Towards Modular Development of Typed Unification Grammars

Yael Sygal*
University of Haifa

Shuly Wintner**
University of Haifa

Development of large-scale grammars for natural languages is a complicated endeavor: Grammars are developed collaboratively by teams of linguists, computational linguists, and computer scientists, in a process very similar to the development of large-scale software. Grammars are written in grammatical formalisms that resemble very-high-level programming languages, and are thus very similar to computer programs. Yet grammar engineering is still in its infancy: Few grammar development environments support sophisticated modularized grammar development, in the form of distribution of the grammar development effort, combination of sub-grammars, separate compilation and automatic linkage, information encapsulation, and so forth.

This work provides preliminary foundations for modular construction of (typed) unification grammars for natural languages. Much of the information in such formalisms is encoded by the type signature, and we subsequently address the problem through the distribution of the signature among the different modules. We define signature modules *and provide operators of* module combination. *Modules may specify only partial information about the components of the signature and may communicate through parameters, similarly to function calls in programming languages. Our definitions are inspired by methods and techniques of programming language theory and software engineering and are motivated by the actual needs of grammar developers, obtained through a careful examination of existing grammars. We show that our definitions meet these needs by conforming to a detailed set of desiderata. We demonstrate the utility of our definitions by providing a modular design of the HPSG grammar of Pollard and Sag.*

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, and so on. Wide-coverage grammars are being developed for various languages (Abeillé, Candito, and Kinyon 2000; XTAG Research

* Department of Computer Science, University of Haifa, 31905 Haifa, Israel.
E-mail: yael.sygal@gmail.com.
** Department of Computer Science, University of Haifa, 31905 Haifa, Israel.
E-mail: shuly@cs.haifa.ac.il.

Submission received: 3 June 2009; revised submission received: 21 June 2010; accepted for publication: 14 September 2010.

Group 2001; Oepen et al. 2002; Hinrichs, Meurers, and Wintner 2004; Bender et al. 2005; King et al. 2005; Müller 2007) in several theoretical frameworks, including TAG (Joshi, Levy, and Takahashi 1975), LFG (Dalrymple 2001), HPSG (Pollard and Sag 1994), and XDG (Debusmann, Duchier, and Rossberg 2005).

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—large-scale grammars can be made up by tens of thousands of line of code (Oepen et al. 2000) and may include thousands of types (Copestake and Flickinger 2000). Modern grammars are written in grammatical formalisms that are often reminiscent of very-high-level, declarative (mostly logical) programming languages, and are thus very similar to computer programs. This raises problems similar to those encountered in large-scale software development (Erbach and Uszkoreit 1990). Although whereas software engineering provides adequate solutions for the programmer, grammar engineering is still in its infancy.

In this work we focus on typed unification grammars (TUG), and their implementation in grammar-development platforms such as LKB (Copestake 2002), ALE (Carpenter and Penn 2001), TRALE (Meurers, Penn, and Richter 2002), or Grammix (Müller 2007). Such platforms conceptually view the grammar as a single entity (even when it is distributed over several files), and provide few provisions for modular grammar development, such as mechanisms for defining modules that can interact with each other through well-defined interfaces, combination of sub-grammars, separate compilation and automatic linkage of grammars, information encapsulation, and so forth. This is the main issue that we address in this work.[1]

We provide a preliminary yet thorough and well-founded solution to the problem of grammar modularization. We first specify a set of desiderata for a beneficial solution in Section 1.1, and then survey related work, emphasizing the shortcomings of existing approaches with respect to these desiderata. Much of the information in typed unification grammars is encoded in the signature, and hence the key is facilitating a modularized development of type signatures. In Section 2 we introduce a definition of signature modules, and show how two signature modules combine and how the resulting signature module can be extended to a stand-alone type signature. We lift our definitions from signatures to full grammar modules in Section 3. In Section 4 we use signature modules and their combination operators to work out a modular design of the HPSG grammar of Pollard and Sag (1994), demonstrating the utility of signature modules for the development of linguistically motivated grammars. We then outline `MODALE`, an implementation of our solutions which supports modular development of type signatures in the context of both ALE and TRALE (Section 5). We show in Section 6 how our solution complies with the desiderata of Section 1.1, and conclude with directions for future research.

1.1 Motivation

The motivation for modular grammar development is straightforward. Like software development, large-scale grammar development is much simpler when the task can be cleanly distributed among different developers, provided that well-defined interfaces govern the interaction among modules. From a theoretical point of view, modularity

1 This article extends and revises Cohen-Sygal and Wintner (2006) and Sygal and Wintner (2008).

facilitates the definition of cleaner semantics for the underlying formalism and the construction of correctness proofs. The engineering benefits of modularity in programming languages are summarized by Mitchell (2003, page 235), and are equally valid for grammar construction:

> In an effective design, each module can be designed and tested independently. Two important goals in modularity are to allow one module to be written with little knowledge of the code in another module and to allow a module to be redesigned and re-implemented without modifying other parts of the system.

A suitable notion of modularity should support "reuse of software, abstraction mechanisms for information hiding, and import/export relationships" (Brogi et al. 1994, page 1363). Similarly, Bugliesi, Lamma, and Mello (1994, page 444) state that

> [a] modular language should allow rich forms of abstraction, parametrization, and information hiding; it should ease the development and maintenance of large programs as well as provide adequate support or reusability and separate and efficient compilation; it should finally encompass a non-trivial notion of program equivalence to make it possible to justify the replacement of equivalent components.

In the linguistic literature, however, modularity has a different flavor which has to do with the way linguistic knowledge is organized, either cognitively (Fodor 1983) or theoretically (Jackendoff 2002, pages 218–230). Although we do not directly subscribe to this notion of modularity in this work, it may be the case that an engineering-inspired definition of modules will facilitate a better understanding of the linguistic notion. Furthermore, although there is no general agreement among linguists on the exact form of grammar modularity, a good solution for grammar development must not reflect the correctness of linguistic theories but rather provide the computational framework for their implementation.

To consolidate the two notions of modularity, and to devise a solution that is on one hand inspired by developments in programming languages and on the other useful for linguists, a clear understanding of the actual needs of grammar developers is crucial. A first step in this direction was done by Erbach and Uszkoreit (1990). In a similar vein, we carefully explored two existing grammars: the LINGO grammar matrix (Bender, Flickinger, and Oepen 2002),[2] which is a framework for rapid development of cross-linguistically consistent grammars; and a grammar of a fragment of modern Hebrew, focusing on inverted constructions (Melnik 2006). These grammars were chosen since they are comprehensive enough to reflect the kind of data large-scale grammars encode, but are not too large to encumber this process.

