The Grammar of Code Switching

MARCUS KRACHT & UDO KLEIN

Abstract. The idea that language is a homogeneous code is a massive simplification. In actual fact, we constantly use a wide array of codes, be they other languages, dialects, registers, or special purpose codes (for bank account numbers, book numbers, shoe sizes, and so on). In this paper we provide a formal analysis of code switching.

1. Introduction

Formal linguistics studies one language at a time; hence utterances are considered to be produced with the help of a single grammar. It is of course known that this is just a simplified picture of reality. Monolingualism is certainly not the norm across the world, see [1]. In recent years multilingualism has become a new focus of research. It opens up new avenues of studying cultural and linguistic diversity; but in addition, multilingualism challenges our picture of language rather substantially. Think about the so called language faculty. If language is a relation between signifiers and signified, what is then *two* languages? It must be two such relations, for sure. But how do we distinguish them? How do we know which is which? How can and should the language faculty deal with these two languages? And if people can speak several languages, which ones are they using at a given moment? How do the interlocutors find out which one they are hearing? And how do they understand them?

As it has turned out, multilingualism is not just the command of several languages. It does not simply consist in several relations between sounds and meanings. Or in the possession of several separate language faculties. Speakers tend to mix two or even more languages in their utterances. The switch between languages may occur even within a single sentence. This is the phenomenon of *code switching*. Any community of bi- or multilingual speakers shows code switching to a great extent. We are thus led to assume that there is a uniform language faculty that is somehow metalinguistic. It seems to be—to some extent at least—independent of any particular language. In fact, we do not even have to look far to become aware that things have to be so. For there is a number of phenomena that are metalinguistic in the same sense: they make use of several languages within one utterance. One such phenomenon is borrowing: speakers use a word or phrase from another language (before it has become fully native, of course). This phenomenon is pretty widespread. Any language we know of is the result of borrowing from other languages, and the process of borrowing must begin with a phase where the borrowed words are introduced via code switching. Of course, there often is a change in form and/or meaning involved. That however is not part of the story told here.

The term "code switching" is typically reserved for the phenomenon of changing an entire language (dialect, register, etc.) in the middle of an utterance. However, in this paper we shall argue that the phenomenon is far more widespread. We shall show that ordinary discourse contains many such points at which speakers switch from one code to another; where by "code" we mean anything from a language, a dialect to a code in the original sense of the word, for example bar codes, ISBN numbers, and so on. Although it clashes somewhat with the established intuitions to use the term "code switching" in this way, we shall argue that this is because linguists do not generally consider codes as part of the language proper. However, we find it hard to draw any boundary between languages and codes. At a closer look, what is taken to be a uniform language (English, French, German and so on) in practice is a vast array of codes or—sometimes highly specialized—languages. And this is not only because there exist different dialects. Technical jargon, for example, can and should be seen as a separate language. This however calls for a thorough analysis. Dictionaries list technical usage side by side with ordinary usage as if these were just different signs belonging to the same language. However, we may alternatively think of the technical jargon as a separate code, though a very impoverished one. Normally, technical languages are too poor to even form a single sentence, and so they are always in need of being embedded in a natural language. So why think of them as separate languages?

The answer is that every code or language has some mechanism by which the code itself is being reified and maintained; this gives the code a certain unity so that it is not just an arbitrary collection of signs. While languages are self-evolving (no one is in control of what words there are and what they mean), technical jargon is the result of a specific subgroup taking care of it, nowadays even by issuing norms (ISO, DIN and so on). Also, there often are several concurrent terminological systems (for example, metrical and nonmetrical units of measurement, including different extensional meanings for the basic units such as the word "mile") and changing them arguably does not affect the language, only a specialized code.

But there is much more. When we talk about shoe sizes, bank accounts, book orders, internet communication and so on, we are often using highly specialized codes (for example IBAN, ISBN, IPv6 and much more). It is certainly not helpful to consider them part of a natural language, as their origin and functioning is really much different. Not only are speakers of English not in control of the composition of ISBN numbers, no one thinks of your competence of English as being somewhat less than native if you do not master ISBN numbers. Finally, the ISBN code is not tied to any language and can be used in any other language as well (though in spoken language the numbers may come out differently). When someone orders a book in France using ISBN numbers it would be strange to declare that he is talking English. ISBN numbers therefore look more like a code that can be imported into any given language.

All this warrants a new perspective on formal semantics. Rather than viewing our linguistic competence as mirrored by a single grammar, we should think of it as a bundle of grammars of varying complexity, with the possibility (or even necessity) of producing utterances with any number of them in parallel. This raises interesting formal questions. First and foremost, if we allow for multiple codes or grammars to be used then it becomes harder to see how we can make ourselves understood. Expressions will become ambiguous to a massive extent.

This problem is pervasive. To a certain extent it is even a necessary ingredient of our discourse. Consider a textbook of nonclassical logic. One and the same formula can be read in many ways, depending on what logic you are using. "$p \to q$" means something different to an intuitionist than to a classical logician. Not only do they disagree in what formulas are tautologies, also the meaning of these symbols is different. But if that is so, how is it possible to read this textbook? We shall show below that there are ways to rein in this ambiguity; but as always, a price must be paid.

