

---

## Type Signature Modules

Yael Sygal and Shuly Wintner

### Abstract

This work provides the essential foundations for modular construction of typed unification grammars for natural languages. Much of the information in such grammars is encoded in the type signature, and hence we focus on modularized development of the signatures. We extend the preliminary results of Cohen-Sygal and Wintner (2006) and define *signature modules*, facilitating module interaction and modular development of grammars. Our definitions are motivated by the actual needs of grammar developers and meet these needs by conforming to a detailed set of desiderata.

**Keywords** TYPE SIGNATURES, MODULARIZATION, GRAMMAR ENGINEERING

### 9.1 Introduction

Development of large-scale grammars for natural languages is an active area of research in human language technology. Such grammars are developed not only for purposes of theoretical linguistic research, but also for natural language applications such as machine translation, speech generation, etc. Wide-coverage grammars are being developed for various languages (Oepen et al., 2002, Hinrichs et al., 2004, King et al., 2005) in several theoretical frameworks, e.g., LFG (Dalrymple, 2001) and HPSG (Pollard and Sag, 1994).

Grammar development is a complex enterprise: it is not unusual for a single grammar to be developed by a team including several linguists, computational linguists and computer scientists. The scale of grammars is overwhelming: for example, the English resource grammar (Copestake and Flickinger, 2000) includes thousands of types. This raises problems reminiscent of those encountered in large-scale software development. Yet while software engineering provides adequate solutions for the programmer, no grammar devel-

opment environment supports even the most basic needs, such as grammar modularization, combination of sub-grammars, separate compilation and automatic linkage of grammars, information encapsulation, etc. (Klint et al., 2005). Referring to grammar engineering, Copestake and Flickinger (2000) note: “to some extent it just has to be accepted that it really is inherently difficult.”

This paper provides a thorough, well-founded solution to this difficult problem. After a review of some basic notions we list a set of desiderata in Section 9.1.2 and discuss related work in Section 9.1.3, highlighting the shortcomings of existing approaches. We review the definitions of Cohen-Sygal and Wintner (2006) in Section 9.2. We then introduce signature modules in Section 9.3 and show how two modules are combined in Section 9.4. We exemplify the definitions on two toy examples, but the solution can be scaled up to real-life grammars. We conclude with directions for future research.

### 9.1.1 Type signatures

We assume familiarity with theories of (typed) unification grammars, as formulated by, e.g., Carpenter (1992) and Penn (2000). The definitions in this section set the notation and recall basic notions. For a partial function  $F$ , ‘ $F(x)\downarrow$ ’ means that  $F$  is defined for the value  $x$ .

**Definition 1** A **type signature** is a tuple  $\langle \text{TYPE}, \sqsubseteq, \text{FEAT}, \text{Approp} \rangle$ , where:

1.  $\langle \text{TYPE}, \sqsubseteq \rangle$  is a finite bounded complete partial order<sup>1</sup> (the **type hierarchy**)
2. FEAT is a finite set, disjoint from TYPE.
3.  $\text{Approp} : \text{TYPE} \times \text{FEAT} \rightarrow \text{TYPE}$  (the **appropriateness specification**) is a partial function such that for every  $F \in \text{FEAT}$ :

**Feature Introduction** there exists a type  $\text{Int}(F)$  such that  $\text{Approp}(\text{Int}(F), F)\downarrow$ , and for all  $t \in \text{TYPE}$ , if  $\text{Approp}(t, F)\downarrow$ , then  $\text{Int}(F) \sqsubseteq t$ ;

**Upward Closure** for all  $s, t \in \text{TYPE}$ , if  $\text{Approp}(s, F)\downarrow$  and  $s \sqsubseteq t$ , then  $\text{Approp}(t, F)\downarrow$  and  $\text{Approp}(s, F) \sqsubseteq \text{Approp}(t, F)$ .

If  $x \sqsubseteq y$ , then  $x$  is a **supertype** of  $y$  and  $y$  is a **subtype** of  $x$ .

### 9.1.2 Desiderata

In defining a framework for grammar modularization we are guided by the following set of desiderata, adapted from Cohen-Sygal and Wintner (2006):

**Partiality:** Modules should provide means for specifying *partial* information about the components of a grammar. Since most of the information in typed formalisms is encoded by the type signature, modularization

---

<sup>1</sup>A partial order is **bounded complete** (BCPO) if every subset that has an upper bound has a unique *least* upper bound.

must be carried out mainly through the distribution of the signature between the different modules.

