Compositional Semantics for Unification-based Linguistic Formalisms

Shuly Wintner
Institute for Research in Cognitive Science
University of Pennsylvania
3401 Walnut Street, Suite 400A
Philadelphia, PA 19104-6228
shuly@linc.cis.upenn.edu

Abstract

Contemporary linguistic formalisms have become so rigorous that it is now possible to view them as very high level declarative programming languages. Consequently, *grammars* for natural languages can be viewed as *programs*; this view enables the application of various methods and techniques that were proved useful for programming languages to the study of natural languages. This paper adapts the notion of *program composition*, well developed in the context of logic programming languages, to the domain of linguistic formalisms. We study alternative definitions for the semantics of such formalisms, suggesting a denotational semantics that we show to be compositional and fully-abstract. This facilitates a clear, mathematically sound way for defining grammar modularity.

1 Introduction

The tasks of developing large scale grammars for natural languages become more and more complicated: it is not unusual for a single grammar to be developed by a team including a number of linguists, computational linguists and computer scientists. The problems that grammar engineers face when they design a broad-coverage grammar for some natural language are very reminiscent to the problems tackled by software engineering (see Erbach and Uszkoreit (1990) for a discussion of some of the problems involved in developing large-scale grammars). It is possible – and, indeed, desirable — to adapt methods and techniques of software engineering to the domain of natural language formalisms.

The departure point for this work is the belief that any advances in grammar engineering must be preceded by a more theoretical work, concentrating on the semantics of grammars. This view reflects the situation in logic programming, where developments in alternative definitions for the semantics of predicate logic led to implementations of various program composition operators. Viewing contemporary linguistic formalisms as very high level declarative programming languages, a *grammar* for a natural language can be viewed as a *program*. The execution of a grammar on an input sentence yields an output which represents the sentence's structure.

This paper adapts well-known results from logic programming languages semantics to the framework of unification-based linguistic formalisms. While most of the results we report on are probably not surprising, we believe it is important to derive them directly for linguistic formalisms for two reasons. First, practitioners of linguistic formalisms usually do not view them as instances of a general logic programming framework, but rather as first-class programming environments which deserve independent study. Second, there are some crucial differences between linguistic formalisms and, say, Prolog: the basic elements — typed feature structures — are more general then first-order terms, the notion of unification is different, and computations amount to parsing, rather than SLD-resolution. The fact that we can derive similar results in this new domain is encouraging, and should not be considered trivial.

In the next section we review the literature on modularity in logic programming. We discuss alternative approaches, both operational and denotational, to the semantics of unification-based linguistic formalisms in section 3. In section 4 we show that the standard semantics are "too crude" and present an alternative semantics, which we show to be compositional (with respect to grammar union, a simple syntactic combination operation on grammars). However, this definition is "too fine": we show that it is not fully-abstract, and in section 5 we present an adequate, compositional and fully-abstract semantics for unification-based linguistic formalisms. We conclude with suggestions for further research.

2 Modularity in logic programming

The seminal work in the semantics of logic programming is Van Emden and Kowalski (1976), where three alternative definitions for the meanings of predicate logic sentences are given and proven equivalent: the model theoretic definition, viewing the denotation of the program as its minimal model, the intersection of all its Herbrand models; the operational definition, viewing the denotation of a program as the set of consequences derivable from the data structures it manipulates; and the fixpoint semantics, by which the meaning of a program is the least fixpoint of its immediate consequence operator. These results are elaborated upon by Apt and Van Emden (1982), further exploiting the application of fixpoints by relating them to SLD resolution and accounting also to finite failure.

A different approach is pursued by Lassez and Maher (1984): viewing a logic program as the set of its rules, distinct from the facts, they define an immediate consequence operator for the rules only. The denotation of a grammar is the fixpoint of this operator, starting from the set of facts. This approach facilitates composition of programs, as the meaning of any composite sentence can be expressed in terms of the meanings of its immediate constituents.

Gaifman and Shapiro (1989) observe that the classical Herbrand-base semantics of logic programs is inadequate, as it identifies programs that should be distinguished and vice versa. They define the notions of compositional semantics and fully-abstract semantics and show that the reason for the inadequacy of the standard model-theoretic semantics lies in the fact that it captures only the derivable atoms in its Herbrand base, whereas a query can introduce new function symbols that are not part of the program's signature. Rather than resort to a solution that incorporates a "universal" signature for logic programs, they change the notion of program equivalence by taking as invariant the set of all most general atomic logical consequences of the program. This definition of semantics is shown to be both compositional and fully-abstract.

Gaifman and Shapiro (1989) define compositional semantics as follows: let $\mathcal{P}$ be a class of programs (or program parts); let Ob be a function associating a set of objects, the observables, with a program $P \in \mathcal{P}$. Let Com be a class of composition functions over $\mathcal{P}$ such that for every n-ary function $f \in Com$ and every set of n programs $P_1, \ldots, P_n \in \mathcal{P}$, $f(P_1, \ldots, P_n) \in \mathcal{P}$. A compositional equivalence (with respect to Ob, Com) is an equivalence relation '$\equiv$' that preserves observables (i.e., whenever $P \equiv Q$, $Ob(P) = Ob(Q)$), and for every $f \in Com$, if for all $1 \leq i \leq n$, $P_i \equiv Q_i$ then $f(P_1, \ldots, P_n) \equiv f(Q_1, \ldots, Q_n)$.

The need for a modular extension for logic programming languages has always been agreed upon, as relations were viewed as providing too fine-grained abstraction for the design of large programs. Two major tracks were taken: one, referred to as *programming in the large* and inspired by O'keefe (1985), suggests a meta-linguistic mechanism: modules are viewed as sets of Horn clauses and their composition is modeled in terms of operations on the components (such as union, deletion, closure etc.) The other approach, known as *programming in the small* and originating with Miller (1986; 1989), enhances logic programming with linguistic abstraction mechanisms that are richer than those offered by Horn clauses (see Bugliesi, Lamma, and Mello (1994) for a review of modularity in logic programming).

Unlike O'keefe (1985), who defines the denotation of a program to be the transformation operator obtained by applying deduction steps arbitrarily many times, Mancarella and Pedreschi (1988) move from an object level to a function level and interpret a logic program as its immediate consequence operator, modeling a single deduction step. They define an algebra over operators, suitable for defining a *union* composition operator over sub-programs. These results are extended in Brogi et al. (1990), where an additional operator, intersection, is added to the union operator. The composition operators are characterized both model-theoretically and using meta-interpreter techniques; both approaches are shown to be equivalent to each other and to the original algebraic characterization.

A different approach is taken by Brogi, Lamma, and Mello (1992), who observe that the reason for the inadequacy of the standard Herbrand model semantics for compositionality stems from the adoption of the closed world assumption. When program composition is to be supported, programs should be view as *open*, to be completed by additional knowledge. Viewing the denotation of a program as the set of all its Herbrand models is inadequate; rather, program meaning is defined to be a subset of its models, namely its *admissible* models – those that are supported by the assumption of a set of hypotheses. A natural notion of program composition is thus obtained. This work is extended by Brogi, Lamma, and Mello (1993), adding a *close* operator and exemplifying its usage.