It may be worthwhile to put the present research into perspective. We do *not* aim at dealing with code switching in the classical sense. Rather, we wish to elaborate on the possibilities that the technique of code switching itself offers, and the problems that it also incurs. Suffice it to say that there are many details to be considered about the exact way in which code switching happens (see [9] for an attempt at formalizing the underlying grammar of code switching). Our approach is therefore similar to that of Joseph Goguen, in particular the idea of algebraic semiotics, see [5]. However, the implementation is quite different. [5] does not spell out semiotic systems as relations, and keeps an overall abstract view of sign systems. The connection between signs and their meanings is provided by some morphisms. Here it is inbuilt in the notion of the sign itself. Among other things this allows to treat codes as *relations* rather than *functions* between expressions and meanings. In effect, our notion of sign is Saussurean, whereas Goguen seems to lean towards a Peircean notion. Morphisms will be introduced here as structural maps of actual sign systems (and hence cannot be used to introduce metaphor, for instance). They provide the glue for the various codes and show how to switch between them, without however altering the signification relation as such. [1]

2. Formal Prerequisites

A *language* or *code* (these two words will be used interchangeably throughout this paper) is a set L of pairs $\sigma = \langle e, m\rangle$, called *signs*, where e is called the *exponent* and m the *meaning* of σ. Hence, a language is a subset of $E \times M$, where E and M are the sets of exponents and meanings, respectively. There are no further requirements. Formal languages usually are required to satisfy that L is a partial function; that is, if $\langle e, m\rangle, \langle e, m'\rangle \in L$ then $m = m'$. However, for natural languages this does not hold.

[1]This work has been conducted as part of the project A8 of the Sonderforschungsbereich 673 "Alignment in Communication". We gratefully acknowledge the financial support of the Deutsche Forschungsgemeinschaft. We wish to thank two reviewers for useful comments.

Given two languages L and M, also $L \cup M$, $L^{\smile}$ and $L \cdot M$ are languages, where

(1)
$$\begin{aligned} L^{\smile} &= \{\langle m, e\rangle : \langle e, m\rangle \in L\} \\ L \cdot M &= \{\langle e, n\rangle : \text{ there is } m \text{ such that } \langle e, m\rangle \in L \text{ and } \langle m, n\rangle \in M\} \end{aligned}$$

Here, $L \cdot M$ is a cascade of L and M. Think, for example, of L as the numeral names of English (containing, say, the pair $\langle$`twenty-six`$, 26\rangle$) and M the binary code (containing the pair $\langle$`11010`$, 26\rangle$). Then $M^{\smile}$ is a language, where the natural numbers are the exponents standing for (= signifying) sequences of binary digits. Then $L \cdot M^{\smile}$ also is a language, relating expressions of English with binary sequences representing the same number; it contains for example the pair $\langle$`twenty-six`, `11010`$\rangle$.

A *generating system* for L is a finite set Σ of partial functions on $E \times M$ such that L is the least set closed under these functions. The *lexicon* of Σ is the subset of 0-ary functions. If g is zeroary, we have $g() \in L$. (We also write g in place of $g()$.) The lexicon is effectively a finite list of signs from L. It is easy to see that L is non-empty just in case the lexicon of Σ contains at least one element. A member of Σ that is not lexical is called a *rule*. A *grammar* is a pair $G = (\Omega, I)$ such that $\Omega : F \to \mathbb{N}$ is a finite signature (that is F is finite and Ω is a function from F to $\mathbb{N}$), and I a function assigning to each function symbol f an $\Omega(f)$-ary partial function $I(f)$ on $E \times M$. G is a *grammar for L* iff the image of F under I is a generating system for L. If G is a grammar for L, we say that L is the language generated by G and write $L = L(G)$. Terms for the signature Ω are defined as follows.

[a] A variable ξ_i, $i \in \mathbb{N}$, is an Ω-term.
[b] If $u_0, \cdots, u_{\Omega(f)-1}$ are Ω-terms and $f \in F$, then $f(u_0, \cdots, u_{\Omega(f)-1})$ also is an Ω-term.
[c] Nothing can be an Ω-term that is not produced using the two previous rules.

A term *unfolds* to a sign in the following way. Let β be a function from variables to signs. Then

[a] $[\xi_i]^\beta := \beta(\xi_i)$.
[b] $[f(u_0, \cdots, u_{\Omega(f)-1})]^\beta := I(f)([u_0]^\beta, \cdots, [u_{\Omega(f)-1}]^\beta)$.

Notice that the value does not always exist due to the partiality of $I(f)$. A *constant* term is a term without variables. For a constant term, the value does not depend on the assignment; but it need not exist. A constant term is *definite* if its value exists. Notice that it is not guaranteed that the value of a term t is in $L(G)$. For the values of variables may be outside of $L(G)$. If t is constant, though, we have $[t]^\beta \in L(G)$ on condition that t is definite.

A grammar G is *compositional* if the meaning of the sign

(2) $$I(f)(\langle e_0, m_0\rangle, \cdots, \langle e_{\Omega(f)-1}, m_{\Omega(f)-1}\rangle)$$

does not depend on the choices of e_i, $i < \Omega(f)$, for given m_i, $i < \Omega(f)$. (We ignore here certain details due to the partiality of the functions. See [7] for an extensive discussion.) G is *autonomous* if the exponent of the sign

(3) $$I(f)(\langle e_0, m_0\rangle, \cdots, \langle e_{\Omega(f)-1}, m_{\Omega(f)-1}\rangle)$$

does not depend on the choices of the m_i, $i < \Omega(f)$, for fixed choices of e_i, $i < \Omega(f)$. G is *independent* if it is both autonomous and compositional. An independent grammar can also be formulated with the help of two functions in the following way.

A *bigrammar* is a triple $B = (\Omega, I^{\varepsilon}, I^{\mu})$ such that Ω is a signature and I^{ε} a function assigning to each f a partial $\Omega(f)$-ary function on E; and I^{μ} assigns to each f a partial $\Omega(f)$-ary function on M. The grammar associated with B is the pair (Ω, J) where

$$(4) \qquad J(f)(\langle e_0, m_0\rangle, \cdots, \langle e_{\Omega(f)-1}, m_{\Omega(f)-1}\rangle) \\ := \langle I^{\varepsilon}(f)(e_0, \cdots, e_{\Omega(f)-1}), I^{\mu}(f)(m_0, \cdots, m_{\Omega(f)-1})\rangle$$

with the right hand side being defined if and only if the left hand side is defined.

3. Using Multiple Languages

The preceding definitions cover the case of a single language with a single generating system. The main challenge of using multiple codes or languages lies in the following. We do not generally use one code per utterance or text. Rather, codes may be embedded in other codes. For example, when talking about shoes we use a code for expressing shoe sizes; when talking about some topic of interest we may use a special purpose language. And, more radically, multilingual speakers may embed entire chunks of one language into another and vice versa. The same happens in computers. One can embed code of one language (say, PHP) into expressions of another language (HTML).

