \models L \). (We say that \( T \) logically implies \( L \).) The logical closure \( Cl(T) \) of \( T \) is the set \( \{ L | T \models L \} \). A theory \( T \) is logically closed if \( T = Cl(T) \).

Notice that any theory containing the empty clause logically implies any clause.

Theorem 3.2 (Compactness). If \( L \) is a logical consequence of \( T \) then \( L \) is a logical consequence of a finite subset of \( T \).
Sketch of proof. One can prove that the space of logically closed theories is an algebraic bcpo under the inclusion ordering; further, that the compact elements of this bcpo are exactly the logical closures of finite theories. (This is not immediate; for a proof see [9].) Now suppose \( T \models L \). The set of closures of finite subtheories of \( T \) is directed, and its least upper bound is the closure of \( T \) (by algebraicity). The closure of \( L \) is a subset of the closure of \( T \) by hypothesis. Since it is itself a compact element, it must be subsumed by the closure of one of the finite subcollections of \( T \).

The compactness theorem does not really depend on much about feature logic; only on the fact that the domain of feature structures is an algebraic bcpo, and that the compact elements are the finite feature structures.

3.1. A resolution proof system. We introduce a (generalized) resolution rule. In the following, \( K, L, M \) are clauses.

\[
\frac{K \quad L \quad f \in K \quad g \in L \quad \{f \cup g\} \models M}{M \cup (K - f) \cup (L - g)}
\]

The side condition \( f \cup g \) means the singleton clause consisting of the unification of \( f \) and \( g \) if that exists, and the empty clause otherwise. Notice that \( f \cup g \) is the minimal satisfier of \( \phi \land \psi \), where \( f \) is a minimal satisfier of \( \phi \) and \( g \) of \( \psi \). Also, when the consequent clause is derived, we throw out any non-minimal structure in it.

To see how this rule generalizes the usual one, let \( a \) and \( a' \) be inconsistent types, and consider clauses \( K, L \) with \( a \in K \) and \( a' \in L \). Then \( \{a \sqcup a'\} = \emptyset \models \emptyset \), so the conclusion of the rule is \( (K - a) \cup (L - a') \).

This rule is sound; suppose the side conditions in the rule hypothesis hold. We claim that for any \( h \), if \( h \models K \) and \( h \models L \), then \( f \models M \cup (K - f) \cup (L - g) \). The only way this could not happen is for \( f \sqsubseteq h \) and \( g \sqsubseteq h \). But then by the last condition, \( f \models M \).

We add two more standard rules to our proof system:

- **Initial:**
  \[
  \begin{array}{c}
  \{1\} \\
  \end{array}
  \]

- **Inconsistency:**
  \[
  \begin{array}{c}
  \emptyset \\
  M \\
  \end{array}
  \]

Our objective is

**Theorem 3.3** (Completeness). If \( T \models L \) then \( L \) is provable from \( T \) using resolution.

*Proof.* By the compactness theorem, \( L \) is a logical consequence of a finite subset \( \{K_1, \ldots, K_n\} \) of \( T \). We show that if \( \{K_1, \ldots, K_n\} \models L \) then we can prove \( L \) from the finitely many assumptions \( K_i \).

One case of this is relatively easy: the theory is a single clause \( K \). Suppose \( K \models L \); we wish to show \( K \vdash L \), where \( \vdash \) means “proves using resolution”. The case \( K = \emptyset \) is trivial by the inconsistency rule, so suppose \( K \neq \emptyset \). We first show that there is a subset \( L' \) of \( L \) such that \( K \vdash L' \). We do this by removing elements of \( K \) and adding elements of \( L \) as follows: If \( K \subseteq L \) then we do not need the resolution rule at all; we can take \( L' = K \). Suppose \( f \in K \setminus L \). Then since \( K \models L \) there must be \( g \in L \) with \( g \subseteq f \). By the resolution rule \( K \vdash (K - f) \cup \{g\} \). Continue this process, eventually removing all \( f \in K \setminus L \), and obtaining an \( L' \subseteq L \) such that \( K \vdash L' \). Now we want to show \( L' \vdash L \). This is clear if \( L' = \emptyset \). If not, fix a \( d \in L' \). For any \( c \in L \setminus L' \) we have \( L' \vdash L' - \{d\} \cup \{d, c\} \). Continue this addition of \( c \)’s to \( L' \) until we have \( L' \vdash L \). We then have \( K \vdash L' \vdash L \). This takes care of the “easy” case.

Now let \( P \) and \( Q \) be clauses. The cross-unification \( P \bowtie Q \) is the clause \( \{p \cup q \mid p \in P, q \in Q\} \). This is, in view of our identification of clauses as disjunctions, and unification as producing the most general
satisfier of a conjunction, a clause which is logically equivalent to \( P \land Q \), by the distributive law. In fact, \( \{ P, Q \} \) is a theory which has the same set of models as \( P \bowtie Q \). Therefore, up to logical equivalence, \( \bowtie \) is associative.

**Claim 3.4.** \( \{ P, Q \} \vdash P \bowtie Q \).

Assuming the claim, we can finish the proof of the completeness theorem. Suppose that \( T \models L \). By the compactness theorem, there are finitely many clauses \( X_1, \ldots, X_n \) such that \( \{ X_1, \ldots, X_n \} \models L \). By the claim and induction,

\[
X_1, \ldots, X_n \vdash X = \text{def} \ X_1 \bowtie \ldots \bowtie X_n.
\]

Now the cross-unification clause \( X \) is logically equivalent to the conjunction of \( X_1, \ldots, X_n \). In particular \( L \) must be a logical consequence of \( X \), so that by the first case of the proof, \( X \vdash^* L \).

It remains to prove the claim. This we omit! (A proof can be found in [9].) Instead we give an example: we show that \( \{ f \}, \{ g, h \} \vdash \{ fg, fh \} \):

\[
\begin{array}{c}
\{ f \} \\
\{ f \} \\
\{ fg, fh \}
\end{array} \quad \begin{array}{c}
\{ g, h \} \\
\{ fg \} \\
\{ fh \}
\end{array}
\]

where we have written \( f \sqcup g \) as \( fg \). In the second inference we reuse the assumption \( \{ f \} \).

\[
\square
\]

4. Logic Programming with Feature Logic

**4.1. Definition and examples.** We introduce a different kind of logic programming. The typical LP language, like Prolog, is based on first-order logic. Our logic programming is a very close relative of the recursive type constraint systems in Carpenter’s book [3, Chapter 15]. The difference is that feature logic programming is presented as an example of a much more general kind of disjunctive logic programming over coherent algebraic domains [9]. The flexibility of the rules and rule schemas allow several types of tree mappings and grammars to be modeled.

**Definition 4.1.** A logic programming rule is a construct

\[
f \rightarrow L
\]

where \( f \) is a feature structure and \( L \) is a clause.

**Example 4.2.** Consider a CFG with one nonterminal \( S \), one terminal \( a \), start symbol \( S \), and the productions

\[
S \rightarrow aS \mid a.
\]

We create a type system as follows. The nonterminal \( S \) is one type, and the symbol \( a \) is the other. To represent trees as feature structures, we have the feature names \( l \) and \( r \) to be used for binary branching, and \( c \) for unary branching.

The productions can be displayed as logic programming rules:

\[
\bot \rightarrow \{ S \}
\]

\[
S \rightarrow \{ S[l:a, r:S], S[c:a] \}
\]

**Definition 4.3.** A logic program is a set of logic programming rules.

A logic programming rule will be of the form \( f_0 \rightarrow \{ f_1, \ldots, f_n \} \), with the left side of the rule a unit clause The set on the right is still a clause, but is interpreted operationally as a (forced) nondeterministic choice.
In the official semantics, the rules are used to create a theory. The structures sanctioned by the grammar (logic program) are the structures satisfying all clauses of the theory. We therefore need to give a program inference rule allowing to add clauses to an already existing theory. We add to the logical resolution rules above the following resolution-style inference rule $U_P$ for a program $P$:

$$
\frac{M \models g \quad f \rightarrow L \in P \quad f \sqsubseteq g}{L \cup (M \setminus \{g\})}
$$

**Definition 4.4.** The theory $\text{Th}(P)$ of a program $P$ is the set of all clauses derivable using the logical and program inference rules.

Notice that the theory of $P$ is logically closed; if $\text{Th}(P) \models M$ then $\text{Th}(P) \vdash M$ by the completeness theorem, whence by definition of $\text{Th}(P)$, $M \in \text{Th}(P)$.

**Example 4.5.** What is the theory of the CFG program above? By the logical resolution rule, we get $\{\bot\}$ for free. Then we get $\{S\}$ using the program inference rule. From this we get the clause $\{S[l:a, r:S], S[c:a]\}$

The program inference rule cannot be applied to this clause in an irredundant way. Going back to the logical syntax, the theory consists of all logical consequences of $\{(S[l:a, r:S], S[c:a])\}$.

Example syntax: $S[l:a, r:S]$ is a model of the last clause, and of the theory of the program. This is only a partial CFG tree, and we might insist that our program produce a theory which admits as models only “unexpandable” derivation trees. How should we do this? We want to be able to reuse the rules as we do CFG productions, to “expand” the lowest $S$ in the partial tree. We propose to generate new program rules with a rule generating scheme allowing us to work inside a feature structure (equivalently, a conjunctive formula).

**Definition 4.6.** Let $f$ be a feature structure and $\alpha$ a sequence of features (path). By $\alpha : f$ we mean the feature structure whose paths are $\{\alpha \pi \mid \pi \text{ is a path of } f\}$ (more precisely, the prefix closure of this set). The type of any proper prefix of $\alpha$ is $\bot$, and the type of $\alpha \pi$ is the type of $\pi$. The prefix-extension of a rule $f \rightarrow L \in P$ is the set of rules of the form $\alpha : f \rightarrow \{\alpha : g \mid g \in L\}$, where $\alpha$ is any path.

