

# On Two Extensions of Abstract Categorical Grammars

Philippe de Groote<sup>1</sup>, Sarah Maarek<sup>2</sup>, and Ryo Yoshinaka<sup>1</sup>

<sup>1</sup> LORIA & INRIA-Lorraine  
Philippe.de.Groote@loria.fr,  
Ryo.Yoshinaka@loria.fr

<sup>2</sup> LORIA & Université Nancy 2  
Sarah.Maarek@loria.fr

## 1 Introduction

The abstract categorical grammars (ACGs, for short) are a type-theoretic grammatical formalism intended for the description of natural languages [1]. It is based on the implicative fragment of multiplicative linear logic, which results in a rather simple framework.

From a language-theoretic standpoint, however, this simplicity is not synonymous of a weak expressive power [2,4]. In particular, the string languages generated by the second-order ACGs, whose parsing is known to be polynomial, corresponds to the class of mildly context sensitive languages [7,11]. Nevertheless, in [5], we have argued that it would be interesting to increase the intentional expressive power of the formalism by providing high level constructs.

From a formal point of view, to provide ACGs with new constructs consists in extending the type system of the formalism. In the present paper, we study two such type-theoretic extensions of the ACGs. They consist in providing the ACG type system with Cartesian product and dependent product, respectively. We prove that both extensions result in Turing-complete formalisms that allow any recursively enumerable language to be specified.

The paper is organized as follows. In the next section, we remind the reader of the definition of an abstract categorical grammar. In section 3, we study ACGs with Cartesian product. In section 4, we study ACGs with dependent product.

## 2 Abstract Categorical Grammars

Let  $A$  be a set of atomic types. The set  $\mathcal{T}_A$  of *linear implicative types* built upon  $A$  is inductively defined by the following rules:

$$\mathcal{T}_A ::= A \mid (\mathcal{T}_A \multimap \mathcal{T}_A)$$

A *higher-order linear signature* is defined to be a triple  $\Sigma = \langle A, C, \tau \rangle$ , where:

1.  $A$  is a finite set of atomic types;
2.  $C$  is a finite set of constants;

3.  $\tau$  is a mapping from  $C$  to  $\mathcal{T}_A$ .

A higher-order linear signature will also be called a *vocabulary*. In the sequel, we will write  $A_\Sigma$ ,  $C_\Sigma$ , and  $\tau_\Sigma$  to designate the three components of a signature  $\Sigma$ , and we will write  $\mathcal{T}_\Sigma$  for  $\mathcal{T}_{A_\Sigma}$ .

The set of untyped  $\lambda$ -terms is defined as usual, and one takes the relation of  $\beta\eta$ -conversion as the notion of equality between  $\lambda$ -terms. Then, given a signature  $\Sigma$ , the set of well-typed linear  $\lambda$ -terms  $A_\Sigma$  is the set of  $\lambda$ -terms that may be assigned a linear implicative types by the following typing rules.

$$\begin{array}{c} \vdash_{\Sigma} c : \tau_{\Sigma}(c) \quad (\text{cons}) \\ \\ x : \alpha \vdash_{\Sigma} x : \alpha \quad (\text{var}) \\ \\ \frac{\Gamma, x : \alpha \vdash_{\Sigma} t : \beta}{\Gamma \vdash_{\Sigma} (\lambda x. t) : (\alpha \multimap \beta)} \quad x \notin \text{dom}(\Gamma) \quad (\text{abs}) \\ \\ \frac{\Gamma \vdash_{\Sigma} t : (\alpha \multimap \beta) \quad \Delta \vdash_{\Sigma} u : \alpha}{\Gamma, \Delta \vdash_{\Sigma} (tu) : \beta} \quad \text{dom}(\Gamma) \cap \text{dom}(\Delta) = \emptyset \quad (\text{app}) \end{array}$$

In the above rules, as usual,  $\Gamma$  and  $\Delta$  range over typing environments, i.e., finite sets of declarations of the form ' $x : \alpha$ ' such that each variable is declared at most once. ' $\Gamma, \Delta$ ' stands for  $\Gamma \cup \Delta$ , and ' $\Gamma, x : \alpha$ ' for  $\Gamma \cup \{x : \alpha\}$ . Finally,  $\text{dom}(\Gamma)$  denotes the set of variable declared in  $\Gamma$ .

Given two signatures  $\Sigma$  and  $\Xi$ , a *lexicon*  $\mathcal{L}$  from  $\Sigma$  to  $\Xi$  (in notation,  $\mathcal{L} : \Sigma \rightarrow \Xi$ ) is defined to be a pair  $\mathcal{L} = \langle \eta, \theta \rangle$  such that:

1.  $\eta$  is a mapping from  $A_\Sigma$  into  $\mathcal{T}_\Xi$ ;
2.  $\theta$  is a mapping from  $C_\Sigma$  into  $\Lambda_\Xi$ ;
3. for every  $c \in C_\Sigma$ , the following typing judgement is derivable:

$$\vdash_{\Xi} \theta(c) : \hat{\eta}(\tau_{\Sigma}(c)),$$

where  $\hat{\eta} : \mathcal{T}_\Sigma \rightarrow \mathcal{T}_\Xi$  is the unique homomorphic extension of  $\eta$ .<sup>1</sup>

As stated in Condition 3 of the above definition, there exists a unique type homomorphism  $\hat{\eta} : \mathcal{T}_\Sigma \rightarrow \mathcal{T}_\Xi$  that extends  $\eta$ . Similarly, there exists a unique  $\lambda$ -term homomorphism  $\hat{\theta} : \Lambda_\Sigma \rightarrow \Lambda_\Xi$  that extends  $\theta$ .<sup>2</sup> In the sequel,  $\mathcal{L}$  will denote both  $\hat{\eta}$  and  $\hat{\theta}$ , the intended meaning being clear from the context. In addition, when  $\Gamma$  denotes a typing environment ' $x_1 : \alpha_1, \dots, x_n : \alpha_n$ ', we will write  $\mathcal{L}(\Gamma)$  for ' $x_1 : \mathcal{L}(\alpha_1), \dots, x_n : \mathcal{L}(\alpha_n)$ '. Using these notations, we have that Condition 3 induces the following property:

$$\text{if } \Gamma \vdash_{\Sigma} t : \alpha \text{ then } \mathcal{L}(\Gamma) \vdash_{\Xi} \mathcal{L}(t) : \mathcal{L}(\alpha).$$

We now give the main definition of this section. An abstract categorial grammar is a quadruple  $\mathcal{G} = \langle \Sigma, \Xi, \mathcal{L}, s \rangle$  where:

<sup>1</sup> That is  $\hat{\eta}(a) = \eta(a)$  and  $\hat{\eta}(\alpha \multimap \beta) = \hat{\eta}(\alpha) \multimap \hat{\eta}(\beta)$ .

<sup>2</sup> That is  $\hat{\theta}(c) = \theta(c)$ ,  $\hat{\theta}(x) = x$ ,  $\hat{\theta}(\lambda x. t) = \lambda x. \hat{\theta}(t)$ , and  $\hat{\theta}(tu) = \hat{\theta}(t) \hat{\theta}(u)$ .

1.  $\Sigma$  and  $\Xi$  are two higher-order linear signatures, which are called the *abstract vocabulary* and the *object vocabulary*, respectively;
2.  $\mathcal{L} : \Sigma \rightarrow \Xi$  is a lexicon from the abstract vocabulary to the object vocabulary;
3.  $s \in \mathcal{T}_\Sigma$  is a type of the abstract vocabulary, which is called the *distinguished type* of the grammar.