Inspired by established criteria for modularity in programming languages, and motivated by our observation of actual grammars, we define the following desiderata for a beneficial solution for (typed unification) grammar modularization:

Signature focus: Much of the information in typed formalisms is encoded by the signature. This includes the type hierarchy, the appropriateness specification, and the

2 The LINGO grammar matrix is not a grammar per se, but rather a framework for grammar development for several languages. We focused on its core grammar and several of the resulting, language-specific grammars.

type constraints. Hence, modularization must be carried out mainly through the distribution of the signature between the different modules.[3]

Partiality: Modules should provide means for specifying *partial* information about the components of a grammar: both the grammar itself and the signature over which it is defined.

Extensibility: Although 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.

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. 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 all the information encoded in each of the modules and the information resulting from the combination operation. Additional information must only be added if it is essential to render the module well-defined.

Associativity: Module combination must be associative and commutative: The order in which modules are combined must not affect the result. However, this desideratum is not absolute—it is restricted to cases where the combination formulates a simple union of data. In other cases, associativity and commutativity should be considered with respect to the benefit the system may enjoy if they are abandoned.

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

The solution we advocate here satisfies all these requirements. It facilitates collaborative development of grammars, where several applications of modularity are conceivable:

- A single large-scale grammar developed by a team.
- Development of parallel grammars for multiple languages under a single theory, as in Bender et al. (2005), King et al. (2005), or Müller (2007). Here, a *core* module is common to all grammars, and language-specific fragments are developed as separate modules.
- A sequence of grammars modeling language development, for example language acquisition or (historical) language change (Wintner, Lavie, and MacWhinney 2009). 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.

3 We restrict ourselves to standard type signatures (as defined by Carpenter [1992] and Penn [2000]), ignoring type constraints which are becoming common in practical systems. We defer an extension of our results to type constraints to future work.

1.2 Related Work

1.2.1 Modularity in Programming Languages. Vast literature addresses modularity in programming languages, and a comprehensive survey is beyond the scope of this work. As unification grammars are in many ways very similar to logic programming languages, our desiderata and solutions are inspired by works in this paradigm.

Modular interfaces of logic programs were first suggested by O'keefe (1985) and Gaifman and Shapiro (1989). Combination operators that were proved suitable for Prolog include the algebraic operators $\oplus$ and $\otimes$ of Mancarella and Pedreschi (1988); the union and intersection operators of Brogi et al. (1990); the closure operator of Brogi, Lamma, and Mello (1993); and the set of four operators (encapsulation, union, intersection, and import) defined by Brogi and Turini (1995). For a comprehensive survey, see Bugliesi, Lamma, and Mello (1994).

The 'merge' operator that we present in Section 2.4.2 is closely related to union operations proposed for logic programming languages. We define no counterpart of intersection-type operations, although such operations are indeed conceivable. Our 'attachment' operation is more in line with Gaifman and Shapiro (1989).

1.2.2 Initial Approaches: Modularized Parsing. Early attempts to address modularity in linguistic formalisms share a significant disadvantage: The modularization is of the parsing process rather than the grammar.

Kasper and Krieger (1996) describe a technique for dividing a unification-based grammar into two components, roughly along the syntax/semantics axis. Their motivation is efficiency; observing that syntax usually imposes constraints on permissible structures, and semantics usually mostly adds structure, they propose to parse with the syntactic constraints first, and apply the semantics later. This is achieved by recursively deleting the syntactic and semantic information (under their corresponding attributes in the rules and the lexicon) for the semantic and syntactic parsers, respectively. This proposal requires that a single grammar be given, from which the two components can be derived. A more significant disadvantage of this method is that coreferences between syntax and semantics are lost during this division (because reentrancies that represent the connection between the syntax and the semantics are removed). Kasper and Krieger observe that the intersection of the languages generated by the two grammars does not yield the language of the original grammar.

Zajac and Amtrup (2000) present an implementation of a pipeline-like composition operator that enables the grammar designer to break a grammar into sub-grammars that are applied in a sequential manner at run-time. Such an organization is especially useful for dividing the development process into stages that correspond to morphological processing, syntax, semantics, and so on. The notion of composition here is such that sub-grammar G_{i+1} operates on the output of sub-grammar G_i; such an organization might not be suitable for all grammar development frameworks. A similar idea is proposed by Basili, Pazienza, and Zanzotto (2000); it is an approach to parsing that divides the task into sub-tasks, whereby a module component P_i takes an input sentence at a given state of analysis S_i and augments this information in S_{i+1} using a knowledge base K_i. Here, too, it is the processing system, rather than the grammar, which is modularized in a pipeline fashion.

1.2.3 Modularity in Typed Unification Grammars. Keselj (2001) presents a modular Head-driven Phrase Structure Grammar (HPSG), where each module is an ordinary HPSG grammar, including an ordinary type signature, but each of the sets FEAT, TYPE, and RULES is divided into two disjoint sets of private and public elements. The public

sets consist of those elements that can communicate with elements from corresponding sets in other modules, and private elements are those that are internal to the module. Merging two modules is then defined by set union; in particular, the type hierarchies are merged by unioning the two sets of types and taking the transitive closure of the union of the two BCPOs (see Definition 2). The success of the merge of two modules requires that the union of the two BCPOs be a BCPO.

While this work is the first to concretely define signature modules, it provides a highly insufficient mechanism for supporting modular grammar development: The requirement that each module include a complete type hierarchy imposes strong limitations on the kind of information that modules can specify. It is virtually impossible to specify partial information that is consistent with the complete type hierarchy requirement. Furthermore, module composition becomes order-dependent as we show in Example 8 (Section 2.4.2). Finally, the only channel of interaction between modules is the names of the types. Our work is similar in spirit to Keselj (2001), but it overcomes these shortcomings and complies with the desiderata of Section 1.1.

Kaplan, King, and Maxwell (2002) introduce a system designed for building a grammar by both extending and restricting another grammar. An LFG grammar is presented to the system in a priority-ordered sequence of files containing phrase-structure rules, lexical entries, abbreviatory macros and templates, feature declarations, and finite-state transducers for tokenization and morphological analysis. The grammar can include only one definition of an item of a given type with a particular name (e.g., there can be only one NP rule, potentially with many alternative expansions), and items in a file with higher priority override lower priority items of the same type with the same name. The override convention makes it possible to add, delete, or modify rules. However, when a rule is modified, the entire rule has to be rewritten, even if the modifications are minor. Moreover, there is no real concept of modularization in this approach because the only interaction among files is overriding of information.

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. This framework addresses modularity in the sense that the entire grammar is distributed between several fragments that can be combined in different ways, according to the user's choice. However, the notion of modularity is rather different here, as modules are pre-written pieces of code which the grammar designer does not develop and whose interaction he or she has little control over.