**Example 4.7.** If we form the prefix extension of the rule $S \rightarrow (S[l:a, r:S])$, then we have the rule

$$r:S \rightarrow \{r:S[l:a, r:S]\}$$

in our CFG program. This rule allows inference of the clause

$$\{r:S[l:a, r:S], S[c:a]\}.$$

Cross-unifying this clause with $\{S[l:a, r:S], S[c:a]\}$ gives the clause

$$\{S[l:a, r:S[r:S[l:a, r:S]], S[c:a]]\}.$$

We illustrate prefixing in Figure 1.

**Remark 4.8.** Closing rules under prefix-extension is a way of expressing an idea from modal logic called the universal modality. If $\phi$ is a modal formula, by $\square_u \phi$ is meant a formula which is supposed to hold at all worlds “transitively” accessible from the current one, not just those reachable by one iteration of the accessibility relation. We could use the notation $\square_u (f \rightarrow L)$ to express the prefix-extension of a rule, but since $f \rightarrow L$ is a logic programming rule, and not a logic formula, we do not do this.
Remark 4.9. Warnings:

- Disjunction in LP rules cannot be replaced by “listing out” the disjunction as productions. For example, consider the simple program

\[ \bot \rightarrow S \]
\[ S \rightarrow \{ l \big\downarrow r, c \big\} \]

Original program

\[ r \mid S \rightarrow \{ r \mid S \}
\]
\[ a \big\downarrow S \]
\[ a \big\downarrow S \]

Additional rule generated by prefixing

\[ \bot \rightarrow S \]
\[ S \rightarrow a \]
\[ S \rightarrow \overline{a} \]

where \( a \) and \( \overline{a} \) are incompatible types, but both subsumed by \( S \). This is very different from

\[ \bot \rightarrow S \]
\[ S \rightarrow a \]
\[ S \rightarrow \overline{a} \]

The first program has two models \( a \) and \( \overline{a} \), but the second one has no models.

- There is another way we cannot use rules like productions: they have the effect of making productions obligatory instead of optional. Consider the program generated by applying prefixing to the rule \( S \rightarrow c : S \), and adding the single rule \( \bot \rightarrow S \). Then every path of \( cs \) must be extended by another \( c \), making the program describe the infinite feature structure consisting of an infinite path of \( cs \), all nodes labeled with \( S \).

4.2. Theory. In this subsection we try to justify our definitions on theoretical grounds. Corresponding to a logic program \( P \) we introduce an operator \( T_P \) on logically closed theories. This operator is a continuous function on the Scott domain of closed theories, and it has therefore a least fixed point \( \mu T_P \). We show that this is equal to \( Th(P) \). Then we invoke a theorem of Rounds and Zhang [9] to show that this theory is the set of clauses entailed by all “models of \( P \)” in the traditional sense of logic programming.
Consider the program resolution rule
\[
\frac{M \quad g \in M \quad f \rightarrow L \in P \quad f \sqsubseteq g}{L \cup (M \setminus \{g\})}
\]
We say that \(L \cup (M \setminus \{g\})\) is a \(P\)-consequence of \(M\). Further, we say that \(N\) is a \(P\)-consequence of a theory \(T\) if it is a \(P\)-consequence of some \(M \in T\).

**Definition 4.10.** Let \(T\) be a logically closed theory over \(D\), and let \(P\) be a program. We define
\[
T_P(T) = \text{Cl}\{Y \mid Y \text{ is a } P\text{-consequence of } T\}
\]
where \(\text{Cl}\) is the closure of a theory under logical resolution.

\(T_P(T)\) is the logical closure of the theory obtained from \(T\) by taking one step of the program inference rule in all possible ways. Clearly it is monotonic in the set inclusion ordering on theories.

**Theorem 4.11.** \(T_P\) is continuous in the inclusion order.

**Proof.** Let \(\Lambda\) be a directed set of theories. We have to show that
\[
T_P(\bigcup \Lambda) = \bigcup_{T \in \Lambda} T_P(T).
\]
Since \(T_P\) is monotone, the inclusion of right in left is obvious. To go from left to right, suppose that \(Z\) is a logical consequence of the set of \(P\)-consequences of \(\bigcup \Lambda\). Then by compactness, \(Z\) is a logical consequence of a finite number of these \(P\)-consequences. This means that in fact \(Z\) is a logical consequence of the set of \(P\)-consequences of some finite number \(X_1, \ldots, X_m\) of clauses of \(\bigcup \Lambda\), and therefore of some particular \(T \in \Lambda\), since \(\Lambda\) is directed. This shows that \(Z\) is a member of the right hand side as well. \(\square\)

By the standard least-fixed point theorem, \(T_P\) has the least fixed point \(\bigcup_{n} T_P^n(\bot)\), where \(\bot\) is the theory consisting of all consequences of \(\{\text{true}\}\).

**Proposition 4.12.** \(\mu T_P = \text{Th}(P)\).

**Proof.** Suppose a clause is in \(\mu T_P\). Then it is in \(T_P^n(\bot)\) for some \(n\). By induction, it has a proof from \(\bot\) involving at most \(n\) uses of the \(P\) rule, and so is in \(\text{Th}(P)\). The converse direction just reverses this argument. \(\square\)

**Definition 4.13.** Let \(P\) be a logic program. A feature structure \(g\) is said to be a model of \(P\) if for every rule \(f \rightarrow L\) in \(P\), if \(f \sqsubseteq g\), then \(g \models L\). A clause \(M\) is a model-theoretic consequence of \(P\) if every model of \(P\) satisfies \(M\). We write \(\text{ModCons}(P)\) for the set of all clauses which are model-theoretic consequences of \(P\).

The following theorems are from Rounds and Zhang [9]. The first result is actually needed for the second.

**Theorem 4.14.** Every minimal model of \(\text{Th}(P)\) is a model of \(P\).

**Theorem 4.15.** \(\text{ModCons}(P) = \text{Th}(P)\).

**Corollary 4.16.** Every model of \(P\) is a model of \(\text{Th}(P)\).

Unfortunately, the set of models of \(P\) need not be upward closed; one can find \(P\) with a model \(m\) and \(n \sqsupseteq m\) but \(n\) not a model.

5. **TWO COMMON GRAMMAR FORMALISMS**

In this section we review tree transducers and tree adjoining grammars.
5.1. **Tree Transducers.** In the original papers on tree transducers, including my own, trees have nodes labeled with elements of a set of symbols \( \Sigma \), each element of which has a non-negative integer rank assigned to it, say by a function \( \text{arity}(\sigma) \), determining the number of children of a node labeled with \( \sigma \). We let \( \Sigma_n = \{ \sigma \mid \text{arity}(\sigma) = n \} \).

**Definition 5.1.** Let \( X \) be a countably indexed set of variables. The set of **ranked trees** over \( \Sigma \) and \( X \), \( T_{\Sigma}(X) \) is the smallest set such that

- Nullary symbols or variables at leaves: \( \Sigma_0 \cup X \subseteq T_{\Sigma}(X) \);
- Internal nodes: \( f(t_1, \ldots, t_n) \in T_{\Sigma}(X) \) for all \( f \in \Sigma_n \) and \( t_1, \ldots, t_n \in T_{\Sigma}(X) \).

We can analogously define, for any set \( S \) disjoint from \( \Sigma \), the set \( T_{\Sigma}(S) \), trees whose leaves are labeled with elements of \( \Sigma_0 \cup S \).

Plainly, trees without variables (ground trees) correspond to feature structures in which \( \equiv \) is just equality of paths, where appropriateness conditions given by the arity conditions, and where the feature names are implicitly given by integers representing the order of subtrees occurring in a given tree. Let \( T_{\Sigma}^0 \) be this set.

**Definition 5.2.** Let \( \Sigma_n(X) = \{ f(x_1, \ldots, x_n) \mid f \in \Sigma_n \} \). A **nondeterministic top-down tree transducer** is a tuple \( M = (Q, q_0, \Sigma, \Delta, \rightarrow) \), where

- \( Q \) is a finite set of states;
- \( q_0 \in Q \) is the initial state;
- \( \Sigma \) is a ranked input alphabet;
- \( \Delta \) is a ranked output alphabet;
- \( \rightarrow \) is a finite transition relation between \( Q \times \Sigma_n(X) \) and \( T_{\Delta}(Q \times X_n) \), where \( n \) can be any arity value except 0, when we require the output tree to be ground.

**Definition 5.3.** The one-step rewriting relation \( \vdash \) induced by a tree transducer is defined as follows. Given a tree \( t \in T_{\Delta}[Q \times T_{\Sigma}] \) we select an occurrence of some \( (q, f(t_1, \ldots, t_n)) \in Q \times T_{\Sigma} \) in \( t \). If \( (q, f(t_1, \ldots, t_n)) \rightarrow s \) is a transition rule, form the tree \( s' \) obtained by substituting \( t_i \) for \( x_i \) in the pairs \( (r, x_i) \) occurring as leaves of \( s \), and substitute the resulting tree in place of \( (q, f(t_1, \ldots, t_n)) \) in \( t \), giving a new tree \( t' \). Then \( t \vdash t' \).

As usual, we let \( \vdash^* \) be the reflexive transitive closure of \( \vdash \). The tree relation defined by \( M \) is then the set of pairs \( (t, t') \in T_{\Sigma}^0 \times T_{\Delta}^0 \) such that \( (q_0, t) \vdash^* t' \).