Several questions now arise:

[a] How is the use and the change of code signalled?
[b] How can we understand utterances composed by switching between codes?
[c] What is required of a language to be open for code switching?

Take the first problem. In its simplest form the problem is this. Consider an expression e used as an utterance. Suppose that it belongs to a single language. (Thus, no code switching takes place in it.) How is it possible to know what language is being used in uttering e? The answer may be easy: if two English speakers talk to each other, it is English. If two French people talk to each other it is French. But what if a Frenchman talks to an English person? Will he be using French or English? How can we know? [2]

Often, the choice of language can be left implicit. For a short piece of the utterance may suffice to determine what language this utterance is in. So, in presence of several languages, the algorithm is to first determine what language the utterance is in and then parse it accordingly. In the worst case we need to buffer it so as not to lose out on the initial part. In casual conversations, this is what we do and it is

[2]The first author recalls the following incident. At one conference he was asking Hans Kamp a question at the end of a lecture. However, all he got in return was a blank stare. He made one more attempt with the same result. Then, at the third time, Hans suddenly exclaimed: “Oh, you are talking German to me! I thought you were talking in French.” So his “parser” had been expecting French and could not make head nor tail from the input.

sufficient. But in general this strategy is problematic. Consider a technical jargon. Typically, jargons use a mixture of new words and existing ones. In mathematics, for example, there are plenty of uses of the words “normal”, “canonical” and so on. And they mean very different things depending on what objects we are talking about; each of the meanings is precise, but which one we have to choose depends on the context. In this case, it is not possible to judge the language from the occurring words alone, we also need to understand part of what is being said before we can make a decision which jargon we have to use. If that sounds circular then that is because *it is circular*.

It is interesting to compare this to computer science. In programming, any ambiguity should be avoided. Yet there are many programming languages and highly specialized codes. The situation is therefore quite analogous. There simply is no single language, in fact, there are dozens and dozens being used by a single machine. The solution however is different. In programming, we avoid ambiguity at all costs. Hence, whenever the machine is about to parse some code it chooses a specific one, and there are precise rules for doing that. For example, computer languages often use envelopes that contain information as to what language is being used. This information is given right at the beginning (as in executable files containing a single line indicating what shell is to be used to interpret them). Other information is provided by file name suffixes (.pdf, .js, .html, .tar and so on) that indicate to the system how to process or display the data.

Standardly, we think of the preamble as metadata. That could be for example the line “`#!bin/bash`” in a bash-shell telling the computer what program is to be used to interpret that program, or it is the line “`<?xml version=1.0?>`” in an XML-file. However, it is a very special kind of metadata. There is a reason why it is has to be given right before every other data. It shows the machine what it needs to do in order to parse the remainder. It announces the code or language used in the sequel. For this to be successful, though, the machine needs to understand how to read these files in the first place at least when it comes to the first line. There has to be some kind of metacode if you like that specializes in how to set the code in the first instance.

4. Joining two Codes

The preceding discussion showed how we can make sure a message is understood provided it belongs to the set theoretic union of the languages. However, there is also the possibility of producing messages that are entirely new, that is, that belong to none of the languages. To see how this works, we need to begin with a definition. Let $G = (\Omega, I)$ and $H = (\Psi, J)$ be grammars. The fusion, $G \oplus H = (\Omega \oplus \Psi, I \oplus J)$, is defined as follows. Let $\Omega : F \to \mathbb{N}$ and $\Psi : G \to \mathbb{N}$. Assume that F and G are disjoint (if not, change to a disjoint signature).

$$(5) \qquad (\Omega \oplus \Psi)(f) = \begin{cases} \Omega(f) & \text{if } f \in F, \\ \Psi(f) & \text{if } f \in G. \end{cases}$$

Similarly,

$$(6) \qquad (I \oplus J)(f) = \begin{cases} I(f) & \text{if } f \in F, \\ J(f) & \text{if } f \in G. \end{cases}$$

We call this the *fusion* of the two codes. Notice that the function symbols are arbitrary, so it does not matter if we rename them. It may be the case, for example, that G and H contain some symbols f and g such that $I(f) = J(g)$. If $f \neq g$ then the new grammar contains both f and g, but we still have $(I \oplus J)(f) = (I \oplus J)(g)$. If $f = g$, however, we replace f by f^1 and g by g^2, say, and then proceed as in the first case. Either way we get *two* abstract function symbols that have the same interpretation. This is however completely harmless.

The language generated by $G \oplus H$ contains $L(G)$ and $L(H)$. (It is in general *not* the union defined in [9], since we do not work with nonterminals.) However, it may contain new signs not generated by either of them. When mixing G and H we obtain a signature $\Omega \oplus \Psi$. The terms of this signature are obtained by applying symbols of Ω and Ψ indiscriminately to each other. We may look at these terms as terms in their own right, of a newly formed grammar. Or we may look at them with respect to the origin of the function symbols. Say that a term t is Ω-*headed* if it has the form

$$(7) \qquad t = f(u_0, \cdots, u_{n-1})$$

where $f \in \text{dom}(\Omega)$; and that it is Ψ-headed otherwise, i.e. when $f \in \text{dom}(\Psi)$. Say that t is a (Ω, Ψ)-*switch* (or simply a *switch* when the identity of the signature is clear) if it is Ω-headed but contains an immediate subterm that is Ψ-headed or if it is Ψ-headed but contains an immediate subterm that is Ω-headed. Switches contain an immediate subterm of a different signature. In principle, we can have switches inside switches. The terms where this does not happen, however, are of interest in their own right. A term t is called *layered* if it does not contain two switches u and v that are proper subterms of each other; that is, if u and v are subterms of t and switches, then neither is u a proper subterm of v nor is v a proper subterm of u. In that case, t results from substituting Ψ-terms for some variables into some Ω-term or Ω-terms for some variables into some Ψ-term (see [3] for a specific example). They are interesting because they allow to think of the term as belonging to a single language with certain subexpressions "imported" from some other language. This is often assumed to be the general type of expressions in code switching. But see [9].

