Minimalist Grammars in the Light of Logic

Sylvain Salvati

Minimalist Grammars in the Light of Logic

Sylvain Salvati

Abstract

In this paper, we aim at understanding the derivations of minimalist grammars without the shortest move constraint. This leads us to study the relationship of those derivations with logic. In particular we show that the membership problem of minimalist grammars without the shortest move constraint is as difficult as provability in Multiplicative Exponential Linear Logic. As a byproduct, this result gives us a new representation of those derivations with linear λ-terms. We show how to interpret those terms in a homomorphic way so as to recover the sentence they analyse. As the homorphisms we describe are rather evolved, we turn to a proof-net representation and explain how Monadic Second Order Logic and related techniques allow us both to define those proof-nets and to retrieve the sentence they analyse.

Since Stabler defined Minimalist Grammars[1] (MGs) as a mathematical account of Chomsky's minimalist program, an important effort of research has been dedicated to give a logical account of them. MGs use a feature checking system which guides the derivations and as those features behave similarly to resources, it seemed possible to represent those derivations in some substructural logic. There have been a lot of propositions (among others [LR99], [LR01], [Lec03], [Lec04], [Amb07]), but we do not think that any of them establishes a relation between MGs and logic in a satisfactory way. These propositions are, in most cases, describing a way of building proofs in a certain logic so as to describe minimalist derivations. But they cannot be considered as a logical account of minimalist derivations since they use extra and non-logical constraints that rule out proofs that would not represent a minimalist derivation. Those propositions solve nevertheless some problem that is inherent to Stabler's formalism. Indeed, in Stabler's formalism, the derivations are ambiguous in the sense that they can be interpreted into different sentences that have different meanings. Thus when dealing with semantic interpretations, one needs to interpret derivations both syntactically and semantically so as to build the syntax/semantic relation.

In the present paper, we give a logical account of minimalist grammars as proofs in Multiplicative Exponential Linear Logic (MELL) [Gir87]. We claim that this account is *accurate* and *of logical nature* for two reasons; first, because we prove that the membership problem for minimalist grammars is Turing-equivalent to provability in MELL; second, we define minimalist derivations

[1] Across the paper, unless stated otherwise, when we refer to Minimalist Grammars, we are referring to Stabler's Minimalist Grammars **without** the Shortest Move Constraint.

as being all the proofs of a particular sequent (in the article we actually give the equivalent presentation in terms of closed linear λ-terms of a certain type). Nevertheless, even though linear logic is dealing with resources, in our approach, linear logic is not modeling the fact that features are treated as resources in MGs. While this is somehow a defect of our approach, it shows that MGs' derivations are dealing with other kinds of resources that we call *moving pieces*, but which correspond to the linguistic notion of *traces*.

The idea that has motivated this work is an idea that is not so wide-spread in the community of computational linguistics. It consists in making a clear distinction between derivations and surface realisations. This idea can be traced back to Curry [Cur61], but is very common in compiling and has been recently reintroduced in computational linguistics by the works of Muskens [Mus01] and de Groote [dG01]. So we start thinking about Minimalist Grammars only from the point of view of their derivations, trying to find a good representation and to understand how to build them. We continue by studying how to retrieve the surface form, or the string, that the derivation is analyzing. This step is harder than one would expect, but it also shows an interesting feature. Indeed, we have not been able to find a way to interpret our derivations without the use of a context which is quite similar to the context that de Groote proposes for semantics [dG07]. Finally since, so as to find a more satisfactory way of reading sentences out of derivations, we turn to Formal Language Theory and use techniques related to Monadic Second Order Logic (MSO). This leads us to a fairly simple account of both the structure of derivations and the way of interpreting them.

The paper is organized as follows. In section 1 we introduce linear λ-calculus and minimalist grammars. We show that the languages defined by minimalist grammars are closed under intersection with recognizable sets. This allows us to prove the Turing-equivalence of the emptiness problem and of the membership problem for MGs. In section 2 we show that the emptiness problem for MGs is Turing-equivalent to provability in MELL. We proceed in two steps, first we show that the emptiness problem for a particular class of automata, k-VATA, can be reduced to the emptiness problem for MGs. As the emptiness problem for k-VATA is as difficult as provability in MELL, this reduces provability in MELL to the emptiness of MGs. Second, we show an encoding of minimalist derivations as linear λ-terms and we study some consequences of that encoding. Section 3 shows how the representation of minimalist derivations as linear λ-terms can be interpreted into sentences and the limitations of this interpretation. Then section 4, tries to overcome those limitations with Monadic Second Order Logic. Section 5 gives some conclusions on this work.

1 Preliminaries

In this section we introduce two technical notions the linear λ-calculus and minimalist grammars. The λ-calculus has been introduced so as to define a theory of functions. But it captures the notion of binding and has therefore

been extensively used in formal semantics of natural languages. For the syntax of natural languages the linear λ-calculus can be seen as a representation of deduction of type logical grammars via the Curry-Howard isomorphism. A more explicit and systematic use of the linear λ-calculus in syntax is proposed by the Abstract Categorial Grammars [dG01] and the λ-grammars [Mus01]. The interest of linear λ-calculus in modeling syntax is that it naturally extends the notion of syntactic tree by providing it with the possibility of representing *traces* with linear λ-abstraction. So even though there seems *a priori* to be very little relationships between minimalist grammars and linear λ-calculus, they can at least be related by the fact that traces occupy a central position in minimalist grammars and that linear λ-calculus offers the possibility to represent traces.

1.1 Linear λ-calculus.

We now present the linear λ-calculus. Linear types are built from a given finite set of atoms A by using the infix operator $\multimap$. The set of types built from A, $\mathcal{T}_{\multimap}(A)$, is constructed according to the following grammar:

$$\mathcal{T}_{\multimap}(A) ::= A | (\mathcal{T} \multimap \mathcal{T})$$

We adopt the convention that $\multimap$ associates to the right and that $\alpha_1 \multimap \cdots \multimap \alpha_n \multimap \beta$ represents the type $(\alpha_1 \multimap \cdots (\alpha_n \multimap \beta) \cdots)$. As usual $ord(\alpha)$, the order of a simple type α of $\mathcal{T}_{\multimap}(A)$, is defined to be 1 when α is atomic (*i.e.* α is an element of A), and $\max(ord(\alpha_1)+1, ord(\alpha_2))$ when $\alpha = \alpha_1 \multimap \alpha_2$.

A higher order signature Σ, is a triple (A, C, τ) where A is a finite set of atoms, C is a finite set of constants and τ is a function from C to $\mathcal{T}_{\multimap}(A)$. A signature is said of n^{th} order if $\max_{c \in C}(ord(\tau(c)) \leq n$. We use a type system *à la Church*, which means that variables explicitly carry their types. We adopt the notation x^{α}, to specify that x is a variable of type α. The family $(\Lambda_{\Sigma}^{\alpha})_{\alpha \in \mathcal{T}_{\multimap}(A)}$ is defined by:

1. $c \in \Lambda_{\Sigma}^{\tau(c)}$ when $c \in C$,
2. $x^{\alpha} \in \Lambda_{\Sigma}^{\alpha}$,
3. $(t_1 t_2) \in \Lambda_{\Sigma}^{\alpha}$ if $t_1 \in \Lambda_{\Sigma}^{\beta \multimap \alpha}$, $t_2 \in \Lambda_{\Sigma}^{\beta}$ and $FV(t_1) \cap FV(t_2) = \emptyset$,
4. $\lambda x^{\beta}.t \in \Lambda_{\Sigma}^{\alpha \multimap \beta}$ if $t \in \Lambda_{\Sigma}^{\beta}$ and $x^{\beta} \in FV(t)$.

where $FV(t)$ is the set of free variables (defined as usual) of t. The λ-terms that are in $\Lambda_{\Sigma}^{\alpha}$ are said *linear*, because a variable may at most have one free occurrence in a term and because every bound variable has exactly one free occurrence below the λ that binds it.

When they are not relevant or when they can easily be infered from the context, we will omit the typing annotations on the variables. We also use the convention that $t_0 t_1 \ldots t_n$ denotes the term $(\cdots (t_0 t_1) \cdots t_n)$ and that $\lambda x_1 \ldots x_n.t$ denotes the term $\lambda x_1 . \ldots \lambda x_n . t$. We take for granted the notions of α-conversion,

β-contraction and η-contraction. We always consider λ-terms up to α-convertibility, and we respectively write (with $\gamma \in \{\beta; \eta; \beta\eta\}$) $\rightarrow_\gamma$, $\stackrel{*}{\rightarrow}_\gamma$, $=_\gamma$, the relation of γ-contraction, γ-reduction and γ-conversion. A term is *closed* when its set of free variables is empty.

Contexts are λ-terms with a hole which are written $C[]$. The operation of grafting a term N in the hole of a context $C[]$ is written $C[N]$. For example, if $C[] = \lambda x.[]$ and $N = x$ then $C[N] = \lambda x.x$.

The linear λ-calculus is a conservative extension of the notion of ranked trees. A signature $\Sigma = \langle A, C, \tau \rangle$ is said to be a *tree signature* when $A = \{o\}$ and for all $c \in C$, $\tau(c)$ is of the form $o \multimap \cdots \multimap o \multimap o$. We will in general write $o^n \multimap o$ for the type $\underbrace{o \multimap \cdots \multimap o}_{n\times} \multimap o$ (when $n = 0$, $o^n \multimap o$ simply denotes o). Trees are then denoted in the obvious way by closed linear λ-terms of type o in normal form. Tree signatures may also be called *ranked alphabets*. We may denote with $\Sigma^{(n)}$ the set of constants declared in Σ which have type $o^n \multimap o$. If $\Sigma_1 = \langle \{o\}, C_1, \tau_1 \rangle$ and $\Sigma_2 = \langle \{o\}, C_2, \tau_2 \rangle$ are two ranked alphabet such that $C_1 \cap C_2 = \emptyset$ we write $\Sigma_1 \cup \Sigma_2$ to refer to the ranked alphabet $\langle \{o\}, C_1 \cup C_2, \tau_1 \cup \tau_2 \rangle$. A *multi-sorted tree signature* or a *multi-sorted ranked alphabet* is simply a second order signature. When we deal with ranked trees, we assume that they are represented by linear λ-terms in normal form and we represent a subtree of a tree t as a pair $(C[], v)$ such that $C[v] = t$.

1.2 Minimalist Grammars

A *minimalist grammar* G is a tuple (V, B, F, C, c) where V is a finite set of *words*, B is a finite set of *selection features*, F is a finite set of *licensing features*, C is a *lexicon* (a finite set of *lexical entries* that are defined below) and $c \in B$. Features are used in two different forms, a positive form and a negative form. The positive form of a selection feature b (*resp.* licensing feature f) is denoted by $=b$ (*resp.* $+f$) while its negative form is denoted by b (*resp.* $-f$). The set of positive (*resp.* negative) features of G will be denoted by B^+ and F^+ (*resp.* B^- and F^-).

The elements of C, the *lexical entries*, are pairs (v, l) where $v \in V \cup \{\epsilon\}$ and l is a string built using symbols taken from $B^- \cup B^+ \cup F^- \cup F^+$. These strings are not arbitrary, they have a special structure, they are of the form $l_1 a l_2$ where:

1. $a \in B^-$,
2. l_1 is a string (possibly empty) of elements taken from $B^+ \cup F^+$ and which must start by an element of B^+ when it is not empty,
3. l_2 is a string (possibly empty) of elements taken only from F^-.

The set of *feature suffixes* of G is the set $\mathrm{Suff}(G) = \{l_2 | \exists (v, l_1 l_2) \in C\}$, the set of *moving suffixes* of G is the set $\mathrm{Move}(G) = \{l \in \mathrm{Suff}(G) | l \in (F^-)^*\}$. A *lexical construction* is a pair (w, l) such that $w \in V^*$ and $l \in \mathrm{Suff}(G)$; a *moving piece* is a lexical construction (w, l) such that $l \in \mathrm{Move}(G)$.

The *derivations* of minimalist grammars G are defined on a tree signature of the form: $\mathrm{Der}(G) = (\{o\}, \{\mathbf{merge}; \mathbf{move}\} \cup C, \rho)$ where $\rho(\mathbf{merge}) = o \multimap o \multimap o$, $\rho(\mathbf{move}) = o \multimap o$ and $\rho(c) = o$ when $c \in C$. The set of trees that can be built on $\mathrm{Der}(G)$ will be written $d(G)$.

In order to produce the strings that derivations are representing, we use a transformation $\mathcal{H}$ that interprets the elements of $d(G)$ as pairs $\langle s, L\rangle$ where:

1. s is a lexical construction, *the head* of the derivation, and,
2. L is a finite multiset of moving pieces.

We consider multisets built on a set A as functions from A to $\mathbb{N}$. Such a multiset L is said finite when $\sum_{a\in A} L(a)$ is finite. Given a and a multiset L we say that a has $L(a)$ occurrences in L. We will confuse a and the multiset L_a which contains one occurrence of a and no occurrence of elements different from a. We write $\emptyset$ to denote the multiset which contains no occurrence of any element. Given two multisets L_1 and L_2 we write $L_1 \cup L_2$ the multiset such that $(L_1 \cup L_2)(a) = L_1(a) + L_2(a)$. We may represent finite multisets L with a *list* notation $[e_1, \ldots, e_n]$, with the understanding that for each a there is exactly $L(a)$ e_i that are equal to a . The fact that we use multisets of moving pieces is a first hint for understanding the relation between MGs and MELL. Indeed, contexts of hypotheses in MELL are best represented as multisets of formulae.

The transformation $\mathcal{H}$ is defined as follows:

1. $\mathcal{H}(\mathbf{merge}\, t_1\, t_2) = \langle (w_2, l_2), (w_1, l_1) \cup L_1 \cup L_2\rangle$ if $\mathcal{H}(t_1) = \langle (w_1, al_1), L_1\rangle$, l_1 is not empty and $\mathcal{H}(t_2) = \langle (w_2, {=}al_2), L_2\rangle$,
2. $\mathcal{H}(\mathbf{merge}\, t_1\, t_2) = \langle (w_1w_2, l_2), L_1 \cup L_2\rangle$ if $\mathcal{H}(t_1) = \langle (w_1, a), L_1\rangle$, $\mathcal{H}(t_2) = \langle (w_2, {=}al_2), L_2\rangle$ and t_2 is not an element of C
3. $\mathcal{H}(\mathbf{merge}\, t_1\, t_2) = \langle (w_2w_1, l_2), L_1 \cup L_2\rangle$ if $\mathcal{H}(t_1) = \langle (w_1, a), L_1\rangle$, $\mathcal{H}(t_2) = \langle (w_2, {=}al_2), L_2\rangle$ and t_2 is an element of C
4. let's assume $\mathcal{H}(t_1) = \langle (w, {+}al), (w', {-}al') \cup L\rangle$ then,
$$\mathcal{H}(\mathbf{move}\, t_1) = \begin{cases} \langle (w, l), (w', l') \cup L\rangle \text{ when } l' \text{ is not empty} \\ \langle (w'w, l), L\rangle \text{ otherwise} \end{cases}$$
5. in the other cases $\mathcal{H}(t)$ is undefined.