1.2.4 Modularity in Related Formalisms. The previously mentioned works emphasize the fact that existing approaches to modular grammar development in the area of unification grammars are still insufficient. The same problem has also been addressed in some

other, related, formalisms; we now survey such works and discuss the applicability of the proposed solutions to the problem of modularity in typed unification grammars.

Wintner (2002) defines the concept of modules for CFGs: The set of nonterminals is partitioned into three disjoint classes of **internal**, **exported**, and **imported** elements. The imported elements are those that are supplied to the module by other modules, the exported elements are those it provides to the outside world, and the internal ones are local to it. Two modules can be combined only if the set of internal elements of each module is disjoint from the exported and imported sets of the other module as well as if the exported sets are disjoint. Then the combination of two modules is done by simple measures of set union. This is the infrastructure underlying the definition of modular HPSG discussed earlier (Keselj 2001).

Provisions for modularity have also been discussed in the context of tree-adjoining grammars (TAG) (Joshi, Levy, and Takahashi 1975). A wide-coverage TAG may contain hundreds or even thousands of elementary trees, and syntactic structure can be redundantly repeated in many of them (XTAG Research Group 2001; Abeillé, Candito, and Kinyon 2000). Consequently, maintenance and extension of such grammars is a complex task. To address these issues, several high-level formalisms were developed (Vijay-Shanker 1992; Candito 1996; Duchier and Gardent 1999; Kallmeyer 2001). These formalisms take the **metagrammar approach**, where the basic units are tree *descriptions* (i.e., formulas denoting sets of trees) rather than trees. Tree descriptions are constructed by a tree logic and combined through conjunction or inheritance; a module in this approach is merely a tree description, and modules are combined by means of the control logic. When trees are semantic objects, (i.e., the denotation of tree descriptions), there can be various ways to refer to nodes in the trees in order to control the possible combination of grammar modules. Several mechanisms have been suggested to facilitate reference across modules (Candito 1996; Perrier 2000; Crabbé and Duchier 2004; Kahane 2006).

The solution that we propose here embraces the idea of moving from concrete objects (e.g., a concrete type signature) to descriptions thereof; but we take special care to do so in a way that maintains the associativity of the main grammar combination operator, in contrast to some earlier approaches (Sygal and Wintner 2009).

Debusmann, Duchier, and Rossberg (2005) introduce Extensible Dependency Grammar (XDG), which is a general framework for dependency grammars that supports modular grammar design. An XDG grammar consists of **dimensions**, **principles**, and a lexicon; it characterizes a set of well-formed analyses. Each dimension is an attributed labeled graph, and when a grammar consists of multiple dimensions (e.g., multigraphs), they share the same set of nodes. A lexicon for a dimension is a set of total assignments of nodes and labels. The main mechanism XDG uses to control analyses are principles, that can be either **local** (imposing a constraint on the possible analysis of a specific dimension) or **multi-dimensional** (constraining the analysis of several dimensions with respect to each other). In XDG, principles are formulated using a type-system that includes several kinds of elementary types (e.g., nodes, edges, graphs, and even multigraphs) and complex types that are constructed incrementally over the elementary types. Then, parameters range over types to formulate parametric principles. A feasible XDG analysis amounts to a labeled graph in which each dimension is a subgraph, such that all (parametric) principles are maintained (this may require nodes in different subgraphs to be identified). XDG supports modular grammar design where each dimension graph is a grammar module, and module interaction is governed through multi-dimensional parametric principles.

This work emphasizes the importance of types as a mechanism for modularity. Our work shares with XDG the use of graphs as the basic components and the use

of parameters to enforce interaction among modules. In both works, each module introduces constraints on the type system and interaction among modules through parameters is used to construct a multigraph in which some of the nodes are identified. In our approach, however, the type system is part of the grammar specification, and modules are combined via explicit combination operations. In contrast, in XDG the type mechanism is used externally, to describe objects, and a general description logic is used to impose constraints. Another major difference has to do with expressive power: Whereas unification grammars are Turing-equivalent, XDG is probably mildly context-sensitive (Debusmann 2006).

The **grammar formalism** (GF) (Ranta 2007) is a typed functional programming language designed for multilingual grammars. Ranta introduces a module system for GF where a module can be either one of three kinds: **abstract**, **concrete**, or a **resource** module. Each of them reflects the kind of data this module may include. A module of type abstract includes abstract syntax trees which represent grammatical information, such as semantic or syntactic data. A module of type concrete includes relations between trees in the abstract module and relations between strings in the target language. Communication between modules of these two types is carried out through inheritance hierarchies similarly to object-oriented programs. Resource modules are a means for code-sharing, independently of the hierarchies. The system of modules supports development of multilingual grammars through replacement of certain modules with others. A given grammar can also be extended by adding new modules. Additionally, to avoid repetition of code with minor variations, GF allows the grammar writer to define operations which produce new elements.

GF is purposely designed for multilingual grammars which share a core representation, and individual extensions to different languages are developed independently. As such, the theoretical framework it provides is tailored for such needs, but is lacking where general purpose modular applications are considered (see section 1.1 for examples of such conceivable applications). Mainly, GF forces the developer to pre-decide on the relations between *all* modules (through the concrete module and inheritance hierarchies), whereas in an ideal solution the interaction between all modules should be left to the development process. Each module should be able to independently declare its own interface with other modules; then, when modules combine they may do so in any way that is consistent with the interfaces of other modules. Furthermore, reference to mutual elements in GF is carried out only through naming, again resulting in a weak interface for module interaction. Finally, the operations that the grammar writer can define in GF are macros, rather than functions, as they are expanded by textual replacement.

2. Modularization of the Signature

2.1 Typed Signatures

We assume familiarity with theories of (typed) unification grammar, as formulated by, for example, Carpenter (1992) and Penn (2000). The definitions in this section set the notation and recall basic notions. For a partial function F, '$F(x)\downarrow$' ('$F(x)\uparrow$') means that F is defined (undefined) for the value x; '$F(x) = F(y)$' means that either F is defined both for x and for y and assigns them equal values or it is undefined for both.

Definition 1
Given a partially ordered set $\langle P, \leq \rangle$, the set of **upper bounds** of a subset $S \subseteq P$ is the set $S^u = \{y \in P \mid \forall x \in S \;\; x \leq y\}$.

For a given partially ordered set $\langle P, \leq \rangle$, if $S \subseteq P$ has a least element then it is unique, and is denoted $min(S)$.

Definition 2
A partially ordered set $\langle P, \leq \rangle$ is a **bounded complete partial order** (BCPO) iff for every $S \subseteq P$ such that $S^u \neq \emptyset$, S^u has a least element, called a **least upper bound (lub)** and denoted $\bigsqcup S$.

Definition 3
A **type hierarchy** is a non-empty, finite, bounded complete partial order $\langle \textsc{Type}, \sqsubseteq \rangle$.