The notion of compositionality employed in these two works is slightly different: given a program P, let $[\![P]\!]$ be the denotation of P. Let $\cup$ be a composition operator on programs, and $\bullet$ be a composition operator on denotations. A semantics is compositional if $\cup$ and $\bullet$ commute, that is, if $[\![P \cup Q]\!] = [\![P]\!] \bullet [\![Q]\!]$. To avoid confusion, we say that a semantics is *commutative* with respect to the operator $\bullet$ iff it is compositional by this definition. Commutativity is a stronger notion than the definition of Gaifman and Shapiro (1989) above: if a semantics is commutative with respect to some operator then it is compositional (with respect to a singleton set containing the same operator).

Proposition 1. *Let $\mathcal{P}$ be a class of programs, $[\![\cdot]\!]$ a denotation operator, $\cup$ a combination operator on programs and $\bullet$ a combination operator on denotations. For $P, Q \in \mathcal{P}$, let $P \equiv Q$ iff $[\![P]\!] = [\![Q]\!]$. If $[\![\cdot]\!]$ is commutative with respect to $\cup, \bullet$, then $\equiv$ is compositional with respect to $\cup$.*

Proof. Assume that $P_1 \equiv P_2$ and $Q_1 \equiv Q_2$.

$$\begin{array}{rcll} [\![P_1 \cup Q_1]\!] & = & [\![P_1]\!] \bullet [\![Q_1]\!] & \text{commutativity of } [\![\cdot]\!] \\ & = & [\![P_2]\!] \bullet [\![Q_1]\!] & \text{since } P_1 \equiv P_2 \\ & = & [\![P_2]\!] \bullet [\![Q_2]\!] & \text{since } Q_1 \equiv Q_2 \\ & = & [\![P_2 \cup Q_2]\!] & \text{commutativity of } [\![\cdot]\!] \end{array}$$

$\square$

3 Semantics of linguistics formalisms

Viewing grammars as formal entities that share many features with computer programs, it is natural to consider the notion of *semantics* of unification-based formalisms. Analogously to logic programming languages, the denotation of unification based grammars can be defined using various techniques. We review in this section the operational definition of Shieber, Schabes, and Pereira (1995) and the denotational definition of, e.g., Pereira and Shieber (1984) or Carpenter (1992, pp. 204-206). We show that these definitions are equivalent and that none of them supports compositionality.

3.1 Basic notions

We assume familiarity with theories of feature structure based unification grammars, as formulated by, e.g., Carpenter (1992) or Shieber (1992). Grammars are defined over *typed feature structures* (TFSs) which can be viewed as generalizations of first-order terms (Carpenter, 1991). TFSs are partially ordered by subsumption, with $\bot$ the least (or most general) TFS. A *multi-rooted structure* (MRS, see Sikkel

(1997) or Wintner and Francez (1999)) is a sequence of TFSs, with possible reentrancies among different elements in the sequence. Meta-variables A, B range over TFSs and σ, ρ – over MRSs. MRSs are partially ordered by subsumption, denoted '$\sqsubseteq$', with a least upper bound operation of *unification*, denoted '$\sqcup$', and a greatest lowest bound denoted '$\sqcap$'. We assume the existence of a fixed, finite set WORDS of words. A *lexicon* associates with every word a set of TFSs, its **category**. Meta-variable a ranges over WORDS and w – over strings of words (elements of WORDS*). Grammars are defined over a *signature* of types and features, assumed to be fixed below.

Definition 1 (Grammars). *A* **rule** *is an MRS of length greater than or equal to 1 with a designated (first) element, the* **head** *of the rule. The rest of the elements form the rule's* **body** *(which may be empty, in which case the rule is depicted as a TFS). A* **lexicon** *is a total function from* WORDS *to finite, possibly empty sets of TFSs. A* **grammar** $G = \langle \mathcal{R}, \mathcal{L}, A^s \rangle$ *is a finite set of rules* $\mathcal{R}$*, a lexicon* $\mathcal{L}$ *and a* **start symbol** A^s *that is a TFS.*

Figure 1 depicts an example grammar,[1] suppressing the type hierarchy on which it is based.[2] When the initial symbol is not explicitly depicted, it is assumed to be the most general feature structure that is subsumed by the head of the first listed rule.

$$A^s = (cat : s)$$

$$\mathcal{R} = \left\{ \begin{array}{lcll} (cat : s) & \rightarrow & (cat : n) & (cat : vp) \\ (cat : vp) & \rightarrow & (cat : v) & (cat : n) \\ (cat : vp) & \rightarrow & (cat : v) & \end{array} \right\}$$

$$\mathcal{L}(\text{John}) = \mathcal{L}(\text{Mary}) = \{(cat : n)\}$$
$$\mathcal{L}(\text{sleeps}) = \mathcal{L}(\text{sleep}) = \mathcal{L}(\text{loves}) = \{(cat : v)\}$$

Figure 1: An example grammar, G

The definition of unification is lifted to MRSs: let σ, ρ be two MRSs of the same length; the *unification* of σ and ρ, denoted $\sigma \sqcup \rho$, is the most general MRS that is subsumed by both σ and ρ, if such an MRS exists. Otherwise, the unification *fails*.

Definition 2 (Reduction). *An MRS* $\langle A_1, \ldots, A_k \rangle$ **reduces to** *a TFS* A *with respect to a grammar* G *(denoted* $\langle A_1, \ldots, A_k \rangle \Rightarrow_G A$*) iff there exists a rule* $\rho \in \mathcal{R}$ *such that* $\langle B, B_1, \ldots, B_k \rangle = \rho \sqcup \langle \bot, A_1, \ldots, A_k \rangle$ *and* $B \sqsubseteq A$*. When* G *is understood from the context it is omitted. Reduction can be viewed as the bottom-up counterpart of derivation.*

For two functions f, g, over the same (set) domain, $f + g$ is defined as $\lambda I.f(I) \cup g(I)$. Let $\mathbb{N}_0$ denote the set $\{0, 1, 2, 3, \ldots\}$. Let ITEMS be the set $\{[w, i, A, j] \mid w \in \text{WORDS}^*, A \text{ is a TFS and } i, j \in \mathbb{N}_0\}$. Let $\mathcal{I} = 2^{\text{ITEMS}}$. Meta-variables x, y range over items and I – over sets of items. When $\mathcal{I}$ is ordered by set inclusion it forms a complete lattice with set union as a least upper bound (lub) operation. A function $T : \mathcal{I} \rightarrow \mathcal{I}$ is monotone if whenever $I_1 \subseteq I_2$, also $T(I_1) \subseteq T(I_2)$. It is continuous if for every chain $I_1 \subseteq I_2 \subseteq \cdots$, $T(\bigcup_{j<\omega} I_j) = \bigcup_{j<\omega} T(I_j)$. If a function T is monotone it has a least fixpoint (Tarski-Knaster theorem); if T is also continuous, the fixpoint can be obtained by iterative application of T to the empty set (Kleene theorem): $lfp(T) = T \uparrow \omega$, where $T \uparrow 0 = \emptyset$ and $T \uparrow n = T(T \uparrow (n-1))$ when n is a successor ordinal and $\bigcup_{k<n}(T \uparrow n)$ when n is a limit ordinal.

[1] Grammars are displayed using a simple description language, where ':' denotes feature values and ',' denotes conjunction.
[2] Assume that in all the example grammars, the types s, n, v and vp are maximal and (pairwise) inconsistent.

3.2 An operational semantics

As Van Emden and Kowalski (1976) note, "to define an operational semantics for a programming language is to define an implementational independent interpreter for it. For predicate logic the proof procedure behaves as such an interpreter." Shieber, Schabes, and Pereira (1995) view parsing as a deductive process that proves claims about the grammatical status of strings from assumptions derived from the grammar. We follow their insight and notation and list a deductive system for parsing grammars formalized as in the previous section. As the special properties of different parsing algorithms are of little interest here, we limit the definition to a simple bottom-up procedure.