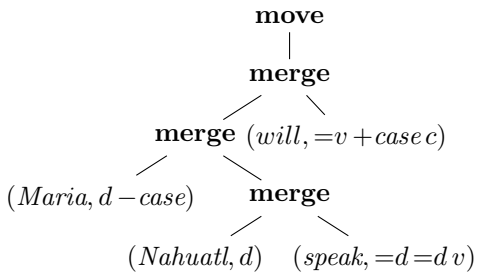
In this way G defines two languages:

1. the language of its derivations $\mathcal{D}(G) = \{t | \mathcal{H}(t) = \langle (w, c), \emptyset\rangle\}$,
2. the string language $\mathcal{L}(G) = \{w \in V^* | \exists t. \mathcal{H}(t) = \langle (w, c), \emptyset\rangle\}$

Example 1 *In the course of this paper, we will use an example adapted from [Sta97] with a grammar using the following lexical entries:*

$$(\mathit{Maria}, d - \mathit{case}), (\mathit{speak}, {=}d\, {=}d\, v), (\mathit{will}, {=}v\, {+}\mathit{case}\, c), (\mathit{Nahuatl}, d)$$

With this grammar we can give an analysis of the sentence Maria will speak Nahuatl *with the derivation t that is represented by the following term:*



We here give the details of the computation of $\mathcal{H}(t)$*:*

1. *let* $u_1 = \mathbf{merge}\,(Nahuatl, d)(speak, {=}d\,{=}d\,v)$ *then (case 3 of the definition)*

$$\mathcal{H}(u_1) = \langle (speak\ Nahuatl, {=}d\,v), \emptyset \rangle$$

2. *let now* $u_2 = \mathbf{merge}\,(Maria, d-case)\,u_1$ *we have that (case 1 in the definition)*

$$\mathcal{H}(u_2) = \langle (speak\ Nahuatl, v), [(Maria, -case)] \rangle$$

3. *let* $u_3 = \mathbf{merge}\,u_2\,(will, {=}v\,{+}case\,c)$ *and then (case 2 of the definition)*

$$\mathcal{H}(u_3) = \langle (will\ speak\ Nahuatl, {+}case\,c), [(Maria, -case)] \rangle$$

4. *finally (case 4 of the definition)* $\mathcal{H}(t) = \langle (Maria\ will\ speak\ Nahuatl, c), \emptyset \rangle$

An element t from $d(G)$ is said to satisfy the Shortest Move Constraint (SMC) if $\mathcal{H}(t)$ is defined and is of the form $\langle s, L \rangle$ where for each licensing feature f of G there is at most one occurrence in L of a moving piece of the form $(w, -fl)$. A term t is said to hereditarily satisfy the SMC when t and each of its subterm satisfy the SMC (we write that t is HSMC).

With the SMC, G defines two languages:

1. the language of its SMC-derivations $\mathcal{D}_{SMC}(G) = \{t | \mathcal{H}(t) = \langle (w, c), \emptyset \rangle$ and t is HSMC$\}$,
2. the string SMC-language $\mathcal{L}_{SMC}(G) = \{w \in W^* | \exists t \in \mathcal{D}_{SMC}(G).\mathcal{H}(t) = \langle (w, c), \emptyset \rangle\}$

In the general case (with derivations that do not satisfy the SMC), the mapping $\mathcal{H}$ cannot be seen as a homomorphism. Indeed, the interpretations of **merge** or **move** via $\mathcal{H}$ lead to functions which have to inspect their arguments in order to possibly compute a result. Moreover, the interpretation of **move** is not deterministic, since one can pick any element in the multiset of moving pieces which exhibits the required feature. There would be an easy way of turning $\mathcal{H}$ into a homomorphism, simply by:

- distinguishing the domains in which elements of $\mathcal{C}$ and complex expressions are interpreted by $\mathcal{H}$

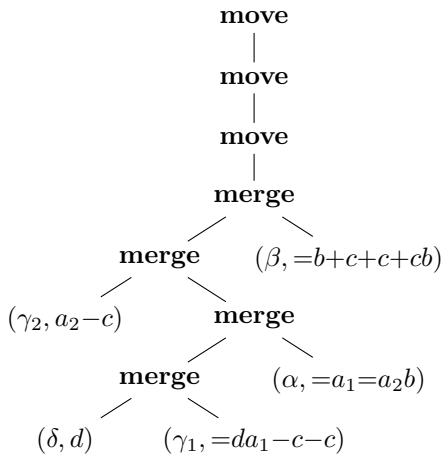
- interpreting the term t as the set of pairs $\langle s, L\rangle$ in which $\mathcal{H}$ can interpret t

But, this technique does not help to grasp any interesting criterion, apart from the actual computation, that allows to understand in which cases $\mathcal{H}(t)$ gives an actual result. Thus this presentation of minimalist grammars does not give a satisfactory account of the mathematical nature of the derivations and can be seen as an algorithm that computes derived structures. Another reason why this technique of turning $\mathcal{H}$ into a homomorphism is not worthwhile is that in general minimalist grammars are not only concerned with syntax but also with the interface between syntax and semantics. And when $\mathcal{H}(t)$ outputs several results, these results should in general be put in relation with different semantic representations. Therefore, derivation terms do not determine completely the relation between syntax and semantics, this relation really depends on the actual computation $\mathcal{H}$ is doing on the terms. So that understanding the mathematical nature of the derivations of minimalist grammars should lead to the definition of derivations on which it would be both possible to check easily whether they denote a correct syntactic object and to relate this syntactic object uniquely to some semantic representation.

Example 2 *We here give an artificial example of an ambiguous derivation. We use a grammar with the following lexical entries:*

$$(\alpha, {=}a_1{=}a_2 b), (\beta, {=}b{+}c{+}c{+}cb), (\gamma_1, {=}da_1{-}c{-}c), (\gamma_2, a_2{-}c), (\delta, d)$$

and we build the derivation t represented by the term:



We can now compute the possible values of $\mathcal{H}(t)$:

1. *let* $u_1 = \mathbf{merge}(\mathbf{merge}(\delta, d)(\gamma_1, {=}da_1{-}c{-}c))(\alpha, {=}a_1{=}a_2 b)$ *then* $\mathcal{H}(u_1) = \langle(\alpha, {=}a_2 b), [\gamma_1\delta, {-}c{-}c)]\rangle$,
2. *let* $u_2 = \mathbf{merge}\,(\mathbf{merge}\,(\gamma_2, a_2{-}c{-}c)\,(\,u_1\,))\,(\beta, {=}b{+}c{+}c{+}cb)$, *we easily obtain that*

$$\mathcal{H}(u_2) = \langle(\beta\alpha, {+}c{+}c{+}cb), [(\gamma_1\delta, {-}c{-}c), (\gamma_2, {-}c)]\rangle$$

3. *let* $u_3 = \mathbf{move}(u_2)$ *then we have two possible results for* $\mathcal{H}(u_3)$*:*

$$\langle(\beta\alpha, +c{+}cb), [(\gamma_1\delta, -c), (\gamma_2, -c)]\rangle \text{ and } \langle(\gamma_2\beta\alpha, +c{+}cb), [(\gamma_1\delta, -c{-}c)]\rangle$$

4. *let* $u_4 = \mathbf{move}(u_3)$*, there are two possible results for* $\mathcal{H}(u_4)$*:*

$$\langle(\gamma_1\delta\beta\alpha, +cb), [(\gamma_2, -c)]\rangle \text{ and } \langle(\gamma_2\beta\alpha, +cb), [(\gamma_1\delta, -c)]\rangle$$

5. *finally there are two possible results for* $\mathcal{H}(t)$*:*

$$\langle(\gamma_2\gamma_1\delta\beta\alpha, b), \emptyset\rangle \text{ and } \langle(\gamma_1\delta\gamma_2\beta\alpha, b), \emptyset\rangle$$

We now show that the emptiness problem for minimalist grammars is Turing-equivalent to the membership problem. First, the emptiness problem can be reduced to the membership problem simply by replacing every element (v, l) in C by (ϵ, l), it is then trivial to check that the emptiness problem of the former grammar is equivalent to the membership of ϵ to the language of the new grammar. We can then state that:

Lemma 1 *If one can decide whether a sentense s belongs to the language of a minimalist grammar G then one can decide whether the language of a minimalist grammar is empty.*

In order to prove the converse property we show that the class of languages that can be defined with minimalist grammars is closed under intersection with recognizable sets of strings.

Lemma 2 *Given a minimalist grammar $G = (V, B, F, C, c)$ and a regular set of strings $Reg \subseteq V^*$, then there is a minimalist grammar G' whose language is the intersection of the language defined by G and Reg.*

Proof. Let us suppose that *Reg* is recognized by the following deterministic finite state automaton $\mathcal{A} = (V, Q, \delta, q_{\text{init}}, Q_f)$ where δ is the transition function from $V \times Q$ to Q (we make the confusion between δ and its homomorphic extension to the free monoid V^*, *i.e.* we consider that $\delta(\epsilon, q) = q$ and that, for w from V^*, $\delta(w, q)$ is the state that the automaton reaches when reading w from state q), $q_{\text{init}} \in Q$ is the initial state and $Q_f \subseteq Q$ is the set of final states.

We define $G' = (V, \{d\} \cup B', F', C', d)$ with $B' = B \times Q \times Q$, $F' = F \times Q \times Q$ and d not in $B'^- \cup B'^+ \cup F'^- \cup F'^+$. We let

$$C' = \{(\epsilon, =(c, q_{\text{init}}, q)d) | q \in Q_f\} \cup \bigcup_{(v,l)\in C} \varphi(v, l).$$

where $\varphi(v, l)$ is defined as follows[2]:

[2]In the definition of φ, we adopt the convention that ϵ_i is $=$ when b_i is in B and $+$ when b_i is in F.

1. l is of the form $=b\epsilon_1 b_1 \dots \epsilon_k b_k a - h_1 \dots - h_{n+1}$ then we will have

$$\varphi(v,l) = \{(v, =(b,q_0,q)\epsilon_1(b_1,q_2,q_1)\dots\epsilon_k(b_k,q_{k+1},q_k)(a,q'_0,q'_0) \\ -(h_1,q'_1,q'_1)\dots-(h_n,q'_n,q'_n)-(h_{n+1},q_{k+1},q))| \\ q,q_0,\dots,q_{k+1},q'_0,\dots q'_n \in Q \wedge \delta(v,q_1) = q_0\}$$

2. l is of the form $=b\epsilon_1 b_1 \dots \epsilon_k b_k a$ then we will have

$$\varphi(v,l) = \{(v, =(b,q_0,q)\epsilon_1(b_1,q_2,q_1)\dots\epsilon_k(b_k,q_{k+1},q_k)a(q_{k+1},q))| \\ q,q_0,\dots,q_{k+1} \in Q \wedge \delta(v,q_1) = q_0\}$$

3. l is of the form $a - h_1 \dots - h_{n+1}$ the we let:

$$\varphi(v,l) = \{(v, (a,q'_0,q'_0)-(h_1,q'_1,q'_1)\dots-(h_n,q'_n,q'_n)-(h_{n+1},q_1,q_0))| \\ q_0,q_1,q'_0,\dots,q'_n \in Q \wedge \delta(v,q_1) = q_0\}$$

4. l is of the form a then we let:

$$\varphi(v,l) = \{(v,(a,q_1,q_0))|q_0,q_1 \in Q \wedge \delta(v,q_1) = q_0\}$$

We here give an rough explanation about the definition $\varphi(v,l)$. If we look at a lexical entry produced as in the first case of that definition, it will be used to build a lexical construction of the form

$$\langle (w_k w_{k-1} \dots w_1 v w_0, (a,q'_0,q'_0)-(h_1,q'_1,q'_1)\dots-(h_n,q'_n,q'_n)-(h_{n+1},q_k,q)), M\rangle$$

where the word w_0 (possibly the empty string) comes from the lexical construction to which the lexical entry is first merged and the w_i (possibly empty strings) are the words that are put in front of the lexical construction through successive **merge** and **move** operations. The construction we give guaranties that $\delta(w_i, q_{i+1}) = q_i$ when $i > 0$ and $\delta(w_0, q_0) = q$ so that, knowing that $\delta(v, q_1) = q_0$, we have that $\delta(w_k w_{k-1} \dots w_1 v w_0, q_{k+1}) = q$. This can help to understand how the states are related to each other in the positive part of the list of features of the lexical entry. Afterwards, this lexical construction can be merged and then moved several times in another lexical construction leaving the head of this construction unchanged until the final **move** operation. Thus in the negative part of the list of features, the first negative features just contain pairs of identical states because they correspond the fact that the head of the lexical construction in which it is a moving piece is left unchanged. Then, when the last **move** operation happens, the string $w_k w_{k-1} \dots w_1 v w_0$ will be put in front of the head and the fact that when reading it the automaton goes from state q_k to state q must be consistently used.

Let's now follow this intuition and turn to a sketch of a proof that $\mathcal{L}(G')$ is equal to $\mathcal{L}(G) \cap Reg$.

In each of the cases of the definition $\varphi(v,l)$, if s is in $\varphi(v,l)$ (we suppose that s is written as in the cases defining the set $\varphi(v,l)$) then we write $range(s)$ for the pair of states (q_1, q_0).

Given a list of features l from Suff(G') we define $rg(l)$ as follows[3]:

1. if l starts with a positive feature and $l = \epsilon(b, q, q_1)l'\epsilon'(f, q', q_0)$ (l' being a list of features and $\epsilon'(f, q', q_0)$ a negative feature), then $rg(l) = (q_1, q_0)$
2. if l does not start with a positive feature, then $l = l'\epsilon'(f, q_1, q_0)$ and $rg(l) = (q_1, q_0)$.