The intuition behind this definition is that the abstract vocabulary is used to express the parse structures of the grammar while the object vocabulary corresponds somehow to the terminal symbols of the grammar. This explains that an ACG generates two languages: an *abstract language* and an *object language*. The abstract language is the set of closed linear  $\lambda$ -terms that are built on the abstract vocabulary, and whose type is the distinguished type:

$$\mathcal{A}(\mathcal{G}) = \{t \in \Lambda_\Sigma \mid \vdash_\Sigma t : s \text{ is derivable}\}$$

On the other hand, the object language is defined to be the image of the abstract language by the lexicon:

$$\mathcal{O}(\mathcal{G}) = \{t \in \Lambda_\Xi \mid \exists u \in \mathcal{A}(\mathcal{G}). t = \mathcal{L}(u)\}$$

Then, given some term  $t \in \Lambda_\Xi$ , the membership problem for  $\mathcal{G}$  consists in deciding whether  $t$  belongs to  $\mathcal{O}(\mathcal{G})$ , i.e., whether there exists  $u \in \mathcal{A}(\mathcal{G})$  such that  $\mathcal{L}(u) = t$ . The decidability of this problem is open. What is known, is that it is equivalent to the decidability of the multiplicative exponential fragment of linear logic [4,12]. It is also known that, for second-order ACGs (i.e., ACGs whose abstract constants are at most second-order) membership is decidable in polynomial time [11,7].

### 3 Abstract Categorical Grammars with Cartesian Product

Feature structures, which are akin to records, are one of the main primitives of unification based grammatical formalisms such as HPSG. Records themselves are intensively used in Ranta’s GF [10]. This explains our motivation in defining an extension of the ACGs where a notion of record would be available. From a theoretical point of view this amounts to extend the ACG typing system with a Cartesian product.

#### 3.1 Definition

From the perspective of linear logic, the Cartesian product corresponds to the additive conjunction. Consequently, the set of types must be extended as follows:

$$\mathcal{T}_A ::= A \mid (\mathcal{T}_A \multimap \mathcal{T}_A) \mid (\mathcal{T}_A \& \mathcal{T}_A)$$

Then the set of untyped  $\lambda$ -terms must be extended with a pair constructor together with its two projection operators:

$$T ::= c \mid x \mid \lambda x. T \mid (TT) \mid \langle T, T \rangle \mid (\pi_1 T) \mid (\pi_2 T)$$

The notion of equality between  $\lambda$ -terms must be adapted accordingly by taking into account the following additional reduction rules:

$$\begin{aligned} \pi_1 \langle t, u \rangle &\rightarrow t && \text{(left projection)} \\ \pi_2 \langle t, u \rangle &\rightarrow u && \text{(right projection)} \\ \langle \pi_1 t, \pi_2 t \rangle &\rightarrow t && \text{(surjective pairing)} \end{aligned}$$

Finally, the three following rules are added to the typing system:

$$\frac{\Gamma \vdash_{\Sigma} t : \alpha \quad \Gamma \vdash_{\Sigma} u : \beta}{\Gamma \vdash_{\Sigma} \langle t, u \rangle : \alpha \& \beta} \quad \text{(pair)}$$

$$\frac{\Gamma \vdash_{\Sigma} t : \alpha \& \beta}{\Gamma \vdash_{\Sigma} \pi_1 t : \alpha} \quad \text{(left proj.)} \qquad \frac{\Gamma \vdash_{\Sigma} t : \alpha \& \beta}{\Gamma \vdash_{\Sigma} \pi_2 t : \beta} \quad \text{(right proj.)}$$

Then, the very definitions of a lexicon and of an abstract categorial grammar may be kept unchanged.

### 3.2 Turing Completeness

As we already mentioned, membership for purely implicative ACGs is equivalent to provability in multiplicative exponential linear logic (the decidability of which is still open). Analogously, one may expect that membership for ACGs with Cartesian product is equivalent to provability in multiplicative additive exponential linear logic (which is known to be undecidable [9]). This is indeed the case, as shown in this section.

Given any recursively enumerable set of integers, we will construct an ACG whose object language is this given set. As well known,  $k$ -counter machines compute arbitrary recursive functions [8]. A  $k$ -counter machine is a quadruple  $M = \langle Q, \delta, q_0, q_f \rangle$  where  $Q$  is a finite set of *states*,  $\delta$  is a finite set of *transition rules*,  $q_0 \in Q$  is the *initial state* and  $q_f \in Q$  is the *final state*. A machine has  $k$  counters in each of which one natural number is stored. Each transition rule in  $\delta$  has one of the following forms:

$$\langle q \mathbf{Inc} \ i \ r \rangle, \quad \langle q \mathbf{Dec} \ i \ r \rangle, \quad \langle q \mathbf{Zero?} \ i \ r \rangle$$

for some  $i \in \{1, \dots, k\}$  and  $q, r \in Q$ . An *instantaneous description (ID)* is an element of  $Q \times \mathbb{N}^k$ , where  $\mathbb{N}$  is the set of natural numbers  $0, 1, 2, \dots$ . When the machine is in the state  $q$  and the increment rule  $\langle q \mathbf{Inc} \ i \ r \rangle$  is applied, then the machine increments the  $i$ th counter by 1 and it enters the state  $r$ . The decrement rule  $\langle q \mathbf{Dec} \ i \ r \rangle$  is applied to the machine only when it is in the state  $q$  and moreover the entry of the  $i$ th counter is not zero. In that case, the machine decrements the  $i$ th counter by 1 and it goes to the state  $r$ . The zero-test rule  $\langle q \mathbf{Zero?} \ i \ r \rangle$

is applied to the machine only when it is in the state  $q$  and moreover the entry of the  $i$ th counter is exactly zero. In that case, the machine moves to the state  $r$ . For two IDs  $X, Y \in Q \times \mathbb{N}^k$ , we write  $X \rightarrow Y$  when the transition from  $X$  to  $Y$  is possible. We say that  $M$  *accepts*  $\mathbf{m} \in \mathbb{N}^k$  if and only if  $\langle q_0, \mathbf{m} \rangle \xrightarrow{*} \langle q_f, 0^k \rangle$ , where  $\xrightarrow{*}$  is the reflexive transitive closure of  $\rightarrow$  and  $0^k$  is short for the sequence of 0s of length  $k$ . The following theorem is an alternative presentation of Lambek's result [8].