**Extensibility:** While modules can specify partial information, it must be possible to deterministically extend a module (which can be the result of the combination of several modules) into a full grammar.

**Privacy:** Modules should be able to hide (encapsulate) information and render it unavailable to other modules.

**Consistency:** Contradicting information in different modules must be detected when modules are combined.

**Flexibility:** The grammar designer should be provided with as much flexibility as possible. The definition of modules should not be unnecessarily constrained.

**(Remote) Reference:** A good solution should enable one module to refer to entities defined in another. Specifically, it should enable the designer of module  $M_i$  to use an entity (e.g., a type or a feature structure) defined in  $M_j$  without specifying the entity explicitly.

**Parsimony:** When two modules are combined, the resulting module must include only information encoded in each of the modules and information resulting from the combination operation itself.

Summing up, a good solution for grammar modularization should facilitate collaborative development of grammars, whether it is a single large-scale grammar developed by a team, or a set of grammars for different languages sharing some core fragments and principles (Bender et al., 2005, King et al., 2005), or a sequence of grammars reflecting language development. The solution we advocate here satisfies all these requirements and can be used for these and similar applications.

### 9.1.3 Related work

Several works address the issue of grammar modularization in unification formalisms. Moshier (1997) views HPSG, and in particular its signature, as a collection of constraints over maps between sets. This allows the grammar writer to specify any partial information about the signature, and provides the needed mathematical and computational capabilities to integrate the information with the rest of the signature. Pendar (2007) outlines an approach to incorporate soft constraints into grammars, to resolve conflicting requirements. These works do not explicitly define grammar modules and the way they interact. There are no mechanisms for privacy and information encapsulation.

A modular version of context-free grammars is given by Wintner (2002). Based on it, Keselj (2001) defines modular HPSG, where each module is an ordinary type signature, but each of the sets FEAT and TYPE is divided into

two disjoint sets of private and public elements. In this solution, modules cannot specify partial information; module combination is not associative; and the only channel of interaction between modules is the names of types.

King et al. (2005) augment LFG with a makeshift signature to allow modular development of *untyped* unification grammars. In addition, they suggest that any development team should agree in advance on the feature space. This work emphasizes the observation that the modularization of the signature is the key for modular development of grammars. However, the proposed solution is ad-hoc and cannot be taken seriously as a concept of modularization. In particular, the suggestion for an agreement on the feature space undermines the essence of modular design.

To support rapid prototyping of deep grammars, Bender and Flickinger (2005) propose a framework in which the grammar developer can select pre-written grammar fragments, accounting for common linguistic phenomena that vary across languages (e.g., word order, yes-no questions and sentential negation). The developer can specify how these phenomena are realized in a given language, and a grammar for that language is automatically generated, implementing that particular realization of the phenomenon, integrated with a language-independent grammar core. While Bender and Flickinger (2005) refer to such pre-written components as *modules*, these are clearly not modules in the usual sense: they are pre-written fragments of code which the grammar developer does not develop, and they do not interact freely with other fragments of the grammar.

## 9.2 Partially specified signatures

Our work extends and improves the results of Cohen-Sygal and Wintner (2006), who introduce *partially specified signatures*, which we review and evaluate in this section. The key here is a move from concrete type signatures to descriptions thereof; rather than specify types, a description uses nodes to denote types and arcs to denote elements of the subsumption and appropriateness relations of signatures. We assume enumerable, disjoint sets TYPE of types, FEAT of features and NODES of nodes, over which PSSs are defined.

**Definition 2** A **partially specified signature (PSS)** over TYPE, FEAT and NODES is a finite, directed labeled graph  $S = \langle Q, T, \preceq, Ap \rangle$ , where:

1.  $Q \subseteq \text{NODES}$  is a finite, nonempty set of nodes.
2.  $T : Q \rightarrow \text{TYPE}$  is a partial one to one function, marking some of the nodes with types.
3.  $\preceq \subseteq Q \times Q$  is an antireflexive relation specifying (immediate) subsumption; its reflexive-transitive closure, ' $\preceq^*$ ', is antisymmetric.
4.  $Ap \subseteq Q \times \text{FEAT} \times Q$  is a relation specifying appropriateness.