Given an element t of Der(G'), such that $\mathcal{H}(t) = \langle (w, l), M\rangle$ we write range(t) for the pair of states defined as follows:

1. if t is of the form s where s is in C' then range(t) = $range(s)$,
2. otherwise range(t) = $rg(l)$.

For a list of features l of G', we write $\bar{l}$ for the list of features of G such that $\overline{\epsilon(b, q, q')l'} = \epsilon b\overline{l'}$.

An easy induction on t' in $d(G')$ proves that if the following properties are verified:

1. $\mathcal{H}(t') = \langle (w, l), [(w_1, l_1), \ldots, (w_n, l_n)]\rangle$, and
2. range(t') = (q, q'), $rg(l_1) = (q_1, q'_1)$, $\ldots$, $rg(l_n) = (q_n, q'_n)$

then we have:

1. $\delta(w, q) = q'$, $\delta(w_1, q_1) = q'_1$, $\ldots$, $\delta(w_n, q_n) = q'_n$,
2. there is t in $d(G)$ such that $\mathcal{H}(t) = \langle (w, \bar{l}), [(w_1, \overline{l_1}), \ldots, (w_n, \overline{l_n})]\rangle$.

Another simple induction on t in $d(G)$ shows that whenever

$$\mathcal{H}(t) = \langle (w, l), [(w_1, l_1), \ldots, (w_n, l_n)]\rangle$$

then for every pairs of states (q, q'), (q_1, q'_1), $\ldots$, (q_n, q'_n) such that $\delta(w, q) = q'$, $\delta(w_1, q_1) = q'_1$, $\ldots$, $\delta(w_n, q_n) = q'_n$ and every l', l'_1, $\ldots$, l'_n from Suff(G'), such that range(l') = (q, q') and $\overline{l'} = l$, range(l'_1) = (q_1, q'_1) and $\overline{l'_1} = l_1$, $\ldots$, range(l'_n) = (q_n, q'_n) and $\overline{l'_n} = l_n$ there is t' in $d(G')$ such that $\mathcal{H}(t') = \langle (w, l'), [(w_1, l'_1), \ldots, (w_n, l'_n)]]\rangle$.

These two properties have the consequence that a term t' from $d(G')$ verifies $\mathcal{H}(t') = \langle (w, (c, q_{\text{init}}, q_f)), \emptyset\rangle$ with q_f if and only if w is in $\mathcal{L}(G) \cap Reg$. Thus a sentence w is in $\mathcal{L}(G')$ (*i.e.* there is t' in $d(G)$ such that $\mathcal{H}(t') = \langle (w, d), \emptyset\rangle$) if and only if w is in $\mathcal{L}(G) \cap Reg$.

Thus the class of languages defined by minimalist grammars is closed under intersection with regular sets. □

Note that this proof gives an actual construction of G' and has therefore the next lemma as a consequence.

[3]We adopt the convention that ϵ is either = or + and ϵ' is either − or empty (when the feature $\epsilon' b$ is a negative base feature).

Lemma 3 *If the emptiness problem for minimalist grammars is decidable then the membership problem for those grammars is decidable.*

Proof. If we want to know whether w belongs to the language defined by G, since $\{w\}$ is a regular set, we construct G' the minimalist grammar whose language is the intersection of the language of G and of $\{w\}$. The language of G' is empty if and only if w belongs to the language defined by G. □

Theorem 1 *The emptiness problem and the membership problem for minimalist grammars are Turing-equivalent.*

2 Minimalist grammars and MELL

We are now going to show that provability in MELL is Turing-equivalent to the emptiness problem for minimalist grammars. We prove each direction of the equivalence separately.

First we reduce the provability in MELL to the emptiness of MGs. For this purpose we use a class of tree automata, k-VATA, introduced in [dGGS04], and which generalizes the notion of Vector Addition Systems (VAS) which are equivalent to Petri nets. It is proved in [dGGS04] that the decidability of the emptiness problem for k-VATA is equivalent to the decidability of the provability of sequents in MELL.

Second we show how to represent derivations in MGs as linear λ-terms built over a certain signature. It is well-known [dGGS04] that deciding provability in MELL is equivalent to deciding the existence of such linear λ-terms.

2.1 Emptiness of k-VATAs reduced to emptiness of MGs

A k-VATA is a tuple (Σ, Q, δ, C_f) where:

1. Σ is a tree signature,
2. Q is a finite set of states,
3. δ is a finite set of rules of the form:

$$f\,(q_1, \mathbf{x}_1) \ldots (q_n, \mathbf{x}_n) \longrightarrow \Big(q, \sum_{i\in 1}^{n} (\mathbf{x}_i - \mathbf{z}_i) + \mathbf{z}\Big)$$

   where f is a constant of Σ and $\mathbf{x}_i$ are variables and $\mathbf{z}_i$ and $\mathbf{z}$ are elements of $\mathbb{N}^k$.

4. C_f is a finite subset of $Q \times \mathbb{N}^k$, the *accepting configurations*

For a k-VATA, a configuration is an element of $Q \times \mathbb{N}^k$, a k-VATA is rewriting terms built on the tree signature Σ and that can have as leaves configurations. Thus given a rule of the considered k-VATA, a tree t which is equal to $C[f(q_1, \mathbf{p}_1) \dots (q_n, \mathbf{p}_n)]$ and a rule

$$f(q_1, \mathbf{x}_1) \dots (q_n, \mathbf{x}_n) \longrightarrow \Big(q, \sum_{i=1}^{n} (\mathbf{x}_i - \mathbf{z}_i) + \mathbf{z}\Big)$$

then it rewrites t to $t' = C[(q, \sum_{i=1}^{n} (\mathbf{p}_i - \mathbf{z}_i) + \mathbf{z})]$ provided that for all i in $[1; n]$, $\mathbf{p}_i - \mathbf{z}_i$ is an element of $\mathbb{N}^k$. In such a case, we write $t \longrightarrow_{\mathcal{A}} t'$ and $\xrightarrow{*}_{\mathcal{A}}$ denotes the reflexive and transitive closure of $\longrightarrow_{\mathcal{A}}$. The language of a k-VATA $\mathcal{A} = (\Sigma, Q, \delta, C_f)$ is the set $\mathcal{L}(\mathcal{A}) = \{t \in \mathcal{T}(\Sigma) | t \xrightarrow{*}_{\mathcal{A}} (q, \mathbf{p}) \wedge (q, \mathbf{p}) \in C_f\}$.

For a given k, we write 0 (*resp.* $\mathbf{e}_i$) to denote the element of $\mathbb{N}^k$ whose components are all zero (*resp.* except the i^{th} which is 1). A k-VATA is in *normal form* if it has only one accepting configuration which is of the form $(q, 0)$ and if all its rules are in one of the following form:

1. $c \longrightarrow (q, \mathbf{e}_i)$ for some i in $[1; k]$,
2. $f(q_0, \mathbf{x}) \longrightarrow (q, \mathbf{x} - \mathbf{e}_i)$ for some i in $[1; k]$,
3. $f((q_1, \mathbf{x}_1), (q_2, \mathbf{x}_2)) \longrightarrow (q, \mathbf{x}_1 + \mathbf{x}_2)$

As it is showed in [dGGS04], the emptiness problem for k-VATA in normal form is as difficult as for general k-VATA. Furthermore, the theorem is proved by giving an effective construction.

Theorem 2 *Given a k-VATA $\mathcal{A}$, there is a k-VATA $\mathcal{B}$ in normal form such that $\mathcal{L}(\mathcal{A})$ is empty if and only if $\mathcal{L}(\mathcal{B})$ is empty.*

We can now reduce the emptiness problem of a k-VATA in normal form to the emptiness of a minimalist grammar. Suppose that we are given a k-VATA in normal form $\mathcal{A} = (\Sigma, Q, \delta, \{(q_f, 0)\})$, we construct the following MG $G_{\mathcal{A}} = (\emptyset, Q, [1; k], C, q_f)$ where C contains the following entries:

- $(\epsilon, q{-}i)$ when there is a rule of the form $c \longrightarrow (q, \mathbf{e}_i)$ in δ
- $(\epsilon, {=}q_1{=}q_2 q)$ when there is a rule of the form $f((q_1, \mathbf{x}_1), (q_2, \mathbf{x}_2)) \longrightarrow (q, \mathbf{x}_1 + \mathbf{x}_2)$ in δ
- $(\epsilon, {=}q_0{+}iq)$ when there is a rule of the form $f(q_0, \mathbf{x}) \longrightarrow (q, \mathbf{x} - \mathbf{e}_i)$ in δ

We are going to prove that $\mathcal{L}(G_{\mathcal{A}})$ is empty if and only if $\mathcal{L}(\mathcal{A})$ is empty by giving an interpretation of the derivations of $G_{\mathcal{A}}$ as configurations of $\mathcal{A}$. Given t from $d(G_{\mathcal{A}})$, t can be interpreted as an element of $\mathbb{N}^k$ when $\mathcal{H}(t)$ is defined, then the i^{th} component of that vector is the number of occurrences of the feature $-i$ in $\mathcal{H}(t)$. We write $\mathcal{V}(t)$ for the vector denoted by t when it is defined. The state of t, denoted by $\mathcal{Q}(t)$, is the element of Q such that $\mathcal{H}(t) = ((v, ql), L)$. Note that $\mathcal{Q}(t)$ is not defined when $\mathcal{H}(t) = ((v, lql'), L)$ and l is not empty.

Thus t in $d(G_{\mathcal{A}})$ is interpreted as a configuration of $\mathcal{A}$ by $\mathrm{conf}(t) = (\mathcal{Q}(t), \mathcal{V}(t))$. This configuration is defined only when $\mathcal{Q}(t)$ is defined (note that when $\mathcal{Q}(t)$ is defined, obviously $\mathcal{H}(t)$ is defined and thus so is $\mathcal{V}(t)$).

Lemma 4 *Given $\mathbf{v} \in \mathbb{N}^k$ and $q \in Q$, there is t in $d(G_{\mathcal{A}})$ such that $\mathrm{conf}(t) = (q, \mathbf{v})$ if and only if there is a term t' such that $t' \xrightarrow{*}_{\mathcal{A}} (q, \mathbf{v})$.*

Proof. We first remark that whenever $\mathrm{conf}(t)$ is defined, then t is in one of the three following forms:

1. $t = (\epsilon, q{-}i)$, where $(\epsilon, q - i)$ is in C,
2. $t = \mathbf{merge}\, t_2(\mathbf{merge}\, t_1(\epsilon, {=}q_1{=}q_2 q))$ where $\mathcal{H}(t_1) = ((\epsilon, q_1 l_1), L_1)$ and $\mathcal{H}(t_2) = ((\epsilon, q_2 l_2), L_2)$,
3. $t = \mathbf{move}(\mathbf{merge}\, u\, (=q_0{+}\mathbf{i}q))$ where $\mathcal{H}(u) = ((\epsilon, q_0 l), L)$ and the i^{th} component of $\mathcal{V}(u)$ is strictly positive.

We now prove the existence of t' by induction on t.

In case $t = (\epsilon, q{-}i)$, then, by definition of $G_{\mathcal{A}}$ there is a rule in δ which is of the form $c \longrightarrow (q, \mathbf{e}_i)$. Then it suffices to take $t' = c$.

In case $t = \mathbf{merge}\, t_2(\mathbf{merge}\, t_1(\epsilon, {=}q_1{=}q_2 q))$ then, by induction hypothesis, we have the existence of t'_1 and t'_2 such that $t'_i \xrightarrow{*} \mathrm{conf}(t_i)$. Moreover, by definition of C, there is a rule of the form $f((q_1, \mathbf{x}_1), (q_2, \mathbf{x}_2)) \longrightarrow (q, \mathbf{x}_1 + \mathbf{x}_2)$ in C. We then let t' be $f\, t'_1\, t'_2$.

In case $t = \mathbf{move}(\mathbf{merge}\, u\, (=q_0{+}\mathbf{i}q))$, then, by induction hypothesis, there is u' such that $u' \xrightarrow{*}_{\mathcal{A}} \mathrm{conf}(u)$. Furthermore, we know that the i^{th} component of $\mathcal{V}(u)$ is strictly positive, and that, by definition of C, there is in δ a rule of the form $f(q_0, \mathbf{x}) \longrightarrow (q_0, \mathbf{x} - \mathbf{e}_i)$. We can then choose t' to be $f(u')$.

The proof of the converse property is using a similar induction. □

The parallel that is drawn between minimalist grammars and k-VATA allows us to give a negative answer to the conjecture raised in [GM07] that MGs (without SMC) define only semi-linear languages. To prove it we use the notion of k-VAS (k- Vector Addition Systems). A k-VAS can be seen as a k-VATA whose signature contains only nullary and unary operators, and, given a state q, the sets of vectors that are accessible at q for a k-VAS $\mathcal{A}$ is $\mathrm{Acc}(\mathcal{A}, q) = \{\mathbf{v} | \exists t.t \xrightarrow{*}_{\mathcal{A}} (q, \mathbf{v})\}$. It is known that the sets of the form $\mathrm{Acc}(\mathcal{A}, q)$ may not be semi-linear [HP79]. As for k-VATA, there is a normal form for k-VAS, where the rules are of the following form:

1. $c \longrightarrow (q, \mathbf{0})$
2. $f(q_1, \mathbf{x}) \longrightarrow (q_2, \mathbf{x} - \mathbf{e}_i)$ for some i in $[1; k]$,
3. $f(q_1, \mathbf{x}) \longrightarrow (q_2, \mathbf{x} + \mathbf{e}_i)$ for some i in $[1; k]$

The important property is that if $\mathcal{A}$ is a k-VAS, and q is a state of $\mathcal{A}$, then there is a k-VAS in normal form $\mathcal{B}$ and a state q' of $\mathcal{B}$ such that $\mathrm{Acc}(\mathcal{A}, q) = \mathrm{Acc}(\mathcal{B}, q')$.