Definition 3 (Deductive parsing). *The deductive parsing system associated with a grammar* $G = \langle \mathcal{R}, \mathcal{L}, A^s \rangle$ *is defined over* ITEMS *and is characterized by:*

Axioms:

$$\begin{array}{ll} [\epsilon, i, A, i] & \textit{if } B \textit{ is an } \epsilon\textit{-rule in } \mathcal{R} \textit{ and } B \sqsubseteq A \\ [a, i, A, i+1] & \textit{if } B \in \mathcal{L}(a) \textit{ and } B \sqsubseteq A \end{array}$$

Goals:

$$[w, 0, A, |w|] \ \textit{where } A \sqsupseteq A^s$$

Inference rules:

$$\frac{[w_1, i_1, A_1, j_1], \ldots, [w_k, i_k, A_k, j_k]}{[w_1 \cdots w_k, i, A, j]} \qquad \begin{array}{l} \textit{if } j_l = i_{l+1} \textit{ for } 1 \leq l < k \textit{ and} \\ i = i_1 \textit{ and } j = j_k \textit{ and} \\ \langle A_1, \ldots, A_k \rangle \Rightarrow_G A \end{array}$$

Notice that the domain of items is infinite, and in particular that the number of axioms is infinite. Also, notice that the goal is to deduce a TFS which is subsumed by the start symbol, and when TFSs can be cyclic, there can be infinitely many such TFSs (and, hence, goals) – see Wintner and Francez (1999).

When an item $[w, i, A, j]$ can be deduced, applying k times the inference rules associated with a grammar G, we write $\vdash_G^k [w, i, A, j]$. When the number of inference steps is irrelevant it is omitted.

Definition 4 (Operational semantics). *The* **operational denotation** *of a grammar* G *is* $[\![G]\!]_{Op} = \{x \mid \vdash_G x\}$. $G_1 \equiv_{Op} G_2$ *iff* $[\![G_1]\!]_{Op} = [\![G_2]\!]_{Op}$.

We use the operational semantics to define the *language* generated by a grammar G. The language of a grammar G is $L(G) = \{\langle w, A \rangle \mid [w, 0, A, |w|] \in [\![G]\!]_{Op}\}$. Notice that a language is not merely a set of strings; rather, each string is associated with a TFS through the deduction procedure. Note also that the start symbol A^s does not play a role in this definition; this is equivalent to assuming that the start symbol is always the most general TFS, $\perp$.

The most natural observable for a grammar would be its language, either as a set of strings or augmented by TFSs. Thus we take $Ob(G)$ to be $L(G)$ and by definition, the operational semantics '$[\![\cdot]\!]_{Op}$' preserves observables.

3.3 Denotational semantics

As an alternative to the operational semantics discussed above, we consider in this section denotational semantics through a fixpoint of a transformational operator associated with grammars. This is essentially similar to the definition of Pereira and Shieber (1984) and Carpenter (1992, pp. 204-206). We then show that the algebraic semantics is equivalent to the operational one.

Associate with a grammar G an operator T_G that, analogously to the immediate consequence operator of logic programming, can be thought of as a "parsing step" operator in the context of grammatical formalisms. For the following discussion fix a particular grammar $G = \langle \mathcal{R}, \mathcal{L}, A^s \rangle$.

Definition 5. *Let $T_G : \mathcal{I} \to \mathcal{I}$ be a transformation on sets of items, where for every $I \subseteq$ ITEMS, $[w, i, A, j] \in T_G(I)$ iff either*

- *there exist $y_1, \ldots, y_k \in I$ such that $y_l = [w_l, i_l, A_l, j_l]$ for $1 \leq l \leq k$ and $i_{l+1} = j_l$ for $1 \leq l < k$ and $i_1 = 1$ and $j_k = j$ and $\langle A_1, \ldots, A_k \rangle \Rightarrow A$ and $w = w_1 \cdots w_k$; or*
- *$i = j$ and B is an ϵ-rule in G and $B \sqsubseteq A$ and $w = \epsilon$; or*
- *$i + 1 = j$ and $|w| = 1$ and $B \in \mathcal{L}(w)$ and $B \sqsubseteq A$.*

Theorem 2. *For every grammar G, T_G is monotone.*

Proof. Suppose $I_1 \subseteq I_2$. If $x \in T_G(I_1)$ then x was added by one of the three operations in the definition of T_G. Notice that the last two clauses of definition 5 are independent of I: they add the same items in each application of the operator. If $x \in T_G(I_1)$ due to them, then $x \in T_G(I_2)$, too. If x was added by the first clause, then there exist items $y_1, \ldots, y_k$ in I_1 to which this operation applies. Since $I_1 \subseteq I_2$, $y_1, \ldots, y_k$ are in I_2, too, and hence $x \in T_G(I_2)$, too. □

Theorem 3. *For every grammar G, T_G is continuous.*

Proof. First, T_G is monotone. Second, let $I = I_0 \subseteq I_1 \subseteq \ldots$ be a chain of items. If $x \in T_G(\bigcup_{i \geq 0} I_i)$ then there exist $y_1, \ldots, y_k \in \bigcup_{i \geq 0} I_i$ as required, due to which x is added. Then there exist $i_1, \ldots, i_k$ such that $y_1 \in I_{i_1}, \ldots, y_k \in I_{i_k}$. Let m be the maximum of $i_1, \ldots, i_k$. Then $y_1, \ldots, y_k \in I_m$, $x \in T_G(I_m)$ and hence $x \in \bigcup_{i \geq 0} T_G(I_i)$.
If $x \in \bigcup_{i \geq 0} T_G(I_i)$ then there exists some i that $x \in T_G(I_i)$. $I_i \subseteq \bigcup_{i \geq 0} I_i$ and since T_G is monotone, $T_G(I_i) \subseteq T_G(\bigcup_{i \geq 0} I_i)$, and hence $x \in T_G(\bigcup_{i \geq 0} I_i)$. Therefore T_G is continuous. □

Corollary 4. *For every grammar G, the least fixpoint of T_G exists and $lfp(T_G) = T_G \uparrow \omega$.*

Following the paradigm of logic programming languages, define a fixpoint semantics for unification-based grammars by taking the least fixpoint of the parsing step operator as the denotation of a grammar.

Definition 6 (Fixpoint semantics). *The fixpoint denotation of a grammar G is $[\![G]\!]_{fp} = lfp(T_G)$. $G_1 \equiv_{fp} G_2$ iff $lfp(T_{G_1}) = lfp(T_{G_2})$.*

The denotational definition is equivalent to the operational one:

Theorem 5. *For $x \in$ ITEMS, $x \in lfp(T_G)$ iff $\vdash_G x$.*

Proof.