5. (Relaxed Upward Closure) for all  $q_1, q'_1, q_2 \in Q$  and  $F \in \text{FEAT}$ , if  $(q_1, F, q_2) \in Ap$  and  $q_1 \preceq^* q'_1$ , then there exists  $q'_2 \in Q$  such that  $q_2 \preceq^* q'_2$  and  $(q'_1, F, q'_2) \in Ap$

A PSS is a finite directed graph whose nodes denote types and whose edges denote the subsumption and appropriateness relations. Nodes can be *marked* by types through the function  $T$  but can also be *anonymous* (unmarked). Anonymous nodes facilitate reference, in one module, to types that are defined in another.  $T$  is one-to-one since two marked nodes must denote different types.

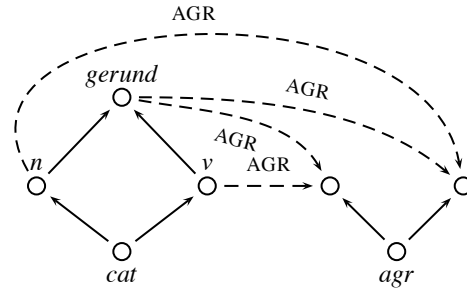
The ' $\preceq$ ' relation specifies an immediate subsumption order over the nodes, with the intention that this order hold later for the types denoted by nodes. This is why ' $\preceq^*$ ' is required to be a partial order. The type hierarchy of PSSs is partially ordered but this order is not necessarily a bounded complete one, thus allowing more flexibility in grammar design.

In contrast to type signatures, the appropriateness relation  $Ap$  is not required to be a function. Rather, it is a relation which may specify *several* appropriate nodes for the values of a feature  $F$  at a node  $q$ . The intention is that the eventual value of  $Approp(T(q), F)$  be the *lub* of the types of all those nodes  $q'$  such that  $Ap(q, F, q')$ . The  $Ap$  relation is restricted by a relaxed version of upward closure. Finally, the feature introduction condition of type signatures is not enforced by PSSs, again, to allow more flexibility for the grammar designer.

**Example 1** A simple PSS  $S_1$  is depicted in Figure 1, where solid arrows represent the ' $\preceq$ ' (subsumption) relation and dashed arrows, labeled by features, the  $Ap$  relation.  $S_1$  stipulates two subtypes of *cat*,  $n$  and  $v$ , with a common subtype, *gerund*. The feature AGR is appropriate to all three categories, with distinct (but anonymous) values for  $Approp(n, \text{AGR})$  and  $Approp(v, \text{AGR})$ .  $Approp(\text{gerund}, \text{AGR})$  will eventually be the *lub* of  $Approp(n, \text{AGR})$  and  $Approp(v, \text{AGR})$ , hence the multiple outgoing AGR arcs from *gerund*.

Observe that in  $S_1$ , ' $\preceq$ ' is not a BCPO,  $Ap$  is not a function and the feature introduction condition does not hold.

An additional restriction is imposed on PSSs: a PSS is *well-formed* if it contains no redundant arcs and nodes. A subsumption arc  $(q_1, q_2)$  is *redundant* if it is a member of the transitive closure of  $\preceq$ , where  $\preceq$  excludes  $(q_1, q_2)$ . A PSS does not contain redundant appropriateness arcs if any two nodes that are appropriate for the same node and feature are not related by subsumption. If they are, then the appropriateness arc whose target is the smaller node is redundant due to the 'lub' intention of appropriateness arcs. Finally, a PSS includes no redundant nodes if any two anonymous nodes are *distinguishable*, i.e., if each node encodes unique information. Given a PSS  $S$ , it can

FIGURE 1 A partially specified signature,  $S_1$ 

be *compact*ed into a PSS,  $compact(S)$ , by removing all the redundant arcs and by unifying all the indistinguishable nodes in  $S$ . Two nodes, only one of which is anonymous, can still be otherwise indistinguishable. Such nodes will, eventually, be coalesced, but only after all modules are combined.

To combine two PSSs, Cohen-Sygal and Wintner (2006) define the *merge* operator which fundamentally unions two PSSs, taking care to coalesce nodes that are marked by the same type, as well as pairs of indistinguishable anonymous nodes. An anonymous node cannot be coalesced with a typed node, even if they are otherwise indistinguishable, to guarantee the associativity of the operation. Anonymous nodes are assigned types only after all modules combine. Two PSSs can be merged only if the resulting subsumption relation is indeed a partial order, where the only obstacle can be the antisymmetry of the resulting relation.

After all the modules are combined, the PSS is extended into a bona fide signature by assigning types to anonymous nodes, extending ' $\preceq$ ' to a BCPO and extending  $Ap$  to a full appropriateness specification.