**Theorem 1.** *For any recursive  $n$ -ary function  $\phi$ , there is a  $k$ -counter machine  $M$  with some  $k > n$  such that*

$$\phi(m_1, \dots, m_n) = m_0 \text{ iff } M \text{ accepts } \langle m_0, m_1, \dots, m_n, 0^{k-n-1} \rangle.$$

Our encoding of  $k$ -counter machines by ACGs is given in a way similar to Lincoln et al.'s technique for showing the undecidability of the multiplicative additive exponential linear logic [9]. They have introduced a variant of two-counter machines, which they call *and-branching two-counter machines without zero-test* (2-ACMs, for short), and shown that 2-ACMs simulate standard two-counter machines. An *and-branching  $k$ -counter machine without zero-test* ( $k$ -ACM, for short) is also a quadruple  $M = \langle Q, \delta, q_0, q_f \rangle$ , where  $\delta$  has no zero-test rules. Instead, it has *fork rules* of the form  $\langle q \text{ Fork } r_1 r_2 \rangle$  with  $q, r_1, r_2 \in Q$ , which allow us to simulate zero-test rules. An ID of a  $k$ -ACM is a finite sequence of elements of  $Q \times \mathbb{N}^k$ . The fork rule  $\langle q \text{ Fork } r_1 r_2 \rangle$  allows the machine to move from  $X_1 \langle q, \mathbf{m} \rangle X_2$  to  $X_1 \langle r_1, \mathbf{m} \rangle \langle r_2, \mathbf{m} \rangle X_2$ , where  $\mathbf{m} \in \mathbb{N}^k$  and  $X_1, X_2 \in (Q \times \mathbb{N}^k)^*$ . Only fork rules increase the number of elements of an ID of the machine. The transition by an increment or decrement rule is defined in the same way as standard  $k$ -counter machines. A sequence of the form  $\langle q_f, 0^k \rangle \dots \langle q_f, 0^k \rangle$  is called an *accepting ID*. One says that  $M$  *accepts from an ID*  $X$  if and only if  $X \xrightarrow{*} \langle q_f, 0^k \rangle \dots \langle q_f, 0^k \rangle$ . One also says that  $M$  *accepts*  $\mathbf{m} \in \mathbb{N}^k$  if and only if  $M$  accepts from  $\langle q_0, \mathbf{m} \rangle$ .

Lincoln et al.'s proof for that a 2-ACM simulates an arbitrary standard two-counter machine is also applied to  $k$ -counter machines. It is easy to modify a  $k$ -counter machine so that it has no transition rule going out from the final state while keeping the acceptable  $k$ -tuples of natural numbers. Then a zero-test rule  $\langle q \text{ Zero? } i r \rangle$  is simulated by the following rules of a  $k$ -ACM:

$$\langle q \text{ Fork } r s_i \rangle, \quad \langle s_i \text{ Dec } j s_i \rangle \text{ for all } j \neq i, \quad \langle s_i \text{ Fork } q_f q_f \rangle$$

where  $s_i$  is a new state not in the original set of states.

**Lemma 1.** *Any  $k$ -counter machine is simulated by a  $k$ -ACM.*

Now, to any  $k$ -ACM  $M = \langle Q, \delta, q_0, q_f \rangle$ , we associate the following ACG  $\mathcal{G}_M = \langle \Sigma_M, \Sigma_{\mathbb{N}}, \mathcal{L}, s \rangle$ :

$$\begin{array}{l}
 \Sigma_M \left\{ \begin{array}{l}
 s : \text{type}, \\
 a_i : \text{type} \quad \text{for all } i \in \{1, \dots, k\}, \\
 q : \text{type} \quad \text{for all } q \in Q, \\
 c_f : q_f, \\
 c_\rho : \alpha_\rho \quad \text{for all } \rho \in \delta, \\
 \text{where } \alpha_\rho = \begin{cases} (a_i \multimap r) \multimap q & \text{for } \rho = \langle q \mathbf{Inc} \ i \ r \rangle, \\
 r \multimap (a_i \multimap q) & \text{for } \rho = \langle q \mathbf{Dec} \ i \ r \rangle, \\
 (r_1 \ \& \ r_2) \multimap q & \text{for } \rho = \langle q \mathbf{Fork} \ r_1 \ r_2 \rangle, \end{cases} \\
 d_0 : q_0 \multimap s, \\
 d_i : (a_i \multimap s) \multimap s.
 \end{array} \right. \\
 \\
 \Sigma_N \left\{ \begin{array}{l}
 o : \text{type}, \\
 0 : o, \\
 S : o \multimap o,
 \end{array} \right. \\
 \\
 \mathcal{L} \left\{ \begin{array}{l}
 s := (o^k \multimap o) \multimap o, \\
 a_i := o \multimap o \quad \text{for all } i \in \{1, \dots, k\}, \\
 q := o \multimap o \quad \text{for all } q \in Q, \\
 c_f := \lambda z. z, \\
 c_\rho := \begin{cases} \lambda x. x (\lambda z. z) & \text{for } \rho = \langle q \mathbf{Inc} \ i \ r \rangle, \\
 \lambda x y z. x (y z) & \text{for } \rho = \langle q \mathbf{Dec} \ i \ r \rangle, \\
 \lambda x. \pi_1 x & \text{for } \rho = \langle p \mathbf{Fork} \ q \ r \rangle, \end{cases} \\
 d_0 := \lambda x y. x (y 0^k), \\
 d_i := \lambda x y. x (\lambda z. z) (\lambda z_1 \dots z_k. y z_1 \dots z_{i-1} (S z_i) z_{i+1} \dots z_k).
 \end{array} \right.
 \end{array}$$

We now prove that  $\lambda y. y (S^{m_1} 0) \dots (S^{m_k} 0)$  belongs to the object language of  $\mathcal{G}_M$  if and only if  $M$  accepts  $\langle m_1, \dots, m_k \rangle$ , for any  $k$  natural numbers  $m_1, \dots, m_k$ . The proof consists of four technical lemmas, the first two of which establish the if part of the property. For notational convenience, to each  $\mathbf{m} = \langle m_1, \dots, m_k \rangle \in \mathbb{N}^k$ , we assign the typing environment

$$\Gamma_{\mathbf{m}} = x_{1,1} : a_1, \dots, x_{1,m_1} : a_1, \dots, x_{k,1} : a_k, \dots, x_{k,m_k} : a_k.$$

**Lemma 2.** *If  $M$  accepts from an ID  $X$ , then for each element  $\langle q, \mathbf{m} \rangle$  of  $X$ , there is  $t$  such that  $\Gamma_{\mathbf{m}} \vdash_{\Sigma_M} t : q$  and  $t$  does not contain  $d_i$  for any  $i$ .*

*Proof.* By induction on the length of the transition from  $X$  to an accepting ID  $\langle q_f, 0^k \rangle \dots \langle q_f, 0^k \rangle$ . The constant  $c_f$  of type  $q_f$  satisfies the lemma for the zero step transition.

CASE 1. Suppose that  $\rho = \langle q \mathbf{Inc} \ i \ r \rangle \in \delta$  induces the first step of the transition as

$$X_1 \langle q, \mathbf{m} \rangle X_2 \rightarrow X_1 \langle r, \mathbf{m}' \rangle X_2 \xrightarrow{*} \langle q_f, 0^k \rangle \dots \langle q_f, 0^k \rangle$$

where  $\mathbf{m}'$  is obtained by incrementing the  $i$ th element of  $\mathbf{m} = m_1, \dots, m_k$  by 1. We have  $\Gamma_{\mathbf{m}'} = \Gamma_{\mathbf{m}}, x_{i,m_i+1} : a_i$  and  $\Gamma_{\mathbf{m}}, x_{i,m_i+1} : a_i \vdash_{\Sigma_M} t : r$  for some  $t$

by induction hypothesis.  $\Sigma_M$  contains the constant  $c_\rho$  of type  $(a_i \multimap r) \multimap q$ . We have

$$\Gamma_{\mathbf{m}} \vdash_{\Sigma_M} c_\rho (\lambda x_{i,m_i+1}. t) : q.$$

CASE 2. Suppose that  $\rho = \langle q \mathbf{Dec} \ i \ r \rangle \in \delta$  induces the first step of the transition as