So given a k-VAS in normal form $\mathcal{A}$ and its final state p, we can define the following MG $G_{\mathcal{A}} = ([1; k], Q \cup d, [1; k], C, d)$:

1. (ϵ, q) if there is a rule $e \longrightarrow (q, \mathbf{0})$ is in δ,
2. $(\epsilon, =q_1{+}iq_2)$ if there is a rule $f(q_1, \mathbf{x}) \longrightarrow (q_2, \mathbf{x} - \mathbf{e}_i)$ in δ,
3. $(\epsilon, =q_1q_2{-}i)$ if there is a rule $f(q_1, \mathbf{x}) \longrightarrow (q_2, \mathbf{x} + \mathbf{e}_i)$ in δ,
4. $(\epsilon, =pd)$
5. $(i, =d{+}id)$ for all i in $[1; k]$

Similarly to the proof of the Lemma 4, it can be showed that whenever $\mathbf{v}$ is accessible at q in $\mathcal{A}$ then there is t such that $\mathrm{conf}(t) = (q, \mathbf{v})$. Then the lexical entries $(\epsilon, =pd)$ and $(i, =d{+}id)$ (where i is in $[1; k]$) transform the vector $\mathbf{v}$ into a word of $[1; k]^*$ such that, for all i in $[1; k]$, it contains exactly $\mathbf{v}_i$ occurrences of i (if $\mathbf{v}_i$ is the i^{th} component of $\mathbf{v}$) so that the language defined by $G_{\mathcal{A}}$ is the set of elements of $[1; k]^*$ whose Parikh image is $\mathrm{Acc}(\mathcal{A}, q)$. Thus the language of $G_{\mathcal{A}}$ is semi-linear if and only if the set of vectors accessible by $\mathcal{A}$ form a semi-linear set. Thus, we have the following theorem.

Theorem 3 *The class of languages defined by MGs is not semi-linear.*

2.2 Representing MG derivations as proofs in MELL

We here give an account of the derivations of an MG with the linear λ-terms built over a certain signature. It is known (*c.f.* [dGGS04]) that finding such λ-terms is in general Turing-equivalent to provability in MELL. This encoding thus completes the proof that the membership problem for MGs is Turing-equivalent to provability in MELL.

For a given MG, $G = (B, F, W, C, c)$, we define Σ_G which declares the following set of atomic types:

1. $e(l)$ if there is (w, l) in C,
2. $d(l)$ if l is an element of $\mathrm{Suff}(G)$,
3. $h(l)$ if l is an element of $\mathrm{Move}(G)$

Even though we use a predicate-like notation, note that since C, $\mathrm{Suff}(G)$ and $\mathrm{Move}(G)$ are finite, there are finitely many types that are declared in Σ_G. The type $e(l)$ represents the type of a lexical entry, $d(l)$ represents the type of a derivation whose head is of the form (w, l) and $h(l)$ is the type of a moving piece of the form (w, l). We make the distinction between $e(l)$ and $d(l)$ so as to know how to interpret **merge** in a homomorphic way.

Σ_G also declares the following constants:

1. $(w, l) : e(l)$ if (w, l) is in C,
2. $merge[k(al_1), k'(=al_2)] : k(al_1) \multimap k'(=al_2) \multimap h(l_1) \multimap d(l_2)$ where l_1 is not empty and with k, k' in $\{d; e\}$
3. $merge[k(a), k'(=al_2)] : k(a) \multimap k'(=al_2) \multimap d(l_2)$ with k, k' in $\{d; e\}$

4. $move[h(-al_1), d(+al_2)] : (h(-al_1) \multimap d(+al_2)) \multimap h(l_1) \multimap d(l_2)$ where l_1 is not empty,

5. $move[h(-a), d(+al)] : (h(-a) \multimap d(+al)) \multimap d(l)$

We will show that there are closed terms of type $k(c)$ (with k in $\{d; e\}$) if and only if $\mathcal{L}(G)$ is not empty. Terms of $d(G)$ that can be interpreted with the function $\mathcal{H}$ are represented as terms built on Σ_G whose types are $d(l)$ or $e(l)$ and whose free variables have types of the form $h(l)$.

Lemma 5 *There is $t \in \Lambda_{\Sigma_G}^{k(l)}$ (with k being either d or e) such that $FV(t) = \{x_1^{h(l_1)}; \ldots; x_n^{h(l_n)}\}$ is derivable if and only if there is t' in $d(G)$ such that $\mathcal{H}(t') = \langle (w, l), [(w_1, l_1) \ldots (w_n, l_n)] \rangle$.*

Proof. We first construct t' by induction on t. We suppose without loss of generality that t is in normal form.

If $t = (w, l)$ then it suffices to take $t' = (w, l)$.

If $t = merge[k(al_1), k'(= al_2)]t_1\, t_2\, t_3$, then because there is no constant that has the type $h(l_1)$ as a conclusion and because free variables all have a type of the form $h(l)$, it is necessary that, for some i, $t_3 = x_i^{h(l_i)}$ and $h(l_i) = h(l)$. Thus, we have that $t_1 \in \Lambda_{\Sigma_G}^{k(al)}$ with $FV(t_1) = \{x_{i_1}^{h(l_{i_1})}; \ldots; x_{i_p}^{h(l_{i_p})}\}$, $t_2 \in \Lambda_{\Sigma_G}^{k'(=al_2)}$ with $FV(t_2) = \{x_{j_1}^{h(l_{j_1})}; \ldots; x_{j_q}^{h(l_{j_p})}\}$ and $\langle \{i_1; \ldots; i_p\}, \{j_1; \ldots; j_q\}, \{i\} \rangle$ forms a partition of $[1; n]$. By induction hypothesis, we get the existence of t'_1 and t'_2 verifying the right properties and it suffices to take $t' = \mathbf{merge}\, t'_1\, t'_2$.

If $t = merge[k(a), k'(= al_2)]t_1\, t_2$, then we proceed similarly to the previous case.

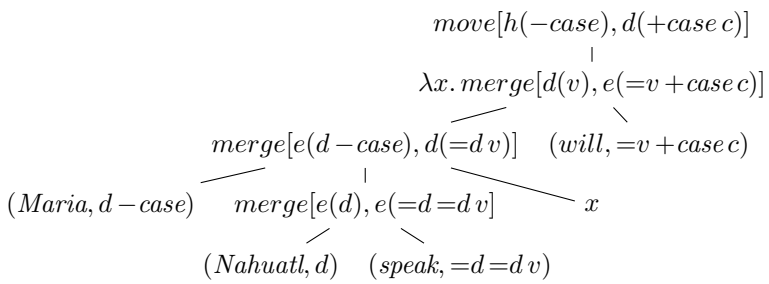
If $t = move[h(-al), d(+al')]\, (\lambda x^{h(-al)}.t_1)t_2$ then, similarly to the previous case, we have that t_2 must be one of the $x_i^{h(l_i)}$. We suppose without loss of generality, that $t_2 = x_1^{h(l_1)}$ and then we have that $t_1 \in \Lambda_{\Sigma_G}^{d(+al')}$ with $FV(t_1) = \{x_2^{h(l_2)}; \ldots; x_n^{h(l_n)}; x^{h(-al)}\}$. Then we obtain a t'_1 from t_1 by using the induction hypothesis and it suffices to take $t' = \mathbf{move}(t'_1)$ by assuming that **move** is operating on a moving piece of the form $(v, -al)$ which by induction hypothesis must exist.

If $t = move'[h(-a), d(+al)]\, \lambda x.t_1$ then we proceed in a way similar to the previous case.

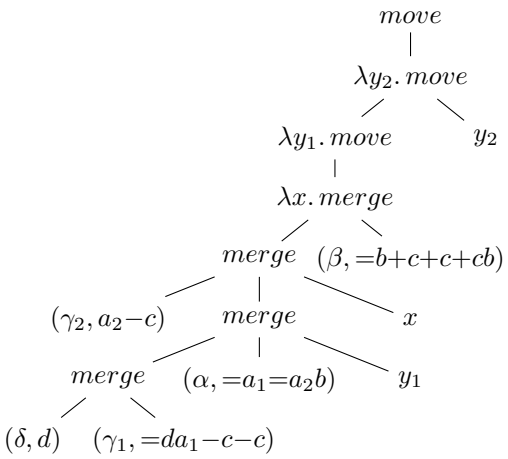
The converse does not present more difficulty and is then left to the reader. □

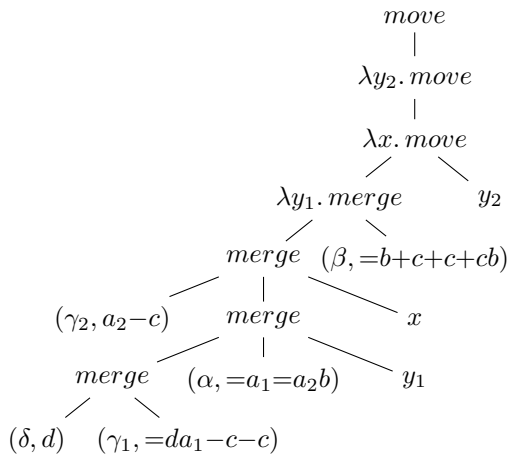
This Lemma together with Lemma 4 answers the question of the mathematical nature of the derivations of minimalist grammars. It shows that these derivations can be seen as closed linear λ-terms of type $d(c)$ or $e(c)$. Thus, with such a representation, checking whether such a derivation is correct does not amount to compute whether it can be interpreted as a string, but merely amounts to type checking.

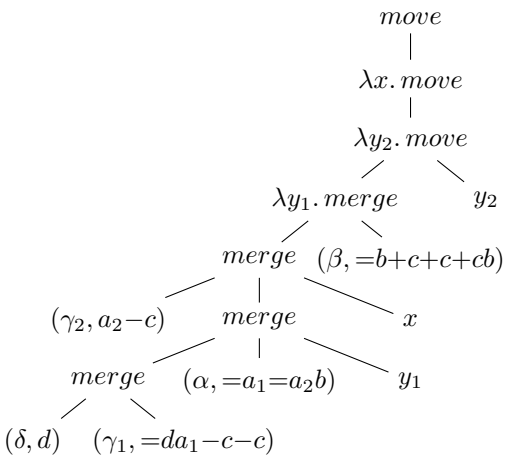
Example 3 *We show here the representation of the derivation of the example 1 as a linear λ-term:*



On the other hand, the derivation of the example 2 can be represented by three different linear λ-terms (for the sake of concision we erase the squared brackets $[\alpha, \beta]$ of the constants move$[\alpha, \beta]$ and merge$[\alpha, \beta]$):







In this presentation of minimalist derivations λ-variables represent the moving pieces of a derivation. When a derivation t_1 is merged to another derivation t_2 and must be moved afterwards then the new derivation is of the form $merge\, t_1\, t_2\, x$ where the λ-variable x materialises t_1 as a moving piece.

*Each time a **move** operation is applied, it is applied to a third-order term of the form $\lambda x.t$ where x indicates which moving piece is actually moved. When a constant of the form $move[\alpha, \beta]$ has two arguments, it means that the moving piece that the **move** operation has moved still has to be moved and then the second argument is a λ-variable that materialises the actualisation of this moving piece.*

In the representation of minimalist derivations we propose, it becomes explicit that the **move** operation corresponds to binding some variable, but we can go a little further in the logical interpretation of **move**. Indeed, it would be possible to define a signature Π_G which only contains constants representing the lexical entries of G and no operator representing **move** or **merge**. The types used by Π_G are: $d(la)$ where la is such that there is $(w, l_1 la l_2)$ in C and l is either starting with an element of F^+ or is empty. For every entry $(w, la{-}f_1 \ldots {-}f_n)$ in C, Π_G contains constants of type

$$((\ldots(((g(la) \multimap d(+f_1 l_1)) \multimap g(l_1) \multimap d(+f_2 l_2)) \multimap g(l_2))\ldots) \multimap d(+f_n l_n)) \multimap g(l_n)$$

for every possible atomic type of the form $d(+f_i l_i')$ and where $g(l)$ is equal to $d(a_1) \multimap \cdots d(a_k) \multimap d(l'b)$ if l is of the form $=a_1 \ldots =a_k l' b$ and l' if either starting with some element of F^+ or is empty. The idea is that the $d(+f_i l_i')$ and e_i represent the features that the head has to licence when the i^{th} movement of the entry happens.

Example 4 *If we applied such a transformation to the grammar we used in our examples then we would get the following type assignment for the lexical entries for the derivations we showed:*

$$\begin{array}{rcl}
(Maria, d{-}case) & : & (d(d) \multimap d(+case\, c)) \multimap d(c) \\
(will, {=}v\, {+}case\, c) & : & d(v) \multimap d(+case\, c) \\
(Nahuatl, d) & : & d(d) \\
(speak, {=}d\, {=}d\, v) & : & d(d) \multimap d(d) \multimap d(v)
\end{array}$$

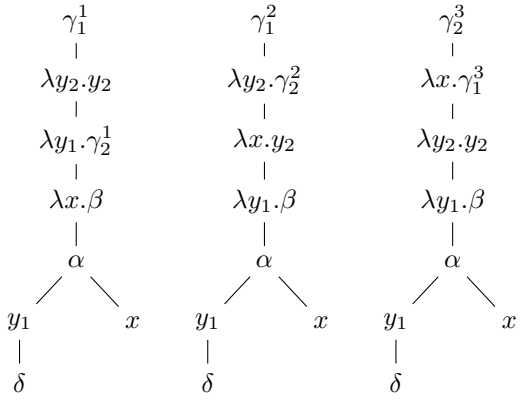
Then the derivation is represented by the linear λ-term:

$$\begin{array}{c}
(Maria, d{-}case) \\
| \\
\lambda x.\, (will, {=}v\, {+}case\, c) \\
| \\
(speak, {=}d\, {=}d\, v) \\
\swarrow \quad \searrow \\
(Nahuatl, d) \qquad x
\end{array}$$

For the second example, we may represent the derivations given as examples with the following constants:

$$
\begin{array}{rcl}
\gamma_2^1 & : & (((d(a_2) \multimap d(+c{+}cb)) \multimap d(+c\,b)) \multimap d(+c)) \multimap d(b) \\
\gamma_2^2 & : & (((d(a_2) \multimap d(+c{+}c{+}cb)) \multimap d(+c{+}cb)) \multimap d(+c)) \multimap d(b) \\
\gamma_2^3 & : & (((d(a_2) \multimap d(+c{+}c{+}cb)) \multimap d(+c{+}c\,b)) \multimap d(+c{+}c)) \multimap d(+cb) \\
\gamma_1^1 & : & ((d(d) \multimap d(a_1)) \multimap d(+c{+}c{+}cb)) \multimap d(+c{+}c\,b) \\
\gamma_1^2 & : & ((d(d) \multimap d(a_1)) \multimap d(+c{+}cb)) \multimap d(+c\,b) \\
\gamma_1^3 & : & ((d(d) \multimap d(a_1)) \multimap d(+cb)) \multimap d(b) \\
\beta & : & d(b) \multimap d(+c{+}c{+}cb) \\
\alpha & : & d(a_1) \multimap d(a_2) \multimap d(b) \\
\delta & : & d(d)
\end{array}
$$

With these constants, the derivations can be represented by the linear λ-terms:



We can now understand the technical contributions of **move** and **merge** by comparing Σ_G and Π_G. First, we remark that in Π_G each entry conveys explicitly the context in which it is used; in particular it specifies the characteristics of the head at each step of its movements. It has the inconvenience that Π_G has a size that is a $\mathcal{O}(|G|^{n+1})$ where n is the maximum number of movements that a moving piece can do whereas Σ_G is much more compact and has a size in $\mathcal{O}(|G|^2)$. Furthermore, by making the types of the constants $merge[k(al_1), k'(= al_2)]$, $move[h(-al_1), d(+al_2)]$ and $move[h(-a), d(+al)]$ be polymorphic in a, l_1, l_2 or l we obtain a grammar whose size is linear with respect to the size of G. This polymorphism has also the advantage that we can add new entries without having to change the grammar in any respect. We can also use a notion of polymorphism to Π_G, but it needs to be stronger a notion. Indeed, while in Σ_G, polymorphism instantiates differently atomic types, in Π_G, because we use a function $g(l)$ that gives a complex type depending on the shape of l, this polymorphism requires to have a notion of functions from feature string to types.

