Features Moving Madly: A formal perspective on feature percolation in the minimalist program*

Gregory M. Kobele

June 13, 2005

Abstract

I show that adding a mechanism of feature percolation (via specifier head agreement) to Minimalist Grammars (MGs) [Stabler, 1997] takes them out of the class of context-sensitive grammar formalisms. The main theorem of the paper is that adding a mechanism of feature percolation to MGs allows them to implement infinite abaci [Lambek, 1961], which can simulate any Turing Machine computation. As a simple corollary, I show that, for any computable $f : \mathbb{N} \rightarrow \mathbb{N}$, MGs thus enhanced can generate the language $L_{a^{f(n)}} = \{a^{f(n)} : n \in \mathbb{N}\}$.

1 Features

Complex categories are ubiquitous in linguistic theory. From the attribute value matrices of HPSG to the algebraic types in categorial grammars, we have found it incredibly useful to structure the once simplex distributional information represented by categories. Amongst the offspring of the Government and Binding family, categories are viewed as feature bundles, a notion rarely made precise. There seem to be two intuitions about what this means, as evidenced by the kinds of operations that deal with features. These will

*This paper will appear in **Research in Language and Computation** in a slightly modified form. Citations should, when necessary, be made to the official version of this paper. Comments are welcome!

be called here the 'features-as-properties' view, and the 'features-as-building-blocks' view. The latter view is suggested by operations involving features such as checking, deletion, and especially movement, and is a natural reading of [Chomsky, 1995, Frampton and Gutmann, 2000, Koopman, 1996, Koopman and Szabolcsi, 2000]. The features-as-properties view can be found in later Chomsky [2001], where dependencies are established between elements by a long-distance operation of AGREE (the abandonment of establishing syntactic relations under a condition of immediate locality was motivated in part by participial agreement facts from Icelandic). In the AGREE-based theory, operations on features like those described above are exchanged for the single operation of identity checking (called matching, or valuation). In order to achieve the level of precision which is an indispensable aid in determining the predictions of the theory, we need to decide how to formalize the role features play in the Chomskyian frameworks. It is not immediately obvious that the there is any real difference between viewing features as properties of expressions, and viewing features as pieces of expressions - linguists have long made use of the equivalence between elements of a set and the set of properties shared by a particular element in moving between individuals to their type-raised counterparts in a Montagovian semantic theory. The two intuitions seem to diverge in their answer to the question of whether it is possible for a feature bundle to have multiple instances of the same feature type (and if so, whether this is to be distinguished from having a single instance). So is there any reason to think that having the same feature twice is different from having it just once? The answer to this is of course bound up with both the theory of morphology and the morphology-syntax interaction one adopts. The checking theory advanced by Chomsky [1993] provides us with an approach which seems committed to this possibility. Recall that according to the checking theory, words are built up in the lexicon, and are the building blocks of syntactic derivations. A derivation proceeds by checking off features of words against functional (and thus, phonetically empty) heads. The relation between morphology and syntax is mediated solely by the featural component of the words built up in the lexicon. Adopting this perspective for the moment, notice that in a language like Bolivian Quechua with morphological causatives, a word with two causative morphemes is interpreted differently than a word with just one:[1]

[1]Examples are from Stabler [1994]

(1)

Riku	*-chi*	*-ni*
see	make	1S

‘I show it’ or ‘I make him see it’

(2)

Riku	*-chi*	*-chi*	*-ni*
see	make	make	1S

‘I have it shown’

The sentences (or words) above differ syntactically as well - the ‘doubly causativized’ form in (2) licenses (up to) four arguments in the clause in which it occurs (the first causer, the second causer, the underlying subject, and the object), the form in (1) only three. As we are for the time being strict-lexicalists, and believe that the construction of both words above proceeds independently of the machinations of the syntactic component of the grammar, we need to ask: how is the number of heads introducing causers (possibly v) enforced (i.e. why do we not see sentences with four arguments using the form in (1))? One option is to allow the syntax to overgenerate (i.e. to decide that causative heads do not have features which need to be checked off in the course of a derivation, and thus can be freely added with no appreciable syntactic consequences), and for the offending predictions to be excised from the theory at an interface (one natural account might have it that the derivations which mismatch arguments with the valency of the verb won’t be interpretable, given reasonable assumptions about the semantics). An option more in line with the idea that interface conditions do not act as filters on derivations [Frampton and Gutmann, 2002] is to allow mutiple ‘causer’ features to be hosted by the verb in (2), each of which licenses up to one argument introducing projection.

Allowing feature bundles to have multiple token instances of a single feature type brings the Chomskyian perspective on grammar into line with ideas on resource sensitivity in logic [Girard, 1987]. Viewing features as resources in a minimalist setting has been investigated in other works [Michaelis, 1998, 2001, Harkema, 2001, Kobele, 2002],[2] as has the relation between minimalist

[2]This paper also addresses a problem left open by Kobele [2002]. There a formalization of Mirror Theory [Brody, 2000] was presented, which utilized a sort of feature percolation. Placing a restriction on the formalism’s feature percolation allowed a proof of the restricted formalism’s expressive equivalence with Minimalist Grammars. The result in this paper extends straightforwardly to the unrestricted Mirror Theory formalism. Given a program $p = \langle p_1, \ldots, p_n \rangle$, in which k registers are referenced, we construct a mirror theoretic lexicon

grammars and resource sensitive logics [Lecomte and Retoré, 1999, Retoré and Stabler, 2004, Vermaat, 2004]. In the remainder of this paper I study the results of adding the widely used mechanism of feature percolation when features are viewed as resources.

1.1 Feature Percolation

It has been well-known since Ross [1967] that sometimes the phrase which we analyze as having undergone movement properly includes the elements which we think of as introducing the motivation for this movement. Consider the English prenominal genitive, which is traditionally analyzed with the genitive marked phrase a dependent of the noun so-modified (i.e. in the specifier of said noun's phrase). A wh-word, an introducer of features requiring movement, as a genitive marked specifier, is not itself allowed to move to check its features (4), but instead requires 'pied-piping' of the largest noun-phrase it is a specifier (of a specifier ...) of (5).

(3) I saw Mary's brother's dentist's college roommate.

(4) *Who(se) did you see ('s) brother's dentist's college roommate?

(and hence the grammar) as follows. For each instruction p_m in p, add the following lexical items:

p_m is:	lexical items:
$w^+(k)$	m= k_1 -w
	=k_1 +w k_1 -w
	=k_1 k
$w^-(j)(k)$	m= k_1 -w
	k_1= k_2
	=k_2 +w k
	m= k_3
	=k_3 +w j

Now, let $\mathbf{a}_1$, ..., $\mathbf{a}_k$ be the k registers used in the program. Add the following lexical items to the lexicon:

a_1	$\vdots$
=a_1 a_1 -a_1	a_{k-1}= a_k
a_1= a_2	=a_k a_k -a_k
$\vdots$	a_k= 1

(5) Whose brother's dentist's college roommate did you see?

The phenomenon of pied-piping is standardly analyzed as involving the transmission of one element's features to another (see e.g. [Moritz and Valois, 1994]).[3] In theories where structure is ultimately projected from the lexicon, a mechanism of feature percolation allows for a simplification of the description of certain grammatical phenomena by separating the statement of which items introduce the features from where these features ultimately wind up. Feature percolation is limited to certain (local) structural relationships, although exactly which such configurations allow feature percolation is not universally agreed upon. Most theories of locality agree (though see [Hallman, 2004]) that the specifier-head relationship is about as local as one gets, and thus that, if feature percolation happens at all, it can happen in the spec-head configuration.