**Example 5.4.** Here is an arbitrary example. Let \( \Sigma = \{ c, a, b \} \) with \( \text{arity}(c) = 2, \text{arity}(a) = \text{arity}(b) = 0 \), and let \( \Delta = \Sigma \). Let \( Q = \{ q, p, s \} \) with \( q_0 = q \). Define a tree transducer \( M \) by the rules

\[
\begin{align*}
(q, c(x_1, x_2)) & \rightarrow c((p, x_2), (q, x_1)) \\
(q, c(x_1, x_2)) & \rightarrow c((p, x_1), (s, x_2)) \\
(p, b) & \rightarrow a \\
(p, a) & \rightarrow a \\
(s, b) & \rightarrow b
\end{align*}
\]

A derivation using \( M \) is given in Figure 2. In the first transition, the left subtree has been flipped to the right using the first rule; at the bottom the second rule has been applied to the flipped subtree.

5.2. **Tree Adjoining Grammars.** The following description of TAG is quoted from the PhD thesis of [Gert van Noord](11). Those familiar with TAGs can skip this section.
A tree adjoining grammar consists of a set of elementary trees, partitioned into initial and auxiliary trees. These trees constitute the basic building blocks of the formalism. The operations of adjunction and substitution build derived trees from elementary trees.

An initial tree is a tree of which the interior nodes are all labelled with non-terminal symbols, and the nodes on the frontier are either labelled with terminal symbols, or with non-terminal symbols, which are marked with the substitution marker (\(\downarrow\)).

An auxiliary tree is defined as an initial tree, except that exactly one of its frontier nodes must be marked as the foot node (\(*\)). The foot node must be labelled with a non-terminal symbol which is the same as the label of the root node.

As an example, consider the initial and auxiliary trees in Figure 3. In the initial tree \(\alpha_1\), the np \(\downarrow\) node is a substitution node, and the word left is a terminal symbol labeling a frontier node. In the auxiliary tree \(\beta_2\) the foot node is the rightmost daughter of the root node. The word very is a terminal symbol at a frontier node.

Derived trees are built from initial and auxiliary trees by substitution and adjunction. Substituting a tree \(\alpha\) in a tree \(\alpha'\) simply replaces a substitution node in \(\alpha\) with \(\alpha'\), under the convention that the non-terminal symbol of the substitution node is the same as the root node of \(\alpha'\). For example, substituting \(\alpha_2\) in \(\alpha_3\) is illustrated in Figure 4.

Adjoining an auxiliary tree \(\beta\) at some node \(n\) of a derived tree \(\gamma\) proceeds as follows. First, the non-terminal symbol of the root node (and hence the non-terminal symbol of the foot node) of \(\beta\) should be the same as the non-terminal symbol associated with \(n\). The subtree \(t\) of rooted by \(n\) is removed from \(\gamma\), and...
FIGURE 3. Three initial and two auxiliary trees as an example of a TAG.

$\beta_2$ is substituted for it instead; then $t$ is substituted in the foot node of $\beta$. An illustration of the adjunction operation is presented in Figure 5. The derived tree representing “the boy” serves as a host for the adjunction of the auxiliary tree $\beta_3$ at the node labeled $n$. The auxiliary tree is substituted at node $n$, and the “boy” subtree of $n$ is substituted in the foot node of the auxiliary tree.
6. MODELLING GRAMMAR FORMALISMS

We can model tree transductions and derivations for tree-adjoining grammars in our feature logic formalism. The key idea is a feature structure version of Jim Rogers’ \textit{n-dimensional trees}. As in Rogers, only three dimensions will actually be necessary. Before proceeding to the mathematics of this, we pause to explain these structures.

6.1. Multidimensional feature structures. The title of the subsection is actually a misnomer. We can represent structures very like multidimensional trees using ordinary feature structures. All we have to do is to use special numbered features (say the feature number 3) to represent “selection of a 3-dimensional structure rooted at the current node.” This mechanism, in fact, only needs to kick in at dimension 3 or higher, because ordinary feature structures are the two-dimensional version of 2-dimensional trees.

An example is worth many words. We give Rogers’ example of a 3-dimensional tree and its representation as a 3-dimensional feature structure in Figure 6. The top part (i) of the figure is Rogers’ tree. The indexing scheme attaches certain tuples to nodes in order to address each node uniquely. At any node, say \((1, \epsilon)\), there can appear a 3D tree whose base is a new “level” of the big tree. The base of this tree would be the bold triangle labeled with \((1, \epsilon)\) at its root.

The representation (ii) displays the same tree as an ordinary feature structure. Following an arc labeled 3 moves you to a new level in the 3-dimensional tree. Instead of having a new 3D tree as part of the yield of the 2D tree, the levels are “split out.” Instead of bracketed numbers, we use names like \(a\), \(b\), and \(c\) on the branches of the 2D levels.

It is clear that one can uniquely address nodes in our 3D feature structure. For example, the node labeled by \((1, \epsilon)\) in (i) is accessed by the path \(3b3\) in (ii). Counting the number of 3s in a path tells you how deep
you are in dimension 3. For higher dimensions, we can use “dimension features” 4, 5, and so on. We do not pursue higher dimensions here.

A listing of the corresponding nodes will clarify the relation between the addressing schemes.

\[
\begin{align*}
\epsilon & \mapsto \epsilon \\
\langle \epsilon \rangle & \mapsto 3 \\
\langle \langle 0 \rangle \rangle & \mapsto 3a \\
\langle \langle 1 \rangle \rangle & \mapsto 3b \\
\langle \langle 2 \rangle \rangle & \mapsto 3c \\
\langle \langle 1, 0 \rangle \rangle & \mapsto 3ba \quad (1, 0 \text{ really is } 1 \cdot 0) \\
\langle \langle 1, 1 \rangle \rangle & \mapsto 3bb \\
\langle \langle 1, \epsilon \rangle \rangle & \mapsto 3b3 \quad (\text{introducing the root of a new level}) \\
\langle \langle 1, 0 \rangle \rangle & \mapsto 3b3a \quad (\langle \langle 1, 0 \rangle \rangle = \langle \langle 1 \rangle \cdot \langle 0 \rangle \rangle, \text{ and corresponds to } 3b \cdot 3a) \\
\end{align*}
\]

etc.
6.2. **Formalization of the encoding.**

**Definition 6.1.** Let \( L \) be a finite set of feature names not containing “3”. A 3DFS over \( L \) just an ordinary feature structure over \( L \cup \{3\} \).

**Definition 6.2 (Rogers paths and 3D trees).** Under the same conditions on \( L \), we stipulate the following:

- Any element of \( L^* \) is a 2D path;
- A 3D path is either \( \langle \rangle \) or \( \langle p_1, \ldots, p_n, y \rangle \), where \( p_i \) and \( y \) are 2D paths. To denote concatenation in dimension 3 we will explicitly use the raised dot: instead of \( \langle p_1, p_2, p_3 \rangle \), we will use \( p_1 \cdot p_2 \cdot p_3 \). Ordinary concatenation over \( L \) will be expressed without a dot. We will use \( w, x, y \) to stand for 3D paths, and \( p, q \) to stand for ordinary paths.
- A 3D tree domain is a nonempty hereditarily prefix-closed set \( T \) of 3D paths. By the last we mean that if \( w \cdot \langle p \rangle \in T \) and \( l \in L \) then \( w \in T \) and \( w \cdot \langle p[l] \rangle \in T \).

**Definition 6.3.** We define a map \( \pi \) from 3D paths to strings over \( (L \cup 3)^* \) as follows:

- \( \pi(\langle \rangle) = 3 \);
- \( \pi(w \cdot \langle p \rangle) = \pi(w)3p \).

In the reverse direction, we map a string \( \alpha \) over \( (3(L \cup 3))^* \) to \( \pi_3(\alpha) \) as follows:

- \( \pi_3(3) = \langle \rangle \);
- \( \pi_3(\alpha3p) = \pi_3(\alpha) \cdot \langle p \rangle \).

Notice that the image \( \pi[W] \) of the set \( W \) of all 3D paths is \( 3(L \cup 3)^* \), because the set \( W \) is hereditarily prefix-closed.

**Proposition 6.4.** The maps \( \pi \) and \( \pi_3 \) are inverses, when we redefine the codomain of \( \pi \) to be \( 3(L \cup 3)^* \).

**Proof.** Since \( \pi \) is now onto, we only have to show \( \pi_3(\pi(w)) = w \). This is clear if \( w = \langle \rangle \). Assume \( \pi_3(\pi(w)) = w \). Then \( \pi_3(\pi(w \cdot \langle p \rangle)) = \pi_3(\pi(w)3p) = \pi_3(w) \cdot \langle p \rangle = w \cdot \langle p \rangle \) as desired. \( \square \)

6.3. **Yields.** We would like to read off a 2DFS as the yield of a 3DFS; for this we employ a different scheme from that found in Rogers. In this subsection we work with trees \( (P, \tau) \), where we ignore the relation \( \equiv \). At this point we recall the inductive definition [5.1] of finite trees over the (type) set \( \Sigma \) and set \( L \) of feature names.

**Definition 6.5.** A tree over the \( L \)-ranked alphabet \( \Sigma \) is given by

- \( \sigma \) is a (zero-ary) tree if \( \sigma = \sigma(\emptyset) \);
- If \( \sigma = \sigma(l, r) \), then \( s \) and \( u \) are trees, then \( \sigma(l : s, r : u) \) is a tree.
- If \( \sigma = \sigma(c) \) and \( s \) is a tree then \( \sigma(c : s) \) is a tree.

In the definition, we denote two typical appropriateness sets by \( \sigma(l, r) \) and \( \sigma(c) \). Then, for example, \( \sigma(l, r, 3) \) denotes the type of a ternary tree with a “3” subtree. We move back and forth freely between this inductive definition and the one using the FS description.