Code switching is often done without importing a full grammar. For example, one may use some English words or Latin idioms when speaking French. Still the construction applies to these cases as well. It turns out that the fusion of two grammars produces in the worst case only the union of the languages. The resulting language of the fusion is not predictable from the languages of the individual grammars that are being fused.

Proposition 1. *$L(G \oplus H) \supseteq L(G) \cup L(H)$. Equality need not hold. Furthermore, there are grammars G, G' and H such that $L(G) = L(G')$ but $L(G \oplus H) \neq L(G' \oplus H)$.*

The proof is simple. Take $G = (\Omega, I)$, $G' = (\Omega', I')$ and $H = (\Psi, J)$, where $F = \{f_0, f_1, f_2, f_3, f_4, f_5\}$, $F' = \{f_0', f_1', f_2'\}$, and $K = \{g\}$ with $\Omega : f_i \mapsto 0$, $\Omega' : f_0', f_1' \mapsto 0, f_2' \mapsto 2$, and $\Psi : g \mapsto 0$.

$$(8)\quad \begin{array}{lcl} I(f_0)() & = & \langle \mathsf{a}, 0\rangle \\ I(f_1)() & = & \langle \mathsf{b}, 1\rangle \\ I(f_2)() & = & \langle \mathsf{aa}, 0\rangle \\ I(f_3)() & = & \langle \mathsf{ab}, 1\rangle \\ I(f_4)() & = & \langle \mathsf{ba}, 2\rangle \\ I(f_5)() & = & \langle \mathsf{bb}, 3\rangle \end{array}$$

$$(9)\quad \begin{array}{lcl} I'(f_0')() & = & \langle \mathsf{a}, 0\rangle \\ I'(f_1')() & = & \langle \mathsf{b}, 1\rangle \\ I'(f_2')(\langle e, m\rangle, \langle e', m'\rangle) & = & \langle e^\frown e', 2m + m'\rangle \end{array}$$

where the condition on e and e' is that they each contain just one letter. Finally, let $J(g)() = \langle \mathsf{c}, 2\rangle$. Then $G \oplus H$ allows to form only the terms f_i ($i < 6$) and g, and so $L(G \oplus H) = L(G) \cup L(H)$. However, $L(G' \oplus H)$ produces the sign $\langle \mathsf{cc}, 6\rangle$, which is neither in $L(G')$ nor in $L(H)$. Its term is $f_2'(g, g)$.

What this says is that the mixing of codes is determined only in part by their language; the other part is the generating system. Hence, the more abstract the generating system the more signs it may generate under fusion with some other grammar.

Here is another example to show that abstractness alone is not enough. Consider a third grammar, G'' just like G' but where $I''(f_2')$ is defined only for the signs $\langle \mathsf{a}, 0\rangle$ and $\langle \mathsf{b}, 1\rangle$ (that is, for $e, e' \in \{\mathsf{a}, \mathsf{b}\}$). Then once again $L(G'' \oplus H) = L(G'') \cup L(H)$ even though G'' shares with G' a more abstract analysis. The crucial point to note is that G'' is an extensional variant of G', where H' is defined to be an *extensional variant* of H if $L(H') = L(H)$. In addition to being an extensional variant of G', the interpretations of the function symbols in G'' is the same *when restricted to the language generated by* G' ([7]).

Proposition 2. *Let G and H be grammars. Then there are extensional variants G' and H' of G and H, respectively, such that $L(G' \oplus H') = L(G) \cup L(H)$.*

Simply take $I'(f) = I(f) \upharpoonright L(G)$ for a function symbol of G, and $J'(f) = J(f) \upharpoonright L(H)$ for a function symbol of H. The definite terms of this union $G' \oplus H'$ are either terms of G or terms of H (up to inessential variations, see below).

These facts show that for every language there is a grammar that does not tolerate any extension via code fusion except for the trivial union. Think for example of mixing ISBN numbers and shoe sizes. Both codes are "closed". The ISBN numbers are not made of parts, since an ISBN number must be of fixed length, namely 13. Thus no parts have meaning in and of themselves. The code consists only in those 13 digit numbers with a correct check digit. (The last digit is determined by the previous 12 and serves to check for the correctness of the input; this is why it is called a check digit.) Despite its simplicity, it is a very complex code since it consists only of those 13 digit numbers that have been issued. Hence, what we described is only the shape of correct ISBN numbers, not the entire code. For that we would have to add for each published book b the pair consisting of its ISBN number and b.

Definition 3. *A grammar $G = (\Omega, I)$ is called* closed *if for all f, $I(f) \restriction L(G) = I(f)$.*

Closed grammars effectively allow the use of their constructions only for those signs that are in the generated language. It is tempting to conclude that the definite constant terms for a closed grammar can only consist of function symbols of its own signature. But this is not exactly true.

Proposition 4. *Let $G = (\Omega, I)$ and $H = (\Psi, J)$. If G is closed, then for every definite constant term of $G \oplus H$ there is a definite constant term with identical value that is obtained from a Ψ-term by substituting Ω-terms for some variables.*

We cannot conclude that every definite constant term is layered, more specifically that it is the result of putting Ω-terms inside Ψ-terms. Think of the possibility that some Ω-term t and some Ψ-term u have the same value. Then we may freely substitute t and u for each other in a definite term and get a definite term. So, if t is a subterm of a definite Ω-term, we may substitute u for t and thus obtain a layered term of the other kind: an Ω-term containing some Ψ-term as a subterm. However, this is the only exception to the rule.

Definition 5. *A* spurious ambiguity *for a grammar G is a pair of constant terms t and u such that $t \neq u$ but they unfold to the same sign.*

Given this definition we can say that if G is closed, any definite term for $G \oplus H$ is *up to spurious ambiguity* a layered term, where Ψ-terms may be contained inside Ω-terms.