- If $\vdash_G^n [w, i, A, j]$ then $[w, i, A, j] \in T_G \uparrow n$. By induction on n: if $n = 1$ then $[w, i, A, j]$ is an axiom and therefore either $|w| = 1$, $j = i+1$ and $B \in \mathcal{L}(w)$ for $B \sqsubseteq A$; or $w = \epsilon$, $i = j$ and B is an ϵ-rule for $B \sqsubseteq A$. By the definition of T_G, $T_G(\emptyset) = \{[\epsilon, i, A, j] \mid B \sqsubseteq A \text{ is an } \epsilon\text{-rule}\} \cup \{[w, i, A, i+1] \mid |w| = 1$ and $B \in \mathcal{L}(w)$ for $B \sqsubseteq A\}$, so we have $[w, i, A, j] \in T_G \uparrow 1$.
  Assume that the hypothesis holds for $n - 1$; assume that $\vdash_G^n [w, i, A, j]$ for $n > 1$. Then the inference rule must be applied at least once, i.e., there exist items $[w_1, i_1, A_1, j_1], \ldots, [w_k, i_k, A_k, j_k]$ such that $j_l = i_{l+1}$ for $1 \leq l < k$ and $i = i_1$ and $j = j_k$ and $\langle A_1, \ldots, A_k \rangle \Rightarrow A$ and $w = w_1 \cdots w_k$. Furthermore, for every $1 \leq l \leq k$, the item $[w_i, i_l, A_l, j_l]$ can be deduced in $n - 1$ steps: $\vdash_G^{n-1} [w_l, i_l, A_l, j_l]$. By the induction hypothesis, for every $1 \leq l \leq k$, $[w_l, i_l, A_l, j_l] \in T_G \uparrow (n-1)$. By the definition of T_G, applying the first clause of the definition, $[w, i, A, j] \in T_G(T_G \uparrow (n-1)) = T_G \uparrow n$.

- If $[w,i,A,j] \in T_G \uparrow n$ then $\vdash_G^n [w,i,A,j]$. By induction on n: if $n=1$, that is, $[w,i,A,j] \in T_G \uparrow 1$, then either $i=j$, $w=\epsilon$ and $B \sqsubseteq A$ is an ϵ-rule in G, or $i+1=j$ and $B \in \mathcal{L}(w)$ for $B \sqsubseteq A$. In the first case, $\vdash_G^1 [w,i,A,j]$ by the first axiom of the deductive procedure; in the other case, $\vdash_G^1 [w,i,A,j]$ by the second axiom.
  Assume that the hypothesis holds for $n-1$ and that $[w,i,A,j] \in T_G \uparrow n = T_G(T_G \uparrow (n-1))$. Then there exist items $y_1,\ldots,y_k \in T_G \uparrow (n-1)$ such that $y_l = [w_l, i_l, A_l, j_l]$ for $1 \leq l \leq k$ and $i_{l+1} = j_l$ for $1 \leq l < k$ and $i_1 = 1$ and $j_k = j$ and $\langle A_1,\ldots,A_k\rangle \Rightarrow A$ and $w = w_1 \cdots w_k$. By the induction hypothesis, for every $1 \leq l \leq k$, $\vdash_G^{n-1} [w_l, i_l, A_l, j_l]$, and the inference rule is applicable, so by an additional step of deduction we obtain $\vdash_G^n [w,i,A,j]$.

$\square$

Corollary 6. *The relation '$\equiv_{fp}$' preserves observables: whenever $G_1 \equiv_{fp} G_2$, also $Ob(G_1) = Ob(G_2)$.*

3.4 Compositionality

While the operational and the denotational semantics defined above are standard for complete grammars, they are too coarse to serve as a model when the composition of grammars is concerned. When the denotation of a grammar is taken to be $[\![G]\!]_{Op}$, important characteristics of the internal structure of the grammar are lost. To demonstrate the problem, we introduce a natural composition operator on grammars, namely union of the sets of rules (and the lexicons) in the composed grammars.

Definition 7 (Grammar union). *If $G_1 = \langle \mathcal{R}_1, \mathcal{L}_1, A_1^s\rangle$ and $G_2 = \langle \mathcal{R}_2, \mathcal{L}_2, A_2^s\rangle$ are two grammars over the same signature, then the* **union** *of the two grammars, denoted $G_1 \cup G_2$, is a new grammar $G = \langle \mathcal{R}, \mathcal{L}, A^s\rangle$ such that $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$, $\mathcal{L} = \mathcal{L}_1 + \mathcal{L}_2$ and $A^s = A_1^s \sqcap A_2^s$.*

Figure 2 exemplifies the union operation on grammars. Observe that $G_1 \cup G_2 = G_2 \cup G_1$.

$$
\begin{array}{lllll}
G_1: & A^s = (cat : s) & & & \\
 & (cat : s) & \rightarrow & (cat : n) & (cat : vp) \\
 & \mathcal{L}(\text{John}) = \{(cat : n)\} & & & \\
G_2: & A^s = (\bot) & & & \\
 & (cat : vp) & \rightarrow & (cat : v) & \\
 & (cat : vp) & \rightarrow & (cat : v) & (cat : n) \\
 & \mathcal{L}(\text{sleeps}) = \mathcal{L}(\text{loves}) = \{(cat : v)\} & & & \\
G_1 \cup G_2: & A^s = (cat : s) & & & \\
 & (cat : s) & \rightarrow & (cat : n) & (cat : vp) \\
 & (cat : vp) & \rightarrow & (cat : v) & \\
 & (cat : vp) & \rightarrow & (cat : v) & (cat : n) \\
 & \mathcal{L}(\text{John}) = \{(cat : n)\} & & & \\
 & \mathcal{L}(\text{sleeps}) = \mathcal{L}(\text{loves}) = \{(cat : v)\} & & &
\end{array}
$$

Figure 2: Grammar union

Proposition 7. *The equivalence relation '$\equiv_{Op}$' is not compositional with respect to $Ob, \{\cup\}$.*

Proof. Consider the following grammars:

$$
\begin{array}{llll}
G_1: & A^s = (cat:s) & & \\
 & (cat:s) & \rightarrow (cat:n) & (cat:vp) \\
 & \mathcal{L}(\text{John}) = \{(cat:n)\} & & \\
G_2: & A^s = (\bot) & & \\
 & \mathcal{L}(\text{loves}) = \{(cat:v)\} & & \\
G_3: & A^s = (\bot) & & \\
 & (cat:vp) & \rightarrow (cat:v) & (cat:n) \\
 & \mathcal{L}(\text{loves}) = \{(cat:v)\} & & \\
G_1 \cup G_2: & A^s = (cat:s) & & \\
 & (cat:s) & \rightarrow (cat:n) & (cat:vp) \\
 & \mathcal{L}(\text{John}) = \{(cat:n)\} & & \\
 & \mathcal{L}(\text{loves}) = \{(cat:v)\} & & \\
G_1 \cup G_3: & A^s = (cat:s) & & \\
 & (cat:s) & \rightarrow (cat:n) & (cat:vp) \\
 & (cat:vp) & \rightarrow (cat:v) & (cat:n) \\
 & \mathcal{L}(\text{John}) = \{(cat:n)\} & & \\
 & \mathcal{L}(\text{loves}) = \{(cat:v)\} & &
\end{array}
$$

Note that

$$[\![G_2]\!]_{Op} = [\![G_3]\!]_{Op} = \{[\text{“loves”}, i, (cat:v), i+1] \mid i \geq 0\}$$

but

$$\{[\text{“John loves John”}, i, (cat:s), i+3 \mid i \geq 0\} \subseteq [\![G_1 \cup G_3]\!]_{Op}$$
$$\{[\text{“John loves John”}, i, (cat:s), i+3 \mid i \geq 0\} \not\subseteq [\![G_1 \cup G_2]\!]_{Op}$$

Thus $G_2 \equiv_{Op} G_3$ but $(G_1 \cup G_2) \not\equiv_{Op} (G_1 \cup G_3)$, hence '$\equiv_{Op}$' is not compositional with respect to $Ob, \{\cup\}$. □

The implication of the above proposition is that while grammar union might be a natural, well defined syntactic operation on grammars, the standard semantics of grammars is too coarse to support it. Intuitively, this is because when a grammar G_1 includes a particular rule ρ that is inapplicable for reduction, this rule contributes nothing to the denotation of the grammar. But when G_1 is combined with some other grammar, G_2, ρ might be used for reduction in $G_1 \cup G_2$, where it can interact with the rules of G_2. The question to ask is, then, in what sense is a grammar a union of its rules? We suggest an alternative, fixpoint based semantics for unification based grammars that naturally supports compositionality.