A succinct definition of the yield mapping on finite trees is available in case the input is described inductively. We simply define a top-down “deleting” tree homomorphism (one-state top down transducer) operating on these structures. The definition is as follows.
Definition 6.6. The homomorphic yield $y_h(f)$ of a 3D finite tree is given as follows:

$$y_h(\sigma(l:t_l, r:t_r, 3:t_3)) = y_h(t_3);$$
$$y_h(\sigma(l:t_l, r:t_r)) = \sigma(l:y_h(t_l), r:y_h(t_r));$$
$$y_h(\sigma) = \sigma.$$

Example 6.7. Here is a 3D tree and its yield.

Paths like $bc$ and $bd3f$ do not appear in the yield. Why is this? The answer will appear shortly: the sub-trees of $c$ and $d$ will be copied into the next level using path equations, when we apply a logic programming rule involving a transducer transition or an adjunction. This copying represents substitution.

Rogers’ version of “yield” requires one to designate head-to-foot paths (spines) in all of the 2D trees occurring on the different levels of a 3D tree, so it is clearly a different notion. He does not consider infinite trees, but we can give a feature-based definition of “yield”, one which applies to at least some of these.

Definition 6.8. Let $u$ be an acyclic but not necessarily finite 3D tree. By a principal path of $u$ we mean a maximal path $\alpha$ of $u$ such that if $\alpha = \alpha_13p\alpha_2$, then the path $p \in L^+$ has no proper prefix $q$ such that $\alpha_1q3$ is a path of $u$. The Let $d_3(\alpha)$ be the string homomorphism deleting all 3’s from $\alpha$. Then the set of paths of the F-yield $y_f(u)$ of $u$ is the prefix-closure of

$$\{d_3(\alpha) \mid \alpha \text{ is a principal path of } u\}.$$  

In the figure, the principal paths of $u$ are $3b3k$ and $3a3p$. They are found by traversing from the root and “jumping in the 3rd dimension” at the very first opportunity. The F-yield of the tree is the set of paths $\{e, a, b, ap, bk\}$ with the typing shown.
To define \( y_f(u) \) completely we must specify how to assign types to each path in the yield set. For this we have to make an assumption about 3D trees: there are no infinite paths consisting entirely of 3s. Call such FTs 3-fair. Our grammar models in general have 3-fair infinite trees as models, so we can take the yields of these if we wish.

We give an algorithm to assign types to any path \( \pi \) of the form \( d_3(\alpha) \) for \( \alpha \) a principal path of a 3-fair tree \( u \).

1. To assign a type to \( \epsilon \), follow a string of 3’s until there are no more. That is, find \( m \) such that \( 3^m \) is a path of \( u \), but not \( 3^{m+1} \). Then let the type of \( \epsilon \) be the type of \( 3^m \) in \( f \). Record the path \( 3^m \) as the path so far.
2. Suppose that the path so far is \( \alpha \) and the type of \( \pi' \) has been assigned, and let \( \pi = \pi'a \) be a path in the yield set. We claim that \( \alpha a \) is a path of \( f \). If not, there is some other principal path \( \beta \) such that \( \beta a \) is a path of \( f \) and \( d_3(\beta) = \pi' \). Look at the first point in \( f \) where \( \alpha \) and \( \beta \) diverge. This cannot be at a point where there is a 3 transition, because both paths are principal. Hence it has to be at a point where there are only “real” choices for the next step, say features \( l \) for \( \alpha \) and \( r \) for \( \beta \). But then it is impossible for \( d_3(\alpha) = d_3(\beta) \).

We make the invariant assertion that \( \alpha \) has no 3-continuation. This was true initially, and we now assume it inductively. Then \( \alpha a \) is a principal path, and we again find \( m \) such that \( \alpha a 3^m \) has no 3-continuation. We now assign the type of \( \alpha a 3^m \) to \( \pi = \pi'a \), and reset the path so far to \( 3^m \).

With the specification of paths and types, we have now defined a tree \( y_f(u) \), called the F-yield of \( u \), to any 3-fair 3D tree.

**Theorem 6.9.** For any finite 3D tree \( t \), \( y_h(t) = y_f(t) \).

**Proof.** By induction on \( t \) using the definition of \( y_h \).

- If \( t \) has just one node, the theorem is obvious.
- Let \( t = (l : t_l, r : t_r) \) have no immediate 3-subtree. Then \( y_h(t) = \sigma(l : y_h(t_l), r : y_h(t_r)) \). By induction \( y_h(t_l) = y_f(t_l) \) and \( y_h(t_r) = y_f(t_r) \). We must then verify that \( y_f(\sigma(l : t_l, r : t_r)) = \sigma(y_f(t_l), y_f(t_r)) \). The type assigned to \( \epsilon \) \( y_f(\sigma(t_l, t_r)) \) is clearly \( \sigma \) because we have no 3-continuation. The paths for \( y_f(\sigma(l : t_l, r : t_r)) \) consist then of \( l \) prefixed to a path of \( y_f(t_l) \) and \( r \) prefixed to a path of \( y_f(t_r) \). The principal paths for both terms are the same – they start either with \( l \) or \( r \) – so \( y_f(\sigma(l : t_l, r : t_r)) \) and \( \sigma(y_f(t_l), y_f(t_r)) \) have the same paths. It is then easy to check that the type assignment algorithm produces the same answers.
- Finally, we consider the case \( t = (l : t_l, r : t_r, 3 : t_3) \). In this case the homomorphism definition gives \( y_h(t) = y_h(t_3) \), and inductive hypothesis gives \( y_h(t_3) = y_f(t_3) \). This is the same as \( y_f(t) \) because the first step in finding \( y_f(t) \) is to take an (erasable) step along the 3 path in \( t \). The rest of the process is then to find the yield of the resulting tree.

\[ \square \]

### 6.4. Modelling tree transducers in feature logic

Our feature logic programs for tree transductions construct transducer derivation trees as models. Each derivation is (roughly) responsible for adding an ordered pair \((u, v)\) to the tree relation associated to the transducer. The input \( u \) appears as a given component in a 3D tree on the “first level”, and the output is constructed by the program step by step, each transduction step adding a component at the next higher level. The input can be constructed using a regular tree grammar as in our previous example, but we focus here on the transduction aspect.

Once again the process is clearest with an example. Consider the transducer given in Example 5.4. The input tree \( c(c(a, b), b) \) is given as a FS with \( l \) and \( r \) branches, at the end of the \( \text{in} \) arrow in the top part of

Figure 7. The initial rule applied (as in Figure 2) is \((q, c(x_1, x_2)) \rightarrow c((p, x_2), q(x_1))\). This flips the left and right subtrees of the input tree. The effect of applying this rule is given by the feature logic rule:

\[
q[\text{in} : [l : \bot, r : \bot]] \rightarrow 3 : c[l : p, r : q] \sqcup [\text{in}r \equiv 3\text{lin}] \sqcup [\text{in}l \equiv 3\text{rin}]
\]

Since there is a nondeterministic choice in the tree transducer at the top node, we really need the rule whose right side is a two-element clause containing the other choice; but this leads to readability problems.
Here, the states appear as new types, ordered as a flat order, and all of which are less specific than the type c, a and b used as node labels. Notice that on the right, the FS $c[l : p, r : q]$ appears in a new level, after the “3” arc. Each time a step of transduction is applied, the piece of output is inserted at a fresh level. In order to make complete sense of such a rule, we have really to insist that the feature “3” is appropriate for the state names $p$. Quite often we will simply not mention this, it being implicit when a “3” arc is present.

The “in” index into the second level is treated as a pointer into the input tree. The effect of substituting the tree $c(a, b)$ for $x_1$ in $c((p, x_2), (q, x_1))$ is given by the path equation $in \cdot l \equiv 3 \cdot r \cdot in$, which also creates the new “in” arc pointing one level down in the input. Similarly, the copying of $b$ into the $x_2$ position is given by $in \cdot r \equiv 3 \cdot l \cdot in$. This gives the second part of Figure 7.

The bottom part of the figure is obtained by applying the rule shown (modified by the prefixing $r$ to both sides), which corresponds to the transducer transition $(q, c(x_1, x_2)) \rightarrow c((p, x_1), (s, x_2))$. You can now see how the output tree is grown on successive new 3D levels of the original 2D incomplete feature structure.

At the end of the transduction process, we obtain a 3D feature structure with re-entrancy, which does not correspond to an official 3D tree. However, recall that a feature structure is defined as a prefix-closed set of paths with a congruence relation identifying which paths are to be made equivalent. We obtain a tree from this feature structure by replacing this congruence relation by the identity relation. Once this is done, we obtain the output tree simply by taking the yield.

We have presented this example as if the feature logic rules were productions, but in fact it is only the last tree which counts as a model of the logic program associated with a tree transducer. Models of the logic program obtained are structures representing ordered pairs in the tree relation defined by the transducer. More specifically, if $t$ is a given input tree, $t^*$ is its representation as a feature structure, and $u_t$ is a feature tree (intuitively) derivable from $3 : q[in : t^*]$, after ignoring re-entrancies, then $(u_t/(in), y(u_t))$ is such an ordered pair. Here $y$ is the yield.

Notice that nonlinear transductions are accommodated by having $in$ pointers to the same place in the input. It is also fairly clear that the feature logic representation of transductions can model “transductions with lookahead”. The left side of a logic programming rule merely needs to be a tree with more than one 2D level. Our formalization does not treat lookahead, but doing so involves only minor changes to the rules.