A more interesting remark concerning the difference between Π_G and Σ_G concerns their respective order. Indeed Σ_G is a third order signature whereas

Π_G is a signature whose order is between $n+1$ and $n+2$ where n is the maximum number of movements a moving piece can do. If G did not contain any moving feature then both Σ_G and Π_G would be second order signatures. Thus movement is responsible for the higher order types in Σ_G and Π_G. We can see the **move** operation as being responsible of transforming higher order into third order. Transforming higher order into third order is quite usual in intuitionistic implicative logics. Indeed, in minimal logic as well as in intuitionisitic implicative linear logic, a sequent $\Gamma \vdash \alpha$ which may contain formulae of any order can be transformed in a sequent $\Delta \vdash \beta$ which contains only formulae that are at most third order and which is provable if and only if $\Gamma \vdash \alpha$ is provable. But interestingly if we use the construction that underpins this property on Π_G we will not obtain Σ_G as a result. This is due to the fact that **move** and **merge** also break down the complexity of the polymorphism that would be necessary to make Π_G be of a reasonable size. The interesting thing about comparing Π_G and Σ_G is to show how the linguistic ideas that lead to **merge** and **move** operations make drastic simplifications in the mathematical objects used to represent derivations. It also gives an opening towards the understanding of the mathematical nature of these simplifications.

3 Interpreting minimalist derivations

In this section we show how to interpret the terms of the signature Σ_G so as to obtain the string that they analyse. In the previous section we have already remarked that these terms were explicitly representing via variable binding which moving piece is actually moved when a **move** operation is used. This allows us to give a homomorphic interpretation of these terms that yield the unique strings that they represent. Thus this representation of derivations makes the linearisation of the derivations become independent from their semantic interpretation. This should make it easier to describe the interface between syntax and semantics for MGs.

We have already seen that interpreting trees of $d(G)$, for an MG G, requires that we have a list of moving pieces. In our homomorphic interpretation of the terms built on Σ_G, we will also need such a list. This list will be used very similarly to the context that is introduced in [dG07] for semantic purposes. This shows that the mapping from minimalist derivations to the surface structure is far from being trivial.

In order to define our context we need to work in a system with at least as much computational power as Gödel's system T. We do not give all the implementation details because they are not of much interest. We wish to convince the reader that the homomorphic interpretation of minimalist derivations requires a rather sophisticated implementation and technical details would obfuscate the reasons why it is so. A *syntactic context* is defined as a pair $\langle L, n\rangle$ where:

1. L is a list of pairs (w, p) with w belonging to W^* and p being an integer,
2. n is an integer.

Integers are used here so as to *give a name to* moving pieces. They allow to make the distinction between several moving pieces. Thus, L is a list that associates specific integers to moving pieces so as to retrieve them and n is a fresh integer so as to be able to extend and search the list. We use the following constants and operations on *syntactic contexts*:

- the empty list is $[]$,
- concatenation $(L_1, p_1) \bullet (L_2, p_2) = (L_1 @ L_2, \max(p_1, p_2))$, where $@$ is the operation of list concatenation
- adding an element $push(e, (L, p)) = (e :: L, p)$, where $::$ is the operation that adds an element to a list,
- incrementation $incr(L, p) = (L, p + 1)$,
- getting a fresh name $fresh(L, p) = p + 1$,
- selection $sel(k, (L, p))$ which sends the string associated to k in L.

We now give the interpretation of the types of Σ_G

- $e(l)$ et $d(l)$ are interpreted as being of type $str \times \gamma$ where str is the type of strings and γ that of syntactic contexts.
- $h(l)$ is interpreted as being the type $\mathbb{N}$, the type of integers.

For the sake of simplicity, we write $\lambda(x_1, x_2).t$ and use some ***let*** $\ldots$ ***and*** $\ldots$ ***in*** constructions instead of the usual projection operators π_1 and π_2. Furthermore as we are interested in the language of an MG and as derivations are interpreted as a pair of type $str \times \gamma$ we introduce two new constants in the signature: $realise_1$ of type $d(c) \multimap s$ and $realise_2$ of type $e(c) \multimap s$ and whose purpose is to get the string part of a valid derivation. When t represents a derivation of the MG G as a term built on the signature Σ_G, the resulting string is obtained using a homomorphism $\mathcal{I}$ on the term $realise_1(t)$ or $realise_2(t)$ depending on whether t have type $d(c)$ or $e(c)$. In what follows, as we will need to concatenate strings, we will write $w \cdot w'$ the concatenation of two terms that are strings in order to avoid confusion with term application.

The interpretation of the constants of Σ_G is (in what follows $k(l)$ and $k'(l)$ may denote either $d(l)$ or $e(l)$):

1. for $realize_1 : d(c) \multimap s$ we have $\mathcal{I}(realize_1) = \lambda(w, L).w$
2. for $realize_2 : e(c) \multimap s$ we have $\mathcal{I}(realize_2) = \lambda(w, L).w$
3. for $(w, l) : e(l)$, we have $\mathcal{I}((w, l)) = (w, ([], 0))$
4. for $merge[k(al_1), k'(=al_2)] : k(al_1) \multimap k'(=al_2) \multimap h(l_1) \multimap d(l_2)$ where l_1 is not empty we have:

   $$\mathcal{I}(merge[k(al_1), k'(=al_2)]) = \lambda(w_1, \mathbf{s}_1)(w_2, \mathbf{s}_2)p.(w_2, (push\,(w_1, p)\,(\mathbf{s}_1 \bullet \mathbf{s}_2)))$$

5. for $merge[k(a), d(=al_2)] : k(a) \multimap d(=al_2) \multimap d(l_2)$ we have:

$$\mathcal{I}(merge[k(a), d(=al_2)]) = \lambda(w_1, \mathbf{s}_1)(w_2, \mathbf{s}_2).(w_1 \cdot w_2, (\mathbf{s}_1 \bullet \mathbf{s}_2))$$

6. for $merge[k(a), e(=al_2)] : k(a) \multimap e(=al_2) \multimap d(l_2)$ we have:

$$\mathcal{I}(merge[k(a), d(=al_2)]) = \lambda(w_1, \mathbf{s}_1)(w_2, \mathbf{s}_2).(w_2 \cdot w_1, (\mathbf{s}_1 \bullet \mathbf{s}_2))$$

7. for $move[h(-al_1), d(+al_2)] : (h(-al_1) \multimap d(+al_2)) \multimap h(l_1) \multimap d(l_2)$ (by definition l_1 is not empty) we have:

$$\mathcal{I}(move[h(-al_1), d(+al_2)]) = \lambda fp.fp$$

8. for $move[h(-a), d(+al_2)] : (h(-a) \multimap d(+al_2)) \multimap d(l_2)$ we have:

$$\begin{aligned}\mathcal{I}(move[h(-al_1), d(+al_2)]) = \lambda f.&\textbf{let } (\_, \mathbf{s}) = (f\ 0)\\ &\textbf{and } n = fresh(\mathbf{s})\\ &\textbf{and } (w, \mathbf{s}') = (fn)\\ &\textbf{in } ((sel(n, \mathbf{s}')) \cdot w, (incr(\mathbf{s}')))\end{aligned}$$

The only operation that is complicated is the $\mathcal{I}(move[h(-a), d(+al_2)])$, because this is the one where a moving piece stops moving and is incorporated to the head. First we retrieve the context $\mathbf{s}$, by giving 0 as a dummy argument to f, this allows us to obtain n as a *fresh name* to give to the moving piece, we then apply f to n and we get the string of the head and a context $\mathbf{s}'$ which associates n to the moving piece that will be incorporated to the head. The operation $sel(n, \mathbf{s}'))$ retrieves the string of that moving piece and the context is incremented so that in a next use of the context, that moving piece won't be chosen. We could have deleted it from the list, but it is not necessary here. Deletion in list is in general a more complex operation than selection in the λ-calculus.

This usage of the context of type γ is typical of the continuation passing style of programming. Even though it is technical, it is quite easy to prove that the set $\{w | \exists t \in \Lambda^{d(c)}_{\Sigma_G}.t$ is closed and $\mathcal{I}(realise_1\, t) = w\} \bigcup \{w | \exists t \in \Lambda^{e(c)}_{\Sigma_G}.t$ is closed and $\mathcal{I}(realise_2\, t) = w\}$ is equal to $\mathcal{L}(G)$; the induction is very similar to the one we used to prove Lemma 4.

In this paper, for clarity reasons, we deliberately use the simplest notion of minimalist grammars as possible. In particular, we omitted weak features that are necessary in most reasonable linguistic models. At the level of the derivation structures, the addition of weak features almost does not change anything; changes occur at the level of the interpretation. We will not enter in the details of a possible homomorphic interpretation of derivations with weak features, but we can say that it is much more evolved than the homomorphism $\mathcal{I}$.

3.1 The shortest move constraint

Now we can see that the shortest move constraint can be expressed on minimalist derivations represented as terms built on Σ_G as the constraint that in any subterm of such a term there is at most one free variable having a type of the form $h(-al)$ for each licencing feature a. With the Curry-Howard isomorphism, we can see linear λ-terms as proofs in implicative linear logic which establishes judgments of the form $\Gamma \vdash \alpha$ where Γ is a multiset of linear types and α is a linear type. The restriction that there is at most one free variable having a type of the form $h(-al)$ is interpreted in implicative linear logic as constraining the possible judgement as being of the form $\Gamma \vdash \alpha$ where Γ contains at most one occurrence of a formula of the form $h(-al)$. This means that the possible Γ may only contain a number of type that is bounded by the number of movement features, the size of $\mathcal{F}$. And thus, there are finitely many possible Γ that obey this constraint. This is the key property that makes minimalist grammars with the shortest move constraint languages of multiple context free grammars (MCFL).

Indeed, because of the finiteness of the possible Γ and of the subformula property, there are only finitely many possible judgements that may have to be proved. We can therefore represent the set of all proofs in an algebraic setting; it suffices to take all the possible instances of the *elimination rules* and of the *introduction rules* of the intuitionistic implicative linear logic. For a given signature Σ_G, we do this by building the following multi-sorted tree signature $\mathrm{SMC}(\Sigma_G)$:

1. the types are $[h(-a_1 l_1), \ldots, h(-a_n l_n) \vdash \alpha]_I$ and $[\Gamma \vdash \alpha]_E$ where $-a_i l_i$ is in $\mathrm{Move}(G)$, the a_i are pairwise distinct and α is a subformula of the type of a constant in Σ_G
2. $c : [\vdash \alpha]_E$ for each constant $c : \alpha$ of the signature (c can be either a lexical entry or one of the constants representing the **move** and **merge operations**.
3. $E_1 : [\Gamma \vdash \alpha \multimap \beta]_E \multimap [\Delta \vdash \alpha]_E \multimap [\Gamma, \Delta \vdash \beta]_E$ if α is atomic,
4. $E_2 : [\Gamma \vdash \alpha \multimap \beta]_E \multimap [\Delta \vdash \alpha]_I \multimap [\Gamma, \Delta \vdash \beta]_E$ if α is not atomic,
5. $I_1 : [\Gamma, \alpha \vdash \beta]_E \multimap [\Gamma \vdash \alpha \multimap \beta]_I$ if β is atomic,
6. $I_2 : [\Gamma, \alpha \vdash \beta]_I \multimap [\Gamma \vdash \alpha \multimap \beta]_I$ if β is not atomic.

To be rigorous, we should have, similarly to the definitions of the constants representing **merge** and **move** in the signature Σ_G, several versions of the constants E_1, E_2, I_1 and I_2 for their various possible typing. But for the sake of the simplicity of the notations, we simply use those four constants and consider that they have several types.

In order to have a unique representation of each proof we have annotated types with I or E, types with an I as subscript are types of proofs that finish with an introduction rule, whereas the one with an E are the other cases. The

representation we have chosen corresponds to β-normal and η-long terms built on Σ_G.

We now give the interpretation of the terms built on SMC(Σ_G) with an homomorphism $\mathcal{D}$ that retrieves the λ-term of Σ_G that is represented:

1. $\mathcal{D}([\alpha_1, \ldots, \alpha_n \vdash \alpha]_E) = \mathcal{D}([\alpha_1, \ldots, \alpha_n \vdash \alpha]_I) = \alpha_1 \multimap \cdots \multimap \alpha_n \multimap \alpha$
2. $\mathcal{D}(c) = c$
3. $\mathcal{D}(E_1) = \lambda f g x_1 \ldots x_n y_1 \ldots y_p . f x_1 \ldots x_n (g y_1 \ldots y_p)$
4. $\mathcal{D}(E_2) = \lambda f g x_1 \ldots x_n y_1 \ldots y_p . f x_1 \ldots x_n (g y_1 \ldots y_p)$
5. $\mathcal{D}(I_1) = \lambda f . f$
6. $\mathcal{D}(I_2) = \lambda f . f$

We can also define a homomorphism $\mathcal{J}$ that transforms a tree t of SMC(Σ_G) in the same string as $\mathcal{D}(\mathcal{I}(t))$ but with the property that every constant is interpreted as an affine λ-term. The idea behind the definition of $\mathcal{J}$ is that we can represent a p-tuple of strings $(s_1, \ldots, s_p)$ that are used to build a string by a term of the form $P = \lambda f . f s_1 \ldots s_p$, then, as an example, we can use P to form a string $w_1 s_1 \cdots w_p s_p w_{p+1}$ simply with the following λ-term: $P(\lambda x_1 ... x_p . w_1 \cdot x_1 \cdot \ldots \cdot w_p \cdot x_p \cdot w_{p+1})$.