1.2 Outline of the Remainder of the Paper

The main purpose of this paper is to show that adding feature percolation to the MG framework endows MGs with more expressive power than is thought strictly necessary for the description of natural languages. To show this, I need to introduce MGs, which I do in § 2.1. In § 2.2 I introduce infinite abaci, the turing equivalent computing device I show in § 3 that MGs with feature percolation can simulate.

2 Infinite Minimalism

Let's begin with a quick review of some basic concepts and abbreviations used in this paper. $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ is the set of natural numbers. For a finite set A, $|A|$ denotes its cardinality. The unique set of cardinality 0 is denoted $\emptyset$. Given two sets A and B, their difference is $A - B = \{a : a \in A \wedge a \notin B\}$, and their cross-product $A \times B = \{\langle a, b\rangle : a \in A \wedge b \in B\}$ is the set of ordered pairs whose first component is in A and whose second is in B. Given a set

[3]Whether this featural 'transmission' is literal movement of features from one object to another, or merely an expanding 'sphere of influence' of features which haven't moved at all is not of vital import for the results obtained herein. A literal feature movement approach is adopted here because of the resulting simplicity of the statement of the generating functions. For a presentation in terms of the expanding sphere of influence approach, see [Kobele et al., 2002].

A, a string over A is a finite sequence of elements $x = x_1 \dots x_n$, $x_i \in A$ for $1 \leq i \leq n$. If $n = 0$ then x is the empty string and is denoted ϵ. The length of a string x is denoted $|x|$. In particular, $|\epsilon| = 0$. The concatenation of two strings x and y is denoted by their juxtaposition xy. The n-fold iteration of a string x is

$$x^n = \underbrace{xx \dots x}_{n\ times}$$

If A and B are sets of strings, then $AB = \{xy : x \in A \text{ and } y \in B\}$ is the set of strings gotten by concatenating a string in A with a string in B. We set $A^0 = \{\epsilon\}$ and define $A^{n+1} = A^n A$. A^n is the n-fold iteration of strings in A. We define $A^* = \bigcup_{n=0}^{\infty} A^n$, and $A^+ = \bigcup_{n=1}^{\infty}$.

2.1 Minimalist Grammars

Linguistic structures are commonly given as (partially ordered sets of) trees, or as graphs. However, much of this structure seems to be functionally inert, and there is a very real (and important) question as to what kinds of structure we actually need to compute in the course of a derivation. Here we adopt a minimal representation (a null hypothesis), where we keep track of only those elements that are visible to our syntactic operations. Further work is needed to determine the full set of needed syntactic operations, as well as the appropriate level of structural richness necessary for adequate descriptions of (our knowledge of) natural language.

The minimalist grammars presented here are variants of the chain-based formalism given in [Stabler and Keenan, 2003].[4] A linguistic expression is understood to be a sequence of bounded length. Each element of the sequence represents a linguistic object which has not yet been linearized with respect to the root. A linguistic object is linearizable only after all of its licensee features have been checked (i.e. after all of its required dependencies have been satisfied). The first element of the sequence represents both the root (that is, the object which projects over everything else) as well as the (syntactically inert) linguistic material which has already been linearized.

Features come in attractor/attractee pairs. For an operation to apply to a (pair of) expression(s), both an attractor and an attractee feature (of the appropriate type) must be accessible to it. The operations defined on

[4]See [Stabler, 1997] for a representationally enriched tree-based version.

expressions check off syntactic features each time they apply (and can thus only apply if there are features to check).

Let Σ be a finite alphabet. A minimalist grammar over Σ is given by specifying three things:

1. a finite set **sel** of *selection* feature types, where for $f \in \mathbf{sel}$, $=f$ is a *selector* feature (i.e. $=np$ allows an np to be base generated by the carrier), and f is a *selectee* feature.

$$\mathbf{selector} \equiv \{=f : f \in \mathbf{sel}\} \text{ and } \mathbf{selectee} \equiv \{f : f \in \mathbf{sel}\}$$

2. a finite set **lic** of *licensing* feature types, where for $f \in \mathbf{lic}$, $+f$ is a *licensor* feature, and $-f$ is a *licensee* feature.

$$\mathbf{licensor} \equiv \{+f : f \in \mathbf{lic}\} \text{ and } \mathbf{licensee} \equiv \{-f : f \in \mathbf{lic}\}$$

3. a finite lexicon of expressions, where

   - $\mathbf{Feat} \equiv \mathbf{selector} \cup \mathbf{licensor} \cup \mathbf{selectee} \cup \mathbf{licensee}$ is the set of syntactic features
   - $\Sigma^* \times \mathbf{Feat}^*$ is the set of *initial chains*
   - $\Sigma^* \times \mathbf{licensee}^+$ is the set of *non-initial chains.*

   An expression e consists of an initial chain e_0 followed by up to $n = |\mathbf{lic}|$ non-initial chains $e_1, \ldots, e_n$, with the restriction that no two non-initial chains begin with the same licensee feature - this is a strict way of enforcing a shortest move constraint (SMC) [Chomsky, 1995].[5] **Exp** is the set of expressions.

For c a chain, c_1 is the phonetic component of c and c_2 is the syntactic component of c. If $c_2 = g\gamma$ for $g \in \mathbf{Feat}$ and $\gamma \in \mathbf{Feat}^*$, then g is the first feature of c. For $f \in \mathbf{lic}$ and e an expression, $e(f)$ is the unique chain e_i such that the first feature of e_i is $-f$, if such exists, and is the empty string

[5]The shortest move constraint is usually formulated in terms like the following:

> the move required to meet the demands of some higher projection, e.g. to check Case, a *wh*-feature, or a V-feature, must be met by the closest expression that could in principle meet that requirement [Hornstein et al., 2005]

otherwise. An expression e is complete iff e is fully linearized (i.e. e has no non-initial chains), and the root of e (e_0) has only a single selectee feature ($(e_0)_2 \in \textbf{selectee}$).

The SMC is ensured by restricting the domains of the structure-building operations, *merge* and *move*, which means that we can view it as a descriptive generalization ultimately due to the nature of our rules and/or our expressions, instead of as a filter on completed derivations.

- *merge* has two cases, given below. In both cases, an expression is merged into the complement position of the root.[6] The cases differ in whether or not the phonetic features of the new complement are linearized with respect to the root (*merge1*) or not (*merge2*).
  for $\sigma, \tau \in \Sigma^*$, $\gamma \in \textbf{Feat}^*$, $\delta \in \textbf{Feat}^+$, such that every ϕ_i and ψ_j has a different first feature (the SMC) and (for *merge2*) δ begins with a different feature from any ϕ_i or ψ_j:

$$\frac{(\sigma, {=}x\gamma), \phi_1, \ldots, \phi_n \quad (\tau, x), \psi_1, \ldots, \psi_m}{(\sigma\tau, \gamma), \phi_1, \ldots, \phi_n, \psi_1, \ldots, \psi_m}\ merge1$$

$$\frac{(\sigma, {=}x\gamma), \phi_1, \ldots, \phi_n \quad (\tau, x\delta), \psi_1, \ldots, \psi_m}{(\sigma, \gamma), \phi_1, \ldots, \phi_n, (\tau, \delta), \psi_1, \ldots, \psi_m}\ merge2$$

- *move* also has two cases. In both cases, the moved element can be thought of as moving to a specifier position of the root. The cases differ again as to whether the new specifier is linearizable (*move1*) or not (*move2*).
  for $\sigma, \tau \in \Sigma^*$, $\gamma \in \textbf{Feat}^*$,$\delta \in \textbf{Feat}^+$, and every ϕ_i has a different first feature (the SMC),

$$\frac{(\sigma, {+}x\gamma), \phi_1, \ldots, \phi_{i-1}, (\tau, {-}x), \phi_{i+1}, \ldots, \phi_m}{(\tau\sigma, \gamma), \phi_1, \ldots, \phi_{i-1}, \phi_{i+1}, \ldots, \phi_m}\ move1$$