The solution of Cohen-Sygal and Wintner (2006) adheres to the desiderata of section 9.1.2, but provides very limited means for modules to interact: A module  $M_i$  can use an entity defined in  $M_j$  only by specifically referring to its type or by specifying all its attributes. Practically,  $M_i$  can only incorporate information from  $M_j$  by being familiar with all the information encoded in  $M_j$ . In the following section we open up new channels of communication among modules, inspired by modules in programming languages. We show that our definitions neatly support collaborative development of unification grammars.

### 9.3 Signature modules

*Signature modules* extend PSSs and provide a complete, well-defined framework for modular development of type signatures and hence of typed unification grammars. Modules may choose which information to expose to other modules and how other modules may use the information they encode. Our solution extends the denotation of nodes by viewing them as *parameters*: Similarly to parameters in programming languages, these are entities through which information can be imported or exported from other modules.

**Definition 3** A (**signature**) **module** over the sets TYPE, FEAT and NODES is a tuple  $P = \langle S, Int, Imp, Exp \rangle$ , where  $S = \langle Q, T, \preceq, Ap \rangle$  is a PSS, and where:

- $Int \subseteq Q$  is a set of internal types
- $Imp \subseteq Q$  is an ordered set of imported parameters
- $Exp \subseteq Q$  is an ordered set of exported parameters
- $Int \cap Imp = Int \cap Exp = \emptyset$
- for all  $q \in Int, T(q) \downarrow$

We refer to elements of (the repetition-free lists)  $Imp$  and  $Exp$  using indices, with the notation  $Imp[i], Exp[j]$ , respectively.  $S$  is the **underlying PSS** of  $P$ .

A module is a PSS whose nodes are distributed among three sets of *internal*, *imported* and *exported* nodes. If a node is internal it cannot be imported or exported; but a node can be simultaneously imported and exported. A node which does not belong to any of the three sets is called *external*. Nodes denote types, as above, but they differ in the way they communicate with nodes in other modules. As their name implies, internal nodes are internal to one module and cannot interact with nodes in other modules. Such nodes provide a mechanism similar to local variables in programming languages.

Non-internal nodes may interact with the nodes in other modules: Imported nodes expect to receive information from other modules, while exported nodes provide information to other modules. Since anonymous nodes facilitate reference, in one module, to information encoded in another module, such nodes cannot be internal. The order of imported and exported nodes controls the assignment of parameters when two modules are combined, as will be shown below.<sup>2</sup>

A signature module is *compacted* in the same way a PSS is compacted; the classification of nodes is induced from the input module. Internal nodes may be coalesced only with each other, resulting in an internal node. If an

---

<sup>2</sup>In fact,  $Imp$  and  $Exp$  can be general sets, rather than lists, as long as the combination operations can deterministically map nodes from  $Exp$  to nodes of  $Imp$ . For simplicity, we limit the discussion to the familiar case of lists, where matching elements from  $Exp$  to  $Imp$  is done by the location of the element on the list, see definitions 4 and 5.

imported node is coalesced with some other node, the resulting node is imported. Similarly, if one of the nodes is exported then the resulting node is exported. A module is extended to a bona fide type signature by extending its underlying PSS to a type signature, ignoring the classification of nodes.

Figure 2 depicts a module,  $P_1$ , based on the PSS of Figure 1. In the examples, the classification of nodes is encoded graphically as follows:

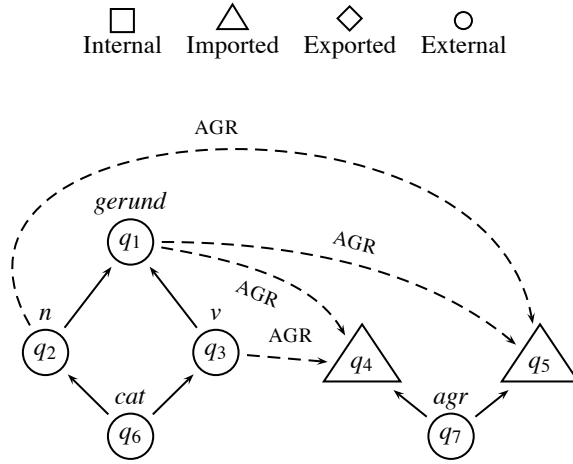


FIGURE 2 A module,  $P_1$

### 9.4 Module Combination

We introduce two operators for combining signature modules. For both of the operators, we assume that the two modules are *consistent*: One module does not include types which are internal to the other module. If this is not the case, the internal nodes can be renamed.