$$X_1 \langle q, \mathbf{m} \rangle X_2 \rightarrow X_1 \langle r, \mathbf{m}' \rangle X_2 \xrightarrow{*} \langle q_f, 0^k \rangle \dots \langle q_f, 0^k \rangle$$

where  $\mathbf{m}'$  is obtained by decrementing the  $i$ th element of  $\mathbf{m} = m_1, \dots, m_k$  by 1. That is,  $m_i > 0$ . We have  $\Gamma_{\mathbf{m}} = \Gamma_{\mathbf{m}'}, x_{i,m_i} : a_i$  and  $\Gamma_{\mathbf{m}'} \vdash_{\Sigma_M} t : r$  for some  $t$  by induction hypothesis.  $\Sigma_M$  contains the constant  $c_\rho$  of type  $r \multimap (a_i \multimap q)$ . We have

$$\Gamma_{\mathbf{m}} \vdash_{\Sigma_M} c_\rho t x_{i,m_i} : q.$$

CASE 3. Suppose that  $\rho = \langle q \mathbf{Fork} \ r_1 \ r_2 \rangle \in \delta$  induces the first step of the transition as

$$X_1 \langle q, \mathbf{m} \rangle X_2 \rightarrow X_1 \langle r_1, \mathbf{m} \rangle \langle r_2, \mathbf{m} \rangle X_2 \xrightarrow{*} \langle q_f, 0^k \rangle \dots \langle q_f, 0^k \rangle.$$

By induction hypothesis, there are  $t_i$  for  $i = 1, 2$  such that  $\Gamma_{\mathbf{m}} \vdash_{\Sigma_M} t_i : r_i$ .  $\Sigma_M$  contains the constant  $c_\rho$  of type  $(r_1 \ \& \ r_2) \multimap q$ . We have

$$\Gamma_{\mathbf{m}} \vdash_{\Sigma_M} c_\rho (t_1, t_2) : q. \quad \square$$

**Lemma 3.** *If  $M$  accepts  $\langle m_1, \dots, m_k \rangle$ , then there is  $t$  such that  $\vdash_{\Sigma_M} t : s$  and  $\mathcal{L}(t) = \lambda y. y (S^{m_1} 0) \dots (S^{m_k} 0)$ .*

*Proof.* By Lemma 2, there exists  $t'$  such that  $\Gamma_{\mathbf{m}} \vdash_{\Sigma_M} t' : q_0$  where  $\mathbf{m} = \langle m_1, \dots, m_k \rangle$ . Let

$$t = d_k (\lambda x_{k,m_k}. \dots d_k (\lambda x_{k,1}. \dots d_1 (\lambda x_{1,m_1}. \dots d_1 (\lambda x_{1,1}. d_0 t') \dots) \dots) \dots).$$

Then  $\vdash_{\Sigma_M} t : s$ . It is easy to check that for any subterm  $t''$  of  $t$  of the form

$$t'' = d_i (\lambda x_{i,j}. \dots d_1 (\lambda x_{1,1}. d_0 t') \dots)$$

where  $1 \leq i \leq k$  and  $1 \leq j \leq m_i$ , we have

$$\mathcal{L}(t'') = \lambda y. u [z := y (S^{m_1} 0) \dots (S^{m_{i-1}} 0) (S^j 0) 0^{k-i}]$$

for some  $u$  that contains no constants. The fact  $\vdash_{\Sigma_N} \mathcal{L}(t) : (o^k \multimap o) \multimap o$  implies that  $\mathcal{L}(t)$   $\beta$ -reduces to  $\lambda y. y (S^{m_1} 0) \dots (S^{m_k} 0)$ .  $\square$

**Lemma 4.** *Let  $\mathbf{m} \in \mathbb{N}^k$  and  $q \in Q$ . If we have  $\Gamma_{\mathbf{m}} \vdash_{\Sigma_M} t : q$  for some  $t$ , then  $M$  accepts from  $\langle q, \mathbf{m} \rangle$ . Moreover,  $t$  does not contain  $d_i$  for any  $i$ .*

*Proof.* Suppose that  $\Gamma_{\mathbf{m}} \vdash_{\Sigma_M} t : q$ . We assume that  $t$  is  $\beta$ -normal. We prove this lemma by induction on the number of occurrences of constants in  $t$ .

CASE 0.  $t = c_f$  and  $q = q_f$ . Then the typing environment  $\Gamma_{\mathbf{m}}$  must be empty, i.e.,  $\mathbf{m} = 0^k$ . Indeed  $M$  accepts from  $\langle q_f, 0^k \rangle$ .

CASE 1.  $t = c_\rho t'$  with  $\rho = \langle q \text{ Inc } i \ r \rangle \in \delta$  and  $t'$  such that  $\Gamma_{\mathbf{m}} \vdash_{\Sigma_M} t' : a_i \multimap r$ . Then  $\Gamma_{\mathbf{m}}, x_{i, m_i+1} : a_i \vdash_{\Sigma_M} t' x_{i, m_i+1} : r$ . Hence, by induction hypothesis,  $M$  accepts from  $\langle r, \mathbf{m}' \rangle$  where  $\mathbf{m}'$  is obtained by incrementing the  $i$ th element  $m_i$  of  $\mathbf{m}$  by 1. Since  $\langle q, \mathbf{m} \rangle \rightarrow \langle r, \mathbf{m}' \rangle$ ,  $M$  also accepts from  $\langle q, \mathbf{m} \rangle$ .

CASE 2.  $t = c_\rho t' t''$  with  $\rho = \langle q \text{ Dec } i \ r \rangle \in \delta$ ,  $\Gamma_1 \vdash_{\Sigma_M} t' : r$  and  $\Gamma_2 \vdash_{\Sigma_M} t'' : a_i$  for some partition  $\Gamma_1$  and  $\Gamma_2$  of the typing environment  $\Gamma_{\mathbf{m}}$ . Now, the only possibility for  $\Gamma_2$  and  $t''$  is  $\Gamma_2 = x_{i, j} : a_i$  and  $t'' = x_{i, j}$  for some  $j \in \{1, \dots, m_i\}$ . Hence,  $m_i \geq 1$  and  $\Gamma_1 = \Gamma_{\mathbf{m}} - \{x_{i, j} : a_i\}$ . By applying induction hypothesis to  $\Gamma_1 \vdash_{\Sigma_M} t' : r$ , we get that  $M$  accepts from  $\langle r, \mathbf{m}' \rangle$  where  $\mathbf{m}'$  is obtained by decrementing the  $i$ th element  $m_i$  of  $\mathbf{m}$  by 1. Since  $\langle q, \mathbf{m} \rangle \rightarrow \langle r, \mathbf{m}' \rangle$ ,  $M$  also accepts from  $\langle q, \mathbf{m} \rangle$ .