We need to pay attention to one kind of transition of a tree transducer: that which corresponds in the string case to replacing a symbol by the null string. An example in the tree case might be $(q, \sigma(x_1, x_2)) \rightarrow (p, x_1)$, where no piece of output tree is produced, and one input tree is deleted. This transition would be expressed as

$$q[in : c[l : \perp, r : \perp]] \rightarrow [3 : p] \sqcup [3in \equiv in \cdot l].$$

This rule is what we want – the output of it will piece “3” arcs together without any actual output tree. If we tried to construct the output tree all on one level, the path equation we would construct for this rule would lead to circularity.

6.5. Modelling tree adjunctions. The feature logic program we build from a TAG produces in essence a TAG derivation tree, which then can be collapsed into the generated tree output of the TAG. Each adjunction adds a new level to the 3D FS, and because adjunctions can occur on top of derived trees, this leads to multiple 3D levels in the structure.

Again we simply give an example\(^4\). Consider the TAG in Example 3. We show how to model the substitutions of $\alpha_2$ into $\alpha_3$, and the subsequent adjunction of $\beta_3$ into the result. We repeat the figures here for easy reference.

\(^4\)The formal definition provides a logic program corresponding to a linear context-free tree grammar.
The process of substitution and adjunction is once again captured by logic programming rules. We consider substitution first. Although this is a special case of adjunction, it illustrates the mechanism. Consider the top part of Figure 10. The FS logic program first builds the corresponding to the initial tree $\alpha_3$; we simply denote this as the rule $\bot \rightarrow \alpha_3$. The node labeled $d$ in $\alpha_3$ is the substitution site (we ignore the ↓); at this site we employ the logic programming rule

$$d \rightarrow 3; [d[c:\text{the}]].$$
This has the effect of moving the little tree headed with \( d \) along the curvy arrow into the required place; the result is the left part of the FS at the bottom. Again we are implicitly assuming that “3” is an appropriate feature for the symbol \( d \).

Adjunction of \( \beta_3 \) into the resulting structure then takes place using the rule

\[
\nu[c: \bot] \rightarrow 3: \beta_3 \sqcup [c = 3rc]
\]

where again we write \( \beta_3 \) instead of its FS description, and assume that “3” is appropriate for \( n \). The result is the right half of the bottom of Figure 10. Notice that the path \( r \) from the root of \( \beta_3 \) to its foot node appears explicitly in the path equation. The structure \( \text{boy} \) has to be copied under the rightmost \( n \). This is done with a path equation, indicated with \( \equiv \).

We present a picture of the general FS adjunction mechanism in Figure 11. In the figure we assume that \( n \) is a nonterminal symbol which is a type appropriate for two features \( l \) and \( r \).
Returning to Figure 10, we still have a 3DFS with just two levels, but if we wanted a “pretty pretty boy” we could adjoin \( \beta \) one more time at either of two further adjunction points, marked by (i) and (ii) in the figure. The differing results of adjunction are presented in Figure 12.

The obligatory nature of the FL rules means that a “sorcerer’s apprentice” phenomenon must take place here; the given rules force the only model of the program to have the sentence the \((\text{pretty})\) as its yield\(^5\).

To deal with termination we let nonterminals \( n \) “decay” to terminals \( \omega \): we merely need to add the choice of \( n[3; n] \) to rules involving \( n \):

\[
n[l: \bot, r: \bot] \rightarrow \{ \omega[3; \beta] \sqcup [l = 3\pi l] \sqcup [r = 3\pi r], \omega[3; \omega] \}
\]

(recall that disjunctive choices must appear in clauses).

7. Formalization

In this section we make precise the translations of regular tree grammars, tree transducers and TAGS into feature logic programs, and characterize transductions and adjunction derivations in terms of models of the FL theories of the respective logic programs.

\(^5\)An idea due originally to Holly [4].
Instead of working with ranked alphabets with numerical “arities”, we anticipate our translations into feature logic and consider arguments to a symbol to be named rather than numbered. So, we fix a set $L$ of feature names (this can be infinite), and define an $L$-ranked alphabet to be a pair $(\Sigma, \text{ap})$, where $\Sigma$ is a finite set of symbols, and $\text{ap}$ is a relation between $\Sigma$ and finite subsets of $L$ (the empty set is allowed, to give 0-ary symbols). Typically $l$, $r$, and $c$ will be feature names. The term $\sigma(a, \sigma(a, b))$, for example, will be written $\sigma(l:a, r:\sigma(l:a, r:b))$.

To repeat, the reason that $\text{ap}$ is a relation is that quite often category symbols like “np” have variable numbers of successors. For example, “np” could have arity $np(l:a, r:b)$ and also $np(c:a)$. To indicate that $\{l, r\}$ is an arity of $\sigma$ we write $\sigma = \sigma(l, r)$. It will usually be that case that we can use $\{l, r\}$ and $\{c\}$ as typical nonempty appropriateness sets, writing $\sigma = \sigma(\emptyset)$ to indicate that $\sigma$ has no features appropriate.

The following definition repeats those given previously, but is the standard way to define trees with occurrences of variables at the leaves.
Definition 7.1. A tree over the $L$-ranked alphabet $\Sigma$ and auxiliary set of 0-ary variables $X$ is defined inductively:

- Each $x \in X$ is a (zero-ary) tree;
- $\sigma$ is a (zero-ary) tree if $\sigma = \sigma(\emptyset)$;
- If $\sigma = \sigma(l, r)$, then and $s$ and $u$ are trees, then $\sigma(l : s, r : u)$ is a tree.
- If $\sigma = \sigma(c)$ and $s$ is a tree then $\sigma(c : s)$ is a tree.

We write $T_{\Sigma}(X)$ as usual for the set of trees, suppressing $F$, and write $T_{\Sigma} = T_{\Sigma}(\emptyset)$.

Definition 7.2. The set of maximal paths $\Pi_m(t)$ of a tree $t \in T_{\Sigma}(X)$ can be defined inductively as follows:

- Any zero-ary tree has maximal path set $\{ \epsilon \}$;
- If $t = \sigma(l : s, r : u)$, then $\Pi_m(t) = l \cdot \Pi_m(s) \cup r \cdot \Pi_m(u)$;
- If $t = \sigma(c : s)$, then $\Pi_m(t) = c \cdot \Pi_m(s)$.

The set $\Pi(t)$ of paths of $t$ is the prefix-closure of $\Pi_m(t)$.

Definition 7.3. If $t$ is a tree and $\pi$ is a path of $t$, then by $t/\pi$ we mean the subtree of $t$ at path address $\pi$. Thus $t/\epsilon = t$, and if $t = \sigma(l : s, r : u)$, then $t/(l \pi) = s/\pi$ and $t/(r \pi) = s/\pi$. We denote the replacement of $t/\pi$ in $t$ by some new tree $t'$ as $t[\pi \leftarrow t']$, without giving the formal definition.

7.1. Regular tree grammars.

Definition 7.4. A regular tree grammar is a tuple $G = (N, \Sigma, P, S)$, where

- $N$ is a set of nonterminal symbols;
- $\Sigma$ is an $L$-ranked alphabet of terminal symbols;
- $P$ is a finite set of productions, each of the form $A \rightarrow t$, where $A \in N$ and $t \in T_{\Sigma}(N)$;
- $S \in N$ is the start symbol.

Definition 7.5. The rewriting relation $\vdash$ defined by a RTG is a subset of $T_{\Sigma}(N) \times T_{\Sigma}(N)$. We say that $u \vdash v$ if there is a rule $A \rightarrow t$ in $P$ such that $u/\pi = A$ for some maximal path $\pi$ of $u$, and $v = u[\pi \leftarrow t]$. The tree language $L(G)$ of $G$ is $\{ t \in T_{\Sigma} : S \vdash^* t \}$.

We start with an RTG, and define its tree language in a non-standard way by using derivation trees. Such a tree is defined as follows. Let $A \in N$. Change the appropriateness set of $A$ so that it has one optional argument called the “3” argument.

Definition 7.6 (RTG derivation tree). We give a simultaneous inductive definition.

- Any nonterminal $A$ is a derivation tree headed by $A$;
- If $u$ is a derivation tree and we form $u'$ by replacing an occurrence of $B$ at a leaf by $B(3 : v)$, where $B \rightarrow v$ is a production, then $u'$ is a derivation tree.
- These are the only derivation trees.

If for every $B$ label occurring on a leaf, there is no production of the form $B \rightarrow t$ in $G$, then the derivation tree is said to be full.

Example 7.7. Let $G$ be the RTG

$$S \rightarrow \sigma(l : T, r : S) \mid \tau(c : a)$$

$$T \rightarrow b \mid d.$$ 

The following is a derivation tree for $G$:
We have displayed this both as a tree and as a term, omitting the feature names in the term. It is not full since the lowest $S$ can be expanded.

Notice that derivation trees simply record which nonterminals were expanded during the course of a derivation, wherever a “3” arc is found.

We translate derivation trees $u$ as expected into 3D feature trees, which we again call $u$. The typing is as follows. Every symbol in $N \cup \Sigma$ will be a type. We use the flat ordering; all symbols incomparable. The appropriateness for $\sigma$ will be the same as it is in the RTG, and the feature 3 will be appropriate for each $A \in N$.

**Lemma 7.8.** $u$ is a derivation $FS$ of $G$ if and only if for every nonmaximal path $\alpha$ of $u$ labeled with $B$, $u/\alpha \sqsupseteq 3:t$ for a rule $B \rightarrow t$ of $G$.

**Proof.** ($\Rightarrow$) We use induction on the formation of $u$. The case $u = A$ is vacuous. If we form $u'$ by replacing an occurrence of $B$ at a leaf by $B(3:v)$, where $B \rightarrow v$ is a production, then the inductive hypothesis gives us what we need for all paths except the ones leading to the given occurrence of $B$ and then into $v$. Any paths leading into $v$ and ending with a terminal are maximal, and the path ending in $v$ satisfies our conclusion.