We begin by lifting the *merge* operation from PSSs to modules. The only change is that the parameters need to be combined: this is achieved by concatenating the internal, imported and exported parameters in the two modules, respectively. The formal definition is suppressed, but see Cohen-Sygal and Wintner (2006) for more details.

#### 9.4.1 Attachment

The novel operation we introduce in this work is *attachment*. While the merge operation is symmetric, this is not the case for attachment, which behaves as a function, where one module is the input for another. Inspired by the concept of parameters in programming languages, the exported nodes of the called

module are viewed as actual parameters which instantiate the imported nodes of the calling module, which are viewed as formal parameters. Informally, a module  $P_1$  receives as input another module  $P_2$ . The information encoded in  $P_2$  is added to  $P_1$  (as in the merge operation), but additionally, the exported parameters of  $P_2$  are assigned to the imported parameters of  $P_1$ : Each of the exported parameters of the called module is forced to coalesce with its corresponding imported parameter in the calling module, regardless of the attributes of these two parameters (i.e., whether they are indistinguishable or not).

**Definition 4** Let  $P_1 = \langle S_1, Int_1, Imp_1, Exp_1 \rangle$  and  $P_2 = \langle S_2, Int_2, Imp_2, Exp_2 \rangle$  be two consistent modules where  $S_1 = \langle Q_1, T_1, \preceq_1, Ap_1 \rangle$  and  $S_2 = \langle Q_2, T_2, \preceq_2, Ap_2 \rangle$  are node-disjoint.  $P_2$  **can be attached** to  $P_1$  if the following conditions hold:

1.  $|Imp_1| = |Exp_2|$
2. for all  $i, 1 \leq i \leq |Imp_1|$ , if  $T_1(Imp_1[i]) \downarrow$  and  $T_2(Exp_2[i]) \downarrow$ , then  $T_1(Imp_1[i]) = T_2(Exp_2[i])$
3.  $S_1$  and  $S_2$  are mergeable
4. for all  $i, j, 1 \leq i \leq |Imp_1|$  and  $1 \leq j \leq |Imp_1|$ , if  $Imp_1[i] \preceq_1^* Imp_1[j]$ , then  $Exp_2[j] \not\preceq_2^* Exp_2[i]$

The first condition requires that the number of formal parameters of the calling module be equal to the number of actual parameters in the called module. The second condition states that if two typed nodes are attached to each other, they are marked by the same type. If they are marked by two different types they cannot be coalesced. Finally, the last two conditions guarantee the antisymmetry of the subsumption relation in the resulting module: The third condition requires that the two underlying PSSs be mergeable, and the last condition requires that no subsumption cycles be created by the attachment of parameters.<sup>3</sup>

**Definition 5** Let  $P_1 = \langle S_1, Int_1, Imp_1, Exp_1 \rangle$  and  $P_2 = \langle S_2, Int_2, Imp_2, Exp_2 \rangle$  be two consistent modules where  $S_1 = \langle Q_1, T_1, \preceq_1, Ap_1 \rangle$  and  $S_2 = \langle Q_2, T_2, \preceq_2, Ap_2 \rangle$  are node-disjoint. If  $P_2$  can be attached to  $P_1$ , then the **attachment** of  $P_2$  to  $P_1$  is<sup>4</sup>  $P_1(P_2) = compact(P)$ , where  $P = \langle \langle Q, T, \preceq, Ap \rangle, Int, Imp, Exp \rangle$  is defined as follows: Let  $\equiv$  be an equivalence relation over  $Q_1 \cup Q_2$  defined by the reflexive and symmetric closure of  $\{(Imp_1[i], Exp_2[i]) \mid 1 \leq i \leq |Imp_1|\}$ . Then:

<sup>3</sup>Relaxed versions of these conditions are conceivable, but we did not find such versions useful. For example, one can require  $|Imp_1| \leq |Exp_2|$  rather than  $|Imp_1| = |Exp_2|$ ; or that  $T_1(Imp_1[i])$  and  $T_2(Exp_2[i])$  be consistent rather than equal.

<sup>4</sup>This is a simplified version; the full definition is more involved and requires some adjustment of the appropriateness arcs, to guarantee relaxed upward closure (definition 2).