4 A compositional semantics

To overcome the problems delineated above, we follow Mancarella and Pedreschi (1988) in moving one step further, considering the grammar transformation operator itself (rather than its fixpoint) as the denotation of a grammar.

Definition 8 (Algebraic semantics). *The algebraic denotation of a grammar G is $[\![G]\!]_{al} = T_G$. $G_1 \equiv_{al} G_2$ iff $T_{G_1} = T_{G_2}$.*

Not only is the algebraic semantics compositional, it is also commutative with respect to grammar union. To show that, a composition operation on denotations has to be defined, and we follow Mancarella and Pedreschi (1988) in its definition:

$$T_{G_1} \bullet T_{G_2} = \lambda I . T_{G_1}(I) \cup T_{G_2}(I)$$

Theorem 8. *The semantics '$\equiv_{al}$' is commutative with respect to grammar union and '$\bullet$': for every two grammars G_1, G_2, $[\![G_1]\!]_{al} \bullet [\![G_2]\!]_{al} = [\![G_1 \cup G_2]\!]_{al}$.*

Proof. It has to be shown that for every set of items I, $T_{G_1 \cup G_2}(I) = T_{G_1}(I) \cup T_{G_2}(I)$.

- if $x \in T_{G_1}(I) \cup T_{G_2}(I)$ then either $x \in T_{G_1}(I)$ or $x \in T_{G_2}(I)$. From the definition of grammar union, $x \in T_{G_1 \cup G_2}(I)$ in any case.
- if $x \in T_{G_1 \cup G_2}(I)$ then x can be added by either of the three clauses in the definition of T_G.
  - if x is added by the first clause then there is a rule $\rho \in \mathcal{R}_1 \cup \mathcal{R}_2$ that licenses the derivation through which x is added. Then either $\rho \in \mathcal{R}_1$ or $\rho \in \mathcal{R}_2$, but in any case ρ would have licensed the same derivation, so either $x \in T_{G_1}(I)$ or $x \in T_{G_2}(I)$.
  - if x is added by the second clause then there is an ϵ-rule in $G_1 \cup G_2$ due to which x is added, and by the same rationale either $x \in T_{G_1}(I)$ or $x \in T_{G_2}(I)$.
  - if x is added by the third clause then there exists a lexical category in $\mathcal{L}_1 \cup \mathcal{L}_2$ due to which x is added, hence this category exists in either $\mathcal{L}_1$ or $\mathcal{L}_2$, and therefore $x \in T_{G_1}(I) \cup T_{G_2}(I)$.

$\square$

Since '$\equiv_{al}$' is commutative, by proposition 1 it is also compositional with respect to grammar union. Intuitively, since T_G captures only one step of the computation, it cannot capture interactions among different rules in the (unioned) grammar, and hence taking T_G to be the denotation of G yields a compositional semantics.

The T_G operator reflects the structure of the grammar better than its fixpoint. In other words, the equivalence relation induced by T_G is finer than the relation induced by $lfp(T_G)$. The question is, how fine is the '$\equiv_{al}$' relation? To make sure that a semantics is not *too* fine, one usually checks the reverse direction.

Definition 9 (Full abstraction). *A semantic equivalence relation '$\equiv$' is* **fully-abstract** *iff*

$$P \equiv Q \textit{ iff for all } R,\ Ob(P \cup R) = Ob(Q \cup R)$$

As it turns out, the selection of T_G as the denotation of G is too fine: '$\equiv_{al}$' is not fully-abstract. To show that, one must provide two grammars, say G_1 and G_2, such that $G_1 \not\equiv_{al} G_2$ (that is, $T_{G_1} \neq T_{G_2}$), but still for every grammar G, $Ob(G \cup G_1) = Ob(G \cup G_2)$.

Proposition 9. *The semantic equivalence relation '$\equiv_{al}$' is not fully abstract.*

Proof. Let G_1 be the grammar

$$A_1^s = \perp, \mathcal{L}_1 = \emptyset, \mathcal{R}_1 = \{(cat : s) \to (cat : np)(cat : vp),\ (cat : np) \to (cat : np)\}$$

and G_2 be the grammar

$$A_2^s = \perp, \mathcal{L}_2 = \emptyset, \mathcal{R}_2 = \{(cat : s) \to (cat : np)(cat : vp)\}$$

- $G_1 \not\equiv_{al} G_2$: because

$$T_{G_1}(\{[\text{“John loves Mary”}, 6, (cat : np), 9]\}) = \{[\text{“John loves Mary”}, 6, (cat : np), 9]\}$$

but

$$T_{G_2}(\{[\text{“John loves Mary”}, 6, (cat : np), 9]\}) = \emptyset$$

- for all G, $Ob(G \cup G_1) = Ob(G \cup G_2)$. The only difference between $G \cup G_1$ and $G \cup G_2$ is the presence of the rule $(cat : np) \to (cat : np)$ in the former. This rule can contribute nothing to a deduction procedure, since any item it licenses must already be deducible. Therefore, any item deducible with $G \cup G_1$ is also deducible with $G \cup G_2$ and hence $Ob(G \cup G_1) = Ob(G \cup G_2)$.

$\square$

A simple solution to the problem would have been to consider, instead of T_G, the following operator as the denotation of G:

$$[\![G]\!]_{id} = \lambda I.T_G(I) \cup I$$

In other words, the semantics is $T_G + Id$, where Id is the identity operator. Evidently, for G_1 and G_2 of the above proof, $[\![G_1]\!]_{id} = [\![G_2]\!]_{id}$, so they no longer constitute a counter example. Also, it is easy to see that the proof of theorem 8 requires only a slight modification for it to hold in this case, so $T_G + Id$ is commutative (and hence compositional).

Unfortunately, this does not solve the problem. Let t_1, t_2, t_3 be (pairwise) inconsistent types (i.e., $t_1 \sqcup t_2 = t_2 \sqcup t_3 = t_1 \sqcup t_3 = \top$). Let G_1 be the grammar

$$A_1^s = \bot, \mathcal{L}_1 = \emptyset, \mathcal{R}_1 = \{(t_1) \to (t_2),\ (t_2) \to (t_3),\ (t_3) \to (t_1)\}$$

and G_2 be the grammar

$$A_1^s = \bot, \mathcal{L}_1 = \emptyset, \mathcal{R}_2 = \{(t_1) \to (t_3),\ (t_2) \to (t_1),\ (t_3) \to (t_2)\}$$

To see that $[\![G_1]\!]_{id} \neq [\![G_2]\!]_{id}$, observe that

$$T_{G_1}(\{[w, i, (t_1), j]\}) = \{[w, i, (t_1), j], [w, i, (t_3), j]\}$$

but

$$T_{G_2}(\{[w, i, (t_1), j]\}) = \{[w, i, (t_1), j], [w, i, (t_2), j]\}$$

To see that $Ob(G \cup G_1) = Ob(G \cup G_2)$ for every G, consider how a derivation in, say, $G \cup G_1$ can make use of, say, the rule $(t_1) \to (t_2)$. For this rule to be applicable, (t_2) must be deducible; hence, by application of the rules $(t_3) \to (t_2)$ and $(t_1) \to (t_3)$, (t_1) is deducible in $G \cup G_2$. Every rule application in either G_1 or G_2 can be modeled by two rule applications in the other grammar, and hence every TFS deducible in $G \cup G_1$ is deducible in $G \cup G_2$ and vice versa.