($\Leftarrow$) Suppose the condition holds for the feature tree $u$. We show how to reconstruct $u$ as a derivation tree. We may start with $A$, the head of $u$ as a feature tree, as the initial step. Suppose now we have reconstructed a $u' \subseteq u$. If $u' \neq u$, then there is some leaf node of $u'$ labeled with a nonterminal $B$, the path $\alpha$ to which is a nonmaximal path of $u$ as a feature tree. By the condition, there is a rule $B \rightarrow t$ of $G$ such that $u/\alpha \sqsupseteq 3:t$. This means we can extend $u'$ at $B$ as demanded by the derivation tree definition and still have a new tree subsumed by $u$. \hfill \Box

**Remark 7.9.** We could use Lemma 7.8 as the primary definition of derivation trees. The condition given by the lemma applies to both finite and infinite feature trees. We will employ this style of definition, which is essentially coinductive, for our other grammar formalisms.

**Definition 7.10.** We use the recursive “transducer” definition of yield for any finite derivation tree:

- $y(A) = A$;
- $y(\sigma(l:t_l,r:t_r)) = \sigma(l:y(t_l),r:y(t_r))$;
- $y(A(t_3)) = y(t_3)$.
For the derivation tree in the example, the yield is \( \sigma(b, \sigma(d, S)) \). The yield function in the case of RTGs simply removes the “3” arcs (and the nonterminals preceding them) from a derivation tree.

**Lemma 7.11.** If \( u \) is a finite derivation tree with root labeled \( A \), then for some \( n \), \( A \vdash^n y(u) \).

**Proof.** By induction on the number of “3” arcs in \( u \). If there are none, then \( u \) is just a nonterminal \( A \). Assume the result for \( n \) and consider a tree \( u' \) with \( n + 1 \) “3” arcs. Then the tree must have been formed by replacing a leaf \( B \) of a derivation tree \( u \) by \( B(3 : v) \) for some production \( B \rightarrow v \). Remove this subtree and replace it with just \( B \), forming a derivation tree \( u \). Then by induction \( A \vdash^* y(u) \). But \( B \) is still a leaf of \( y(u) \). We also have that \( y(u') = y(u)[B \leftarrow v] \) (this needs another induction using the definition of \( y \)). So \( A \vdash^* y(u') \). □

We assert without proof that if \( A \vdash^* t \), there is a derivation tree \( u \) with \( t = y(u) \). (This is straightforward by induction.)

Let \( G \) be a regular tree grammar. We define a FLP \( P(G) \) as follows. For each nonterminal \( B \), form the programming rule set

\[
B \rightarrow \{3 : t \mid B \rightarrow t \text{ is a production of } G \}.
\]

Close these rules under prefix extension; then add the rule \( \bot \rightarrow \{S\} \).

We could now relate \( n \)-step tree derivations with proofs using the \( U_P \) rule; this would involve defining a productive proof as one which at each step derived a new clause not entailed by previously derived clauses. However, it is much more elegant to appeal to Definition 4.13 and Theorems 4.14 and 4.15.

**Theorem 7.12.** A tree is a finite acyclic model of \( P(G) \) if and only if it is a finite full derivation tree of \( G \) headed by \( S \).

**Proof.** For any \( u, u \) is a model of \( P(G) \) if and only if for every path \( \alpha \) of \( u \) labeled with \( B \) for which there is a rule \( B \rightarrow L \) of \( P(G) \), \( u/\alpha \supseteq 3 : t \) for some \( B \rightarrow t \) of \( G \). Moreover, the label of the root is \( S \). (These assertions are just restatements of the definition of being a model of \( P \).) If \( u \) is finite and acyclic, this is equivalent, by Lemma 7.8, to being a full derivation tree of \( G \), because at the leaves of \( u \), there cannot be applicable rules of \( P \). □

**Remark 7.13.** If a \( B \) occurs on some leaf of a full model \( u \), then since \( B \) has the feature 3 appropriate, there are feature structures properly subsuming \( u \). But such structures cannot be models of \( P(G) \), because there are no programming rules sanctioning the extension of a path through \( B \).

**Corollary 7.14.** The finite, acyclic models of \( \text{Th}(P(G)) \) are exactly the finite full derivation trees.

**Proof.** By the theorem above, the full derivation trees are the models of \( P(G) \). By Theorem 4.15, a full derivation tree, which is a model of \( P(G) \), is a model of \( \text{Th}(P) \). Going the other way, we know from Theorem 4.14 that a minimal model of \( P(G) \), hence a minimal full derivation tree, is a model of the theory of \( P \). We have to show that full derivation trees are the only models of \( P(G) \). But a full derivation tree cannot, by definition, be properly extended to another derivation tree. So a minimal model is the only model. □

A ground model of \( P(G) \) is an acyclic model of \( P(G) \) in which every path has maximal type.

**Corollary 7.15.** \( L(G) = \{ y(u) \mid u \text{ is a finite ground model of Th}(P(G)). \} \)
7.2. Tree Transducers. Our objective is now to present the feature logic program \( P(TT) \) for a transducer TT. Once this is done, we have to say what the models of the associated theory represent in terms of the original TT.

**Definition 7.16.** As in the case of RTG, we modify the definition of top-down tree transducers. We start with an \( L \)-ranked alphabet \( \Sigma \) (and another one \( \Delta \)). Instead of nonterminals \( N \) we have a set \( X \) of \( L \)-named variables, say \( \{x_1, x_r, x_c\} \). Then an \( L \)-ranked tree transducer is a tuple \( M = (Q, q_0, \Sigma, \Delta, \rightarrow) \), where

- \( Q \) is a finite set of states;
- \( q_0 \in Q \) is the initial state;
- \( \Sigma \) is an \( L \)-ranked input alphabet;
- \( \Delta \) is an \( L \)-ranked output alphabet;
- \( \rightarrow \) is a finite transition relation between \( Q \times \Sigma_X \) and \( T_\Delta(Q \times X) \), where \( \sigma(x_1, x_r) \) is a typical element of \( \Sigma_X \) provided \( \sigma = \sigma(l, r) \). (We allow transitions like \( (p, \sigma(x_1, x_r) \rightarrow (q, x_l)) \).

**Example 7.17.** Recall Example \([5.4]\):

\[
\begin{align*}
(q, c(x_1, x_2)) & \rightarrow c((p, x_2), (q, x_1)) \\
(q, c(x_1, x_2)) & \rightarrow c((p, x_1), (s, x_2)) \\
(p, b) & \rightarrow a \\
(p, a) & \rightarrow a \\
(s, b) & \rightarrow b
\end{align*}
\]

We rephrase this as a ranked tree transducer. The alphabets remain the same, and we add two feature names \( l \) and \( r \). The symbol \( c = c(l, r) \). The transducer becomes

\[
\begin{align*}
(q, c(x_1, x_r)) & \rightarrow c(l: (q, x_r, r: (q, x_l)) \\
(q, c(x_1, x_2)) & \rightarrow c(l: (p, x_l), r: (s, x_r)) \\
(p, b) & \rightarrow a \\
(p, a) & \rightarrow a \\
(s, b) & \rightarrow b
\end{align*}
\]

Instead of defining ranked transducer rewriting, we go immediately to the definition of a ranked transducer derivation tree. Such a tree has been inserted in a copy of some \( L \)-input tree \( u \), occurring as the initial “in” subtree. This is the tree on the left side of the examples shown in Figure 7. On the right side, we have what corresponds to a derivation tree of an RTG. Corresponding to the bottom third of Figure 7 we have the \( L \)-ranked tree shown in Figure \([13]\).

To specify derivation trees, we add an appropriateness specification \( p(in, 3) \) to state symbols \( p \).

**Definition 7.18.** Given an \( L \)-ranked tree transducer \( M \), and a ranked input tree \( u \) over \( \Sigma \), a **derivation tree** \( d(u) \) for \( M \) is any tree satisfying the following conditions.

- The root of \( d(u) \) is labeled with the initial state \( q_0 \).
- \( d(u)/(in) = u \).
- For every path \( \alpha \) of \( d(u) \) labeled with \( q \), which has successors \( \alpha 3 \) and \( \alpha \cdot in \) such that \( \tau(\alpha \cdot in) = \sigma \), there is a transition \( (q, \sigma) \rightarrow v \) of \( M \) such that (i) for every path \( \pi \) of \( v \) leading to \( (q, x_i) \), we have \( d(u)/(\alpha 3 \pi in) = d(u)/(\alpha \cdot in \cdot i) \) (for \( i \) a feature name), and (ii) for every path \( \rho \) of \( v \), the label of \( \alpha 3 \rho \) in \( d(u) \) is the same as the label of \( \rho \) in \( v \); moreover, the 3 and “in”-free paths \( \rho \) such that \( \alpha 3 \rho \in d(u) \) are the same as in \( v \).
A derivation tree $d(u)$ is said to be **full** if no leaf node can be extended in the sense that there is a path $\alpha$ to a state $p$ for which a rule of $M$ could be applicable to produce another derivation tree. A derivation tree is **ground** if it has only terminals on its leaves.

**Definition 7.19** (Derivation FS). We define a derivation feature structure exactly as we define a derivation tree, replacing "=" with $\equiv$ for path equations.

**Example 7.20.** Let $d$ be as in Figure 13. The corresponding feature structure is shown in Figure 14. In the figure, we label re-entrancies with $\equiv$, reflecting the definition of $d^*(t)$ as an abstract feature structure.