[6]This definition of merge differs from Stabler's in allowing multiple complements to be base generated (and disallowing base generated specifiers). A more complete presentation would add two additional cases to the *merge* operation, corresponding to the merger of a specifier, and differing again in whether the new specifier's phonetic features are linearized with respect to the root. The version given here is both easier on the eye, and sufficient for the purposes of this paper. As no lexical item avails itself of the possibility of multiple complements, the results obtained for this system carry over immediately to the full system.

$$\frac{(\sigma, +x\gamma), \phi_1, \ldots, \phi_{i-1}, (\tau, -x\delta), \phi_{i+1}, \ldots, \phi_m}{(\sigma, \gamma), \phi_1, \ldots, \phi_{i-1}, (\tau, \delta), \phi_{i+1}, \ldots, \phi_m} \; move2$$

To capture the operation of Spec-Head agreement in minimalist grammars, I introduce a new pair of licensee/licensor feature-types (alternatively, a diacritic which may appear on licensee or licensor feature-tokens) with the interpretation that when movement is triggered by a feature with this diacritic, the remaining features on the moved head (recall that movement checks off the involved feature) are transmitted ('percolated') onto the root. There are as many variants of this operation as there are ways to (linearly) order the features percolated and the features of the root. The result is independent of the choice of variant. In the following, I offer a variant independent proof, as well as showing the construction for the variant which involves placing all of the features percolated *before* all of the *licensee* features of the root preserving the original orders of both sequences (what I will refer to as 'stack-based' percolation). The interest of the stack-based percolation is that it is independent of the SMC (and thus shows that the general result is not parasitic on the particular statement of the shortest move constraint).

Formally, let $+\hat{f}, -\hat{f}$, for any $f \in \mathbf{lic}$, be understood as in the above paragraph. Then the move function is extended to include the following case. Here the new specifier is always linearized with respect to the root (as it has no more unchecked syntactic features (having percolated them to the root)).

For $f, f' \in \{g, \hat{g}\}$ such that at least one of f, f' is $\hat{g}$, for $\beta \in \mathbf{Feat}^*$, $\delta \in \mathbf{licensee}^*$, $\sigma, \tau \in \Sigma^*$, and $\alpha_1, \ldots, \alpha_k$ non-initial chains (and the SMC is respected):

$$\frac{(\sigma, +f\ \beta), \alpha_1, \ldots, \alpha_{i-1}, (\tau, -f'\ \delta), \alpha_{i+1}, \ldots, \alpha_k}{(\tau\sigma, \beta \otimes \delta), \alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k} move\text{-}fp$$

The realization of the function $\otimes$ in the definition of *move-fp* is the factor distinguishing the feature percolation variants discussed above. In the stack-based variant, $xy \otimes z = xzy$, for $x \in (\mathbf{Feat} - \mathbf{licensee})^*$ and $y \in \mathbf{licensee}^*$. For the abacus simulation results below to go through, it is sufficient that $\otimes$ neither delete nor insert material not in its arguments.[7]

The expressions generated by a minimalist grammar over Σ are those which can be built up by a finite number of applications of the operations

[7]This requirement is very much in the spirit of Chomsky's Inclusiveness Condition.

merge and *move* starting with the lexical items. For $X \subseteq \mathbf{Exp}$ any arbitrary set of expressions,

$$E(X) = \bigcup_{n=0}^{\infty} E_n(X)$$

and

$$\begin{array}{lcl}
E_0(X) & = & X \\
E_{n+1}(X) & = & E_n(X) \\
& \cup & \{move(e) : e \in dom(move) \cap E_n(X)\} \\
& \cup & \{merge(e, e') : \langle e, e' \rangle \in dom(merge) \cap (E_n(X) \times E_n(X))\}
\end{array}$$

2.2 Bringing out the Abaci

An (infinite) abacus [Lambek, 1961] (see also [Minsky, 1961]) is conceptualized as consisting of some (finite) number of labeled/named locations, an infinite supply of indistinguishable pebbles, and a set of labeled instructions for manipulating the number of pebbles in each location. There is a designated start instruction (with label 0), and a special final label t which does not occur on any instruction. For convenience, we set the labels to be a finite initial segment L of $\mathbb{N}$, and define $t = max(L)$. An instruction over L has one of the following two simple forms:

- $w^+(k)$ - add one pebble to location w and then execute instruction k
- $w^-(k)(j)$ - if location w is empty, execute instruction j, otherwise remove one pebble and execute k

for $k, j \in L$, and $w \in Loc$ (the finite set of locations). The set of instructions over L is $\textsc{Ins}(L)$. Given labels L, a program is a map $p : (L - \{t\}) \to \textsc{Ins}(L)$.

I will present a program p in the following format:

$$\begin{array}{cccccc}
 & 0 & 1 & \ldots & t \\
p : & \downarrow & \downarrow & \ldots & \\
 & p(0) & p(1) & \ldots &
\end{array}$$

A pebble configuration specifies how many pebbles are in each location. We identify pebble configurations with functions $c : Loc \to \mathbb{N}$ from locations to the number of pebbles they contain. Abusing notation somewhat, we can view a program p as a map over pebble configurations such that

$$p(c) = \ulcorner p \urcorner(0)(c)$$

where $\ulcorner p \urcorner : L \times [Loc \to \mathbb{N}] \to [Loc \to \mathbb{N}]$ is a function from instruction labels and pebble configurations to updated pebble configurations such that (defining $c[w^+]$ ($c[w^-]$) to be the pebble configuration identical to c except that $c[w^+](w) = c(w) + 1$ ($c[w^-](w) = c(w) - 1$ if $c(w) > 0$, and undefined otherwise)):

$$\ulcorner p \urcorner(i)(c) = \begin{cases} c & \text{if } i = t \\ \ulcorner p \urcorner(k)(c[w^+]) & \text{if } p(i) = w^+(k) \\ \ulcorner p \urcorner(k)(c[w^-]) & \text{if } c(w) > 0 \text{ and } p(i) = w^-(k)(j) \\ \ulcorner p \urcorner(j)(c) & \text{if } c(w) = 0 \text{ and } p(i) = w^-(k)(j) \end{cases}$$

For example, the following program 'zeroes out' the contents of location a:

$$\begin{array}{lcc} & 0 & t \\ p: & \downarrow & \\ & a^-(0)(t) & \end{array}$$

Given a pebble configuration c such that $c(a) = 2$ we trace the operation of $\ulcorner p \urcorner$ on c:

$$\begin{array}{rcl} p(c) & = & \ulcorner p \urcorner(0)(c) \\ & = & \ulcorner p \urcorner(0)(c[a^-]) \\ & = & \ulcorner p \urcorner(0)((c[a^-])[a^-]) \\ & = & \ulcorner p \urcorner(t)((c[a^-])[a^-]) \\ & = & (c[a^-])[a^-] \end{array}$$

Note that $((c[a^-])[a^-])(a) = ((c[a^-])(a) - 1 = c(a) - 2 = 2 - 2 = 0$. Note also that this is not simply a lucky choice of initial pebble configuration, but that for any pebble configuration c', $p(c')(a) = 0$.

Say that a program p computes a function $f(x_1, x_2, \ldots, x_n) = y$ just in case

1. there is some m such that $Loc = \{x_1, \ldots, x_n, y, t_1, \ldots, t_m\}$
2. on every $c : Loc \to \mathbb{N}$ such that $c(y) = c(t_1) = \ldots = c(t_n) = 0$,
   (a) $p(c)(y) = f(c(x_1), c(x_2), \ldots, c(x_n))$
   (b) $p(c)(x_1) = p(c)(x_2) = \ldots = p(c)(x_n) = 0$
   (c) $p(c)(t_1) = p(c)(t_2) = \ldots = p(c)(t_m) = 0$

Lambek [1961] showed that abaci can compute arbitrary recursive functions.