CASE 3.  $t = c_\rho t'$  with  $\rho = \langle q \text{ Fork } r_1 \ r_2 \rangle \in \delta$ . The type of  $c_\rho$  is  $(r_1 \ \& \ r_2) \multimap q$ . Consequently,  $t' = \langle t_1, t_2 \rangle$  for some  $t_1$  and  $t_2$  such that  $\Gamma_{\mathbf{m}} \vdash_{\Sigma_M} t_i : r_i$  for each  $i = 1, 2$ . By induction hypothesis,  $M$  accepts from both  $\langle r_1, \mathbf{m} \rangle$  and  $\langle r_2, \mathbf{m} \rangle$ , which implies that  $M$  also accepts from  $\langle q, \mathbf{m} \rangle$ , because  $\langle q, \mathbf{m} \rangle \rightarrow \langle r_1, \mathbf{m} \rangle \langle r_2, \mathbf{m} \rangle$ .  $\square$

**Lemma 5.** *For any  $t \in \mathcal{A}(\mathcal{G}_M)$ , we have  $\mathcal{L}(t) = \lambda y. y (S^{m_1} 0) \dots (S^{m_k} 0)$  for some  $m_1, \dots, m_k \in \mathbb{N}$  and moreover  $M$  accepts  $\langle m_1, \dots, m_k \rangle$ .*

*Proof.* By  $\vdash_{\Sigma_M} t : s$ ,  $t$  has the form

$$t = d_{i_1} (\lambda x'_1. \dots d_{i_n} (\lambda x'_n. d_0 t') \dots)$$

for some  $i_1, \dots, i_n \in \{1, \dots, k\}$  and  $t'$  such that  $\Gamma \vdash_{\Sigma_M} t' : q_0$  for  $\Gamma = \{x'_j : a_{i_j} \mid 1 \leq j \leq n\}$ . Let  $m_i$  be the number of occurrences of  $d_i$  in  $t$  and  $\mathbf{m} = \langle m_1, \dots, m_k \rangle$ . By renaming variables, we can assume  $\Gamma = \Gamma_{\mathbf{m}}$ . By Lemma 4,  $t'$  does not contain any  $d_i$ . It is not hard to see that  $\mathcal{L}(t) = \lambda y. y (S^{m_1} 0) \dots (S^{m_k} 0)$ . Moreover Lemma 4 implies that  $M$  accepts  $\mathbf{m}$ .  $\square$

Finally, we obtain the expected property as a direct consequence of Lemmas 1, 3 and 5.

**Proposition 1.** *For any  $k$ -counter machine  $M$ , one can effectively construct an ACG with Cartesian product  $\mathcal{G}_M$  such that*

$$M \text{ accepts } \langle m_1, \dots, m_k \rangle \text{ iff } \lambda y. y (S^{m_1} 0) \dots (S^{m_k} 0) \in \mathcal{O}(\mathcal{G}_M).$$

**Corollary 1.** *For any recursive function  $\phi$ , there exists an ACG with Cartesian product  $\mathcal{G}_\phi$  such that*

$$\phi(a_1, \dots, a_n) = b \text{ iff } \lambda y. y (S^b 0) (S^{a_1} 0) \dots (S^{a_n} 0) \in \mathcal{O}(\mathcal{G}_\phi).$$

*Proof.* By Theorem 1 and Proposition 1. It is easy to modify the definition of  $\mathcal{G}_M$  so that counters not used for representing the arguments and values of the function  $\phi$  are completely suppressed in the object language.  $\square$



## 4 Abstract Categorical Grammars with Dependent Product

Dependent product allows ones to specify types that depend upon terms. In a grammatical setting (where types corresponds to syntactic categories), dependent products are useful in defining generic syntactic categories (for instance, *NP* for *noun phrase*) that can be instantiated according to the value of some feature (for instance,  $(NP f)$  for *feminine noun phrase*,  $(NP m)$  for *masculine noun phrase*, etc.)

### 4.1 Definition

In the presence of dependent products, types may depend upon terms. Consequently, it is no longer the case that the notion of well-formed types may be specified only by means of context-free rules. In the same way terms are assigned types, types will be assigned kinds. Consequently, we first introduce the raw syntax of three forms of expressions, namely, the *kinds* ( $\mathcal{K}$ ), the *types* ( $\mathcal{T}$ ), and the *terms* ( $T$ ).

$$\begin{aligned}\mathcal{K} &::= \text{type} \mid (\mathcal{T})\mathcal{K} \\ \mathcal{T} &::= a \mid (\lambda x. \mathcal{T}) \mid (\mathcal{T} T) \mid (\mathcal{T} \multimap \mathcal{T}) \mid (\Pi x : \mathcal{T}) \mathcal{T} \\ T &::= c \mid x \mid (\lambda^\circ x. T) \mid (\lambda x. T) \mid (T T)\end{aligned}$$

where  $a$  ranges over atomic types, and  $c$  over constants. In addition to atomic types, linear functional types, and dependent products, we have two other type constructs: the abstraction of a  $\lambda$ -variable over a type, and the application of a type to a  $\lambda$ -term. At the level of the  $\lambda$ -terms, we now distinguish between two forms of  $\lambda$ -abstractions: a linear  $\lambda$ -abstraction  $(\lambda^\circ x. T)$ , and a non-linear one  $(\lambda x. T)$ .

Let  $a$  range over atomic types,  $c$  over constants,  $A$  over kinds, and  $\alpha$  over types. A raw signature is then defined as a sequence of declarations either of the form ' $a:A$ ' or of the form ' $c:\alpha$ '. Let  $\Sigma$  be such a raw signature, we define two partial functions. The first one,  $\kappa_\Sigma$ , assigns kinds to atomic types. It is inductively defined as follows:

$$\begin{aligned}\kappa_{()}(a) &\text{ is undefined} \\ \kappa_{\Sigma; a_1:A}(a) &= \begin{cases} A & \text{if } a = a_1 \\ \kappa_\Sigma(a) & \text{otherwise} \end{cases} \\ \kappa_{\Sigma; c:\alpha}(a) &= \kappa_\Sigma(a)\end{aligned}$$

Similarly,  $\tau_\Sigma$ , assigns types to  $\lambda$ -term constants:

$$\begin{aligned}\tau_{()}(c) &\text{ is undefined} \\ \tau_{\Sigma; a:A}(c) &= \tau_\Sigma(c) \\ \tau_{\Sigma; c_1:\alpha}(c) &= \begin{cases} \alpha & \text{if } c = c_1 \\ \tau_\Sigma(c) & \text{otherwise} \end{cases}\end{aligned}$$

We now give the type system of the calculus. It relies on four forms of judgements:

$$\text{sig}(\Sigma) \quad \vdash_{\Sigma} A : \text{kind} \quad \Gamma \vdash_{\Sigma} \alpha : A \quad \Gamma; \Delta \vdash_{\Sigma} t : \alpha$$

where  $A$ ,  $\alpha$ , and  $t$  range over kinds, types, and  $\lambda$ -terms, respectively.  $\Sigma$  is a given signature.  $\Gamma$  and  $\Delta$  range over typing environments, which are now defined to be sequences of declarations of the form ' $x : \alpha$ '.

These four forms of judgements may be paraphrased as follows:

1.  $\Sigma$  is a well-formed signature.
2. Given the signature  $\Sigma$ ,  $A$  is a well-formed kind.
3. Given the signature  $\Sigma$ ,  $\alpha$  is a type of kind  $A$  according to the non-linear typing environment  $\Gamma$ .
4. Given the signature  $\Sigma$ ,  $t$  is a term of type  $\alpha$  according to the non-linear typing environment  $\Gamma$  and the linear typing environment  $\Delta$ .

Finally, the rules of the typing system are as follows.

WELL-FORMED SIGNATURES:

$$\begin{array}{c} \text{sig}() \\ \text{sig}(\Sigma) \quad \vdash_{\Sigma} A : \text{kind} \\ \hline \text{sig}(\Sigma; a : A) \\ \text{sig}(\Sigma) \quad \vdash_{\Sigma} \alpha : \text{type} \\ \hline \text{sig}(\Sigma; c : \alpha) \end{array}$$

In the above rules, the introduced symbols ( $a$  and  $c$ ) must be fresh with respect to  $\Sigma$ .