**Definition 7.21** (Logic program associated to an RTT). We use the feature names as in the definition of derivation tree, and again use the flat ordering on symbols as types, with the same appropriateness. Corresponding to a TT $M$ and input tree $u$ we define the logic program $P(M, u)$.

- The initial rule is $\bot \rightarrow q[in : u^*]$.
- Let $(q, \sigma(x_l, x_r))$ be a left-hand side of a rule. Form the set of all right-hand sides
  $L = \{v \mid (q, \sigma) \rightarrow v \text{ is a transition of } M\}$

Add the LP rule

$$q[in : \sigma[l : \bot, r : \bot]] \rightarrow \{v^* \mid v \in L\}$$

to $P(M, u)$, where $v^*$ is a feature structure whose paths are $\{3\pi \mid \pi \in v\}$, with the typing inherited from $v$ (in other words, the ordinary FS translation of $v$, with states labeling the leaves, and variables omitted.

- If $\pi$ is any of the paths in such a $v$, leading to a state $(s, x_j)$, then unify the path equation $inj = 3\pi in$, where $j$ is either $l$ or $r$ with $v^*$. 

![Figure 13. Ranked transducer derivation tree.](image-url)
• The full logic program is now generated by the prefixing scheme applied to all rules other than the initial one.

**Example 7.22.** Here is the FL program corresponding to our running example of a tree transducer. Let the input tree \( u \) be the one shown in the figures.

\[
\begin{align*}
\bot &\rightarrow q[\text{in}: u^*] \\
q[\text{in}: c[l: \bot, r: \bot]] &\rightarrow \{3: c[l: p, r: q] \sqcup [\text{in} \cdot l \equiv 3 \cdot l \cdot \text{in}] \sqcup [\text{in} \cdot r \equiv 3 \cdot r \cdot \text{in}], \\
3: c[l: p, r: s] &\sqcup [\text{in} \cdot l \equiv 3 \text{in}] \sqcup [\text{in} \cdot r \equiv r \cdot \text{in}] \\
p[\text{in}: b] &\rightarrow 3: a \\
p[\text{in}: a] &\rightarrow 3: a \\
s[\text{in}: b] &\rightarrow 3: b
\end{align*}
\]

Once again, close under prefix-extension of all rules but the initial one.

After all of these definitions, the following theorem is expected.

**Theorem 7.23.** The models of the logic program \( P(M, u) \) are exactly the feature structures derived from (full) derivation trees \( d(u) \). As a consequence the ground feature structures \( y(d^*(u)) \) correspond to \( v \) such that \((u, v)\) is in the tree relation defined by \( M \).

**Proof.** We have only to check that anywhere in a derivation \( fs \) where the left side of a LP rule applies, one of the right sides applies there also; conversely, that any \( fs \) satisfying all such rules is in fact a derivation \( fs \).

Let us carry out the second of these tasks in a bit more detail. Let \( h \) be a FS which is a model of \( P(M, u) \). We verify that \( h \) is a derivation FS.
(1) Clearly the root of $h$ has type $q_0$, and $h/\in = u^\ast$. This part of $h$ cannot be more specific because it is all fully typed.

(2) Consider all the paths $\alpha h$ that are subsumed by the left side of a FLP rule. Because $h$ is a model of $P$, each such $\alpha$ must have successor paths in $h$ which represent the right side of some rule; moreover, $\alpha 3\pi \in = u^\ast$ is a path equation satisfied in $h$. This all implies that $h$ is subsumed by a derivation $FS$; this follows from the conditions in Definition 7.18 and its FS version. But such a derivation $FS$ is a maximal one subsuming $h$, because all of its symbols are inconsistent. In particular, this implies that the 3 and in-free paths of $h$ after any $\alpha 3$ are exactly the ones contained in the right side of some rule. □

7.3. Tree adjoining grammars. We now turn to the formal translation of TAGs into feature logic. As in [10] we want to work with ranked TAGs, and in fact with $L$-ranked TAGs, to facilitate the translation. The actual formal system is a linear context-free tree grammar.

Definition 7.24. An $L$-ranked linear context-free tree grammar is a tuple $G = (N, \Sigma, P, S)$, where

- $N$ is a nonterminal $L$-ranked alphabet, and $\Sigma$ is a terminal $L$-ranked alphabet;
- $P$ is a finite set of productions of the form $A \rightarrow v$, where $A \in N$, and if $A = A(l, r)$ then $v \in T_{N\cup \Sigma}(\{x_l, x_r\});$
- (Linearity) If $A \rightarrow v$ is a production, then there are no two paths in $v$ to the same variable.
- $S \in N$ is the start symbol.

Definition 7.25. The one-step rewriting relation $\vdash$ associated with $G$ is the following. Let $u$ be a ranked tree over the mixed alphabet. Suppose $\rho$ is a path of $u$ with $u/\rho = A(l:u_l, r:u_r)$. If $A \rightarrow v$ is a production, first let $\pi_i$ be a path of $v$ with $v/\pi = x_i$. Form the tree $v = v/[[\pi_i \leftarrow u_i]]$, for all such paths $\pi_i$. Then let $u' = u[\rho \leftarrow v]$. We say that $u \vdash u'$.

Definition 7.26. The tree language $L(G)$ is the set $\{v \in T_\Sigma(\emptyset) \mid S \vdash^* v\}$.

Example 7.27. We start with an example of a LCFTG. It is the TAG fragment treated in Section 6. In the example we have omitted the feature names for ease of reading.

- The nonterminal alphabet $N$ is $\{s, n, d\}$.
- The terminal alphabet is $\{np, adj, d, n, boy, left, pretty, the\}$.
- The productions are

\[
\begin{align*}
  s & \rightarrow d \mid np(d, n(boy)) \\
  d & \rightarrow d(\text{the}) \\
  n(x) & \rightarrow n(adj(\text{pretty}), n(x)) \mid n(x) \\
  n(x, y) & \rightarrow n(adj(\text{pretty}), n(x, y)) \mid n(x, y)
\end{align*}
\]

The redundancy in the last two lines could be eliminated with a convention about the polymorphic nature of $n$; it is either unary or binary.
The following is a derivation in this grammar.

\[
\begin{align*}
s & \vdash np(d, n(\text{boy})) \\
& \vdash np(d(\text{the}), n(\text{boy})) \\
& \vdash np(d(\text{the}), n(\text{adj}(\text{pretty}), n(\text{boy}))) \\
& \vdash np(d(\text{the}), n(\text{adj}(\text{pretty}), n(\text{boy})))) \\
& \vdash np(d(\text{the}), n(\text{adj}(\text{pretty}), n(\text{adj}(\text{pretty}), n(\text{boy}))))))
\end{align*}
\]

**Definition 7.28.** Let \( G \) be an \( L \)-ranked LCFTG. **Derivation trees** are described as trees over a new ranked alphabet in which “3” is allowed as a feature name. For each nonterminal \( A \), replace every appropriate set \( \{x_l, x_r\} \) with \( \{x_l, x_r, x_3\} \). A derivation tree is any tree satisfying the following conditions

- A derivation tree must be of the form \( A(\cdot, t_3) \) for some nonterminal \( A \). It is possible for \( t_3 \) to be null: a one-node tree with type \( \perp \).
- For any subtree of the form \( A(t_l, t_r, t_3) \), where \( t_3 \) is not null, there is a rule \( A \rightarrow v \) of \( G \) such that \( t_3 = v[x_l ← t_l, x_r ← t_r] \).

A derivation tree is full if it has no occurrence of a nonterminal which is not expanded by a rule as above.

This definition is very like the TT and RTG definitions, but it differs subtly and substantially. Namely, the trees \( t_3 \) (moving up a dimension) had the property that the trees \( t_l \) and \( t_r \) copied into \( v \) did not have 3-subtrees (3-arcs in the feature version.) The equations (to be induced in the feature version) assert the identity of derivation subtrees, not the identity of input subtrees as in the TT case. We must therefore convince ourselves that a finite LCFTG derivation tree specifies a derivation, and conversely; moreover, that the yield is in fact a tree generated by the grammar. Once this is done, we will be able to describe these trees naturally in feature logic. We start with an example of a derivation tree.

**Example 7.29.** The following tree corresponds to the above LCFTG derivation.

```
                                      s
                                     /   |
                                    np   |
                                   /    |
                                  d    adj
                                 /     |
                                d  the  n
                               /  \  |
                              boy adj  pretty
                             /  \  |
                            n   adj  boy
                           /  \     |
                          n   n  pretty  boy
```

We have indicated the extra argument as (3). Notice how this tree is beginning to recreate our example in Section 6.

We continue to use the homomorphic definition of yield for finite trees as in the RTG case.

**Proposition 7.30.** If in the original grammar, \( A(t_l, t_r) \vdash^n u \), then there is a derivation tree of the form \( A(t_l, t_r, t_3) \) such that \( y(t_3) = u \).
Remark 7.31. By $A(t_l, t_r)$ above, we mean to illustrate a typical appropriateness condition. In the way we have written it, this derivation tree would have two “substituends” $t_l$ and $t_r$. But there might be no substituends, in which case the tree would just be $A(t_3)$.

Proof. By a result of Kepser and Mönich [6], we may assume the derivation is outside-in, which means that no nonterminals are replaced occurring “in the scope of” other nonterminals. The proof is then by induction on $n$. If $n = 1$ then there is a production of $G$ of the form $A \rightarrow v$, such that $u = v[x_l \leftarrow t_l, x_r \leftarrow t_r]$. The yield of $u$ is just $u$. Thus $A(u') = A(u)$.