Every type hierarchy $\langle \textsc{Type}, \sqsubseteq \rangle$ always has a least type (written $\bot$), because the subset $S = \emptyset$ of $\textsc{Type}$ has the non-empty set of upper bounds, $S^u = \textsc{Type}$, which must have a least element due to bounded completeness.

Definition 4
Let $\langle \textsc{Type}, \sqsubseteq \rangle$ be a type hierarchy and let $x, y \in \textsc{Type}$. If $x \sqsubseteq y$, then x is a **supertype** of y and y is a **subtype** of x. If $x \sqsubseteq y$, $x \neq y$ and there is no z such that $x \sqsubseteq z \sqsubseteq y$ and $z \neq x, y$ then x is an **immediate supertype** of y and y is an **immediate subtype** of x.

We follow the definitions of Carpenter (1992) and Penn (2000) in viewing subtypes as greater than their supertypes (hence the least element $\bot$ and the notion of lub), rather than the other way round (inducing a glb interpretation), which is sometimes common in the literature (Copestake 2002).

Definition 5
Given a type hierarchy $\langle \textsc{Type}, \sqsubseteq \rangle$ and a finite set of features $\textsc{Feat}$, an **appropriateness specification** is a partial function, $Approp : \textsc{Type} \times \textsc{Feat} \rightarrow \textsc{Type}$ such that for every $F \in \textsc{Feat}$:

1. (Feature Introduction) there is a type $Intro(F) \in \textsc{Type}$ such that:
   - $Approp(Intro(F), F)\downarrow$, and
   - for every $t \in \textsc{Type}$, if $Approp(t, F)\downarrow$, then $Intro(F) \sqsubseteq t$, and
2. (Upward Closure / Right Monotonocy) if $Approp(s, F)\downarrow$ and $s \sqsubseteq t$, then $Approp(t, F)\downarrow$ and $Approp(s, F) \sqsubseteq Approp(t, F)$.

Definition 6
A **type signature** is a structure $\langle \textsc{Type}, \sqsubseteq, \textsc{Feat}, Approp \rangle$, where $\langle \textsc{Type}, \sqsubseteq \rangle$ is a type hierarchy, $\textsc{Feat}$ is a finite set of features, $\textsc{Feat}$ and $\textsc{Type}$ are disjoint, and $Approp$ is an appropriateness specification.

Again, note that type constraints are not addressed in this work.

2.2 Overview

We define **signature modules** (also referred to as **modules** herein), which are structures that provide a framework for modular development of type signatures. These structures follow two guidelines.

1. Signature modules contain partial information about a signature: part of the subtyping relation (sometimes referred to in the literature as *type subsumption*) and part of the appropriateness specification. The key here is a move from concrete type signatures to descriptions thereof; rather than specify types, a description is a graph whose nodes denote types and whose arcs denote elements of the subtyping and appropriateness relations of signatures.
2. Modules may choose which information to expose to other modules and how other modules may use the information they encode. The denotation of nodes is extended by viewing them as *parameters*: Similarly to parameters in programming languages, these are entities through which information can be imported to or exported from other modules. This is done similarly to the way parametric principles are used by Debusmann, Duchier, and Rossberg (2005).

We begin by defining the basic structure of signature modules in Section 2.3. We then introduce (Section 2.4) two combination operators for signature modules which facilitate interaction and (remote) reference among modules. We end this section by showing how to extend a signature module into a bona fide type signature (Section 2.5).

2.3 Signature Modules

The definition of a signature module is conceptually divided into two levels of information. The first includes all the genuine information that may be encoded by a signature, such as subtyping and appropriateness relations, types, and so forth. The second level includes the parametric casting of nodes. This casting is not part of the core of a signature, but rather a device that enables advanced module communication. Consequently, we define **signature modules** in two steps. First, we define **partially specified signatures (PSSs)**, which are finite directed graphs that encode partial information about the signature. Then, we extend PSSs to *signature modules* which are structures, based on PSSs, that provide also a complete mechanism for module interaction and (remote) reference.

We assume enumerable, disjoint sets TYPE of types, FEAT of features, and NODES of nodes, over which signatures are defined.

Definition 7
A **partially labeled graph (PLG)** over TYPE and FEAT is a finite, directed labeled graph $P = \langle Q, T, \preceq, Ap \rangle$, where:

1. $Q \subset$ NODES is a finite, nonempty set of nodes.
2. $T : Q \rightarrow$ TYPE is a partial function, marking some of the nodes with types.

3. $\preceq \subseteq Q \times Q$ is a relation specifying (immediate) subtyping.
4. $Ap \subseteq Q \times \textsc{Feat} \times Q$ is a relation specifying appropriateness.

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

5. T is one to one.
6. '$\preceq$' is antireflexive; its reflexive-transitive closure, denoted '$\overset{*}{\preceq}$', is antisymmetric.
7. (Relaxed Upward Closure) for all $q_1, q_1', q_2 \in Q$ and $F \in \textsc{Feat}$, if $(q_1, F, q_2) \in Ap$ and $q_1 \overset{*}{\preceq} q_1'$, then there exists $q_2' \in Q$ such that $q_2 \overset{*}{\preceq} q_2'$ and $(q_1', F, q_2') \in Ap$

A PSS is a finite, directed graph whose nodes denote types and whose edges denote the subtyping 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 module. T is one-to-one (item 5), because we require that two marked nodes denote different types.

The '$\preceq$' relation (item 3) specifies an immediate subtyping order over the nodes, with the intention that this order hold later for the types denoted by nodes. This is why '$\overset{*}{\preceq}$' is required to be a partial order (item 6). The type hierarchy of an ordinary type signature is required to be a BCPO, but current approaches (Copestake 2002) relax this requirement to allow more flexibility in grammar design. Similarly, the type hierarchy of PSSs is partially ordered but this order is not necessarily a bounded complete one. Only after all modules are combined is the resulting subtyping relation extended to a BCPO (see Section 2.5); any intermediate result can be a general partial order. Relaxing the BCPO requirement also helps guarantee the associativity of module combination (see Example 8).[4]

Consider now the appropriateness relation. In contrast to type signatures, 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 (item 4). 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 $Ap(q, F, q')$. This relaxation reflects our initial motivation of supporting partiality in modular grammar development, since different modules may specify different appropriate values according to their needs and available information. After all modules are combined, all the specified values are replaced by a single appropriate value, their *lub* (see Section 2.5). In this way, each module may specify its own appropriate values without needing to know the value specification of other modules. We do restrict the Ap relation, however, by a relaxed version of upward closure (item 7). Finally, the feature introduction condition of type signatures (Definition 5, item 1) is not enforced by signature modules. This, again,

4 The fact that the subtyping relation is only extended to a BCPO *after* all modules are combined implies a lack of incrementality in the system that may be problematic for grammar developers, as modules cannot be tested and evaluated independently. This situation, however, is not unlike the scenario of programming languages, where modules can typically be developed and compiled, but not tested, independently of a complete system.

results in more flexibility for the grammar designer; the condition can be restored, if it is desirable, after all modules combine (see Section 2.5).