5. Code Identifiers

The fusion of grammars can look radically different from either of the two grammars. First, fusing two grammars can introduce massive ambiguity. We shall first study a formal case before discussing natural languages.

Consider representations of numbers in a system of base n. If $n = 2$ we speak of binary representations. Our ordinary representation is a base 10 system. For each of them we postulate a grammar $B_n = (\Omega_n, I_n)$ of the following kind. We have $\Omega_n : F_n \to \mathbb{N}$, where $F_n = \{f_0, \cdots, f_{n-1}, c_n\}$. $\Omega_n(f_i^n) := 0$ and $\Omega_n(c_n) := 2$. I_n is defined as follows. The value will be $I_n(f_i^n)() = \langle i, i\rangle$. (So we consider the digit i to be the same as the number it denotes.) Since this is uniform across the grammars, we shall drop the superscript. Furthermore, each grammar for base n numbers contains an additional binary function symbol c_n whose interpretation is as follows.

$$(10) \qquad I_n(c_n)(\langle \vec{x}, p\rangle, \langle \vec{y}, q\rangle) := \begin{cases} \langle \vec{x}^\frown\vec{y}, np + q\rangle & \text{if } \vec{y} \text{ has length 1} \\ \text{undefined} & \text{else} \end{cases}$$

Furthermore, assume that the concatenation is restricted to sequences that contain only digits with value $< n$. Consider now the sequence “`7010`”. This sequence is multiply ambiguous. Here are some terms and their values:

(11)
$$\begin{array}{lcl}
(I_{10} \oplus I_8)(c_{10}(c_{10}(c_{10}(f_7, f_0), f_1), f_0)) & = & \langle \mathtt{7010}, 7010\rangle \\
(I_{10} \oplus I_8)(c_8(c_8(c_8(f_7, f_0), f_1), f_0)) & = & \langle \mathtt{7010}, 3592\rangle \\
(I_{10} \oplus I_8)(c_{10}(c_8(c_{10}(f_7, f_0), f_1), f_0)) & = & \langle \mathtt{7010}, 5610\rangle \\
(I_{10} \oplus I_2)(c_{10}(c_2(c_2(f_7, f_0), f_1), f_0)) & = & \langle \mathtt{7010}, 290\rangle \\
(I_{10} \oplus I_8 \oplus I_2)(c_2(c_{10}(c_8(f_7, f_0), f_1), f_0)) & = & \langle \mathtt{7010}, 1122\rangle
\end{array}$$

The first term reads this as a base 10 number expressions, the second as octal. Both are also terms of the original grammars. The third reads the first two digits as belonging to a base 10 code, then adding 1 to it as if it were a base 8 expression, which gives $\langle \mathtt{701}, 561\rangle$, because in base 8 shifting means multiplying by 8, not 10. Finally, the expression is reinterpreted as a base 10 expression, and the sign computed is $\langle \mathtt{7010}, 5610\rangle$.

Undefined terms include $c_2(f_7, c_2(f_0, c_2(f_1, f_0)))$ for the reason that $I_2(c_2)$ is not defined on strings containing the digit “7”.

We see that switching between codes opens too much freedom. What can be done? The first remedy that comes to mind is to recall that the function symbols were originally disjoint, so maybe reintroducing the distinction will restrict the fusion accordingly. This is not the case. The problem is that f_1^2 denotes the same sign as f_1^8 or f_1^{10}, and so if $I_n(c_n)$ is defined on one, it is defined on the other. In other words, we create spurious ambiguities.

The next thing we can try is to change the interpretation of the function symbols. Recall the definition of a closed grammar (Def. 3). It is possible to define the codes in such a way that mixing is impossible (up to spurious ambiguity).

(12)
$$I_n(c_n)(\langle \vec{x}, p\rangle, \langle \vec{y}, q\rangle) := \begin{cases} \langle \vec{x}^\frown \vec{y}, np + q\rangle & \text{if } \vec{y} \text{ has length 1,} \\ & \quad \text{and } \langle \vec{x}, p\rangle \in L(B_n) \\ \text{undefined} & \text{else} \end{cases}$$

This has several drawbacks, however. The main one is that this is not an autonomous grammar. In order to understand whether or not we can apply the function $I_n(c_n)$ to the pairs $\langle \vec{x}, p\rangle$ and $\langle \vec{y}, q\rangle$ we cannot just look at whether we can combine $\vec{x}$ and $\vec{y}$, and—independently—whether we can combine p and q. Rather, the combination is licit only in the circumstance that $\vec{x}$ is the B_n-expression for p. This is effectively a measure to enforce layering. Though this is a perfectly viable option, we shall now turn to a solution that is much different.

The idea we shall pursue here is to *regiment the analysis term.* This is tantamount to regimenting the places of code switching. We can say, for example, that a number term may not embed a number term for a number expression of a different base. This is a homogeneity assumption. Notice that it is necessary to have such a restriction since the place of code switching is otherwise unrecoverable, unlike the case of code switching between languages where often the words reveal their language of origin.

Even so a given string is still multiply ambiguous. For example, “$\mathtt{10101101}$” can be seen as a string in base 2, 3, 8, 10 and so on. And here the regimentation is of no use: when we are writing down a string we have to change into some grammar for the number to interpret the expression. When we read a sequence we have to decide what the base is. For most languages the choice is clear: we take a base 10 grammar. However, notice that there are enough contexts in which such an expression must be read differently. In computer science, it may also be the representation in binary. Or in octal. Indeed, to guard against misunderstanding, some metainformation must be issued. One such information is to add the base at the end of the number. We write "$\mathtt{101011101}_2$" to indicate that the sequence "`101011101`" is to be read in binary; and we write "$\mathtt{101011101}_8$" to say that it is to be read in octal. And so on.