Assume the proposition for $n$ and let $A(t_l, t_r) \vdash^{n+1} u$. the first step being $A \vdash v[x_l \leftarrow t_l, x_r \leftarrow t_r]$, for some rule $A \rightarrow v$. We may represent the situation as in Figure 15. The derivation can be continued at the outermost occurrences of nonterminals. By inductive hypothesis, the trees at $B$ and $C$ (all the outermost places a derivation step can take place) have corresponding derivation trees as shown in the bottom half of the figure. Replacing the subtrees headed by $B$ and $C$ in $v$ by their corresponding derivation trees gives a new tree $d(v)$, and forming the derivation tree $A(t_l, t_r, d(v))$ gives us our desired conclusion, because the yield of this tree is the yield of $d(v)$. \hfill $\Box$

Proposition 7.32. If $A(t_l, t_r, t_3)$ is a derivation tree, then $A(t_l, t_r) \vdash^* y(A(t_l, t_r, t_3))$. 

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{figure15}
\caption{The inductive step in Proposition 7.30}
\end{figure}
Proof. By induction on the number of 3-subtrees of $A(t_1, t_r, t_3)$. Suppose there is just one. Then by definition of derivation tree there is a rule $A \rightarrow v$ of $G$ such that $t_3 = v[x_l \leftarrow t_1, x_r \leftarrow t_r]$. This tree has no 3-subtrees, so its yield is itself, and plainly $A(t_1, t_r) \vdash t_3 = y(t_3)$.

Suppose the lemma is true for all derivation trees with at most $k$ 3-subtrees. Consider $A(t_1, t_r, t_3)$, where there are $k + 1$ 3-subtrees. We need to show $A(t_1, t_r) \vdash y(t_3)$. By the definition of derivation tree, we have a rule $A \rightarrow v$ of $G$ such that $t_3 = v[x_l \leftarrow t_1, x_r \leftarrow t_r]$. Since the tree grammar is linear, there are at most $k$ 3-subtrees of $t_3$.

Now in $t_3$ find all the topmost (outermost) occurrences of $L(u_l, u_r, u_3)$. (Refer to Figure 16.) These are subtrees to which the inductive hypothesis applies. So for each occurrence, $L(u_l, u_r) \vdash^* y(u_3)$. By copying $y(u_3)$ into the spot occupied by $L$, we obtain a new tree $t'_3$ such that $t'_3 = y(t_3)$. But then $A(t_1, t_r) \vdash^* t'_3$. □

These propositions justify the correctness of the feature logic program associated to a given LCFTG, which we now define. Let $G = (N, \Sigma, P, S)$. We let the symbols in $N \cup \Sigma$ be types, where the appropriateness sets for $N$ and $\Sigma$ obtain from the definition of $G$-derivation trees. As before, use the flat ordering on $N \cup \Sigma$. The feature names are the same as they are in $G$, with the addition of the feature name “3.”

Definition 7.33. We define a FLP $P(G)$ by giving its basic rules. All other rules are obtained by prefix-extension of everything but the initial rule.

- $\bot \rightarrow \{S\}$;
- For each nonterminal $A$ with arity \{l, r\} add the clause
  \[
  A[l; \bot, r; \bot] \rightarrow \{3; v^* \sqcup (l \doteq 3\lambda) \sqcup (r \doteq 3\rho r) \mid (A \rightarrow v) \text{ is a rule of } G\}
  \]
  in which \(\lambda\) (resp. \(\rho\)) is the path of \(v\) leading to \(x_l\) (resp. \(x_r\)), if there is one, and \(v^*\) is the standard representation of \(v\) as a feature structure.
The program $P(G)$ is constructed so that its models are the FS versions of $G$-derivation trees headed by
the single node $S$, here defined to have arity $\emptyset$.

**Example 7.34.** This is the program corresponding to our small TAG in Example 7.27
\[
\begin{align*}
    \bot & \rightarrow \{s\} \\
    s & \rightarrow \{3:d,3:n[p:l:d, r:n[c:boy]]\} \\
    d & \rightarrow \{3:d[c:the]\} \\
    n[c:\bot] & \rightarrow \{3:n[l:\underline{adj}[c:pretty], r:n[c:\bot]] \sqcup (c \doteq 3rc), 3:n[c:\bot] \sqcup (c \doteq 3c)\} \\
    n[l:\bot, r:\bot] & \rightarrow \{3:n[l:\underline{adj}[pretty], r:n[l:\bot, r:\bot]] \sqcup (l \doteq 3rl) \sqcup (r \doteq 3rr), 3:n[l:\bot, r:\bot] \sqcup (l \doteq 3rl) \sqcup (r \doteq 3rr)\}
\end{align*}
\]
One small point: all nonterminal symbols $n, d, s$ have an implicit appropriateness set allowing the feature $3$. This is because these nonterminals have an implicit appropriateness set allowing the feature $3$.

Once again, we have results identifying the acyclic models of $P(G)$, and again we identify trees and feature structures in the finite case.

**Proposition 7.35.** Let $f$ be a finite acyclic model of $P(G)$ without the initial rule. Then $f$ (regarded as a tree) is a full derivation tree of $G$.

*Proof.* By looking at the program, we see that any model of $P$ must be a tree with an initial 3-subtree.

If $f$ is a model of the program, we know by prefix-extension that for any path $\alpha$ of $f$ such that $\tau(\alpha) = B$ and such that $\alpha 3$ is a path of $f$, there is a clause $B \rightarrow L$ of $P$ such that one of the elements $v$ of $L$ subsumes $f/(\alpha 3)$. Looking at $f/(\alpha 3)$, we see that it must have at least the same 3-free paths as some $v$ such that $B \rightarrow v$ is a production of $G$. Call these the $v$-paths. Then the type assigned to a $v$-path $\rho$ after $\alpha 3$ in $f$ must be the same type that $\rho$ had in $v$.

Let $\lambda$ be a $v$-path leading to an occurrence of $x_l$ in $v$. Then by the path equation $l \doteq 3\lambda l$ in the given clause, we see that $f/(\alpha l) = f/(\alpha 3\lambda l)$. This says that the subtree replacement condition in the definition of derivation tree holds.

Since we have now accounted for all the conditions demanded by the derivation tree definition, we know that $f$ is subsumed by a derivation tree. But since every expandable nonterminal in $f$ must be expanded by the obligatory nature of the rules in a logic program, then $f$ is subsumed by a full derivation tree. Since such trees are maximal, $f$ itself is a full derivation tree.

We assert that a full derivation tree is a model of $P(G)$ without proof. We obtain

**Theorem 7.36.** The full derivation trees are exactly the models of $Th(P(G))$.

As before, we then have that $L(G)$ consists of the yields of finite acyclic ground models of $P(G)$.

8. Undecidability

This short section contains a proof of undecidability of satisfiability for general feature logic programs. We consider the class of feature logic programs generated by a finite number of rules $f \rightarrow L$, some of which are designated to be subject to the prefix extension scheme. The question is whether or not such a program has a model.

**Theorem 8.1.** The satisfiability problem for feature logic programs is undecidable.
Proof. This is a minor adaptation of the proof by Blackburn and Spaan [2] that satisfiability of formulas, in a certain version of the original Kasper-Rounds feature logic, is undecidable. They (and we) use the undecidability of the tiling problem for $\mathbb{N} \times \mathbb{N}$. This is the following problem. We are given a finite set of $n$ tile types, each of which is a $1 \times 1$ square $T$ which has colors on each side. (Tiles have fixed orientation.) Tiles are placed with their center on a natural number lattice point $(i, j)$. Assume that you have infinitely many actual tiles of each type. Then the question is: can you place a tile at each lattice point in such a way that adjacent tiles have the same color on their common edge?

We reduce the tiling problem to the FLP satisfiability problem. For each tile type $T_i$, $1 \leq i \leq n$, we let $\text{left}(T_i)$, $\text{right}(T_i)$, $\text{up}(T_i)$, and $\text{down}(T_i)$ be the colors of the respective sides of $T$. For each $i$ let $\tau_i$ be a type, and make $\tau_i$ inconsistent with $\tau_j$ for any $j \neq i$. If $\tau_i$ is assigned to a path $\alpha$ this will indicate that at this point in a satisfying model (which represents a 2D grid of lattice points) the tile $T_i$ is placed. For feature names we use $u$ (for “move up a step”) and $r$ (for “move right a step”). The infinite grid is represented by the FL rule

$$\bot \rightarrow \{ (ur = ru) \}.$$ 

All of the next rules specify a proper tile placement.

1. The first rule says that there is at least one tile at each node:

$$\bot \rightarrow \{ \tau_1, \ldots, \tau_n \}.$$ 

Because $\tau_i$ and $\tau_j$ are inconsistent, we do not need a rule saying that there is at most one type at a node.

2. We now need to say that if $T_i$ is a tile at some node, then at a node to the right, there must be a tile whose left face matches the right face of $T_i$. This gives rise to $n$ “horizontal” rules

$$\tau_i \rightarrow \{ r : \tau_j \mid \text{left}(T_j) = \text{right}(T_i) \}$$

3. Similarly for nodes above: we have the “vertical” rules

$$\tau_i \rightarrow \{ u : \tau_j \mid \text{down}(T_j) = \text{up}(T_i) \}.$$ 

4. Close all rules under prefix extension.

It is clear that there is a satisfying model for the program consisting of all the above rules iff there is a tiling of $\mathbb{N} \times \mathbb{N}$ with tiles from the given set of tile types. \hfill $\square$

What does this result tell us about our grammar models? In a way it is beside the point, because those logic programs are all consistent. It does tell us that we cannot translate general feature logic programs into MSO. So the problem is really to isolate the properties of FL programs we need to assure us that a translation would be possible. There is some hope for this, because at bottom both systems are logics of trees.

REFERENCES


CSE DIVISION, UNIVERSITY OF MICHIGAN, ANN ARBOR

E-mail address: rounds@umich.edu