WELL-FORMED KINDS:

$$\begin{array}{c} \vdash_{\Sigma} \text{type} : \text{kind} \\ \vdash_{\Sigma} \alpha : \text{type} \quad \vdash_{\Sigma} A : \text{kind} \\ \hline \vdash_{\Sigma} (\alpha) A : \text{kind} \end{array}$$

WELL-KINDED TYPES:

$$\begin{array}{c} \vdash_{\Sigma} a : \kappa_{\Sigma}(a) \quad (\text{type const.}) \\ \vdash_{\Sigma} \alpha : \text{type} \quad \Gamma \vdash_{\Sigma} \beta : A \\ \hline \Gamma, x : \alpha \vdash_{\Sigma} \beta : A \quad (\text{type weak.}) \end{array}$$

$$\frac{\Gamma, x : \alpha \vdash_{\Sigma} \beta : A}{\Gamma \vdash_{\Sigma} \lambda x. \beta : (\alpha) A} \quad (\text{type abs.})$$

$$\frac{\Gamma \vdash_{\Sigma} \alpha : (\beta) A \quad \Gamma; \vdash_{\Sigma} t : \beta}{\Gamma \vdash_{\Sigma} \alpha t : A} \quad (\text{type app.})$$

$$\frac{\Gamma \vdash_{\Sigma} \alpha : \text{type} \quad \Gamma \vdash_{\Sigma} \beta : \text{type}}{\Gamma \vdash_{\Sigma} \alpha \multimap \beta : \text{type}} \quad (\text{lin. fun.})$$

$$\frac{\Gamma \vdash_{\Sigma} \alpha : \text{type} \quad \Gamma, x : \alpha \vdash_{\Sigma} \beta : \text{type}}{\Gamma \vdash_{\Sigma} (\Pi x : \alpha) \beta : \text{type}} \quad (\text{dep. prod.})$$

In Rule (type weak.),  $x$  must be fresh with respect to  $\Gamma$ .

WELL-TYPED TERMS:

$$; \vdash_{\Sigma} c : \tau_{\Sigma}(c) \quad (\text{const.})$$

$$\frac{\Gamma \vdash_{\Sigma} \alpha : \text{type}}{\Gamma; x : \alpha \vdash_{\Sigma} x : \alpha} \quad (\text{lin. var.})$$

$$\frac{\Gamma \vdash_{\Sigma} \alpha : \text{type}}{\Gamma, x : \alpha; \vdash_{\Sigma} x : \alpha} \quad (\text{var.})$$

$$\frac{\Gamma \vdash_{\Sigma} \alpha : \text{type} \quad \Gamma; \Delta \vdash_{\Sigma} t : \beta}{\Gamma, x : \alpha; \Delta \vdash_{\Sigma} t : \beta} \quad (\text{weak.})$$

$$\frac{\Gamma; \Delta_1, x : \alpha, y : \beta, \Delta_2 \vdash_{\Sigma} t : \gamma}{\Gamma; \Delta_1, y : \beta, x : \alpha, \Delta_2 \vdash_{\Sigma} t : \gamma} \quad (\text{perm.})$$

$$\frac{\Gamma; \Delta, x : \alpha \vdash_{\Sigma} t : \beta}{\Gamma; \Delta \vdash_{\Sigma} \lambda^{\circ} x. t : \alpha \multimap \beta} \quad (\text{lin. abs.})$$

$$\frac{\Gamma; \Delta_1 \vdash_{\Sigma} t : \alpha \multimap \beta \quad \Gamma; \Delta_2 \vdash_{\Sigma} u : \alpha}{\Gamma; \Delta_1, \Delta_2 \vdash_{\Sigma} t u : \beta} \quad (\text{lin. app.})$$

$$\frac{\Gamma, x : \alpha; \Delta \vdash_{\Sigma} t : \beta}{\Gamma; \Delta \vdash_{\Sigma} \lambda x. t : (\Pi x : \alpha) \beta} \quad (\text{abs.})$$

$$\frac{\Gamma; \Delta \vdash_{\Sigma} t : (\Pi x : \alpha) \beta \quad \Gamma; \vdash_{\Sigma} u : \alpha}{\Gamma; \Delta \vdash_{\Sigma} t u : \beta[x:=u]} \quad (\text{app.})$$

$$\frac{\Gamma; \Delta \vdash_{\Sigma} t : \alpha \quad \Gamma \vdash_{\Sigma} \beta : \text{type} \quad \alpha =_{\beta\eta} \beta}{\Gamma; \Delta \vdash_{\Sigma} t : \beta} \quad (\text{type conv.})$$

In Rules (lin. var.) and (var.),  $x$  must be fresh with respect to  $\Gamma$ . In Rule (weak.),  $x$  must be fresh with respect to both  $\Gamma$  and  $\Delta$ . Moreover,  $t$  must be either a  $\lambda$ -variable, or a constant. In Rule (abs.),  $x$  cannot occur free in  $\Delta$ .

Let  $\Sigma$  be a signature. We will write  $A_\Sigma$  for the set of atomic types declared in  $\Sigma$ . Similarly, we will write  $C_\Sigma$  for the set of  $\lambda$ -term constants declared in  $\Sigma$ . Finally, given a well-formed signature  $\Sigma$ , we will write  $\mathcal{K}_\Sigma$ ,  $\mathcal{T}_\Sigma$ , and  $\Lambda_\Sigma$  for the corresponding sets of well-formed kinds, well-kinded types, and well-typed terms, respectively.

In order to define a notion of ACG with dependent product, it remains to adapt the definition of a lexicon. Let  $\Sigma$  and  $\Xi$  be two well-formed signatures. A lexicon  $\mathcal{L}$  from  $\Sigma$  to  $\Xi$  is a pair  $\langle \eta, \theta \rangle$  such that:

1.  $\eta$  is a mapping from  $A_\Sigma$  into  $\mathcal{T}_\Xi$ ;
2.  $\theta$  is a mapping from  $C_\Sigma$  into  $\Lambda_\Xi$ ;
3. for every  $c \in C_\Sigma$ , the following typing judgement is derivable:

$$\vdash_{\Xi} \theta(c) : \hat{\eta}(\tau_\Sigma(c)),$$

where  $\hat{\eta} : \mathcal{T}_\Sigma \rightarrow \mathcal{T}_\Xi$  is the unique homomorphic extension of  $\eta$ ;

4. for every  $a \in A_\Sigma$ , the following kinding judgement is derivable:

$$\vdash_{\Xi} \eta(a) : \tilde{\eta}(\kappa_\Sigma(a)),$$

where  $\tilde{\eta} : \mathcal{K}_\Sigma \rightarrow \mathcal{K}_\Xi$  is defined by  $\tilde{\eta}(\text{type}) = \text{type}$  and  $\tilde{\eta}((\alpha)A) = (\hat{\eta}(\alpha))\tilde{\eta}(A)$ .

## 4.2 Turing Completeness

The  $\lambda$ -calculus we have defined in the previous section contains the Edinburgh logical framework [6] as a subsystem.<sup>3</sup> We may therefore expect ACGs with dependent product to be Turing-complete. In order to show it is indeed the case, we explain how to encode any general phrase structure grammar as an ACG with dependent product.