We call this digit at the end a *code identifier*. Notice that in books these code identifiers are introduced as well. There are two ways to think of them. One is as a syntactic device to signal the switches (as defined earlier). In this case they are strictly metasymbolic: there is no term that issues them, they are devices to recover the term in the first place. So, when we see the subscript "$_2$" we know that the sequence of digits is to be read in binary. However, the subscript will be thrown away. Thus, there is no term that returns the sequence "$\mathtt{101011101}_2$". All terms will yield "`101011101`". However, the code identifier excludes most terms as terms for the particular sequence.

Another way to think of them is as part of a fusion of the codes. When we introduce binary numbers we also introduce a special constant s_2. Given a term t, s_2t is another term specifying that the function symbols of t must belong to the grammar of binary expressions. Here is how this symbol is interpreted. We shall say that $I(s_2)(\langle\vec{x}, n\rangle)$ is defined only if $\vec{x}$ is the binary representation of n, and if that is the case, its value is $\langle\vec{x}, n\rangle$.

As it turns out, introducing this symbol makes the grammar strictly noncompositional. The argument is simple. Take the sequence "`10101`". It can be read among other in binary or in octal. In binary it represents 21, in octal 4161. We expect therefore that the syntactic functions $I^{\varepsilon}(s_2)$, $I^{\varepsilon}(s_8)$ must be defined on both of them. Since every natural number has a code in each of the systems, $I^{\mu}(s_2)$ and $I^{\mu}(s_8)$ must be defined on all numbers. Therefore, $I(s_2)$ as well as $I(s_8)$ are both defined on $\langle\mathtt{10101}, 21\rangle$ and $\langle\mathtt{10101}, 4161\rangle$. But that is not how they were originally defined.

We conclude the following.

> Code identifiers are in general non-compositional. They defy an integration into a unitary compositional grammar.

Actually, a similar result obtains if we pursue the following merger of these grammars. Take a single binary symbol c and let its interpretation be the following.

$$(13) \qquad I(c)(\langle\vec{x}, p\rangle, \langle\vec{y}, q\rangle) := \begin{cases} \langle\vec{x}^\frown\vec{y}, pn + q\rangle & \text{if } \vec{y} \text{ has length 1 and } p \text{ is the base } n \\ & \quad\text{value of } \vec{x} \\ \text{undefined} & \text{otherwise} \end{cases}$$

This allows to consider all base *n* grammars as one single grammar with one function to represent the addition of a digit; the price however is loss of compositionality.

We conclude the following. If we want to have code identifiers and maintain compositionality there is no way around the regimentation of analysis terms. In other words, one must assume that switching points are subject to regimentation at the level of structural analysis. [3]

6. Code Embedding

There is a special case to be considered, namely when the first code allows to embed expressions of the second code but requires them to be transformed in a special way. In that case the expressions of the second code do not appear in their original form but in a somewhat distorted form. An example is provided by words borrowed into a language with a different syllabic structure. Japanese for example does not allow branching onsets and generally prefers CV syllables. In order to import words from other languages, Japanese speakers inserts vowels to make them pronounceable (“`miruku`” ‘milk’, “`arubeitu`” ‘short term job’, from German “`Arbeit`” ‘work’).

Another example is regular expressions. Many programming languages offer to use regular expressions but do not themselves resolve them. Rather, whatever they are being used for, the expressions are being passed on to the host system which uses its own mechanism for regular expressions. (The programmers obviously do not like to double their effort by providing the same functionality again in more or less the same way.) For that to work, a regular expression must be tunneled through the syntax of the host language so that it can be used by the host system. For example, in the standard language for regular expressions (see [4]), the period is a special metacharacter that matches any alphabetic symbol. It must be distinguished from the alphabetic character “period”. To do that, the latter is written using a backslash as a general escape character: “`\.`”. However, regular expressions are built from strings in OCaml, and the syntax of strings also uses the backslash as an escape character. If we were to write just “`\.`”, OCaml would understand that we pass on the alphabetic character ‘escaped period’, which does not exist. But we want the actual *sequence of two characters “backslash” followed by “period” to be communicated*. Hence, to make the backslash survive this process, we need to write “`\\.`”. A somewhat different mechanism is used in Python. In a similar manner, think of XPath expressions being passed on to XSLT. The symbols for less-than and greater-than have such a special meaning to the XML parser that it is often necessary to ‘protect’ them by writing “`&lt;`” instead of “<” and “`&gt;`” for “>”, lest they are misunderstood as being part of a tag. Once you know what to look for, this phenomenon suddenly becomes pervasive.

[3]One reviewer rightly pointed out that there are explicit ways to regiment code switching. Consider phrases like “For the rest of this paper X will be assumed to denote ...”. This is clearly meant to remove all ambiguity as to whether certain code switches may take place. Thus, languages contain more than just code identifiers. There are also expressions that could be called “switching regulators”.

What is characteristic of code embedding is that the host code does not provide any mechanism to interpret the expressions of the embedded code; they are passed on to the embedded code for interpretation. The process of embedding is rather widespread. We mention here the fact that in logic variables are usually thought of as being single expressions of the form "p_i", where i is a so-called index. Indices are not further analyzed. However, on further inspection it is assumed that indices are some kind of number expression. There are of course several; the choice of the code for number expressions is however considered immaterial for the 'main code'. Moreover, some transparent indication of code embedding is used such as writing the number as an index or a superscript.

Often, the embedding of code takes a harmless form. In HTML, embedding PHP code is done by wrapping it as follows. Begin with "`<?php`" followed by the undistorted PHP code, and end with "?>". Sometimes one can even toggle between two codes. This is exemplified by OCamlDuce, which integrates CDuce and OCaml and uses double set braces ("{{" and "}}") to mark the switch between codes.

7. Code Flexibility

It is known that codes change, either by explicit ruling or by tacit convention. Languages evolve not only via sound changes but also by constant innovation. One such innovation is the creation of special terminology. In mathematics it is customary to define new terms for example to create shorthands for complex concepts. This means that codes and grammars are actually fluid rather than static. A reflection of this is the fact that predicate logic is actually not a single language but rather a scheme to define an infinity of languages. Here we shall not discuss change in time of a given code or grammar. Rather, we shall focus on the mechanism of explicitly changing between grammars on need.