- $Q = \{[q]_{\equiv} \mid q \in Q_1 \cup Q_2\}$
- $T([q]_{\equiv}) = \begin{cases} (T_1 \cup T_2)(q') & \text{there exists } q' \in [q]_{\equiv} \\ & \text{such that } (T_1 \cup T_2)(q') \downarrow \\ \text{undefined} & \text{otherwise} \end{cases}$
- $\preceq = \{([q_1]_{\equiv}, [q_2]_{\equiv}) \mid (q_1, q_2) \in \preceq_1 \cup \preceq_2\}$
- $Ap = \{([q_1]_{\equiv}, F, [q_2]_{\equiv}) \mid (q_1, F, q_2) \in Ap_1 \cup Ap_2\}$
- $Int = \{[q]_{\equiv} \mid q \in Int_1 \cup Int_2\}$
- $Imp = \{[q]_{\equiv} \mid q \in Imp_1\}$
- $Exp = \{[q]_{\equiv} \mid q \in Exp_1\}$
- the order of  $Imp$  and  $Exp$  is induced by the order of  $Imp_1$  and  $Exp_1$ , respectively.

The attachment of a parametric module  $P_2$  to a parametric module  $P_1$  is done in several stages: First the two graphs are unioned (this is a simple point-wise union of the coordinates of the graph), and all the exported nodes of  $P_2$  are identified with the imported nodes of  $P_1$ , respectively. This is achieved through the equivalence relation, ‘ $\equiv$ ’. In this way, for each imported node of  $P_1$ , all the information encoded by the corresponding exported node of  $P_2$  is added. Then, similarly to the merge operation, pairs of nodes marked by the same type and pairs of indistinguishable anonymous nodes are coalesced via the *compact* operation.

The imported and exported nodes of the resulting module are the equivalence classes of the imported and exported nodes of the calling module,  $P_1$ , respectively. The nodes of  $P_2$  which are neither internal nor exported yield external nodes in the resulting module. This asymmetric view of nodes stems from the view of  $P_1$  as the calling module and  $P_2$  as the called module.

The parametric view of modules facilitates interaction between modules in two channels: by naming or by reference. Through interaction by naming, nodes marked by the same type are coalesced. Interaction by reference is achieved when the imported nodes of the calling module are coalesced with the exported nodes of the called module. The *merge* operation allows modules to interact only through naming, whereas *attachment* facilitates both ways of interaction.

We present two examples that demonstrate the utility of module combination through attachment. Section 9.4.2 provides a toy grammar example, inspired by a linguistically-motivated type signature but necessarily small-scale. In Section 9.4.3 we emphasize the utility of module combination for grammar engineering tasks by implementing parametric lists, a concept that requires heavy machinery in alternative approaches and is natural and simple with signature modules. A large-scale example of the benefits of signature modules is outside the scope of this paper.

### 9.4.2 A simple example

Let  $P_1$  and  $P_2$  be the modules depicted in Figures 2 and 3.  $P_1$  stipulates two distinct (but anonymous) values for  $Approp(n, AGR)$  and  $Approp(v, AGR)$ .  $P_2$  stipulates two nodes, typed  $nagr$  and  $vagr$ , with the intention that these nodes be coalesced with the two anonymous nodes of  $P_1$ . Notice that all nodes in both  $P_1$  and  $P_2$  are non-internal. Let  $Imp_1 = \langle q_4, q_5 \rangle$  and let  $Exp_2 = \langle p_9, p_{10} \rangle$ .  $P_1(P_2)$  is the module depicted in Figure 4. Notice how  $q_4, q_5$  are coalesced with  $p_9, p_{10}$ , respectively, even though  $q_4, q_5$  are anonymous and  $p_9, p_{10}$  are typed and each pair of nodes has different attributes. Such unification of nodes cannot be achieved with the merge operation.