1. let's suppose that the number of licencing features of G is p, then types of the form $[\Gamma \vdash k(l)]_M$ or $[\Gamma \vdash h(l') \multimap k(l)]_M$ with k in k in $\{d; e\}$ and M in $\{I; E\}$ is interpreted as the type $\gamma = (str^{p+1} \multimap str) \multimap str$. We furthermore assume that the set of licensing features is $\{a_1; \ldots; a_p\}$ so that they are implicitly ordered.
2. the types of the form $[\vdash \alpha]_E$, where α is a the type of a **move** constant, are interpreted as $\gamma \multimap \gamma$.
3. the types of the form $[\vdash \alpha]_E$ where α is a the type of a **merge** constant, are interpreted as $\gamma \multimap \gamma \multimap \gamma$.
4. then $\mathcal{J}((w, l)) = \lambda g . g\, w \underbrace{\epsilon \ldots \epsilon}_{p\times}$
5. $\mathcal{J}(merge[k(b{-}a_k l_1),\, k'(=b l_2)]) =$
   $\lambda D_1 D_2 g . D_1 (\lambda s_1 x_1 \ldots x_p .$
   $D_2 (\lambda s_2 y_1 \ldots y_p .$
   $g\, s_2\, (x_1 \cdot y_1) \ldots (x_{k-1} \cdot y_{k-1}) (s_1) \ldots (x_p \cdot y_p)))$
6. $\mathcal{J}(merge[k(b),\, d(=b l_2)]) =$
   $\lambda D_1 D_2 g . D_1 (\lambda s_1 x_1 \ldots x_p .$
   $D_2 (\lambda s_2 y_1 \ldots y_p . g(s_1 \cdot s_2)\, (x_1 \cdot y_1) \ldots (x_p \cdot y_p)))$
7. $\mathcal{J}(merge[k(b),\, e(=b l_2)]) =$
   $\lambda D_1 D_2 g . D_1 (\lambda s_1 x_1 \ldots x_p .$
   $D_2 (\lambda s_2 y_1 \ldots y_p . g(s_2 \cdot s_1)\, (x_1 \cdot y_1) \ldots (x_p \cdot y_p)))$

8. $\mathcal{J}(move[h(-a_k - a_j l_1),\, d(+a_k l_2)]) =$
$\lambda Dg.D(\lambda s x_1 \dots x_p g.g\, s\, x_1 \dots x_{k-1} \epsilon \dots x_{j-1} x_k \dots x_p)$

9. $\mathcal{J}(move[h(-a_k),\, d(+a_k l_2)]) =$
$\lambda Dg.D(\lambda s x_1 \dots x_p g.g\, (x_k \cdot s)\, x_1 \dots x_{k-1} \epsilon \dots x_p))$

10. $\mathcal{J}(E_1) = \lambda f x.f\, x$ and $\mathcal{J}(E_2) = \lambda f x.f\, x$

All together, SMC(Σ_G) and $\mathcal{J}$ define an affine second order string Abstract Categorial Grammar in the sense of [Yos06] which also shows that the language of such a grammar is the language of a *linear* second order string Abstract Categorial Grammar. But it is showed in [Sal07] that such grammars can only define MCFLs.

This construction of an MCFL from an MG with the SMC is not essentially different from the one given in [Mic98], but the transformation we propose preserves in an obvious way (thanks to the homomorphism $\mathcal{D}$) the structure of the derivation, so that it preserves the interface between syntax and semantics. Furthermore, the use of tuples can also replace the complex *semantic contexts* that would be necessary without the SMC so that it would become very similar to Montague semantics.

3.2 Saving computation

One of the interest of representing the derivation by using the signature Σ_G is that it enables to separate the syntactic interpretation of the derivations from their semantic interpretation. Indeed, as in [Kob06], the semantic interpretation of minimalist grammars has to be done in parallel to its syntactic interpretation. Nevertheless, this parallel computation shows that if we want to give a semantic interpretation of the derivations of minimalist grammars, then we will need to implement a context that is at least as complicated as the one we have defined for the syntactic interpretation. In order to avoid similar computations that need to be accomplished both for the syntactic and the semantic interpretations, we can compute an intermediate structure in which all the computations that are necessary for both the syntactic interpretation and the semantic one are already performed.

This intermediate structure is built on a rather simple signature Int_G whose types are the set of features of the minimalist grammar G plus a fresh type $\mathbf{d}$ for complete derivations, $B \cup F \cup \{\mathbf{d}\}$. The constants are defined as follows:

1. $(w, \epsilon_1 b_1 \dots \epsilon_n b_n a - f_1 \dots - f_p) : ((a \multimap f_1 \multimap \dots \multimap (b_1 \multimap \dots \multimap b_n \multimap f_p) \multimap \mathbf{d}) \multimap \mathbf{d}$ for every lexical entry $(w, \epsilon_1 b_1 \dots \epsilon_n b_n a - f_1 \dots - f_p)$ in C, where $p > 0$,

2. $(w, \epsilon_1 b_1 \dots \epsilon_n b_n a) : ((b_1 \multimap \dots \multimap b_n \multimap a) \multimap \mathbf{d}) \multimap \mathbf{d}$ for every lexical entry $(w, \epsilon_1 b_1 \dots \epsilon_n b_n a)$ in C,

3. $r : c \multimap \mathbf{d}$

In this signature derivations are represented by closed terms of the form:

$$e_1(\lambda y_1^1 \dots y_{p_1}^1 x_1 \dots e_n(\lambda y_1^n \dots y_{p_n}^n x_n.r\, t)$$

where the e_i are constants and where t is a term built only with the variables x_i and y_j^i. The x_i represent the last place where e_i is moved and it is glued there with the components that have licenced its features; whereas the y_j^i represent the traces of e_i in the derivation after it has been moved several times. Of course, if we take every closed terms of this form, many of them will not correspond to an actual minimalist derivation. Moreover, this presentation has some shortcomings since it may have several representations of a single derivation. Indeed, without any further constraint, if

$$e_1(\lambda x_1 y_1^1 \dots y_{p_1}^1 \dots e_n(\lambda x_n y_1^n \dots y_{p_n}^n.r\, t)$$

represents a minimalist derivation then, so is

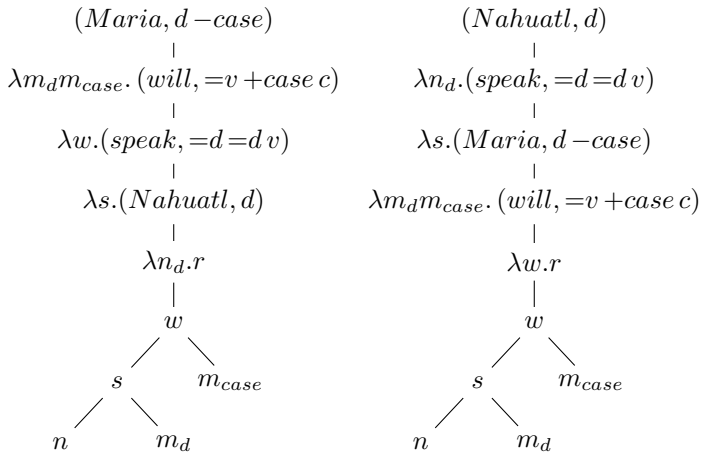
$$e_{\tau(1)}(\lambda x_{\tau(1)} y_1^{\tau(1)} \dots y_{p_{\tau(1)}}^{\tau(1)} \dots e_{\tau(n)}(\lambda x_{\tau(n)} y_1^{\tau(n)} \dots y_{p_{\tau(n)}}^{\tau(n)}.r\, t)$$

for any permutation τ of $[1; n]$. In order to eliminate these *spurious ambiguities* we can constrain the e_i to appear in the same order as in the surface interpretation of the derivation.

Example 5 *A representation of the derivation of example 1 as an intermediate structure can be given using the constants:*

1. $(Maria, d-case) : (d \multimap case \multimap \mathbf{d}) \multimap \mathbf{d}$
2. $(will, =v\, +case\, c) : ((v \multimap case \multimap c) \multimap \mathbf{d}) \multimap \mathbf{d}$
3. $(speak, =d\, =d\, v) : ((d \multimap d \multimap v) \multimap \mathbf{d}) \multimap \mathbf{d}$
4. $(Nahuatl, d) : (d \multimap \mathbf{d}) \multimap \mathbf{d}$

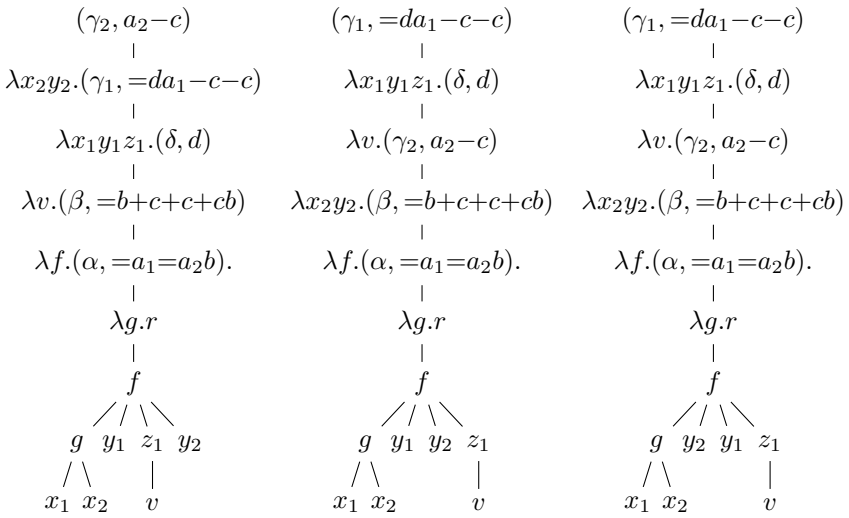
With those constants the derivation is represented by the following λ-terms (only the first one respects the constraint that eliminates spurious ambiguities):



For the example 2, the derivations can be represented using the constants types as:

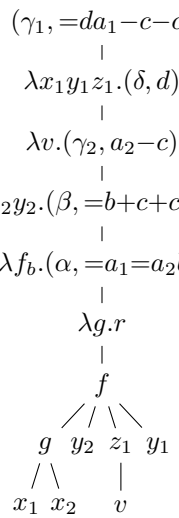
1. $(\alpha, {=}a_1{=}a_2 b) : ((a \multimap a \multimap b) \multimap \mathbf{d}) \multimap \mathbf{d}$
2. $(\beta, {=}b{+}c{+}c{+}cb) : ((b \multimap c \multimap c \multimap c \multimap b) \multimap \mathbf{d}) \multimap \mathbf{d}$
3. $(\delta, d) : (d \multimap \mathbf{d}) \multimap \mathbf{d}$
4. $(\gamma_1, {=}da_1{-}c{-}c) : (a_1 \multimap c \multimap c \multimap \mathbf{d}) \multimap \mathbf{d}$
5. $(\gamma_2, a_2{-}c) : (a_2 \multimap c \multimap \mathbf{d}) \multimap \mathbf{d}$

With this constants, including $r : b \multimap \mathbf{d}$, we can represent the three derivations given in the example 3 with the following λ-terms (here we only give the terms obeying the constraint that avoids spurious ambiguities):



Interestingly the variable v that represent the position of the lexical entry (δ, d), is the argument of the variable z_1 which is the variable that represents the last position the moving piece built around $(\gamma_1, {=}da_1{-}c{-}c)$ occupies after movement. This has to be contrasted with the terms built in example 4 where the lexical entry (δ, d) is placed as an argument of the variable y_1 that represents the first position occupied by $(\gamma_1, {=}da_1{-}c{-}c)$). So with this representation of derivations every movement has already been performed.

Remark that the order in which y_1, z_1 and z_2 appear as arguments of f accounts for the order in which the licencing feature $-c$ of $(\gamma_1, a_1{-}c{-}c)$ and $(\gamma_2, a_2{-}c)$ are checked against the $+c$ feature of $(\beta, {=}b{+}c{+}c{+}cb)$. In particular, this enforces a particular order amongst y_1 and z_1 that represent the two places where $(\gamma_1, a_1{-}c{-}c)$ is moving, y_1 being the first and z_1 being the second. With the choices we made about representing movement order, the following λ-term, even though it is well-typed, does not represent a derivation since it would mean that the second movement of $(\gamma_1, a_1{-}c{-}c)$ has been performed before its first:



There are several things to remark about this representation of minimalist derivations. First of all, contrary to our proposal in MELL, the features are treated explicitly as resources by the logic since they are represented as atomic formulae. The positive or negative versions are represented by the very same atomic type, the way to retrieve whether they are negative or positive amounts to find their polarity in the formula. As polarity in linear logic corresponds to the fact that a formula provides or requires some other formula as resource, the feature checking system of minimalist grammars is adequately modeled that way. This fact has been observed in previous works on logical accounts of minimalist grammars where people have tried to use polarities to elegantly render the feature checking system of minimalist grammars. As we have showed in the example, multiplicative linear logic does not seem to give enough control on the structure of the proofs so as to define derivations as being all the closed terms of a particular type, it explains the reason why those attempts have used logics similar to Lambek calculus. But, since this line of research has not succeeded in defining minimalist derivations uniquely by logical means, it seems to be a difficult problem to define in logical terms exactly the terms that represent minimalist derivations in a signature similar to as Int_G.

Another nice property of the representation of derivations in Int_G, is that, as we wished, the homomorphism that interpret those terms into sentences is quite simple. Indeed, $(w, \epsilon_1 b_1 \dots \epsilon_n b_n a - f_1 \dots - f_p)$ would be interpreted as:

$$\lambda g.g \underbrace{\epsilon \dots \epsilon}_{p\times} (\lambda z_1 \dots z_n.z_n \cdot \dots \cdot z_2 \cdot w \cdot z_1)$$

For the semantic interpretation this would also yield to such simple homomorphisms. Thus being able to compute these intermediate representations factors out the computation that is common to both the syntactic and semantic interpretations of the derivations.