We first remind the reader of some basic definitions. A *phrase structure grammar* is a quadruple  $G = \langle V, T, R, S \rangle$ , where  $V$  is a finite set of *symbols*,  $T \subseteq V$  is a finite set of *terminal symbols*,  $S \in V$  is the *start symbol* and  $R$  is a finite set of *production rules* of the form  $\alpha \rightarrow \beta$  for  $\alpha, \beta \in V^*$ . One writes  $\alpha \Rightarrow \beta$  if  $\alpha = \gamma_1 \alpha' \gamma_2$ ,  $\beta = \gamma_1 \beta' \gamma_2$  and  $\alpha' \rightarrow \beta' \in R$  for some  $\gamma_1, \gamma_2 \in V^*$ . As usual,  $\overset{*}{\Rightarrow}$  is the reflexive, transitive closure of  $\Rightarrow$ . The language generated by  $G$  is defined as  $L(G) = \{\alpha \in T^* \mid S \overset{*}{\Rightarrow} \alpha\}$ .

To any alphabet  $T$ , we associate a signature  $\Sigma_T$ . This signature has one atomic type  $o$  and its set of constants is  $T$ , the elements of which are assigned the type  $o \multimap o$ . Then, every string  $a_1 \dots a_n \in T^*$  may be encoded as  $\lambda^\circ z. a_1 (\dots (a_n z) \dots)$ . Let us write  $/a_1 \dots a_n/$  to denote this last term. We have, in particular, that the empty string  $\varepsilon$  is represented by the linear identity function, i.e.,  $/\varepsilon/ = \lambda^\circ z. z$ . We also have that concatenation is represented by functional composition, and we will write  $t + u$  for  $\lambda^\circ z. t(u z)$ .

<sup>3</sup> Actually, the notion of dependent type we use is slightly weaker.

Now, to any phrase structure grammar  $G = \langle V, T, R, S \rangle$ , we associate the ACG  $\mathcal{G}_G = \langle \Sigma_G, \Sigma_T, \mathcal{L}, s \rangle$  that is defined as follows.

$$\begin{array}{l} \Sigma_G \left\{ \begin{array}{l} o, s : \text{type}, \\ \sigma : (o \multimap o) \text{type}, \\ \tau : (o \multimap o) \text{type}, \\ A : o \multimap o \quad \text{for all } A \in V, \\ c_S : \sigma(/S/), \\ c_{\alpha \rightarrow \beta} : (\Pi x, y \in o \multimap o)(\sigma(x + / \alpha / + y) \multimap \sigma(x + / \beta / + y)) \\ \quad \text{for all } \alpha \rightarrow \beta \in R, \\ \\ d_\varepsilon : \tau(/ \varepsilon /), \\ d_a : (\Pi x \in o \multimap o)(\tau(x) \multimap \tau(/ a / + x)) \quad \text{for all } a \in T, \\ e : (\Pi x \in o \multimap o)(\sigma(x) \multimap \tau(x) \multimap s), \end{array} \right. \\ \\ \Sigma_T \left\{ \begin{array}{l} o : \text{type}, \\ a : o \multimap o \text{ for all } a \in T, \end{array} \right. \\ \\ \mathcal{L} \left\{ \begin{array}{l} o := o, \\ s := o \multimap o, \\ \tau := \lambda x. o \multimap o, \\ \sigma := \lambda x. o \multimap o, \\ A := \lambda^\circ z. z \quad \text{for all } A \in V, \\ c_S := / \varepsilon /, \\ c_{\alpha \rightarrow \beta} := \lambda x y. \lambda^\circ z. z \quad \text{for all } \alpha \rightarrow \beta \in R, \\ d_\varepsilon := / \varepsilon /, \\ d_a := \lambda x. \lambda^\circ y. / a / + y \text{ for all } a \in T, \\ e := \lambda x. \lambda^\circ y z. y + z. \end{array} \right. \end{array}$$

The signature  $\Sigma_G$  consists of two independent parts that are connected through the constant  $e$ . We will establish the two following properties:

1. for every  $\alpha \in V^*$ ,  $S \xrightarrow{*} \alpha$  if and only if there exists  $u_\alpha$  such that  $; \vdash_{\Sigma_G} u_\alpha : \sigma(/ \alpha /)$ ;
2. for every  $\alpha \in V^*$ ,  $\alpha \in T^*$  if and only if there exists  $t_\alpha$  such that  $; \vdash_{\Sigma_G} t_\alpha : \tau(/ \alpha /)$ .

This implies that  $\alpha \in L(G)$  if and only if  $; \vdash_{\Sigma_G} u_\alpha : \sigma(/ \alpha /)$  and  $; \vdash_{\Sigma_G} t_\alpha : \tau(/ \alpha /)$  for some  $u_\alpha$  and  $t_\alpha$ .

**Lemma 6.** *For any  $\alpha \in V^*$ , if  $S \xrightarrow{*} \alpha$ , then there is  $t$  such that  $; \vdash_{\Sigma_G} t : \sigma(/ \alpha /)$  and  $\mathcal{L}(t) \rightarrow_\beta / \varepsilon /$ .*

*Proof.* Induction on the length of the derivation. For  $\alpha = S$ ,  $t = c_S$  satisfies the lemma. Suppose that  $S \xrightarrow{*} \gamma_1 \alpha \gamma_2 \Rightarrow \gamma_1 \beta \gamma_2$  and  $\alpha \rightarrow \beta \in R$ . By induction hypothesis, we have  $t'$  such that  $; \vdash_{\Sigma_G} t' : \sigma(/ \gamma_1 \alpha \gamma_2 /)$  and  $\mathcal{L}(t') \rightarrow_\beta / \varepsilon /$ . Let  $t = c_{\alpha \rightarrow \beta} / \gamma_1 / / \gamma_2 / t'$ . Then we have  $; \vdash_{\Sigma_G} t : \sigma(/ \gamma_1 \beta \gamma_2 /)$  and  $\mathcal{L}(t) \rightarrow_\beta \mathcal{L}(t') \rightarrow_\beta / \varepsilon /$ .  $\square$

**Lemma 7.** *For every  $\alpha \in T^*$ , there is  $t$  such that  $; \vdash_{\Sigma_G} t : \tau(/ \alpha /)$  and  $\mathcal{L}(t) \rightarrow_\beta / \alpha /$ .*

*Proof.* Induction on the length of  $\alpha$ . For  $\alpha = \varepsilon$ ,  $t = d_\varepsilon$  satisfies the lemma. For  $\alpha \neq \varepsilon$ , let  $\alpha = a\alpha'$  with  $a \in T$  and  $\alpha' \in T^*$ . By induction hypothesis, we have  $t'$  such that  $\vdash_{\Sigma_G} t' : \tau(/ \alpha' /)$  and  $\mathcal{L}(t') \twoheadrightarrow_\beta / \alpha' /$ . Let  $t = d_a / \alpha' / t'$ . Then we have  $\vdash_{\Sigma_G} t : \tau(/ \alpha /)$  and  $\mathcal{L}(t) \twoheadrightarrow_\beta / a / + \mathcal{L}(t') \twoheadrightarrow_\beta / \alpha /$ .  $\square$