3 Minimalist Abaci

I will represent a program as a lexicon, with each program instruction being effected by one or more lexical items. In an expression, the initial chain represents the current execution point in the program, and the non-initial chains represent the number of pebbles (licensee feature tokens) in a given location (licensee feature types). Relating this to the discussion above, the arguments to $\ulcorner p \urcorner$ are given by the initial chain (for i), and the non-initial chains (for c). The lexicon contains the inner workings of $\ulcorner p \urcorner$ itself. For example, the expression below represents an abacus at point 1 in the program, with three non-empty locations, **a**, **b**, and **d**, containing one, two, and one pebbles respectively.

$$(\epsilon, 1), (\epsilon, \text{-}\hat{a}), (\epsilon, \text{-}\hat{b}\text{-}\hat{b}), (\epsilon, \text{-}\hat{d})$$

More explicitly, an expression e *corresponds to* a pebble configuration c just in case[8]

1. for each $w \in Loc$, if $c(w) = 0$, then $e(w) = \epsilon$, otherwise $(e(w))_2 = (\text{-}\hat{w})^{c(w)}$
2. $(e_1)_2 \in$ **selectee**

The idea of the proof that MGs with feature percolation can implement infinite abaci is as follows. First, I show that every initial configuration of pebbles in locations can be generated. This will involve providing a set of lexical items – $Lex(x)$ – which can generate all expressions that have any number of pebbles in n locations, where the only feature in their initial chain is x (which I will later call 'being of category x'). The next step is to show how the abacus operations above can be implemented in a MG. For an arbitrary operation of the form $w^{+}(k)$ I add a set of lexical items $Lex(x, w, k)$ to $Lex(x)$ that construct an expression of category j and with one more pebble in location w from each expression of category x. The crucial observation is that the set $Lex(x, w, k)$ can be viewed as a function with domain the set

[8]This definition is for the generic variant. The first condition should be changed for the stack-based (or queue) variants as follows:

1. for each $w \in Loc$, if $c(w) \neq 0$ then $(e(w))_2 = (\text{-}\hat{w})^{c(w)}\text{-}\hat{z}_w$ and $e(z_w) = \epsilon$, otherwise $e(w) = \epsilon$ and $(e(z_w))_2 = \text{-}\hat{z}_w$.

of expressions of category x generated by $Lex(x)$ (the function being (very nearly) $\lambda y.\ulcorner p \urcorner(x)(y)$). If the set $Lex(x, w, k)$ is added not to $Lex(x)$ but to some other set of lexical items, then its domain is effectively limited to just those expressions of category x generated by that other set. This allows us to 'string together' commands. Similarly, operations of the form $w^{-}(k)(j)$ are represented by a set of lexical items $Lex(x, w, k, j)$ which are added to $Lex(x)$. A definition of the sets $Lex(x), Lex(x, w, k)$, and $Lex(x, w, k, j)$ follows.

$$\begin{array}{rcl} Lex(x) & = & \{(\epsilon,\ell_1), (\epsilon,\ell_1\ \text{-}\hat{a}_1)\} \\ & \cup & \{(\epsilon,\text{=}\ell_i\ \text{+}a_i\ \ell_i\ \text{-}\hat{a}_i\ \text{-}\hat{a}_i) : 1 \le i \le |Loc|\} \\ & \cup & \{(\epsilon,\text{=}\ell_i\ \ell_{i+1}), (\epsilon,\text{=}\ell_i\ \ell_{i+1}\ \text{-}\hat{a}_{i+1}) : 1 \le i < |Loc|\} \\ & \cup & \{(\epsilon,\text{=}\ell_{|Loc|}\ x)\} \end{array}$$

$$\begin{array}{rcl} Lex(x, w, k) & = & \{(\epsilon,\text{=}x\ x_0\ \text{-}\hat{w})\} \\ & \cup & \{(\epsilon,\text{=}x\ \text{+}w\ x_0\ \text{-}\hat{w}\ \text{-}\hat{w})\} \\ & \cup & \{(\epsilon,\text{=}x_0\ k)\} \end{array}$$

$$\begin{array}{rcl} Lex(x, w, k, j) & = & \{(\epsilon,\text{=}x\ \text{+}w\ x_0)\} \\ & \cup & \{(\epsilon,\text{=}x_0\ k)\} \\ & \cup & \{(\epsilon,\text{=}x\ x_1\ \text{-}\hat{w})\} \\ & \cup & \{(\epsilon,\text{=}x_1\ \text{+}w\ j)\} \end{array}$$

An entire program can be embedded in a lexicon in the following manner; for $p : (L - \{t\}) \rightarrow \textsc{Ins}(L)$ a program,

1. $Lex(0) \subseteq Lex_p$
2. for $p(i) = w^{+}(k)$, $Lex(i, w, k) \subseteq Lex_p$
3. for $p(i) = w^{-}(k)(j)$, $Lex(i, w, k, j) \subseteq Lex_p$
4. Nothing is in Lex_p if not by virtue of the above.

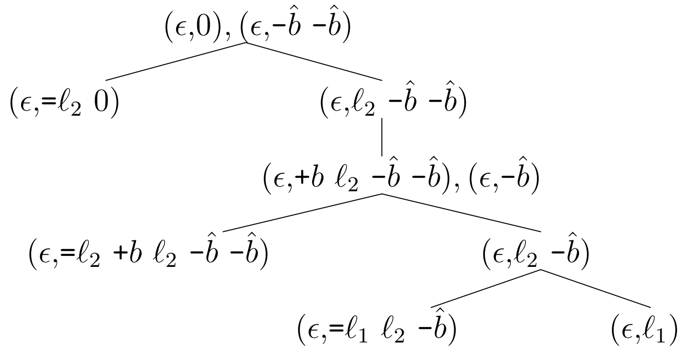
Before the statement of the main theorem, I work through an example, using the following program:

$$p : \begin{array}{ccc} 0 & 1 & t \\ \downarrow & \downarrow & \\ a^{+}(1) & b^{-}(0)(t) & \end{array}$$

I begin by constructing $Lex(0)$ for a program with two locations – **a** and **b**. The first column is for the stack-based variant of feature percolation, in the second column I give a set of lexical items that will work for every variant (including the stack-based one):

STACK	GENERIC
$(\epsilon,\ \ell_1\ \text{-}\hat{z}_a)$	$(\epsilon,\ \ell_1)$
$(\epsilon,\ \ell_1\ \text{-}\hat{a}\ \text{-}\hat{z}_a)$	$(\epsilon,\ \ell_1\ \text{-}\hat{a})$
$(\epsilon,\ \text{=}\ell_1\ \text{+}a\ \ell_1\ \text{-}\hat{a}\ \text{-}\hat{a})$	$(\epsilon,\ \text{=}\ell_1\ \text{+}a\ \ell_1\ \text{-}\hat{a}\ \text{-}\hat{a})$
$(\epsilon,\ \text{=}\ell_1\ \ell_2\ \text{-}\hat{z}_b)$	$(\epsilon,\ \text{=}\ell_1\ \ell_2)$
$(\epsilon,\ \text{=}\ell_1\ \ell_2\ \text{-}\hat{b}\ \text{-}\hat{z}_b)$	$(\epsilon,\ \text{=}\ell_1\ \ell_2\ \text{-}\hat{b})$
$(\epsilon,\ \text{=}\ell_2\ \text{+}b\ \ell_2\ \text{-}\hat{b}\ \text{-}\hat{b})$	$(\epsilon,\ \text{=}\ell_2\ \text{+}b\ \ell_2\ \text{-}\hat{b}\ \text{-}\hat{b})$
$(\epsilon,\ \text{=}\ell_2\ 0)$	$(\epsilon,\ \text{=}\ell_2\ 0)$

I give an example derivation using the generic lexical items – derivations are represented as trees labeled with expressions. A unary branch indicates that the mother is derived from the daughter by an application of *move*, and a binary branching node is derived by an operation of *merge* over its children (the left child is the first argument to *merge*, and the right child the second) – of an expression of category 0 with no pebbles in location **a**, and two in **b**. There is exactly one derivation of each expression of category 0 with this lexicon.