Formally, define a *morphism of grammars* $G = (\Omega, I)$ and $H = (\Psi, J)$ to be a map ι, where $\iota : F \to G$ is such that for all $f \in F$, $\Psi(\iota(f)) = \Omega(f)$ and $J(\iota(f)) = I(f)$. So, the signature of G is translated into the signature of H in such a way that the interpretation of the symbols remains the same. An example is the expansion of a language by some constants or some function symbols. Expanding a language in this way is commonly considered in logic, for example. It is vital that the expansion does not change the interpretation of existing symbols. Grammars form a category $\mathcal{G}$ with these morphisms. The fusion turns out to be the product. A more general construction is the so-called pushout construction, where the grammars share a fragment. (If that fragment is the empty grammar, we get the product.) If G and H share a language K, that is, if we have morphisms $\mu : K \to G$ and $\nu : K \to H$, then the pushout of the embeddings is well-defined, and it consists in a signature which is like $G \oplus H$ except that no distinction is made between modes g and h if there is an f such that $\mu(f) = g$ and $\nu(f) = h$.

(14)
$$\begin{tikzcd}
K \arrow[r, "\mu"] \arrow[d, "\nu"] & G \arrow[d, "!"] \\
H \arrow[r, "!"] & G \oplus_K H
\end{tikzcd}$$

(This diagram says that there is a unique map from G to $G \oplus_K H$, say α, and a unique map from H to $G \oplus_K H$, call it β, such that the diagram "commutes", that is, such that $\alpha \cdot \mu = \beta \cdot \nu$.) This generalization would allow to define the grammars for B_n along a unified set of constants f_i, since their values are the same in grammars wherever they exist in the signature. Suppose O is the empty signature, ie $O = (\varnothing, \varnothing)$. Then for each grammar $G = (\Psi, J)$ there is a unique map $\varepsilon : O \to G$, namely the empty map. This is the initial object in the category $\mathcal{G}$. Then, as indicated above, $G \oplus H \cong G \oplus_O H$, which is to say that they are isomorphic.

The basic scenario is that natural discourse occasionally involves changing the grammar by incorporating new elements or dropping them. A simple mechanism is that of baptism. We find an individual, a particular object or kind and name it. That naming expands the grammar by a new symbol whose meaning is fixed by the act of initial baptism. Whenever that symbol is used, it is used with than meaning *until it is retracted from the grammar.*

This scenario allows to shed light on a problem noted in [8]. Roughly, it is argued there that the semantics of propositional logic and predicate logic cannot be compositional since variables are ambiguous and there is no guarantee that two occurrences of the same variable will have to be given the same value (see also [2] for a similar complaint). For example, if "p" is a genuine variable, "p→p" will not be a tautology, for the values of the two occurrences of "p" are independent of each other, as they are ambiguous between having value 0 and having value 1. This is of course highly problematic since it undermines the usefulness of variables. (Notice that there are interesting connections to 3-valued logic of uncertainty or undefinedness. In Łukasiewicz' logic, the value of "p→q" and "p→p" are the same, namely u (= uncertain), when the value of "p" and "q" are both u as well. However, from a supervaluation point of view, the value of "p→p" should be 1. Evidently, the semantics has no use for the fact that the two occurrences of the variable "p" are coordinated.) In [8] it was proposed to treat variables as schematic expressions for constant propositions. They are therefore metalinguistic expressions, to be filled by actual expressions. Since actual expressions cannot use variables, they must be constant. This means that the logic becomes an artefact of the expressibility of the underlying language, not of the actual semantics. For the semantics determines a potentially larger range for the variables than can be covered by the constant expressions of the language. More precisely, it was proposed that the propositional variables function as indicators of the schematic term into which we substitute constant terms for term variables. Thus "p→p" just indicates that the actual term is $f_\to(\xi_0, \xi_0)$, say, where $f_\to$ is the function symbol of implication-formation and ξ_0 a variable over constant Ω-terms.

The solution proposed here is the following. Variables are schematic over *potential constant expressions*. Thus we consider the possibility to extend the language by some constant, say "c", whose interpretation can be anything we please, but only until that constant is introduced into the language. Once it is there, its value is fixed. This constant is therefore introduced, and "p" is replaced by that constant throughout. The formula "p→p" now becomes "c→c". There is then no risk of ambiguity, as the constant will have one and only one value, by definition. The crux is that the value of "c" could have been anything. So we say that "p→p" is true if, regardless of the constant expansion we choose, and regardless of the constant we use to replace "p" by, the resulting formula is true. It is interesting to note that this is the way in which variables are treated in functional programming (see [6]). In XSLT, you can assign a value to a variable only once. That value cannot be changed. One needs to first wait until the scope of the variable is closed (upon which the variable name is discarded) and then reintroduce the variable with another value. In XSLT, therefore, the difference between variables and constants has been obliterated, as is the case here. Or rather, XSLT knows no variables in the standard sense of the word.

Viewing it this way, however, raises new issues. First, if variables are schematic over languages yet to be built, there is no grammar to parse expressions containing them. Hence, it is easier to think of variables as *arbitrary constants*. That is, introducing a variable means introducing an array of languages, where in each variant the variable is interpreted differently. If the domain is the natural numbers, for example, one language may interpret "x1" as 7, another as 11, and so on. The language thus assimilates the variables into constants, so to speak. However, the difference between variables and constants shows up in the fact that for variables we can choose between languages that give them different values, while for constants this is not the case.