Example 1
A simple PSS P_1 is depicted in Figure 1, where solid arrows represent the '$\preceq$' (subtyping) relation and dashed arrows, labeled by features, the *Ap* relation. P_1 stipulates two subtypes of *cat*, *n* and *v*, with a common subtype, *gerund*. The feature AGR is appropriate for all three categories, with distinct (but anonymous) values for $Approp(n, \text{AGR})$ and $Approp(v, \text{AGR})$. $Approp(gerund, \text{AGR})$ 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 P_1, '$\preceq$' is not a BCPO, *Ap* is not a function, and the feature introduction condition does not hold.

Definition 8
A **pre-signature module** over TYPE and FEAT is a structure $S = \langle P, Int, Imp, Exp \rangle$ where $P = \langle Q, T, \preceq, Ap \rangle$ is a PLG and:

1. $Int \subseteq Q$ is a set of **internal** types
2. $Imp \subseteq Q$ is an ordered set of **imported** parameters
3. $Exp \subseteq Q$ is an ordered set of **exported** parameters
4. $Int \cap Imp = Int \cap Exp = \emptyset$
5. for all $q \in Q$ such that $q \in Int$, $T(q)\downarrow$

We refer to elements of (the sequences) *Imp* and *Exp* using indices, with the notation $Imp[i], Exp[j]$, respectively.

A **signature module** over TYPE and FEAT is a pre-signature module $S = \langle P, Int, Imp, Exp \rangle$ in which P is a PSS.

Signature modules extend the denotation of nodes by viewing them as parameters: Similarly to parameters in programming languages, parameters are entities through which information can be imported from or exported to other modules. The nodes of a signature module 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

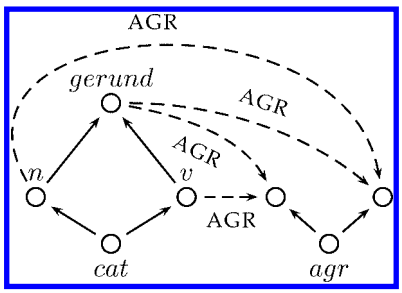

Figure 1
A partially specified signature, P_1.

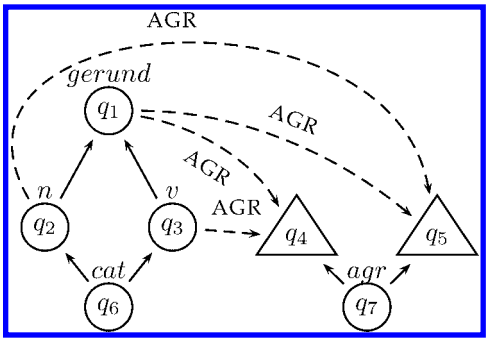

Figure 2
A signature module, S_1.

can be simultaneously imported and exported. A node which does not belong to any of the sets is called **external**. All nodes denote types, 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. External nodes differ from imported and exported nodes in the way they may interact with other modules, and provide a mechanism similar to global variables in programming languages. Because anonymous nodes facilitate reference, in one module, to information encoded in another module, such nodes cannot be internal. The imported and exported nodes are ordered in order to control the assignment of parameters when two modules are combined, as will be shown subsequently.[5] In the examples, the classification of nodes is encoded graphically as follows:

□ △ ◇ ○
Internal Imported Exported External

Example 2
Figure 2 depicts a module S_1, based on the PSS of Figure 1. $S_1 = \langle P_1, Int_1, Imp_1, Exp_1 \rangle$, where P_1 is the PSS of Figure 1, $Int_1 = \emptyset$, $Imp_1 = \{q_4, q_5\}$, and $Exp_1 = \emptyset$.

Herein, the meta-variable q (with or without subscripts) ranges over nodes, S (with or without subscripts) – over (pre-)signature modules, P (with or without subscripts) over PLGs and PSSs, and $Q, T, \preceq, Ap$ (with the same subscripts) over their constituents.

2.4 Combination Operators for Signature Modules

We introduce two operators for combining signature modules. The first operator, **merge**, is a symmetric operation which simply combines the information encoded in the two

5 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 13 and 14.

modules. The second operator, **attachment**, is a non-symmetric operation which uses the concept of parameters and is inspired by function composition. A signature module is viewed as a function whose input is a graph with a list of designated imported nodes and whose output is a graph with a list of designated exported nodes. When two signature modules are attached, similarly to function composition, the exported nodes of the second module instantiate the imported parameters of the first module. Additionally, the information encoded by the second graph is added to the information encoded by the first one.

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 parameters of the calling module are coalesced with the exported nodes of the called module, respectively. The merge operation allows modules to interact only through naming, whereas attachment facilitates both ways of interaction.

For both of the operators, we assume that the two signature modules are **consistent**: One module does not include types which are internal to the other module and the two signature modules have no common nodes. If this is not the case, nodes, and in particular internal nodes, can be renamed without affecting the operation.

Definition 9
Let $S_1 = \langle\langle Q_1, T_1, \preceq_1, Ap_1\rangle, Int_1, Imp_1, Exp_1\rangle$, $S_2 = \langle\langle Q_2, T_2, \preceq_2, Ap_2\rangle, Int_2, Imp_2, Exp_2\rangle$ be two pre-signature modules. S_1 and S_2 are **consistent** iff all the following conditions hold:

1. $\{T_1(q) \mid q \in Int_1\} \cap \{T_2(q) \mid q \in Q_2 \text{ and } T_2(q)\downarrow\} = \emptyset$
2. $\{T_2(q) \mid q \in Int_2\} \cap \{T_1(q) \mid q \in Q_1 \text{ and } T_1(q)\downarrow\} = \emptyset$
3. $Q_1 \cap Q_2 = \emptyset$

We begin by introducing the **compactness** algorithm which is used when two modules are combined as a mechanism to coalesce corresponding nodes in the two modules.

2.4.1 Compactness. When two modules are combined, a crucial step in the combination is the identification of corresponding nodes in the two modules that should be coalesced. Such pairs of nodes can be either of two kinds:

1. Two typed nodes which are labeled by the same type should be coalesced (along with their attributes).
2. Two anonymous nodes which are **indistinguishable**, that is, have **isomorphic** environments, should be coalesced. The environment of a node q is the subgraph that includes all the reachable nodes via any kind of arc (from q or to q) up to and including a typed node. The intuition is that if two anonymous nodes have isomorphic environments, then they cannot be distinguished and therefore should coincide. 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 ensure the associativity of module combination).