$(\epsilon,0), (\epsilon,\text{-}\hat{b}\ \text{-}\hat{b})$
$(\epsilon,\text{=}\ell_2\ 0)$
$(\epsilon,\ell_2\ \text{-}\hat{b}\ \text{-}\hat{b})$
$(\epsilon,\text{+}b\ \ell_2\ \text{-}\hat{b}\ \text{-}\hat{b}), (\epsilon,\text{-}\hat{b})$
$(\epsilon,\text{=}\ell_2\ \text{+}b\ \ell_2\ \text{-}\hat{b}\ \text{-}\hat{b})$
$(\epsilon,\ell_2\ \text{-}\hat{b})$
$(\epsilon,\text{=}\ell_1\ \ell_2\ \text{-}\hat{b})$
(ϵ,ℓ_1)

Note that while the generic lexical items construct the expression just described, the stack based items generate an expression with $\text{-}\hat{z}_i$'s in each component. The intuition is that when the operation $\otimes$ is simple enough, it allows for marking of the 'end' of the list of features (the 'z' is for 'zero (pebbles)'), and then later operations can refer explicitly to this end (i.e. can determine whether there are zero pebbles in a location explicitly; the generic approach relies on the SMC to weed out any derivation that mistakenly assumes that a particular location is not empty).[9]

Next, I construct $Lex(0, a, 1)$, which implements the instruction, $a^{+}(1)$:

Stack	Generic
$(\epsilon,$ =0 +z$_a$ 0_0 -$\hat{a}$ -$\hat{z}_a)$	$(\epsilon,$ =0 0_0 -$\hat{a})$
$(\epsilon,$ =0 +a 0_0 -$\hat{a}$ -$\hat{a})$	$(\epsilon,$ =0 +a 0_0 -$\hat{a}$ -$\hat{a})$
$(\epsilon,$ =0_0 1$)$	$(\epsilon,$ =0_0 1$)$

Combining $Lex(0, a, 1)$ with $Lex(0)$ allows the continuation of the derivation above by adding one pebble to location **a**.

$$\begin{tikzcd}[column sep=small]
 & (\epsilon,\ 1),(\epsilon,\text{-}\hat{a}),(\epsilon,\text{-}\hat{b}\ \text{-}\hat{b}) \arrow[dl, dash] \arrow[dr, dash] & & \\
(\epsilon,\text{=}0_0\ 1) & & (\epsilon,\ 0_0\ \text{-}\hat{a}),(\epsilon,\text{-}\hat{b}\ \text{-}\hat{b}) \arrow[dl, dash] \arrow[dr, dash] & \\
 & (\epsilon,\text{=}0\ 0_0\ \text{-}\hat{a}) & & (\epsilon,\ 0),(\epsilon,\text{-}\hat{b}\ \text{-}\hat{b})
\end{tikzcd}$$

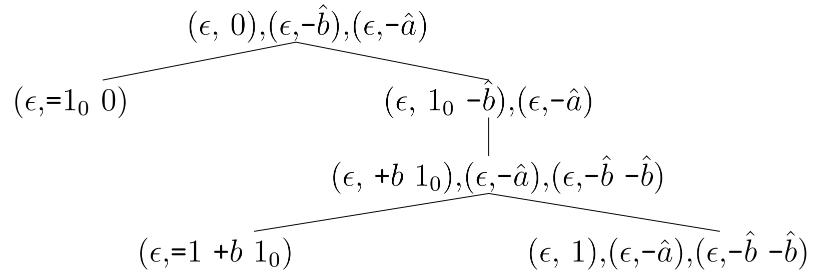
Next, I construct $Lex(1, b, 0, t)$ implementing the instruction $b^{-}(0)(t)$:

Stack	Generic
$(\epsilon,$ =1 +b $1_0)$	$(\epsilon,$ =1 +b $1_0)$
$(\epsilon,$ =1_0 0$)$	$(\epsilon,$ =1_0 0$)$
$(\epsilon,$ =1 +z$_b$ 1_1 -$\hat{z}_b)$	$(\epsilon,$ =1 1_1 -$\hat{b})$
$(\epsilon,$ =1_1 $t)$	$(\epsilon,$ =1_1 +b $t)$

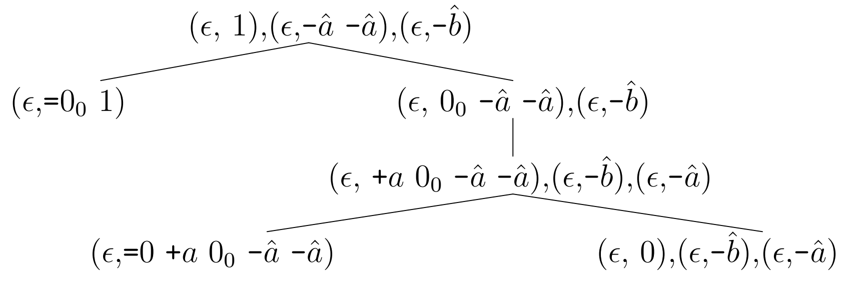
Adding $Lex(1, b, 0, t)$ to $Lex(1)$ will generate both expressions of category 0 and expressions of category t. The generated expressions of category t will all have zero pebbles in location **b**.

As the expression derived above is generated by $Lex(1)$, we can continue the derivation as shown below. First we derive an expression of category 0 by deleting a pebble from location **b**. Note that after the pebble is removed from **b**, the remaining licensee feature percolates onto the initial chain.

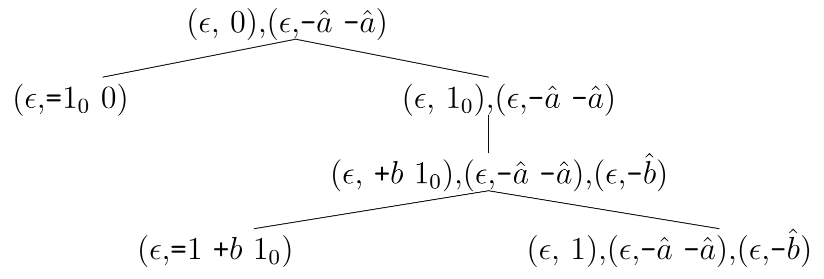
[9]When $x \otimes y = xy$ (the 'queue-based' variant), refering to the edge is still possible, but is less efficient (i.e. it requires more lexical items to effect the translation of a program) than the generic variant. Referring to the edge does, however, remove reliance on the SMC when testing for zero.