5 A compositional, fully abstract semantics

We have shown so far that '$[\![\cdot]\!]_{fp}$' is not compositional, and that '$[\![\cdot]\!]_{id}$' is compositional but not fully abstract. The "right" semantics, therefore, lies somewhere in between: since the choice of semantics induces a natural equivalence on grammars, we seek an equivalence that is cruder than '$[\![\cdot]\!]_{id}$' but finer than '$[\![\cdot]\!]_{fp}$'. In this section we adapt results from Lassez and Maher (1984) and Maher (1988) to the domain of unification-based linguistic formalisms.

Consider the following semantics for logic programs: rather than taking the operator associated with the entire program, look only at the rules (excluding the facts), and take the meaning of a program to be the function that is obtained by an infinite applications of the operator associated with the rules. In our framework, this would amount to associating the following operator with a grammar:

Definition 10. *Let $R_G : \mathcal{I} \to \mathcal{I}$ be a transformation on sets of items, where for every $I \subseteq$ ITEMS, $[w, i, A, j] \in R_G(I)$ iff there exist $y_1, \ldots, y_k \in I$ such that $y_l = [w_l, i_l, A_l, j_l]$ for $1 \le l \le k$ and $i_{l+1} = j_l$ for $1 \le l < k$ and $i_1 = 1$ and $j_k = j$ and $\langle A_1, \ldots, A_k \rangle \Rightarrow A$ and $w = w_1 \cdots w_k$.*

The functional denotation of a grammar G is $[\![G]\!]_{fn} = (R_G + Id)^\omega = \Sigma_{n=0}^{\infty}(R_G + Id)^n$. Notice that R_G^ω is not $R_G \uparrow \omega$: the former is a function from sets of items to set of items; the latter is a set of items.

Observe that R_G is defined similarly to T_G (definition 5), ignoring the items added (by T_G) due to ϵ-rules and lexical items. If we define the set of items $Init_G$ to be those items that are added by T_G independently of the argument it operates on, then for every grammar G and every set of items I, $T_G(I) = R_G(I) \cup Init_G$. Relating the functional semantics to the fixpoint one, we follow Lassez and Maher (1984) in proving that the fixpoint of the grammar transformation operator can be computed by applying the functional semantics to the set $Init_G$.

Definition 11. *For every grammar $G = \langle \mathcal{R}, \mathcal{L}, A^s \rangle$, let*

$$Init_G = \{[\epsilon, i, A, i] \mid B \textit{ is an } \epsilon\textit{-rule in } G \textit{ and } B \sqsubseteq A\} \cup \{[a, i, A, i+1] \mid B \in \mathcal{L}(a) \textit{ for } B \sqsubseteq A\}$$

Theorem 10. *For every grammar G, $(R_G + Id)^\omega(Init_G) = \mathit{lfp}(T_G)$.*

Proof. We show that for every n, $(T_G + Id) \uparrow n = (\Sigma_{k=0}^{n-1}(R_G + Id)^k)(Init_G)$ by induction on n.
For $n = 1$, $(T_G + Id) \uparrow 1 = (T_G + Id)((T_G + Id) \uparrow 0) = (T_G + Id)(\emptyset)$. Clearly, the only items added by T_G are due to the second and third clauses of definition 5, which are exactly $Init_G$. Also, $(\Sigma_{k=0}^{0}(R_G + Id)^k)(Init_G) = (R_G + Id)^0(Init_G) = Init_G$.
Assume that the proposition holds for $n-1$, that is, $(T_G + Id) \uparrow (n-1) = (\Sigma_{k=0}^{n-2}(R_G + Id)^k)(Init_G)$. Then

$$\begin{array}{rcll}
(T_G + Id) \uparrow n & = & (T_G + Id)((T_G + Id) \uparrow (n-1)) & \text{definition of } \uparrow \\
 & = & (T_G + Id)((\Sigma_{k=0}^{n-2}(R_G + Id)^k)(Init_G)) & \text{by the induction hypothesis} \\
 & = & (R_G + Id)((\Sigma_{k=0}^{n-2}(R_G + Id)^k)(Init_G)) \cup Init_G & \text{since } T_G(I) = R_G(I) \cup Init_G \\
 & = & (R_G + Id)((\Sigma_{k=0}^{n-2}(R_G + Id)^k)(Init_G)) & \\
 & = & (\Sigma_{k=0}^{n-1}(R_G + Id)^k)(Init_G) &
\end{array}$$

Hence $(R_G + Id)^\omega(Init_G) = (T_G + Id) \uparrow \omega = \mathit{lfp}(T_G)$. □

The choice of '$[\![\cdot]\!]_{\mathit{fn}}$' as the semantics calls for a different notion of observables. The denotation of a grammar is now a function which reflects an infinite number of applications of the grammar's rules, but completely ignores the ϵ-rules and the lexical entries. If we took the observables of a grammar G to be $L(G)$ we could in general have $[\![G_1]\!]_{\mathit{fn}} = [\![G_2]\!]_{\mathit{fn}}$ but $Ob(G_1) \neq Ob(G_2)$ (due to different lexicons), that is, the semantics would not preserve observables. However, when the lexical entries in a grammar (including the ϵ-rules, which can be viewed as empty categories, or the lexical entries of traces) are taken as *input*, a natural notion of observables preservation is obtained. To guarantee that the semantics is Ob-preserving, we define the observables of a grammar G *with respect to a given input*.

Definition 12 (Observables). *The **observables** of a grammar $G = \langle \mathcal{R}, \mathcal{L}, A^s \rangle$ with respect to an input set of items I are $Ob_I(G) = \{\langle w, A \rangle \mid [w, 0, A, |w|] \in [\![G]\!]_{\mathit{fn}}(I)\}$.*

Corollary 11. *The semantics '$[\![\cdot]\!]_{\mathit{fn}}$' is Ob_I-preserving: if $G_1 \equiv_{\mathit{fn}} G_2$ then for every I, $Ob_I(G_1) = Ob_I(G_2)$.*

The above definition corresponds to the previous one in a natural way: when the input is taken to be $Init_G$, the observables of a grammar are its language.