Additionally, during the combination of modules, some arcs may become redundant (such arcs are not prohibited by the definition of a module). Redundant arcs can be of two kinds:

1. A subtyping 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).
2. An appropriateness arc (q_1, F, q_2) is redundant if there exists $q_3 \in Q$ such that $q_2 \stackrel{*}{\preceq} q_3$ and $(q_1, F, q_3) \in Ap$. (q_1, F, q_2) is redundant due to the lub intention of appropriateness arcs: The eventual value of $Approp(T(q_1), F)$ will be an upper bound of (at least) both q_2 and q_3. Because $q_2 \stackrel{*}{\preceq} q_3$, (q_1, F, q_2) is redundant.

Redundant arcs encode information that can be inferred from other arcs and therefore may be removed without affecting the data encoded by the signature module.

While our main interest is in signature modules, the compactness algorithm is defined over the more general case of pre-signature modules. This more general notion will be helpful in the definition of module combination. Informally, when a pre-signature module is compacted, redundant arcs are removed, nodes marked by the same type are coalesced, and anonymous indistinguishable nodes are identified. Additionally, the parameters and arities are induced from those of the input pre-signature module. All parameters may be coalesced with each other, as long as they are otherwise indistinguishable. If (at least) one of the coalesced nodes is an internal node, then the result is an internal node. Otherwise, if one of the nodes is imported then the resulting parameter is imported as well. Similarly, if one of the nodes is exported then the resulting parameter is exported. Notice that in the case of signature modules, because T is one to one, an internal node may be coalesced only with other internal nodes.

The actual definitions of indistinguishability and the compactness algorithm are mostly technical and are therefore deferred to the Appendix. We do provide two simple examples to illustrate the general idea.

Example 3
Consider the signature module of Figure 3. (q_1, q_4) is a redundant subtyping arc because even without this arc, there is a subtyping path from q_1 to q_4. (q_1, F, q_3) is a redundant appropriateness arc: Eventually the appropriate value of q_1 and F should be the lub of q_3 and q_5, but since q_5 is a subtype of q_3, it is sufficient to require that it be at least q_5.

Example 4
Consider S_2, the pre-signature module depicted in Figure 4. Note that S_2 is not a signature module (because it includes two nodes labeled by a) and that compactness is defined over pre-signature modules rather than signature modules as this is the case for which it will be used during combination. In $compact(S_2)$, q_1 and q_2 are coalesced because they are both marked by the type a. Additionally, q_3 and q_6 are coalesced with q_4 and q_7, respectively, because these are two pairs of anonymous nodes with isomorphic environments. q_5 is not coalesced with q_3 and q_4 because q_5 is typed and q_3 and q_4 are not, even though they are otherwise indistinguishable. q_8 is not coalesced with q_6 and q_7 because they are distinguishable: q_8 has a supertype marked by a whereas q_6 and q_7 have anonymous supertypes.

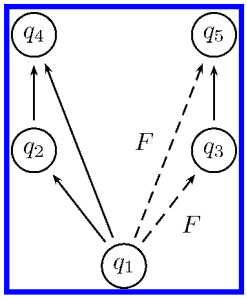

Figure 3
A signature module with redundant arcs.

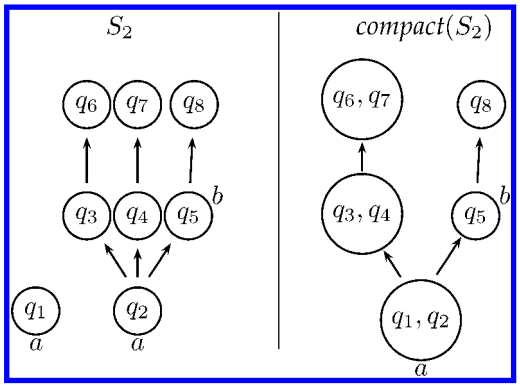

Figure 4
Compactness.

2.4.2 Merge. The merge operation combines the information encoded by two signature modules: Nodes that are marked by the same type are coalesced along with their attributes. Nodes that are marked by different types cannot be coalesced and must denote different types. The main complication arises when two *anonymous* nodes are considered—such nodes are coalesced only if they are indistinguishable.

The merge of two modules is defined in several stages: First, the two graphs are unioned (this is a simple pointwise union of the coordinates of the graph, see Definition 10). Then, the resulting graph is compacted, coalescing nodes marked by the same type as well as indistinguishable anonymous nodes. However, the resulting graph does not necessarily maintain the relaxed upward closure condition, and therefore some modifications are needed. This is done by *Ap-Closure* (see Definition 11). Finally, the addition of appropriateness arcs may turn two anonymous distinguishable nodes into indistinguishable ones and may also add redundant arcs, therefore another compactness step is needed (Definition 12).

Definition 10
Let $S_1 = \langle\langle Q_1, T_1, \preceq_1, Ap_1\rangle, Int_1, Imp_1, Exp_1\rangle$, $S_2 = \langle\langle Q_2, T_2, \preceq_2, Ap_2\rangle, Int_2, Imp_2, Exp_2\rangle$ be two consistent pre-signature modules. The **union** of S_1 and S_2, denoted $S_1 \cup S_2$, is the pre-signature module

$$S = \langle\langle Q_1 \cup Q_2, T_1 \cup T_2, \preceq_1 \cup \preceq_2, Ap_1 \cup Ap_2\rangle, Int_1 \cup Int_2, Imp_1 \cdot Imp_2, Exp_1 \cdot Exp_2\rangle$$

(where '$\cdot$' is the concatenation operator).

Definition 11
Let $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ be a pre-signature module. The **Ap-Closure** of S, denoted $ApCl(S)$, is the pre-signature module $\langle\langle Q, T, \preceq, Ap'\rangle, Int, Imp, Exp\rangle$ where

$$Ap' = \{(q_1, F, q_2) \mid q_1, q_2 \in Q$$

and there exists $q_1' \in Q$ such that

$$q_1' \stackrel{*}{\preceq} q_1 \text{ and } (q_1', F, q_2) \in Ap\}$$

Ap-Closure adds to a pre-signature module the required arcs for it to maintain the relaxed upward closure condition: Arcs are added to create the relations between elements separated between the two modules and related by mutual elements. Notice that $Ap \subseteq Ap'$ by choosing $q_1' = q_1$.

Two signature modules can be merged only if the resulting subtyping relation is indeed a partial order, where the only obstacle can be the antisymmetry of the resulting relation. The combination of the appropriateness relations, in contrast, cannot cause the merge operation to fail because any violation of the appropriateness conditions in signature modules can be deterministically resolved. Note that our specification language does not support inequations; there is no way to specify that two nodes must *not* be identified with each other. Such extensions are indeed possible, but are beyond the scope of this work.