$(\epsilon,\ 0),(\epsilon,\text{-}\hat{b}),(\epsilon,\text{-}\hat{a})$
$(\epsilon,\text{=}1_0\ 0)$
$(\epsilon,\ 1_0\ \text{-}\hat{b}),(\epsilon,\text{-}\hat{a})$
$(\epsilon,\ \text{+}b\ 1_0),(\epsilon,\text{-}\hat{a}),(\epsilon,\text{-}\hat{b}\ \text{-}\hat{b})$
$(\epsilon,\text{=}1\ \text{+}b\ 1_0)$
$(\epsilon,\ 1),(\epsilon,\text{-}\hat{a}),(\epsilon,\text{-}\hat{b}\ \text{-}\hat{b})$

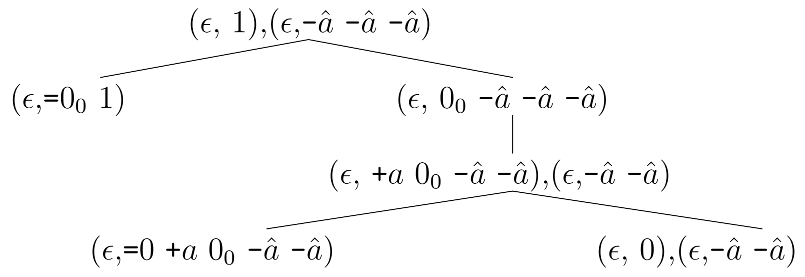
Now that we again have an expression of category 0, we may use the lexical items introduced in $Lex(0, a, 1)$ to derive an expression of category 1 with an additional pebble in location **a**.


$(\epsilon,\ 1),(\epsilon,\text{-}\hat{a}\ \text{-}\hat{a}),(\epsilon,\text{-}\hat{b})$
$(\epsilon,\text{=}0_0\ 1)$
$(\epsilon,\ 0_0\ \text{-}\hat{a}\ \text{-}\hat{a}),(\epsilon,\text{-}\hat{b})$
$(\epsilon,\ \text{+}a\ 0_0\ \text{-}\hat{a}\ \text{-}\hat{a}),(\epsilon,\text{-}\hat{b}),(\epsilon,\text{-}\hat{a})$
$(\epsilon,\text{=}0\ \text{+}a\ 0_0\ \text{-}\hat{a}\ \text{-}\hat{a})$
$(\epsilon,\ 0),(\epsilon,\text{-}\hat{b}),(\epsilon,\text{-}\hat{a})$

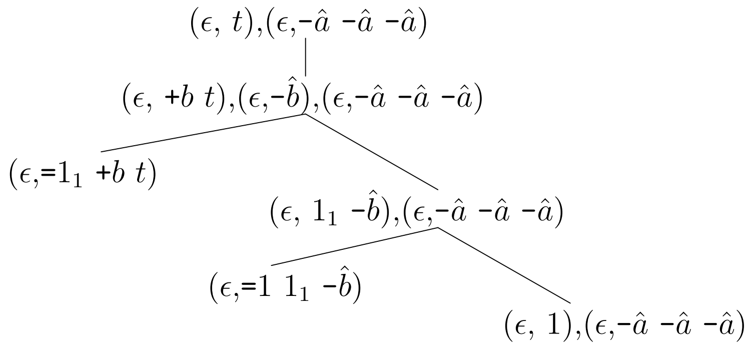
Once more we remove a pebble from location **b** . . .


$(\epsilon,\ 0),(\epsilon,\text{-}\hat{a}\ \text{-}\hat{a})$
$(\epsilon,\text{=}1_0\ 0)$
$(\epsilon,\ 1_0),(\epsilon,\text{-}\hat{a}\ \text{-}\hat{a})$
$(\epsilon,\ \text{+}b\ 1_0),(\epsilon,\text{-}\hat{a}\ \text{-}\hat{a}),(\epsilon,\text{-}\hat{b})$
$(\epsilon,\text{=}1\ \text{+}b\ 1_0)$
$(\epsilon,\ 1),(\epsilon,\text{-}\hat{a}\ \text{-}\hat{a}),(\epsilon,\text{-}\hat{b})$

. . . and add one to **a**. Notice again the feature percolation in the move step.



Finally, there are no more pebbles in location **b**, and so we cannot continue the derivation using the same lexical items (this is the crucial factor). In order to continue the derivation, we must use the other lexical items introduced in $Lex(1, b, 0, t)$. Note that we could not have used them earlier - the SMC would have prohibited the second merge step.



Note that once you have finished ‘setting the pebbles up’ (have finished with $Lex(0)$), there is at most one derivation until you have reached category t. That is, every derivation tree of category 0 generated by $Lex(0)$ is a subtree of at most one derivation tree of category t.

The central claim of this paper is that:

Theorem 1 For any abacus program p there is a minimalist grammar G_p such that for any pebble configurations c, c', and corresponding expressions $e, e' \in E(G_p)$ with $(e_0)_2 = 0$ and $(e'_0)_2 = t$, $p(c) = c'$ iff there is a derivation in G_p from e to e'.

The proof of theorem 1 is deferred to the appendix. A simple corollary shows how to translate this strong generative capacity into weak generative capacity:

Corollary 1 Let $f : \mathbb{N} \to \mathbb{N}$ be an arbitrary recursive function. Then there is a minimalist grammar G such that the string language of G is $\{a^{f(n)} : n \in \mathbb{N}\}$.

Proof: As $f(x) = y$ is a recursive function, there is a program p over $m + 1$ locations that computes it [Lambek, 1961]. By theorem 1 there is a minimalist grammar G_p such that every expression e of category t that, for some $n \in \mathbb{N}$, has exactly $f(n)$ tokens of $\texttt{-}\hat{y}$ features, and for every $n \in \mathbb{N}$, there is some expression e of category t has exactly $f(n)$ $\texttt{-}\hat{y}$ feature tokens. Let G be constructed from G_p by adding the lexical item:

$$(a,\ \texttt{=}t\ \texttt{+}\hat{y}\ t)$$

Set the string language of G to be the yields of the complete expressions of category t:

$$L(G) = \{w : \exists e \in E(G).\ e_0 = (w, t) \wedge \forall f \in \mathbf{lic}.\ e(f) = \epsilon\}$$

□

4 Conclusions

We have seen that, viewing features as resources, adding a simple kind of feature percolation to minimalist grammars increases their expressive power far beyond anything believed necessary for the description of natural language. This result is predicated on assuming that features are percolated linearly (i.e. on the assumption that $\otimes$ neither add nor delete material not in its arguments). We could of course abandon this assumption, or feature percolation, or even the minimalist grammar framework. None of this is required of us, even if we do not believe strictly r.e. string sets to be lurking somewhere in the shadowy unexplored nether regions of the world. The Minimalist Program [Chomsky, 1995] is based on the idea that the explanation for the observed linguistic diversity lies not just on the shoulders of syntax, but is able to be distributed over interactions between 'modules'. We might constrain our super-powerful syntax with a semantic theory that refuses to interpret many of the syntactically 'well-formed' structures, or with a theory of learning that makes large swathes of syntactically possible languages unlearnable from primary linguistic data [Angluin, 1980, Kanazawa, 1998]. An expressivity result of the sort contained herein serves to make explicit the role (or lack thereof) the syntactic component of the grammar will be able to play in delimiting the class of possible human languages. Thus, if we want to do feature percolation linearly, we know that there's a lot of explanatory slack which will need to be picked up somewhere.