Theorem 12. *For every grammar G, $L(G) = Ob_{Init_G}(G)$.*
Proof.

$$\begin{array}{rcll}
L(G) & = & \{\langle w, A \rangle \mid [w, 0, A, |w|] \in [\![G]\!]_{\mathit{Op}}\} & \text{definition of } L(G) \\
 & = & \{\langle w, A \rangle \mid \vdash_G [w, 0, A, |w|]\} & \text{definition 4} \\
 & = & \{\langle w, A \rangle \mid [w, 0, A, |w|] \in \mathit{lfp}(T_G)\} & \text{by theorem 5} \\
 & = & \{\langle w, A \rangle \mid [w, 0, A, |w|] \in [\![G]\!]_{\mathit{fn}}(Init_G)\} & \text{by theorem 10} \\
 & = & Ob_{Init_G}(G) & \text{by definition 12}
\end{array}$$

□

To show that '$[\![\cdot]\!]_{fn}$' is compositional we must define an operator for combining denotations. Unfortunately, the simplest operator, '+', would not do. To see that, consider the following grammars, where the types s, vp and v are pairwise incompatible:

$$\begin{array}{ll} G_1: & A_1^s = \bot, \mathcal{L}_1 = \emptyset, \mathcal{R}_1 = \{(cat:s) \rightarrow (cat:vp)\} \\ G_2: & A_2^s = \bot, \mathcal{L}_2 = \emptyset, \mathcal{R}_2 = \{(cat:vp) \rightarrow (cat:v)\} \end{array}$$

Observe that, for $x = [w, i, (cat:v), j]$,

$$\begin{array}{l} [\![G_1]\!]_{fn}(\{x\}) = \{x\} \\ [\![G_2]\!]_{fn}(\{x\}) = \{x, [w, i, (cat:vp), j]\} \\ [\![G_1 \cup G_2]\!]_{fn}(\{x\}) = \{x, [w, i, (cat:vp), j], [w, i, (cat:s), j]\} \end{array}$$

That is, $([\![G_1]\!]_{fn} + [\![G_2]\!]_{fn})(\{x\}) = [\![G_1]\!]_{fn}(\{x\}) \cup [\![G_2]\!]_{fn}(\{x\}) \neq [\![G_1 \cup G_2]\!]_{fn}(\{x\})$.

However, a different operator does the job. Define $[\![G_1]\!]_{fn} \bullet [\![G_2]\!]_{fn}$ to be $([\![G_1]\!]_{fn} + [\![G_2]\!]_{fn})^{\omega}$. Then '$[\![\cdot]\!]_{fn}$' is commutative with respect to '$\bullet$' and '$\cup$'. This proof is slightly more involved.

Lemma 13. *For every grammar G, $[\![G]\!]_{fn}$ is increasing: $[\![G]\!]_{fn}(I) \supseteq I$ for all I.*

Proof. Since $(R_G + Id)(I) \supseteq I$, also $(R_G + Id)^{\omega}(I) \supseteq I$. □

Definition 13. *If f, g are two functions over the same domain and range, let $f \leq g$ iff for all I, $f(I) \subseteq g(I)$. Let $f \circ g$ denote function composition.*

Lemma 14. *For every two grammars G_1, G_2, $([\![G_1]\!]_{fn} + [\![G_2]\!]_{fn}) \leq ([\![G_1]\!]_{fn} \circ [\![G_2]\!]_{fn})$.*

Proof. $([\![G_1]\!]_{fn} \circ [\![G_2]\!]_{fn})(I) = [\![G_1]\!]_{fn}([\![G_2]\!]_{fn}(I))$. $[\![G_1]\!]_{fn}(I) \supseteq I$, hence $[\![G_1]\!]_{fn}([\![G_2]\!]_{fn}(I)) \supseteq [\![G_2]\!]_{fn}(I)$. From lemma 13 and monotonicity, also $[\![G_1]\!]_{fn}([\![G_2]\!]_{fn}(I)) \supseteq [\![G_1]\!]_{fn}(I)$. Hence $([\![G_1]\!]_{fn} \circ [\![G_2]\!]_{fn})(I) \supseteq ([\![G_1]\!]_{fn} \cup [\![G_2]\!]_{fn})(I)$ and $([\![G_1]\!]_{fn} + [\![G_2]\!]_{fn}) \leq ([\![G_1]\!]_{fn} \circ [\![G_2]\!]_{fn})$. □

Lemma 15. *For every grammar G, $[\![G]\!]_{fn}$ is idempotent: $[\![G]\!]_{fn} \circ [\![G]\!]_{fn} = [\![G]\!]_{fn}$.*

Proof. (Lassez and Maher, 1984) For every I, $([\![G]\!]_{fn} \circ [\![G]\!]_{fn})(I) = ((R_G + Id)^{\omega} \circ (R_G + Id)^{\omega})(I) = (R_G + Id)^{\omega}((R_G + Id)^{\omega}(I)) = \Sigma_{i=0}^{\infty}(R_G + Id)^i(\Sigma_{j=0}^{\infty}(R_G + Id)^j(I)) = \Sigma_{i=0}^{\infty}\Sigma_{j=0}^{\infty}(R_G + Id)^{i+j}(I) = \Sigma_{m=0}^{\infty}(R_G + Id)^m(I) = (R_G + Id)^{\omega}(I) = [\![G]\!]_{fn}$. □

Theorem 16. *$[\![G_1 \cup G_2]\!]_{fn} = [\![G_1]\!]_{fn} \bullet [\![G_2]\!]_{fn}$.*

Proof. (Lassez and Maher, 1984)

$$\begin{array}{rcll} [\![G_1 \cup G_2]\!]_{fn} & = & (R_{G_1} + Id + R_{G_2} + Id)^{\omega} & \\ & \leq & ((R_{G_1} + Id)^{\omega} + (R_{G_2} + Id)^{\omega})^{\omega} & \text{since } R_G + Id \leq (R_G + Id)^{\omega} \text{ for every } G \\ & \leq & ((R_{G_1} + Id)^{\omega} \circ (R_{G_2} + Id)^{\omega})^{\omega} & \text{by lemma 14} \\ & \leq & ((R_{G_1 \cup G_2} + Id)^{\omega} \circ (R_{G_1 \cup G_2} + Id)^{\omega})^{\omega} & \text{since } G_1 \cup G_2 \supseteq G_1 \text{ and } G_1 \cup G_2 \supseteq G_2 \\ & = & ([\![G_1 \cup G_2]\!]_{fn} \circ [\![G_1 \cup G_2]\!]_{fn})^{\omega} & \text{definition of } [\![\cdot]\!]_{fn} \\ & = & ([\![G_1 \cup G_2]\!]_{fn})^{\omega} & \text{by lemma 15} \\ & = & [\![G_1 \cup G_2]\!]_{fn} & \text{since } (f^{\omega})^{\omega} = f^{\omega} \end{array}$$

Thus all the inequations are equations, and in particular $[\![G_1 \cup G_2]\!]_{fn} = ((R_{G_1} + Id)^{\omega} + (R_{G_2} + Id)^{\omega})^{\omega} = ([\![G_1]\!]_{fn}^{\omega} + [\![G_2]\!]_{fn}^{\omega})^{\omega} = [\![G_1]\!]_{fn} \bullet [\![G_2]\!]_{fn}$. □

Since '$[\![\cdot]\!]_{fn}$' is commutative, it is also compositional. For logic programs, Maher (1988) shows that this choice of semantics is also fully abstract, but this proof is not carried out in the algebraic domain. Rather, Maher (1988) shows that two programs are fn-equivalent iff they have the same *Herbrand* model, and then uses logical equivalence to derive full abstraction. A similar technique, using the logical domain, is employed by Bugliesi, Lamma, and Mello (1994) to prove the same result. We pride a more direct, constructive proof below.