Definition 12
Let $S_1 = \langle\langle Q_1, T_1, \preceq_1, Ap_1\rangle, Int_1, Imp_1, Exp_1\rangle$, $S_2 = \langle\langle Q_2, T_2, \preceq_2, Ap_2\rangle, Int_2, Imp_2, Exp_2\rangle$ be two consistent signature modules. S_1, S_2 are **mergeable** if there are no $q_1, q_2 \in Q_1$ and $q_3, q_4 \in Q_2$ such that the following hold:

1. $q_1 \neq q_2$ and $q_3 \neq q_4$
2. $T_1(q_1)\downarrow$, $T_1(q_2)\downarrow$, $T_2(q_3)\downarrow$ and $T_2(q_4)\downarrow$
3. $T_1(q_1) = T_2(q_4)$ and $T_1(q_2) = T_2(q_3)$
4. $q_1 \stackrel{*}{\preceq}_1 q_2$ and $q_3 \stackrel{*}{\preceq}_2 q_4$

If S_1 and S_2 are mergeable, then their **merge**, denoted $S_1 \Cup S_2$, is:

$$compact(ApCl(compact(S_1 \cup S_2)))$$

In the merged module, pairs of nodes marked by the same type and pairs of indistinguishable anonymous nodes are coalesced. An anonymous node cannot be coalesced with a typed node, even if they are otherwise indistinguishable, because that would result in a non-associative combination operation. Anonymous nodes are assigned types only after all modules combine (see Section 2.5.1).

If a node has multiple outgoing *Ap*-arcs labeled with the same feature, these arcs are not replaced by a single arc, even if the *lub* of the target nodes exists in the resulting signature module. Again, this is done to guarantee the associativity of the merge operation (see Example 9).

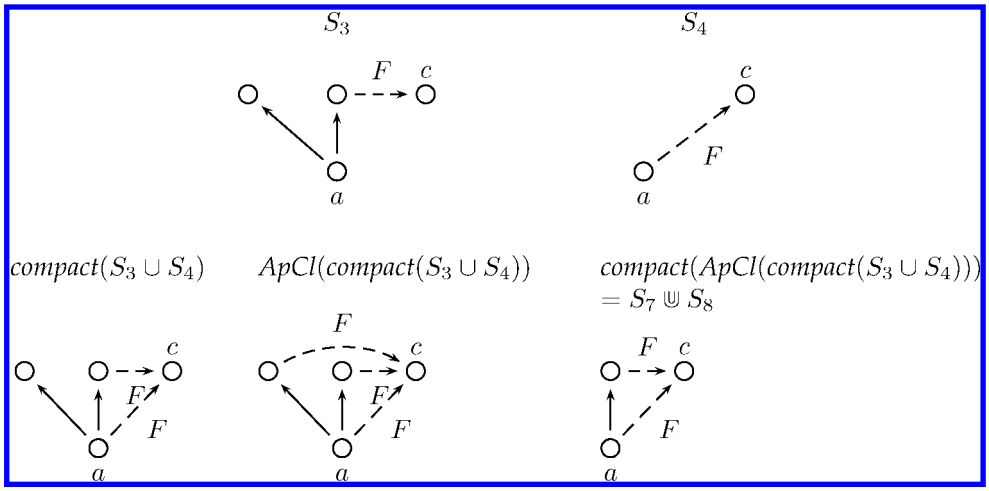

Figure 5
Merge: intermediate steps.

Example 5
Let S_3 and S_4 be the signature modules depicted in Figure 5. $S_3 \Cup S_4$ and the intermediate pre-signature modules are also shown in this figure. First, S_3 and S_4 are unioned. Then, in $compact(S_3 \cup S_4)$ the two nodes typed by *a* are coalesced, as are the nodes typed by *c*. Notice that this pre-signature module is not a signature module because it does not maintain the relaxed upward closure condition. To enforce this condition appropriateness arcs are added to yield $ApCl(compact(S_3 \cup S_4))$, but this signature module includes indistinguishable anonymous nodes and therefore another compactness operation is required to yield the final result.

Example 6
Figure 6 depicts a naive agreement module, S_5. Combined with S_1 of Figure 1, $S_1 \Cup S_5 = S_5 \Cup S_1 = S_6$. All dashed arrows are labeled AGR, but these labels are suppressed for readability.

In what follows, by standard convention, *Ap* arcs that can be inferred by upward closure are not depicted.

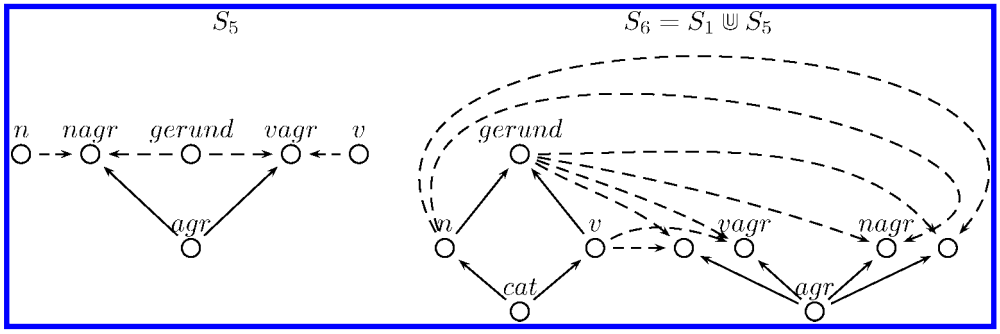

Figure 6
Merge.

Example 7
Let S_7 and S_8 be the signature modules depicted in Figure 7. S_7 includes general agreement information and S_8 specifies detailed values for several specific properties. Then, $S_7 \Cup S_8 = S_8 \Cup S_7 = S_9$. In this way, the high level organization of the agreement module is encoded by S_7, and S_8 provides low level details pertaining to each agreement feature individually.

The following example motivates our decision to relax the BCPO condition and defer the conversion of signature modules to BCPOs to a separate resolution stage (Section 2.5).

Example 8
Let S_{10}, S_{11}, S_{12} be the signature modules depicted in Figure 8. The merge of S_{10} with S_{11} results in a non-BCPO. However, the additional information supplied by S_{12} resolves the problem, and $S_{10} \Cup S_{11} \Cup S_{12}$ is bounded complete.