Since it does not seem possible yet to define by logical means the element built on Int_G that are representing minimalist derivations, we must use language theoretic means and define these elements with a homomorphism from Σ_G to Int_G. The definition of the homomorphism computing this intermediate representation from the derivation requires a technique very similar to the one used for the definition of $\mathcal{I}$ which is transforming derivations into sentences. The actual implementation is a little more complex, because we need to handle lists of traces, but it can be represented using the same computational power as for $\mathcal{I}$ (*i.e.* Gödel system T). Due to space limitations, we cannot give here the technical details of the transformation.

4 The point of view of Monadic Second Order Logic

We have seen that linear λ-terms represent adequately minimalist derivations. But we have also seen that the interpretation of those terms is not trivial. This leads us to the conclusion that terms are not an adequate representation of proofs so as to interpret them as strings. We therefore switch to a proof-net representation of those proofs. We could use the proof-nets that represent the λ-terms we have defined in the previous section. We will however not do so and use the proof-nets introduced by Stabler in [Sta99]. Stabler's proof-nets are tailor-made for representing minimalist derivations and are therefore more concise than the ones we would obtain by a direct representation of the proofs built on Σ_G. We then give the syntactic interpretation of those proof-nets with an MSO-transduction which is fairly simple when compared to the previous homomorphisms. This shows that graph transformations are more natural than homomorphisms when dealing with interpretation of minimalist derivations. This comes from the fact that dealing with graphs has the advantage of avoiding the top-down rigidity of terms. As there is no directionality, we can easily find the right place where to put things. This suggests that MSO-transductions of proof-nets should also be used to deal with the semantic interpretation of minimalist derivations.

We first start by defining Stabler's proof-nets as relational structures. Given a ranked alphabet Ω, an Ω-relational structure, is a tuple $(S, (R_a)_{a\in\Omega})$ where S is a finite set, the *carrier* of the Ω-relational structure, and R_a is a subset of S^n (n being the arity of a). Thus we define Stabler's proof-nets as $R(G)$-relational structures where, given a minimalist grammar G, $R(G)$ is the ranked alphabet whose constants are the lexical entries of G and the arity of such a constant (w, l) is the length of l. Given an $R(G)$-relational structure $\Pi = (S, (R_{(w,l)})_{(w,l)\in R(G)})$, a tuple $(x_1, \ldots, x_{|l|})$ that is in $R_{(w,l)}$ is called a (w, l)-link or a link. We say that x_i *belongs* to that link; the type of x_i in that link is the i^{th} feature of l. Of course, not every possible $R(G)$-relational structure is going to represent a derivation of G and as usual we need a correctness criterion to discriminate structures representing an actual derivation from those that do not. If the tuple

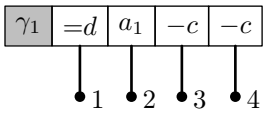
$(x_1, \ldots, x_{|l_1|}, z_0, z_1, \ldots, z_{|l_2|})$ is a $(w, l_1 a l_2)$-link then, x_i is its i^{th} *argument* and z_j is its j^{th} *trace*, z_0 is its *initial trace* and $z_{|l_2|}$ is its *actual trace*. We say that a link l_1 dominates a link l_2 if the initial trace of l_2 is an argument of l_1, we then write $l_2 \lhd l_1$ ($\overset{*}{\lhd}$ is the reflexive transitive closure of $\lhd$). Then the correctness criterion can be stated as:

1. there is a unique element r, the *conclusion* of the proof-net which belongs to exactly one link, its type is c,
2. every element y different from x, belongs to exactly two links p_y and n_y. y is a trace of p_y and an argument of n_y and if the type of y in p_y is a (*resp.* $-f$) then its type in n_y is $=a$ (*resp.* $+f$),
3. on links the relation $\lhd$ forms a tree whose root is p_r,
4. if y_1 and y_2 are respectively the j_1^{th} and the j_2^{th} traces of a link l and if $j_1 < j_2$ then $n_{y_1} \overset{*}{\lhd} n_{y_2}$ and in case $n_{y_1} = n_{y_2}$, y_1 being its i_1^{th} argument and y_2 being its i_2^{th} argument, we have $i_1 < i_2$.

The relational structures that satisfy all those properties are called *proof-nets*. It is easy to prove a property similar to sequentialization so as to show the correspondence between those proof-nets and the closed terms of Σ_G of type $d(c)$. We do not give the formal proof here as it would not bring anything of interest to our exposition.

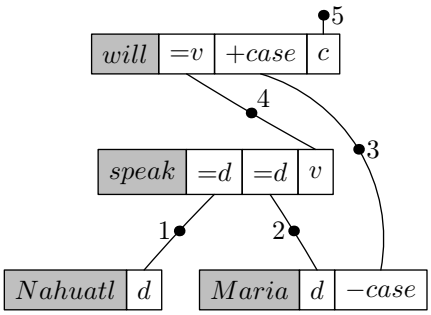
Intuitively the first condition expresses that proof-net form a derivation of the correct type. The second condition expresses that fact that every feature has to be licensed in proof-nets. The third condition enforces the hierarchical construction of the derivation. Finally the last condition makes the movements of a *moving piece* be performed in the right order so that the licensing features are licenced in the linear order in which they appear in the list of features.

Example 6 *We will give a graphical representation of proof-nets. The links of a lexical entry like* $(\gamma_1, {=}da_1{-}c{-}c)$ *will be represented as a hyperedge like:*



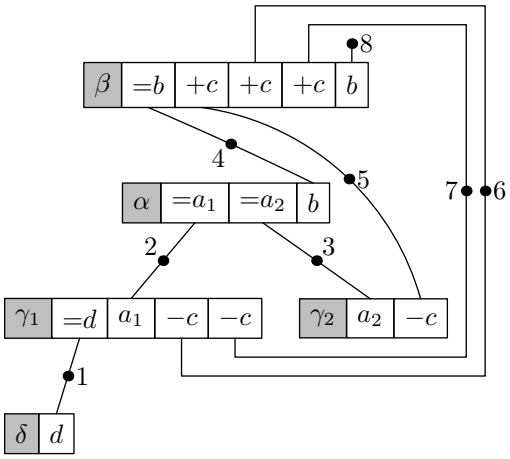
The vertices (the elements of the carrier of the relational structure) are represented with black dots. As in this example, where they are labeled 1, 2, 3 *and* 4*, we will use label in order to designate certain elements of the carrier. Here the* $(\gamma_1, {=}da_1{-}c{-}c)$*-link that is graphically represented is* $(1, 2, 3, 4)$*, this link on has one argument* 1 *it has three traces,* 2, 3 *and* 4*, the actual trace of the link being* 4*;* 1, 2, 3 *and* 4 *respectively have type* $=d$*,* a_1*,* $-c$ *and* $-c$ *in this link. This information is graphically represented in the intuitive way by the origins of the tentacles on the hyperedge.*

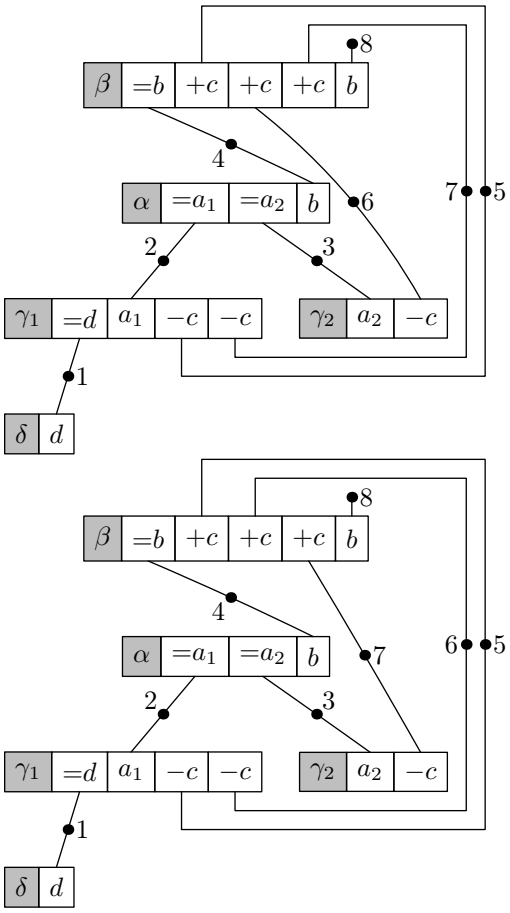
The derivation of example 1 is graphically represented by the following proof-net:



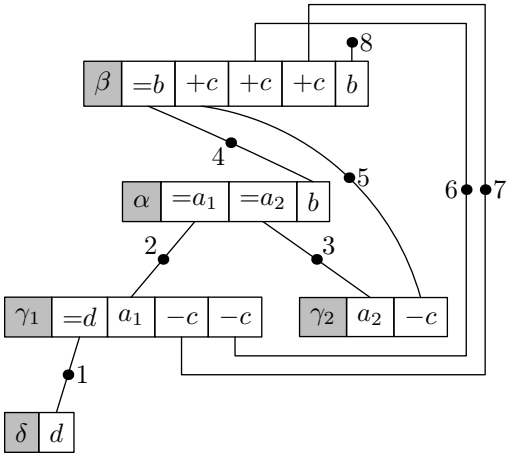
It is easy to check that this proof-structure verifies the two first requirements for being a proof-net. Concerning the third condition, it is fulfilled since we have that the sole $(will, =v{+}casec)$*-link dominates the sole* $(speak, =d{=}dv)$*-link which dominates the* $(Nahuatl, d)$*-link and the* $(Maria, d{-}case)$*-link, and there is no other domination relation. Finally the fourth condition has to be checked only of the vertices* 2 *and* 3 *which are traces of the* $(Maria, d{-}case)$*-link, we can see that* 2 *is an argument of the* $(speak, =d{=}dv)$*-link and that* 3 *is an argument of the* $(will, =v{+}casec)$*-link and that there domination relation agrees with the fourth condition.*

The derivation of example 2 are represented with the following proof-nets:





The fourth condition of the correctness criterion rules out the following structure as being a proof-net:



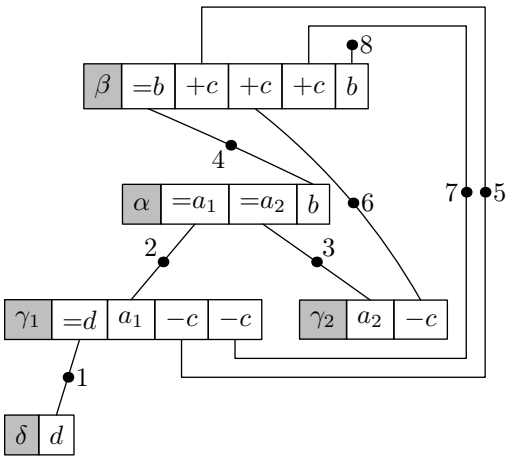
In order to give the syntactic interpretation of those proof-nets, we are going to use the notion of MSO-transduction (see [Cou94]). An MSO-transduction is transforming a Ω-relational structure into a Δ-relational structure as follows:

1. first a finite number of copies of the carrier of the initial relational structure is done,
2. for each copy an MSO-formula specifies which elements of the carrier are kept,
3. then for each relation in Δ, some MSO-formulae specify which tuples of the remaining points verify this relation.

For the sake of simplicity, the MSO-transduction that gives the syntactic interpretation of proof-nets is specified as the composition of two MSO-transductions. A first transduction transforms the proof-nets into a string represented as a relational structure. This string may contain some occurrences of ϵ, the empty string. The second transduction is simply removing these occurrences of ϵ. This second transduction is quite simple and we will not enter into the details of its implementation.

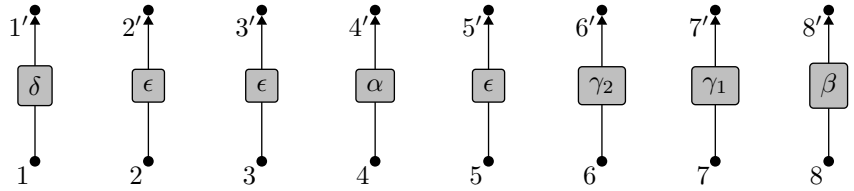
Thus, the first transduction transforms $R(G)$-relational structures $\mathcal{R}$ that are proof-nets into a $W(G)$-relational structures $\mathcal{W}$ where $W(G)$ is the set of binary relations $W \cup \{\epsilon\}$. We first take two copies of the elements of $\mathcal{R}$ and we keep every vertex of each copy in the new structure. We write $fst(x)$ and $snd(x)$ two respectively denote the first and second copy of x in the new structure. In the new structure, we add a relation $\epsilon(fst(x), snd(x))$ if x is not the actual trace of p_x and $w(fst(x), snd(x))$ if x is the actual trace of p_x which is a (w, l)-link.

So as to explain the transduction, we will use the following derivation as a running example.



The first step of the MSO-transduction that we have just described transforms this derivation structure into the following structure. We keep the same label for the vertices of the first copy of the carrier and we use primed labels for

the second copy, furthermore we have put an arrow on one of the tentacle which symbolise the right of the represented letter (the second argument in the represented relation):

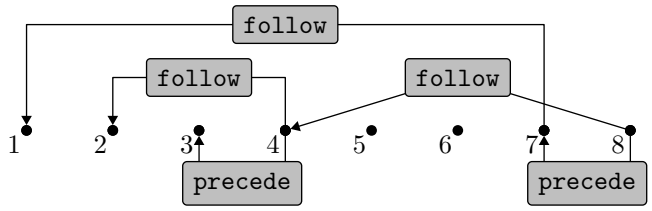


We now have all the ingredients that are necessary to construct the string we want. It just remains to concatenate them suitably. This concatenation will be performed simply by putting epsilon relations where needed. If we want to concatenate the words in which x and y are transformed in $\mathcal{W}$, it suffices to put a relation $\epsilon(snd(x), fst(y))$ in $\mathcal{W}$. This concatenation will be possible only if we can express in MSO the relation $x < y$ which is the linear order of the chain we want to build. If we look at the level of a $(w, l_1 a l_2)$-link $(x_1, \ldots, x_{|l_1|}, z_0, z_1, \ldots, z_{|l_2|})$ in order to build the string around $z_{|l_2|}$, the actual trace, we need to build the string $s_{|l_1|} \ldots s_2 w s_1$ if s_i is the string that is constructed around x_i. To do so, we need to be able to find the elements of $\mathcal{R}$ that start or end the string that is built around an element x. To achieve this, we introduce two binary predicates $\texttt{follow}(x, y)$ and $\texttt{precede}(x, y)$:

1. $\texttt{follow}(x, y)$ if and only if x is the actual trace of the link p_x, p_x has at least one argument, and y is the first argument of p_x,

2. $\texttt{precede}(x, y)$ holds if and only if x is the actual trace of the link p_x, p_x has at least two arguments and y is the last argument of p_x.