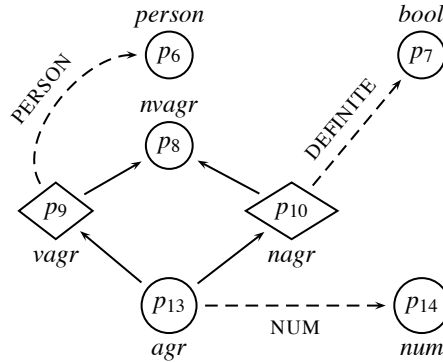


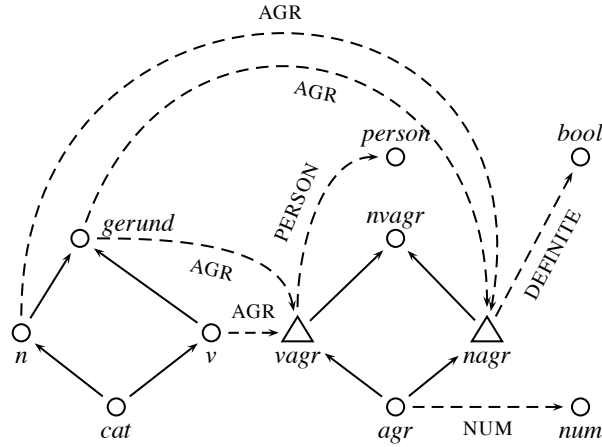
FIGURE 3 An agreement module,  $P_2$

### 9.4.3 A utility example: Parametric lists

Lists and parametric lists are extensively used in typed unification based formalisms, e.g., HPSG. The mathematical foundations for parametric lists were established by Penn (2000), resorting to infinite type signatures. To demonstrate the utility of signature modules, we show how they can be used to construct parametric lists without hampering the finiteness of the signature.

Consider Figure 5. The module *List* depicts a parametric list module. It receives as input, through the imported node  $q_3$ , a node which determines the type of the list members. The entire list can then be used through the exported node  $q_4$ . Notice that  $q_2$  is an external anonymous node. Although its intended denotation is the type  $ne\_list$ , it is anonymous in order to be unique for each copy of the list, as will be shown below. Now, if *Phrase* is a simple module consisting of one exported node, of type *phrase*, then the module obtained by  $List(phrase)$  is depicted in Figure 6.

Other modules can now use lists of phrases; for example, the module

FIGURE 4 Attachment:  $P_1(P_2)$ 

*Struct* of example 5 uses an imported node as the appropriate value for the feature *COMP-DTRS*. Via attachment, this node can be instantiated by *List(Phrase)*, as in Figure 7.

More copies of the list with other list members can be created by different calls to the module *List*. Each such call creates a unique copy of the list, potentially with different types of list elements. Uniqueness is guaranteed by the anonymity of the node  $q_2$  of *List*.

A major difference between our solution and the one of Penn (2000) is that while our solution produces only a finite number of list copies, one copy for each desired list, the solution of Penn (2000) produces a list copy for all the nodes in the signature recursively, resulting in infinite copies (and hence an infinite signature), while only a small finite number of them are indeed necessary.

## 9.5 Conclusions

We presented a complete definition of type signature modules and their interaction. Unlike existing approaches, our solution is formally defined, mathematically proven, can be easily and efficiently implemented, and meets all the desiderata listed in section 9.1.2. Modular construction of signatures is a crucial step toward (typed unification) grammar modularity and an essential requirement for the maintainability and sustainability of large scale grammars.

The examples we used are necessarily simplistic. We believe, however, that our definition of signature modules, along with the operations of *merge*

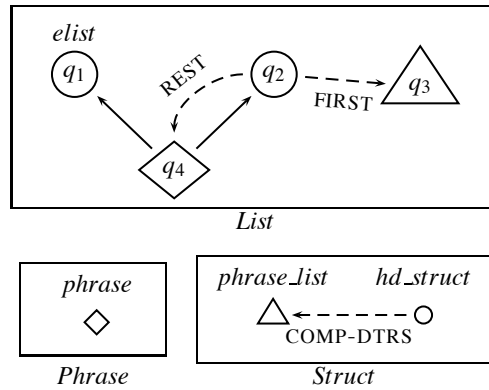
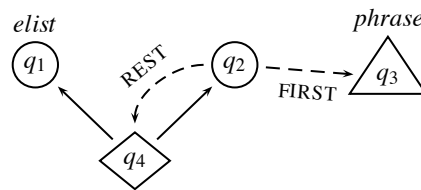


FIGURE 5 Modules defining parametric lists

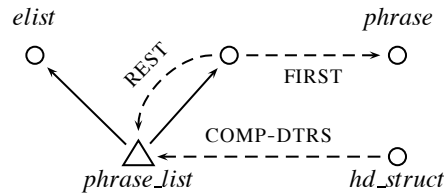
FIGURE 6  $List(Phrase)$ 

and *attachment*, provide grammar developers with powerful and flexible tools for collaborative development of large-scale grammars.