8. Coordination

The present discussion sheds some more light on the problems highlighted in [2]. Fine notes that whenever we use a particular name for a variable we must make it clear what other occurrences of the same variable name it is to be coordinated with. This problem is pervasive. Think of a database or a program full of formulae and expressions that use certain variable names. It becomes clear that we cannot assume that two occurrences of the same name are coordinated. However, there appear to be rules governing the interpretation. Some of them are explicit such as the rules of quantification. Others are rather implicit, as in the case of two formulae versus one. Ordinarily, the occurrences of the same propositional variable in a given formula are coordinated (we are assuming no quantification over propositional variables). However, whether or not two occurrences of the same variable in *different* formulae are coordinated is a matter of convention. We do agree, for example, that in a statement such as "p;p→q⊢q", the variables must be coordinated. In the present schema this says that there may be no code switching inside this expression. In interpreting it, we choose a single grammar which replaces the variables by some constants and interpret that expression. Outside these constituents, however, we are free to proceed to whichever grammar we like. It is in this way that another occurrence of the variable "`p`" can take on a different interpretation. When one tracks the use of variables in mathematical texts, one becomes acutely aware that there are subtle signals which give away what variables must be freed when. The regime of variable is very much as in functional programming (see [6]). In XSLT, a variable once defined must keep its value for the rest of its lifespan. The lifespan is defined by the constituent it occurs in. Once the constituent is complete, the variable disappears together with its value.

9. Openness of Language

Let us briefly return to natural languages. The reality of grammar (in the sense of syntax) and its independence of semantics has been stressed by Chomsky over and over. What this means is that grammars should be autonomous in the sense above. Furthermore, semanticists often make compositionality their priority. Hence we wind up with the condition that grammars are independent, or better, that we have bigrammars. Each function symbol is interpreted on the expression side and on the meaning side independently. The rationale is that as experiments have shown (for example the so-called wug-test) it is possible to introduce new words into a language even without meaning and ask what the plural of them is. Similarly, given a concept of some sort, we can ask what it means for there to be several of them independently of how we would call them. In sum, plural formation appears to be the combination of *two* functions, one for the form and the other for the meaning.

A particular application that comes to mind is the problem whether to think of Malay as a context free language (we owe this problem to Alexis Manaster-Ramer). In Malay, to form the plural you simply reduplicate the noun. So, "`anak-anak`" means "children", because "`anak`" means "child". This sounds as if Malay really must be a copying language of some sort. But since Malay has only finitely many roots, we may as well list all the plural forms independently. Thus, doing the wug-test in Malay will provide some evidence that plural formation in Malay is some function that is defined on more input than the language provides in and of itself. And it may provide evidence that that function really is reduplication. Thus, languages are *open* as opposed to closed. This allows them to incorporate new expressions.

It is however quite another matter for Malay to be open and to *actually* take in a new word. Think of a Malay speaker introducing the English word "`car`" into the language. What will be its plural? There is no unique answer. On the face of it we would expect it to be "`car-car`", but it could also be "`cars`". If it is the latter, it does not mean that the system of Malay is compromised, as long as we think of both words as idioms, that is, as long as we do not attempt to derive "`cars`" from "`car`" in Malay. This is exactly the situation of learned vocabulary items in English (and German). The plural of "`formula`" is sometimes given as "`formulae`" (as it is in Latin) or as "`formulas`" — the regular form in English.

That it can be the latter indicates that plural formation in English does not have to be restricted to its own nouns (using the wug-test that has generally been shown). This openness is the key to the flexibility of languages. However, that languages can follow potentially several trajectories speaks to the freedom of humans who actually make the languages in the first place. Humans are free to choose and form the languages that they speak. In fact, if we are correct, this freedom gets exercised far more often than one is tempted to think.

10. Conclusion

Viewing our communicative powers as not defined by a single language with a fixed grammar, but rather by a multitude of changing and interleaving grammars and languages brings us closer to the observed reality. There are formal tools to deal with this situation. They allow to solve some troubling aspects of semantics, namely how to deal with variables.

As it stands, though, the new approach raises many new questions. Breaking the unity of the code, so to speak, forces us to study the metalinguistic faculty in much more detail. We can no longer rely on the language to be an expression of a fixed code given in advance, but have to make use, often in a multiply interleaving way, of many different codes. There must therefore be rules of when to choose what code. This is what the literature on code switching tries to answer if only for the interleaving use of different human languages. Moreover, as we have seen, the fusion of codes may result in various different codes depending on the generating system we choose for the individual languages. This is a substantial point. It shows that productivity of language can also be understood in a novel way: it can integrate other languages in such a way so as to produce expressions that can be found in neither of the individual languages. Which ones it will produce however is not determined by the individual languages alone, only by their grammars. We expect therefore that different individual grammars of speakers will give rise to different code switching behaviour, for example.

References

[1] Nicholas D. Evans. *Dying Words. Endangered Languages and What They Have To Tell Us*. Wiley-Blackwell, Oxford, 2010.

[2] Kit Fine. *Semantic relationism*. Blackwell, London, 2007.

[3] Marcello Finger and Dov M. Gabbay. Adding a Temporal Dimension to a Logic System. *Journal of Logic, Language and Information*, 1:203–233, 1993.

[4] Jeffrey E. F. Friedl. *Regular Expressions*. O'Reilly, 2002.

[5] Joseph Goguen. An introduction to algebraic semiotics, with an Application to User Interface Design. In C. Nehaniv, editor, *Computation for Metaphors, Analogy and Agents*, number 1562 in Springer Lecture Notes in Artificial Intelligence, pages 242–291. 1999.

[6] Michael Kay. *XSLT 2.0 and XPath 2.0. A Programmer's Reference*. Wrox, 4 edition, 2008.

[7] Marcus Kracht. *Interpreted Languages and Compositionality*. Number 89 in Studies in Linguistics and Philosophy. Springer, 2011.

[8] Marcus Kracht. Are Logical Languages Compositional? *Studia Logica*, 2013.

[9] David Sankoff and Shana Poplack. A Formal Grammar for Code Switching. *International Journal of Human Communication*, 14:3–45, 1981.

Fakultät Linguistik und Literaturwissenschaften, Universität Bielefeld, Postfach 100131, D-33501 Bielefeld, Germany, {udo.klein,marcus.kracht}@uni-bielefeld.de