Appendix

The following sets of expressions will prove useful. For $n > 1$, and for $1 < i \leq |Loc|$ (in the following I implicitly restrict attention to those elements which are in the domains of the various partial functions applied to them):

$$\begin{array}{lcl}
g_1^0 & = & \{(\epsilon,\ell_1)\} \\
g_1^1 & = & \{(\epsilon,\ell_1 \text{-}\hat{w}_1)\} \\
g_1^n & = & \{move(merge((\epsilon,\text{=}\ell_1 \text{+}w_1\ \ell_1 \text{-}\hat{w}_1 \text{-}\hat{w}_1), e)) : e \in g_1^{n-1}\} \\
g_i^0 & = & \{merge((\epsilon,\text{=}\ell_{i-1}\ \ell_i), e) : \exists m.\ e \in g_{i-1}^m\} \\
g_i^1 & = & \{merge((\epsilon,\text{=}\ell_{i-1}\ \ell_i \text{-}\hat{w}_i), e) : \exists m.\ e \in g_{i-1}^m\} \\
g_i^n & = & \{move(merge((\epsilon,\text{=}\ell_i \text{+}w_i\ \ell_i \text{-}\hat{w}_i \text{-}\hat{w}_i), e)) : e \in g_i^{n-1}\} \\
z & = & \{merge((\epsilon,\text{=}\ell_{|Loc|}\ 0), e) : \exists m.\ e \in g_{|Loc|}^m\}
\end{array}$$

Proposition 1 $z = E(Lex(0)) \cap \{e : (e_0)_2 = 0\}$

Proof:
$\subseteq$:
Let $e \in z$ be arbitrary. Clearly, $(e_0)_2 = 0$. If $g_{|Loc|}^m \subseteq E(Lex(0))$, then we would be able to conclude that $e \in E(Lex(0))$, and the proof would be complete. I show $g_i^n \subseteq E(Lex(0))$, for $1 < i \leq |Loc|$, and for $n \in \mathbb{N}$. I first show that $g_1^n \subseteq E(Lex(0))$, all n. This is immediate for g_1^0 and g_1^1, as they are subsets of $Lex(0)$ itself. Now assume that $g_1^m \subseteq E(Lex(0))$ for some $m \geq 1$, and let $e \in g_1^m$ be arbitrary (note that $e_0 = (\epsilon,\ell_1\ (\text{-}\hat{w}_1)^m)$). Then $move(merge((\epsilon,\text{=}\ell_1 \text{+}w_1\ \ell_1 \text{-}\hat{w}_1 \text{-}\hat{w}_1), e)) \in E(Lex(0))$.

Now assume that for all $n \in \mathbb{N}$, $g_k^n \subseteq E(Lex(0))$, and let $n \in \mathbb{N}$ and $e \in g_k^n$ be arbitrary. Note that $e_0 = (\epsilon,\ell_k\ (\text{-}\hat{w}_k)^n)$. Then

1. $merge((\epsilon,\text{=}\ell_k\ \ell_{k+1}), e) \in E(Lex(0))$, and
2. $merge((\epsilon,\text{=}\ell_k\ \ell_{k+1} \text{-}\hat{w}_{k+1}), e) \in E(Lex(0))$

Now assume that $g_{k+1}^m \subseteq E(Lex(0))$, for some $m \geq 1$, and let $e \in g_{k+1}^m$ be arbitrary (again, $e_0 = (\epsilon,\ell_{k+1}\ (\text{-}\hat{w}_{k+1})^m)$). Then, completing the proof, $move(merge((\epsilon,\text{=}\ell_{k+1} \text{+}w_{k+1}\ \ell_{k+1} \text{-}\hat{w}_{k+1} \text{-}\hat{w}_{k+1}), e)) \in E(Lex(0))$
$\supseteq$:
I show instead that for $1 \leq i \leq |Loc|$, $\bigcup_{n=0}^{\infty} g_i^n \supseteq E(Lex(0)) \cap \{e : e_0 \text{ begins with } \ell_i\}$, from whence the original claim will follow. We start with $i = 1$. Let $e \in E(Lex(0)) \cap \{e : e_0 \text{ begins with } \ell_1\}$ be arbitrary. If $e \in E_0(Lex(0)) = Lex(0)$,

then e is either in g_1^0 or g_1^1. Otherwise, let $e \in E_{n+1}(Lex(0))$. The only derived items in $E(Lex(0))$ which begin with ℓ_1 are of the form $move(merge((\epsilon,\text{=}\ell_1\ \text{+}w_1\ \ell_1\ \text{-}\hat{w}_1\ \text{-}\hat{w}_1), e'))$, and thus $e \in \bigcup_{n=0}^{\infty} g_1^n$.

Now if $\bigcup_{n=0}^{\infty} g_j^n \supseteq E(Lex(0)) \cap \{e : e_0 \text{ begins with } \ell_j\}$, for $1 \leq j < |Loc|$, then any $e \in E(Lex(0)) \cap \{e : e_0 \text{ begins with } \ell_{j+1}\}$ must be one of $merge((\epsilon,\text{=}\ell_j\ \ell_{j+1}), e')$, or $merge((\epsilon,\text{=}\ell_j\ \ell_{j+1}\ \text{-}\hat{w}_{j+1}), e')$, or $move(merge((\epsilon,\text{=}\ell_{j+1}\ \text{+}w_{j+1}\ \ell_{j+1}\ \text{-}\hat{w}_{j+1}\ \text{-}\hat{w}_{j+1}), e''))$ for e' such that e'_0 begins with ℓ_j and e'' such that e''_0 begins with ℓ_{j+1}. In either of the first two cases, e is straightforwardly in g_j^0 or g_j^1, respectively. And in the third case, e'' is generated by a finite number of applications of merge and move to ever simpler expressions until an expression in g_j^1 is reached, which means that $e'' \in g_{j+1}^k$, and $e \in g_{j+1}^{k+1}$, some k. □

Proposition 2 For every pebble configuration $c : Loc \to \mathbb{N}$ there is an expression $e \in E(Lex(0))$ of category 0 which corresponds to c, and vice versa.

Proof: I exhibit a bijection $h : z \to [Loc \to \mathbb{N}]$ such that e corresponds to $h(e)$. First I define the relation 'is a constituent of' as $CON = \bigcup_{n=0}^{\infty} CON^n$, where for all $e, e' \in \mathbf{Exp}$,

$$eCON^0e$$
$$eCON^1e' \text{ iff } e' = move(e) \vee \exists e''.\ e' = merge(e, e'') \vee merge(e'', e)$$
$$eCON^{n+1}e' \text{ iff } \exists e''.\ eCON^ne'' \wedge e''CON^1e$$

Then $h(e) = c_e$, where $\forall i \in Loc.\ c_e(i) = max(\{n : \exists e' \in g_i^n.\ e'CONe\})$.

1. h is a function, as for every $e \in z$, and every $1 \leq j \leq |Loc|$, there is a unique $e' \in g_j^n$ such that $e'CONe$ and for any other m such that there is a $e'' \in g_j^m$ which $e''CONe$, $e''CONe'$ as well.

2. h is 1-1, as if $e \neq e'$, then wlg. e has more -$\hat{w}_i$ features than does e', which means that $h(e)(i) > h(e')(i)$.

3. h is onto, as for $c \in [Loc \to \mathbb{N}]$ arbitrary, the $e \in z$ which corresponds to c has as maximal constituents $e^i \in g_i^{c(i)}$. Such an e exists, as the choice of the e^is is independent.

□

Now we are ready to prove

Theorem 1 For any abacus program p there is a minimalist grammar G_p such that for any pebble configurations c, c', and corresponding expressions $e, e' \in E(G_p)$ with $(e_0)_2 = 0$ and $(e'_0)_2 = t$, $p(c) = c'$ iff there is a derivation in G_p from e to e'.

Proof: Let p be a program, and Lex_p as described above. By proposition 2, we see that we have access to all possible pebble configurations via $Lex(0)$. Now we will see that the sets $Lex(i, w, k)$ and $Lex(i, w, k, j)$ map configurations to configurations in the appropriate ways.