The relations $\texttt{follow}(x, y)$ and $\texttt{precede}(x, y)$ hold in a proof-net when y is the element in p_x whose start or end is also the start or the end of the string built around x when x is the actual trace of some link. In the example, if we represent pictorially the relations `follow` and `precede` (the arrow is pointing at the second element of the predicate) then we would obtain:



It is obvious that $\texttt{follow}(x, y)$ and $\texttt{precede}(x, y)$ are MSO-definable predicates. We note $\texttt{follow}^*(x, y)$ and $\texttt{precede}^*(x, y)$ the respective reflexive and transitive closures of $\texttt{follow}(x, y)$ and $\texttt{precede}(x, y)$. These relations are also MSO-definable since transitive closures of MSO-definable relations are also MSO-

definable. We then define the relations $\texttt{start}(x, y)$ and $\texttt{end}(x, y)$ as being:

$$\begin{aligned} \texttt{start}(x, y) &\equiv \texttt{precede}^*(x, y) \wedge \forall z.\texttt{precede}^*(y, z) \Rightarrow y = z \\ \texttt{end}(x, y) &\equiv \texttt{follow}^*(x, y) \wedge \forall z.\texttt{follow}^*(y, z) \Rightarrow y = z \end{aligned}$$

On a proof-net the relation `start` and `end` define functions, *i.e.* for every x there is exactly one y_s and exactly one y_e such that $\texttt{start}(x, y_s)$ and $\texttt{end}(x, y_e)$. According to the definition, we obtain the following table to describe these relations:

x	y_s	y_e
1	1	1
2	2	2
3	3	3
4	3	2

x	y_s	y_e
5	5	5
6	6	6
7	7	1
8	7	2

We are now in position to define the relation $x < y$ which says that the trace introduced by x appears just before the trace introduced by y. If $(x_1, \ldots, x_n, z_0, \ldots, z_p)$ is a $(w, {=}b_1 \ldots \epsilon_n b_n, z_0, \ldots, z_p)$-link then we have that $x < y$ if and only if one of the following holds:

1. if $n > 0$, $\texttt{start}(x_1, x) \wedge y = z_p$,
2. if $n > 1$, $\texttt{end}(x_2, x) \wedge y = z_p$
3. for some $1 < i < n$, $\texttt{end}(x_{i+1}, x) \wedge \texttt{start}(x_i, y)$

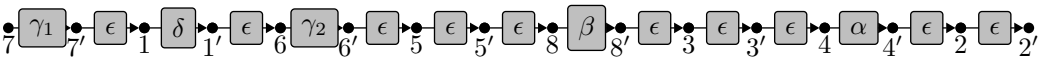
This is enough to define the precedence relation. It is provable that the transitive closure of $<$ is a total order when $\mathcal{R}$ is a proof-net. On our example the relation $<$ would order the set $\{1; \ldots; 8\}$ as follows:

$$7 < 1 < 6 < 5 < 8 < 3 < 4 < 2$$

Indeed, if we look at 8 the conclusion of the proof-net, it is the actual trace of the $(\beta, {=}b{+}c{+}c{+}cb)$-link of the proof-net and, respectively, the first second third and fourth arguments of the link are 4, 5, 6 and 7. As we have $\texttt{start}(4, 3)$, we have $8 < 3$, and as we have $\texttt{end}(7, 1)$ and $\texttt{start}(6, 6)$ we have $1 < 6$, as we have $\texttt{end}(6, 6)$ and $\texttt{start}(5, 5)$, we have $6 < 5$ and as we have $\texttt{end}(5, 5)$ we have $5 < 8$. Similarly, by looking at every vertex that is the actual trace of some link we can complete the relation $<$ as above. Since $\texttt{follow}^*$, $\texttt{precede}^*$ are MSO-definable relations, then `start` and `end` are MSO-definable, and thus $<$ is an MSO-definable relation.

As mentioned previously the relation $<$ describes the way the words represented by binary relations must be concatenated in order to produce the resulting string. As we describe the transduction as the composition of two transductions, the first one building the resulting string interspersed with empty strings and the second one deleting the empty strings, we represent concatenation of two nodes as putting x and y as putting an empty string between the second copy of x, and the first copy of y. So if we add this concatenation requirement to

the definition of ϵ, we have that $\epsilon(fst(x), snd(x))$ holds if and only if x is not the actual trace of any link and $\epsilon(snd(x), fst(y))$ holds if and only if $x < y$. In our example concatenating 5 and 8 amounts to add the pair $(5', 8)$ to the relation ϵ. So after the MSO-transduction we have defined is applied to the derivation we have taken as an example we obtain the following relational structure that represents the string $\gamma_1\epsilon\delta\epsilon\gamma_2\epsilon\epsilon\epsilon\beta\epsilon\epsilon\epsilon\alpha\epsilon\epsilon$:



Then a simple string homomorphism can suppress the occurrences of ϵ. In the example we would then obtain the relational structure representing, as expected, the string $\gamma_1\delta\gamma_2\beta\alpha$. Such string to string transformations can be defined with MSO-transductions. As MSO-transductions are closed under composition, we have showed that the interpretation of proof-nets into strings can be computed using MSO-transductions.

Now if we are concerned with the shortest move constraint, we first need to remark that proof-nets are MSO-definable in $R(G)$-relational structure and then we can easily see that the shortest move constraint gives proof-nets with a bounded treewidth. These proof-nets can therefore be represented as the language of a hyperedge replacement grammar (HR-languages). As HR-languages are closed under MSO-transduction we get that the languages of MGs with shortest move are HR-languages. But it is known that the string languages of HR grammars coincide with MCFLs [Wei92].

5 Conclusion

This work is mainly aiming at clarifying the status of derivations in minimalist grammars without the shortest move constraint. It also tries to promote a certain attitude towards formalisms describing natural languages. This attitude consists in trying to identify and study the abstract syntax of the formalisms. Abstract syntax plays a central role in formalisation because it is the particular place where the connection between syntax and semantics can be made. It is also at the level of abstract syntax that linguistic ideas like **move** and **merge** have the greatest influence. We have emphasized this role from a mathematical point of view by showing that these two operations dramatically reduce the complexity of generating syntactic structure by allowing a rather powerful polymorphism and also by taming variable binding using only third order types.

From the mathematical side, this careful study of the derivations in connection with some simple ideas coming from formal language theory has lead us to several new results. First we have showed that the membership problem for MGs is as difficult as the problem of proof-search in MELL. This result shows that it is not obvious at all that the membership problem for MGs is decidable or not. Second we have obtained the unintuitive result that the languages of MGs may not be semi-linear contradicting a conjecture by [GM07]. Finally

we have obtained a rather interesting and new logical characterization of these derivations as closed linear λ-terms of a certain type. This characterization can be said as being logical since, with the Curry-Howard isomorphism, closed linear λ-terms correspond proofs in implicative linear logic with proper axioms (they play the role of the exponentials) and since we do not appeal to extra constraints in order to rule out terms that would not represent derivations. Furthermore, this way of representing derivations is made even more natural by the aforementioned Turing-equivalence of the membership problem for MGs and proof-search in MELL.

From a linguistic point of view, this careful study also has some outcomes. Indeed, the representation of the derivations of MGs we propose are unambiguous in the sens that, contrary to Stabler's proposal, they only represent the syntactic analysis of one sentence. Furthermore, since the ambiguity of Stabler's proposal makes sentences that have different meanings have the same derivation, the disambiguation we propose should allow a simpler interface between syntax and semantics. Our proposal makes it also clear that movement and traces are adequately rendered by variable binding. It also gives a methodological way of extending MGs. Indeed, the linear λ-calculus and typing theory offers a good framework for devising sensible improvements of MGs.

Nevertheless, our proposal has several defects. First of all, contrary to what would be expected, the feature checking system of MGs is not rendered by using the resource sensitivity of linear logic but it is rather modeled by the particular management of types that we use. Moreover, the linearisation of those structures to string uses a non-trivial homomorphism and most of the computation that are induced by the **move** operation is common to both the syntactic linearisation and the semantic interpretation. This can be fixed by transforming derivations with a homomorphism into intermediate structures. Interestingly at the level of these intermediate structures the feature checking system of MGs is rendered by the resource sensitivity of linear logic, unfortunately we would need extra logical constraints so as to rule out certain terms that do not represent MG derivations and to avoid some spurious ambiguities.

Finally combining proof-nets and MSO related techniques we are able to give a simple interpretation of derivation structures using MSO-transductions. The main problem of this approach is the fact that a proof-net is actually transformed into a string needs to be proved while this is guaranteed by type-checking in the homomorphic approach that maps λ-terms to strings. But, this approach is rather new in mathematical linguistic, so that maybe there could be certain way of guarantying certain properties easily. It would also be interesting to see how this approach could be used for semantic interpretation of derivations.

This work appeals to various techniques from formal language theory. A rule-based one that is rendered by our representation of derivations as linear λ-terms and the homomorphic interpretation of those terms. A descriptive one that resembles Model Theoretic Syntax [PS01] that uses MSO to describe and interpret derivations. This variety of techniques allows us to appeal to many results in the literature to retrieve results like [Mic98] from results on rule-based techniques or from results on MSO related techniques. These two points

of view are complementary. The first one helps to understand computational difficulty, to design parsing algorithm. The second one simplifies greatly the overall description of the formalism.

This duality in formalization helps us to understand in a better way the gap that separates computational linguists from descriptive linguists. While the first are interested in accounting for the way sentences are constructed by designing some generation process, the second are mostly describing linguistic data, explaining how constituents are related in certain circumstances. This opposition seems very similar to the one that opposes MTS to Context Free Grammars.

References

[Amb07] Maxime Amblard. *Calculs de représentations s'emantique et syntaxe générative : les grammaires minimalistes catégorielles*. PhD thesis, Université de Bordeaux 1, 2007.

[Cou94] Bruno Courcelle. Monadic second-order definable graph transductions: A survey. *Theoritical Computer Science*, 126(1):53–75, 1994.

[Cur61] Haskell B. Curry. Some logical aspects of grammatical structure. In Roman Jakobson, editor, *Structure of Language and Its Mathematical Aspects*, pages 56–68. AMS Bookstore, 1961.

[dG01] Philippe de Groote. Towards abstract categorial grammars. In Association for Computational Linguistic, editor, *Proceedings 39th Annual Meeting and 10th Conference of the European Chapter*, pages 148–155. Morgan Kaufmann Publishers, 2001.

[dG07] Philippe de Groote. Towards a montagovian account of dynamics. In *Proceedings of Semantics in Linguistic Theory XVI*. CLC Publications, 2007.

[dGGS04] Philippe de Groote, Bruno Guillaume, and Sylvain Salvati. Vector addition tree automata. *19-th IEEE symposium on Logic in Computer Science*, pages 63–74, 2004.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GM07] Hans-Martin Gärtner and Jens Michaelis. Locality conditions and the complexity of minimalist grammars: a preliminary survey. In James Rogers, editor, *Workshop on Model Theoretic Syntax (MTS@10), Dublin, Ireland*, pages 89–100, 2007.

[HP79] John E. Hopcroft and Jean-Jacques Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theoretical Computer Science*, 8(2):135–159, 1979.

[Kob06] Gregory Michael Kobele. *Generating Copies: An Investigation into Structural Identity in Language and Grammar*. PhD thesis, University of California Los Angeles, 2006.

[Lec03] Alain Lecomte. A computational approach to minimalism. In *Proceedings of ICON-2003, International Conference on Natural Language*, pages 20–31. Central Institute of Indian Languages, 2003.

[Lec04] Alain Lecomte. Derivations as proofs : a logical approach to minimalism. In *Proceedings of CG 2004*, 2004.

[LR99] Alain Lecomte and Christian Retoré. Towards a minimal logic for minimalist grammars: a tranformational use of lambek calculus. In *Formal Grammar 99*, 1999.

[LR01] Alain Lecomte and Christian Retoré. Extending lambek grammars: a logical account of minimalist grammars. In *Proceedings of the 39th meeting of the Association for Computational Linguistics,ACL 2001*, pages 354–361, 2001.

[Mic98] Jens Michaelis. Derivational minimalism is mildly context-sensitive. In Michael Moortgat, editor, *LACL*, volume 2014 of *Lecture Notes in Computer Science*, pages 179–198. Springer, 1998.

[Mus01] Reinhard Muskens. Lambda Grammars and the Syntax-Semantics Interface. In R. van Rooy and M. Stokhof, editors, *Proceedings of the Thirteenth Amsterdam Colloquium*, pages 150–155, Amsterdam, 2001.

[PS01] Geoffrey K. Pullum and Barbara C. Scholz. On the distinction between model-theoretic and generative-enumerative syntactic frameworks. In Philippe de Groote, Glyn Morrill, and Christian Retoré, editors, *LACL*, volume 2099 of *Lecture Notes in Computer Science*, pages 17–43. Springer, 2001.

[Sal07] Sylvain Salvati. Encoding second order string acg with deterministic tree walking transducers. In S. Wintner, editor, *Proceedings FG 2006: the 11th conference on Formal Grammars*, FG Online Proceedings, pages 143–156. CSLI Publications, 2007.

[Sta97] Edward Stabler. Derivational minimalism. In Christian Retoré, editor, *Logical Aspects of Computational Linguistics, LACL'96*, volume 1328 of *LNCS/LNAI*, pages 68–95. Springer-Verlag, 1997.

[Sta99] Edward P. Stabler. Remnant movement and structural complexity. In *Constraints and Resources in Natural Language, Studies in Logic, Language and Information*, pages 299–326. CSLI, 1999.

[Wei92] David J. Weir. Linear context-free rewriting systems and deterministic tree-walking transducers. In *ACL*, pages 136–143, 1992.

[Yos06] Ryo Yoshinaka. Linearization of affine abstract categorial grammars. In *Proceedings of the 11th conference on Formal Grammar*, 2006.