Theorem 17. *The semantics '$[\![\cdot]\!]_{fn}$' is fully abstract: for every two grammars G_1 and G_2, if for every grammar G and set of items I, $Ob_I(G_1 \cup G) = Ob_I(G_2 \cup G)$, then $G_1 \equiv_{fn} G_2$.*

Proof. Assume that $[\![G_1]\!]_{fn} \neq [\![G_2]\!]_{fn}$. Without loss of generality, assume that there exist an item $x = [w, i, A, j]$ and a set I such that $x \in [\![G_1]\!]_{fn}(I)$ but $x \notin [\![G_2]\!]_{fn}(I)$. x is added to $[\![G_1]\!]_{fn}(I)$ through successive applications of the rules in $\mathcal{R}_1$ to items in I. Let $\{y_1, \ldots, y_k\} \subseteq I$ be the (input) items that partake in this sequence. For every $1 \leq i \leq k$, let A_i be the TFSs included in y_i; let $y'_i = [a, i-1, A_i, i]$, for some $a \in$ WORDS.

Recall from the definition of R_G that applicability of a rule to a set of items depends both on the TFSs and on the indices of the items; however, when a rule is applicable to some sequence of items, the same rule is also applicable to different items, with identical TFSs, as long as their indices are consecutive. In other words, the requirement that the indices of an item sequence be consecutive is independent of the requirement that the TFSs in the sequence be unifiable with the rule.

Now consider the set $J = \{y'_1, \ldots, y'_k\}$ and the item $x' = [a^k, 0, A, k]$. We claim that $x' \in [\![G_1]\!]_{fn}(J)$ but $x' \notin [\![G_2]\!]_{fn}(J)$. $x' \in [\![G_1]\!]_{fn}(J)$ for the same reason that $x \in [\![G_1]\!]_{fn}(I)$ – exactly the same rules can be applied in order to generate x'. Now assume $x' \in [\![G_2]\!]_{fn}(J)$; by the same rationale, the same rules that are applied in order to generate x', starting from J, could have been applied in order to generate x starting from I, contradicting the assumption that $x \notin [\![G_2]\!]_{fn}(I)$. Thus we posit a set of items, J, such that $x' \in [\![G_1]\!]_{fn}(J)$ but $x' \notin [\![G_2]\!]_{fn}(J)$. Hence $\langle a^k, A\rangle \in Ob_J(G_1)$ but $\langle a^k, A\rangle \notin Ob_J(G_2)$. □

6 Conclusions

This paper discusses alternative definitions for the semantics of unification-based linguistic formalisms, culminating in one that is both compositional and fully-abstract (with respect to grammar union, a simple syntactic combination operations on grammars). This is mostly an adaptation of well-known results from logic programming to the framework of unification-based linguistic formalisms, and it is encouraging to see that the same choice of semantics which is compositional and fully-abstract for Prolog turned out to have the same desirable properties in our domain.

The functional semantics '$[\![\cdot]\!]_{fn}$' defined here assigns to a grammar a function which reflects the (possibly infinite) successive application of grammar rules, viewing the lexicon as input to the parsing process. We believe that this is a key to modularity in grammar design. A grammar module has to define a set of items that it "exports", and a set of items that can be "imported", in a similar way to the declaration of interfaces in programming languages. We are currently working out the details of such a definition. An immediate application will facilitate the implementation of grammar development systems that support modularity in a clear, mathematically sound way.

The results reported here can be extended in various directions. First, we are only concerned in this work with one composition operator, grammar union. But alternative operators are possible, too. In particular, it would be interesting to define an operator which combines the information encoded in two grammar rules, for example by unifying the rules. Such an operator would facilitate a separate development of grammars along a different axis: one module can define the syntactic component of a grammar while another module would account for the semantics. The composition operator will unify each rule of one module with an associated rule in the other. It remains to be seen whether the grammar semantics we define here is compositional and fully abstract with respect to such an operator.

A different extension of these results should provide for a distribution of the type hierarchy among several grammar modules. While we assume in this work that all grammars are defined over a given signature, it is more realistic to assume separate, interacting signatures. We hope to be able to explore these directions in the future.

This paper is an extended version of Wintner (1999). I am grateful to Nissim Francez for commenting on an earlier version. This work was supported by an IRCS Fellowship and NSF grant SBR 8920230.

References

Apt, Krzysztof R. and M. H. Van Emden. 1982. Contributions to the theory of logic programming. *Journal of the Association for Computing Machinery*, 29(3):841–862, July.

Brogi, Antonio, Evelina Lamma, and Paola Mello. 1992. Compositional model-theoretic semantics for logic programs. *New Generation Computing*, 11:1–21.

Brogi, Antonio, Evelina Lamma, and Paola Mello. 1993. Composing open logic programs. *Journal of Logic and Computation*, 3(4):417–439.

Brogi, Antonio, Paolo Mancarella, Dino Pedreschi, and Franco Turini. 1990. Composition operators for logic theories. In J. W. Lloyd, editor, *Computational Logic – Symposium Proceedings*, pages 117–134. Springer, November.

Bugliesi, Michele, Evelina Lamma, and Paola Mello. 1994. Modularity in logic programming. *Journal of Logic Programming*, 19,20:443–502.

Carpenter, Bob. 1991. Typed feature structures: A generalization of first-order terms. In Vijai Saraswat and Ueda Kazunori, editors, *Logic Programming – Proceedings of the 1991 International Symposium*, pages 187–201, Cambridge, MA. MIT Press.

Carpenter, Bob. 1992. *The Logic of Typed Feature Structures.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

Erbach, Gregor and Hans Uszkoreit. 1990. Grammar engineering: Problems and prospects. CLAUS report 1, University of the Saarland and German research center for Artificial Intelligence, July.

Gaifman, Haim and Ehud Shapiro. 1989. Fully abstract compositional semantics for logic programming. In *16th Annual ACM Symposium on Principles of Logic Programming*, pages 134–142, Austin, Texas, January.

Lassez, J.-L. and M. J. Maher. 1984. Closures and fairness in the semantics of programming logic. *Theoretical computer science*, 29:167–184.

Maher, M. J. 1988. Equivalences of logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann Publishers, Los Altos, CA, chapter 16, pages 627–658.

Mancarella, Paolo and Dino Pedreschi. 1988. An algebra of logic programs. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth international conference and symposium*, pages 1006–1023, Cambridge, Mass. MIT Press.

Miller, D. 1986. A theory of modules in logic programming. In *Proceedings of the symposium on logic programming*, pages 106–114.

Miller, D. 1989. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79–108.

O'keefe, R. 1985. Towards an algebra for constructing logic programs. In J. Cohen and J Conery, editors, *Proceedings of IEEE symposium on logic programming*, pages 152–160, New York. IEEE Computer Society Press.

Pereira, Fernando C. N. and Stuart M. Shieber. 1984. The semantics of grammar formalisms seen as computer languages. In *Proceedings of the 10th international conference on computational linguistics and the 22nd annual meeting of the association for computational linguistics*, pages 123–129, Stanford, CA, July.

Shieber, Stuart, Yves Schabes, and Fernando Pereira. 1995. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1-2):3–36, July/August.

Shieber, Stuart M. 1992. *Constraint-Based Grammar Formalisms.* MIT Press, Cambridge, Mass.

Sikkel, Klaas. 1997. *Parsing Schemata.* Texts in Theoretical Computer Science – An EATCS Series. Springer Verlag, Berlin.

Van Emden, M. H. and Robert A. Kowalski. 1976. The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery*, 23(4):733–742, October.

Wintner, Shuly. 1999. Compositional semantics for linguistic formalisms. In *Proceedings of ACL'99, the 37th Annual Meeting of the Association for Computational Linguistics*, June.

Wintner, Shuly and Nissim Francez. 1999. Off-line parsability and the well-foundedness of subsumption. *Journal of Logic, Language and Information*, 8(1):1–16, January.