Modules provide *abstraction*; for example, the module *List* of Figure 5 defines the structure of a list, abstracting over the type of its elements. In a real-life setting, the grammar designer must determine how to abstract away certain aspects of the developed theory, thereby identifying the interaction points between the defined module and the rest of the grammar. A first step in this direction was done by Bender and Flickinger (2005) (section 9.1.3); we believe that we provide a more general, flexible and powerful framework to achieve the full goal of grammar modularization.

This work can be extended in various ways. First, the definition of modules can be extended to include also parts of the grammar, distributing the rules among several modules. The combination operators we defined for signature modules can be naturally extended to grammar modules. This reflects our observation (section 9.1.2) that most of the information in typed formalisms is encoded by the signature. We are actively pursuing this direction.

While this work is mainly theoretical, it has important practical implica-

FIGURE 7 *Struct(List(Phrase))*

tions. We would like to integrate our solutions in an existing environment for grammar development. An environment that supports modular construction of large scale grammars will greatly contribute to grammar development and will have a significant impact on practical implementations of grammatical formalisms.

Once grammar modules are fully integrated in a grammar development system, two immediate applications of modularity are conceivable. One is the development of parallel grammars for multiple languages under a single theory, as in Bender et al. (2005) or King et al. (2005). Here, a *core* module is common to all grammars, and language-specific fragments are developed as separate modules. A second application is a sequence of grammars modeling language development, e.g., language acquisition or (historical) language change. Here, a “new” grammar is obtained from a “previous” grammar; formal modeling of such operations through module composition can shed new light on the linguistic processes that take place as language develops.

### Acknowledgments

This research was supported by THE ISRAEL SCIENCE FOUNDATION (grant No. 137/06). We are grateful to the participants of the ISF Workshop on Large-scale Grammar Development and Grammar Engineering, held in Haifa, Israel in June 2006, for useful comments and feedback. Special thanks to Nurit Melnik and Gerald Penn for detailed discussions, and to the FG-2008 reviewers for useful comments. All remaining errors are, of course, our own.

### References

- Bender, Emily M. and Dan Flickinger. 2005. Rapid prototyping of scalable grammars: Towards modularity in extensions to a language-independent core. In *Proceedings of IJCNLP-05*. Jeju Island, Korea.
- Bender, Emily M., Dan Flickinger, Fredrik Fouvry, and Melanie Siegel. 2005. Shared representation in multilingual grammar engineering. *Research on Language and Computation* 3:131–138.

- Carpenter, Bob. 1992. *The Logic of Typed Feature Structures*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.
- Cohen-Sygal, Yael and Shuly Wintner. 2006. Partially specified signatures: A vehicle for grammar modularity. In *Proceedings of COLING-ACL*, pages 145–152. Sydney, Australia.
- Copestake, Ann and Dan Flickinger. 2000. An open-source grammar development environment and broad-coverage English grammar using HPSG. In *Proceedings of LREC-2000*. Athens, Greece.
- Dalrymple, Mary. 2001. *Lexical Functional Grammar*. Academic Press.
- Hinrichs, Erhard W., W. Detmar Meurers, and Shuly Wintner. 2004. Linguistic theory and grammar implementation. *Research on Language and Computation* 2:155–163.
- Keselj, Vlado. 2001. Modular HPSG. Tech. Rep. CS-2001-05, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada.
- King, Tracy Holloway, Martin Forst, Jonas Kuhn, and Miriam Butt. 2005. The feature space in parallel grammar writing. *Research on Language and Computation* 3:139–163.
- Klint, Paul, Ralf Lämmel, and Chris Verhoef. 2005. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology* 14(3):331–380.
- Moshier, Andrew M. 1997. Is HPSG featureless or unprincipled? *Linguistics and Philosophy* 20(6):669–695.
- Oepen, Stephan, Daniel Flickinger, J. Tsujii, and Hans Uszkoreit, eds. 2002. *Collaborative Language Engineering: A Case Study in Efficient Grammar-Based Processing*. Stanford: CSLI Publications.
- Pendar, Nick. 2007. Soft constraints at interfaces. In T. H. King and E. M. Bender, eds., *Proceedings of the GEAF07 Workshop*, pages 285–305. Stanford, CA: CSLI.
- Penn, Gerald B. 2000. *The algebraic structure of attributed type signatures*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.
- Pollard, Carl and Ivan A. Sag. 1994. *Head-Driven Phrase Structure Grammar*. University of Chicago Press and CSLI Publications.
- Wintner, Shuly. 2002. Modular context-free grammars. *Grammars* 5(1):41–63.