Let $p(i) = w_m^+(k)$. Then for any c, $\ulcorner p \urcorner(i)(c) = \ulcorner p \urcorner(k)(c[w_m^+])$. By definition, $Lex(i, w_m, k) \subset Lex_p$, and so the following lexical items are in Lex_p:

1. $(\epsilon$, =i +w_m i_i -$\hat{w}_m$ -$\hat{w}_m)$
2. $(\epsilon$, =i i_i -$\hat{w}_m)$
3. $(\epsilon$, =i_i $k)$

I show that for arbitrary e with $(e_0)_2 = i$, the only (successful) derivation it may enter into is one in which $e(w_m)$ increases by 1. First observe that lexical items one and two are the only ones in Lex_p which are able to combine with e. There are two cases, according to whether $e(w_m) = \epsilon$ or not.

case 1 $e(w_m) = \epsilon$
It should be clear that the derivation will crash (i.e. not be able to continue) if lexical item one merges with e. $(merge($=i i_i -$\hat{w}_m, e))_2 = i_i$ -$\hat{w}_m$, which is merged with lexical item three to get e' of category k with exactly one -$\hat{w}_m$ feature. Note that $(e')_2 = k$.

case 2 $(e(w_m))_2 = ($-$\hat{w}_m)^r$
For similar reasons to case 1, merging the second lexical item above with e causes the derivation to crash. Now, let e' be the result of merging the first lexical item with e. Then $(e')_2 =$ +$\hat{w}_m$ i_i -$\hat{w}_m$ -$\hat{w}_m$, and for e'' be the result of applying *move* to e', $(e'')_2 = i_i$ $($-$\hat{w}_m)^{r+1}$. Merging lexical item three to e'' results in e''' of category k with exactly $r + 1$ -$\hat{w}_m$ features. Note that $(e''')_2 = k$.

Let $p(i) = w_m^-(k)(j)$. Then for any c, $\ulcorner p \urcorner(i)(c) = \ulcorner p \urcorner(k)(c[w_m^-])$ if $c(w_m) > 0$, and $\ulcorner p \urcorner(i)(c) = \ulcorner p \urcorner(j)(c)$ otherwise. $Lex(i, w_m, k, j) \subset Lex_p$ by definition, and so the following lexical items are in Lex_p:

1. $=i$ $+w_m$ i_k
2. $=i_k$ k
3. $=i$ i_1 $-\hat{w}_m$
4. $=i_1$ $+w_m$ j

Observe again that lexical items one and three are the only ones in Lex_p which are able to combine with e. There are two cases, according to whether $e(w_m) = \epsilon$ or not.

case 1 $e(w_m) = \epsilon$
As the derivation will crash if lexical item one is used, let instead $e' = merge(=i\ i_1\ -\hat{w}_m, e)$. When item four is merged to e', the $-\hat{w}_m$ is checked again, yielding e'' of category k with no $-\hat{w}_m$ features. Again, $(e'')_2 = j$.

case 2 $(e(w_m))_2 = (-\hat{w}_m)^r$
It is clear that item one will merge with e, one $-\hat{w}_m$ feature will be checked, and then item two will result in some e' of category k. The fact of interest is why the derivation will crash if lexical item three is merged with e instead. Let e'' be the result of merging lexical item three with e. Then $(e''_0)_2 = i_1\ -\hat{w}_m$. Because $(e''(w_m))_2 = (-\hat{w}_m)^r$, lexical item four, the only expression in Lex_p with a $=i_1$ feature, cannot select e'' – *merge* is not defined in this case.

□

References

Dana Angluin. Inductive inference of formal languages from positive data. *Information and Control*, 45:117–135, 1980.

Michael Brody. Mirror theory; syntactic representation in perfect syntax. *Linguistic Inquiry*, 31:29–56, 2000.

Noam Chomsky. Derivation by phase. In Michael Kenstowicz, editor, *Ken Hale: A Life in Language*, pages 1–52. MIT Press, Cambridge, Massachusetts, 2001.

Noam Chomsky. A minimalist program for linguistic theory. In Kenneth Hale and Samuel Jay Keyser, editors, *The View from Building 20.* MIT Press, Cambridge, Massachusetts, 1993.

Noam Chomsky. *The Minimalist Program.* MIT Press, Cambridge, Massachusetts, 1995.

John Frampton and Sam Gutmann. Agreement is feature sharing. ms. Northeastern University, 2000.

John Frampton and Sam Gutmann. Crash-proof syntax. In Samuel David Epstein and T. Daniel Seely, editors, *Derivation and Explanation in the Minimalist Program*, pages 90–105. Blackwell, 2002.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

Peter Hallman. Symmetry in structure building. *Syntax*, 7(1):79–100, 2004.

H. Harkema. A Characterization of Minimalist Grammars. In P. de Groote, G.F. Morrill, and C. Retoré, editors, *Logical Aspects of Computational Linguistics (LACL 2001)*, volume 2099 of *Lecture Notes in Artificial Intelligence.* Springer Verlag, Berlin, Heidelberg, Germany, 2001.

Norbert Hornstein, Jairo Nunes, and Kleanthes K. Grohmann. *Understanding Minimalism: An Introduction to Minimalist Syntax.* Cambridge University Press, 2005.

Makoto Kanazawa. *Learnable Classes of Categorial Grammars.* CSLI Publications, Stanford University., 1998.

Gregory M. Kobele. Formalizing mirror theory. *Grammars*, 5(3):177–221, 2002.

Gregory M. Kobele, Travis Collier, Charles Taylor, and Edward P. Stabler. Learning mirror theory. In *Proceedings of the Sixth International Workshop on Tree Adjoining Grammars and Related Frameworks (TAG+6),* Venezia, 2002.

Hilda Koopman. The spec-head configuration. Syntax at Sunset: UCLA Working Papers in Syntax and Semantics, edited by Edward Garrett and Felicia Lee, 1996.

Hilda Koopman and Anna Szabolcsi. *Verbal Complexes.* MIT Press, Cambridge, Massachusetts, 2000.

Joachim Lambek. How to program an (infinite) abacus. *Canadian Mathematical Bulletin*, 4:295–302, 1961.

Alain Lecomte and Christian Retoré. Towards a minimal logic for minimalist grammars. In *Proceedings, Formal Grammar'99*, Utrecht, 1999.

J. Michaelis. Transforming Linear Context-Free Rewriting Systems into Minimalist Grammars. In P. de Groote, G.F. Morrill, and C. Retoré, editors, *Logical Aspects of Computational Linguistics (LACL 2001)*, volume 2099 of *Lecture Notes in Artificial Intelligence*. Springer Verlag, Berlin, Heidelberg, Germany, 2001.

J. Michaelis. Derivational Minimalism is Mildly Context-Sensitive. In M. Moortgat, editor, *Logical Aspects of Computational Linguistics, (LACL '98)*, volume 2014 of *Lecture Notes in Artificial Intelligence*. Springer Verlag, Berlin, Heidelberg, Germany, 1998.

Marvin Minsky. Recursive unsolvability of Post's problem of 'tag' and other topics in theory of Turing machines. *Annals of Mathematics*, 74:437–454, 1961.

Luc Moritz and Daniel Valois. Pied-piping and specifier-head agreement. *Linguistic Inquiry*, 25(4):667–707, 1994.

Christian Retoré and Edward P. Stabler. Resource logics and minimalist grammars. *Research on Language and Computation*, 2(1):3–25, 2004.

John Robert Ross. *Constraints on Variables in Syntax*. PhD thesis, MIT, 1967. Published in 1986 as *Infinite Syntax*, Ablex.

Edward P. Stabler. The finite connectivity of linguistic structure. In C. Clifton, L. Frazier, and K. Rayner, editors, *Perspectives on Sentence Processing*, pages 303–336. Lawrence Erlbaum, NJ, 1994.

Edward P. Stabler. Derivational minimalism. In Christian Retoré, editor, *Logical Aspects of Computational Linguistics*, pages 68–95. Springer-Verlag (Lecture Notes in Computer Science 1328), NY, 1997.

Edward P. Stabler and Edward L. Keenan. Structural similarity within and among languages. *Theoretical Computer Science*, 293:345–363, 2003.

Willemijn Vermaat. The minimalist move operation in a deductive perspective. *Research on Language and Computation*, 2(1):69–85, 2004.