**Lemma 8.** *For every  $\alpha \in L(G)$ , we have  $/ \alpha / \in \mathcal{O}(\mathcal{G}_G)$ .*

*Proof.* For any  $\alpha \in L(G)$ , we have  $t_1$  and  $t_2$  such that  $\vdash_{\Sigma_G} t_1 : \sigma(/ \alpha /)$ ,  $\mathcal{L}(t_1) \twoheadrightarrow_\beta / \varepsilon /$ ,  $\vdash_{\Sigma_G} t_2 : \tau(/ \alpha /)$  and  $\mathcal{L}(t_2) \twoheadrightarrow_\beta / \alpha /$  by Lemmas 6 and 7. Thus we have  $\vdash_{\Sigma_G} e / \alpha / t_1 t_2 : s$  and  $\mathcal{L}(e / \alpha / t_1 t_2) \twoheadrightarrow_\beta / \alpha /$ .  $\square$

**Lemma 9.** *For any  $\beta \in V^*$ , if  $\vdash_{\Sigma_G} t : \sigma(/ \beta /)$ , then  $S \xrightarrow{*} \beta$  and  $\mathcal{L}(t) \twoheadrightarrow_\beta / \varepsilon /$ .*

*Proof.* Suppose that  $\vdash_{\Sigma_G} t : \sigma(/ \beta /)$ . We assume that  $t$  is  $\beta$ -normal. We prove this lemma by induction on  $t$ . If  $t = c_S$ , then  $\beta = S$  and the lemma holds clearly. Otherwise,  $t$  must have the form  $t = c_{\alpha' \rightarrow \beta'} / \gamma_1 / / \gamma_2 / t'$  with  $\beta = \gamma_1 \beta' \gamma_2$  for some  $t'$  such that  $\vdash_{\Sigma_G} t' : \sigma(/ \gamma_1 \alpha' \gamma_2 /)$ . By induction hypothesis, we have  $S \xrightarrow{*} \gamma_1 \alpha' \gamma_2$  and  $\mathcal{L}(t') \twoheadrightarrow_\beta / \varepsilon /$ . The fact that  $\Sigma_G$  has the constant  $c_{\alpha' \rightarrow \beta'}$  implies that  $\alpha' \rightarrow \beta' \in R$ . We have  $S \xrightarrow{*} \gamma_1 \alpha' \gamma_2 \Rightarrow \gamma_1 \beta' \gamma_2 = \beta$  and  $\mathcal{L}(t) \twoheadrightarrow_\beta \mathcal{L}(t') \twoheadrightarrow_\beta / \varepsilon /$ .  $\square$

**Lemma 10.** *For any  $\beta \in V^*$ , if  $\vdash_{\Sigma_G} t : \tau(/ \beta /)$ , then  $\beta \in T^*$  and  $\mathcal{L}(t) \twoheadrightarrow_\beta / \beta /$ .*

*Proof.* Suppose that  $\vdash_{\Sigma_G} t : \tau(/ \beta /)$ . We assume that  $t$  is  $\beta$ -normal. We prove this lemma by induction on  $t$ . If  $t = d_\varepsilon$ , then  $\beta = \varepsilon$  and the lemma holds clearly. Otherwise,  $t$  must have the form  $t = d_a / \alpha / t'$  for some  $a \in T$ ,  $\alpha \in V^*$  and  $t'$  such that  $\beta = a\alpha$  and  $\vdash_{\Sigma_G} t' : \tau(/ \alpha /)$ . By induction hypothesis, we have  $\mathcal{L}(t') \twoheadrightarrow_\beta / \alpha /$ ,  $\alpha \in T^*$  and thus  $a\alpha \in T^*$ . Besides  $\mathcal{L}(t) \twoheadrightarrow_\beta / a / + \mathcal{L}(t') \twoheadrightarrow_\beta / a\alpha /$ .  $\square$

**Lemma 11.**  *$/ \alpha / \in \mathcal{O}(\mathcal{G}_G)$  implies  $\alpha \in L(G)$ .*

*Proof.* Suppose that  $t \in \mathcal{A}(\mathcal{G}_G)$ .  $t$  must have the form  $t = e / \alpha / t_1 t_2$  with  $\vdash_{\Sigma_G} t_1 : \sigma(/ \alpha /)$  and  $\vdash_{\Sigma_G} t_2 : \tau(/ \alpha /)$  for some  $\alpha \in V^*$ . We have  $S \xrightarrow{*} \alpha$  and  $\mathcal{L}(t_1) \twoheadrightarrow_\beta / \varepsilon /$  by Lemma 9.  $\alpha \in T^*$  and  $\mathcal{L}(t_2) \twoheadrightarrow_\beta / \alpha /$  by Lemma 10. Therefore,  $\alpha \in L(G)$  and  $\mathcal{L}(t) \twoheadrightarrow_\beta / \alpha / \in \mathcal{O}(\mathcal{G})$ .  $\square$

As a consequence of Lemmas 8 and 11 we obtain the main result of this section.

**Proposition 2.** *For any phrase structure grammar, one can find an ACG with dependent product that generates exactly the same language.*

## 5 Conclusions

This work shows that quite simple extensions of the abstract categorical grammars immediately result in undecidable formalisms. This is not quite surprising as it is not even known whether membership is decidable or not for the original ACGs. On the other hand, from a practical point of view, there is a need for powerful constructs such as feature structures, records, or generic types. Consequently, future work must consist in trying to identify fragments of the extended formalism proposed in [5] that offer a good compromise between intentional expressive power and tractability.

## References

1. de Groote, Ph.: Towards abstract categorical grammars. In: Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference, pp. 148–155 (2001)
2. de Groote, Ph.: Tree-Adjoining Grammars as Abstract Categorical Grammars. In: TAG+6, Proceedings of the sixth International Workshop on Tree Adjoining Grammars and Related Frameworks, pp. 145–150 (2001)
3. de Groote, Ph., Guillaume, B., Salvati, S.: Vector Addition Tree Automata. In: LICS 2004. 19th IEEE Symposium on Logic in Computer Science, pp. 64–73. IEEE Computer Society, Los Alamitos (2004)
4. de Groote, Ph., Pogodalla, S.: On the Expressive Power of Abstract Categorical Grammars: Representing Context-Free Formalisms. *Journal of Logic, Language and Information* 13(4), 421–438 (2004)
5. de Groote, P., Maarek, S.: Type-theoretic Extensions of Abstract Categorical Grammars. In: Proceedings of the Workshop on New Directions in Type-theoretic Grammars. ESSLLI (2007)
6. Harper, R., Honsel, F., Plotkin, G.: A framework for defining logics. In: Proceedings of the second annual IEEE symposium on logic in computer science, pp. 194–204. IEEE Computer Society Press, Los Alamitos (1987)
7. Kanazawa, M.: Parsing and Generation as Datalog Queries. In: Association for Computational Linguistics, 45th Annual Meeting, Proceedings of the Conference (to appear, 2007)
8. Lambek, J.: How to program an infinite abacus. *Canadian Mathematical Bulletin* 4, 279–293 (1961)
9. Lincoln, P., Mitchell, J.C., Scedrov, A., Shankar, N.: Decision Problems for Propositional Linear Logic. *Annals of Pure and Applied Logic* 56(1-3), 239–311 (1992)
10. Ranta, A.: Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming* 14(2), 145–189 (2004)
11. Salvati, S.: Problèmes de filtrage et problèmes d’analyse pour les grammaires catégorielles abstraites. Thèse de Doctorat. Institut National Polytechnique de Lorraine (2005)
12. Yoshinaka, R., Kanazawa, M.: The Complexity and Generative Capacity of Lexicalized Abstract Categorical Grammars. In: Blache, P., Stabler, E.P., Busquets, J.V., Moot, R. (eds.) LACL 2005. LNCS (LNAI), vol. 3492, pp. 330–346. Springer, Heidelberg (2005)