Example 9
Let S_{13}, S_{14}, S_{15} be the signature modules depicted in Figure 9. In S_{13} the appropriate value for a and F is b and in S_{14} it is c. Hence $S_{13} \Cup S_{14}$ states that the appropriate value for a and F should be $lub(b, c)$. Although in this module there is no such element, in S_{15} $lub(b, c)$ is determined to be d. In $S_{13} \Cup S_{14} \Cup S_{15}$ the two outgoing arcs from the node marked by a are not replaced by a single arc whose target is the node marked by d, since

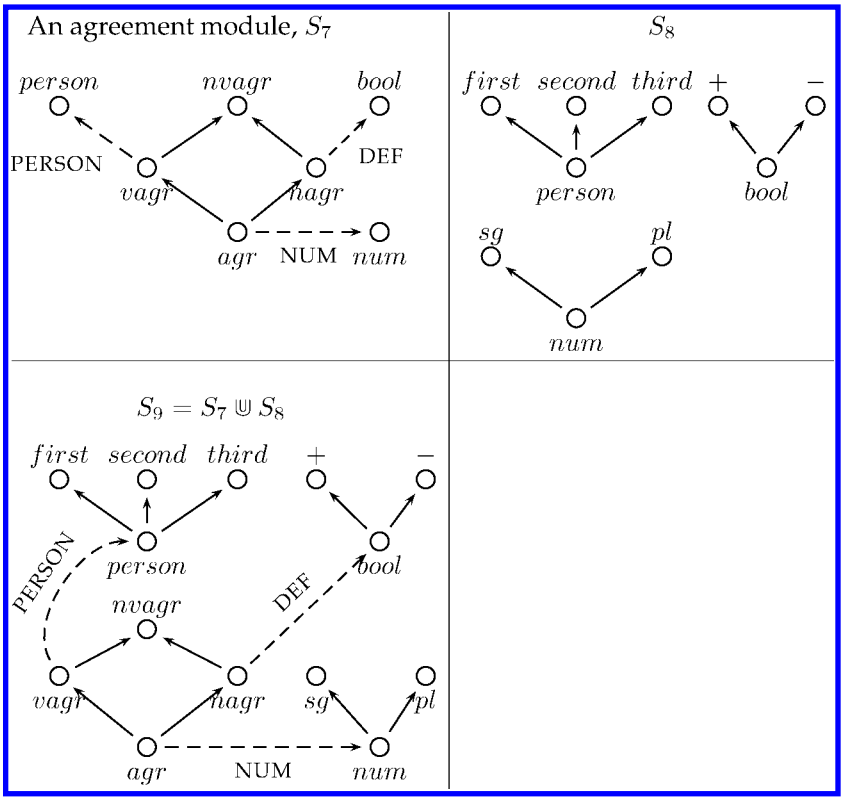

Figure 7
Merge.

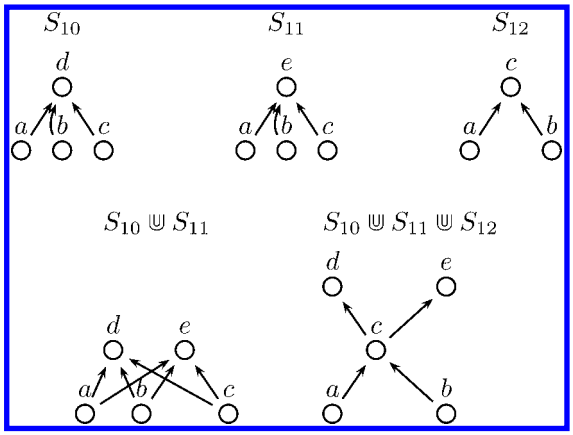

Figure 8
BCPO relaxation.

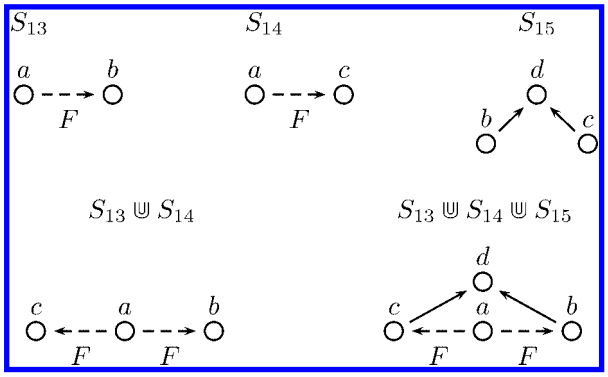

Figure 9
Merge of signature modules.

other signature modules may specify that the *lub* of b and c is some type other than d. These multiple outgoing arcs are preserved to maintain the associativity of the merge operation.

Proposition 1
Merge is commutative: For any two signature modules, S_1, S_2, Let $S = S_1 \Cup S_2$ and $S' = S_2 \Cup S_1$ where P, P' are their underlying PSSs, respectively. Then $P = P'$. In particular, either both are defined or both are undefined.

The proof follows immediately from the fact that the merge operation is defined by set union and equivalence relations which are commutative operations.

Proposition 2
Merge is associative up to isomorphism:[6] for all S_1, S_2, S_3, Let $S = (S_1 \Cup S_2) \Cup S_3$ and $S' = S_1 \Cup (S_2 \Cup S_3)$ where P, P' are their underlying PSSs, respectively. Then $P \sim P'$.

6 The definition of signature module isomorphism is a simple extension of graph isomorphism; see the Appendix for more details.

The proof of associativity is similar in spirit to the proof of the associativity of (polarized) forest combination (Sygal and Wintner 2009) and is therefore suppressed.

2.4.3 Attachment. Consider again S_1 and S_9, the signature modules of Figures 1 and 7, respectively. S_1 stipulates two distinct (but anonymous) values for $Approp(n, \text{AGR})$ and $Approp(v, \text{AGR})$. S_9 stipulates two nodes, typed *nagr* and *vagr*, with the intention that these nodes be coalesced with the two anonymous nodes of S_1. However, the 'merge' operation defined in the previous section cannot achieve this goal, since the two anonymous nodes in S_1 have different attributes from their corresponding typed nodes in S_9. In order to support such a unification of nodes we need to allow a mechanism that specifically identifies two designated nodes, regardless of their attributes. The parametric view of nodes facilitates exactly such a mechanism.

The attachment operation is an asymmetric operation, like function composition, where a signature module, S_1, receives as input another signature module, S_2. The information encoded in S_2 is added to S_1 (as in the merge operation), but additionally, the exported parameters of S_2 are assigned to the imported parameters of S_1: Each of the exported parameters of S_2 is forced to coalesce with its corresponding imported parameter of S_1, regardless of the attributes of these two parameters (i.e., whether they are indistinguishable or not).

Definition 13
Let $S_1 = \langle\langle Q_1, T_1, \preceq_1, Ap_1\rangle, Int_1, Imp_1, Exp_1\rangle$ and $S_2 = \langle\langle Q_2, T_2, \preceq_2, Ap_2, \rangle, Int_2, Imp_2, Exp_2\rangle$ be two consistent signature modules. S_2 can be **attached** to S_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 $Imp_1[i] \overset{*}{\preceq}_1 Imp_1[j]$, then $Exp_2[j] \overset{*}{\not\preceq}_2 Exp_2[i]$

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 parameters are attached to each other, they are marked by the same type. If they are marked by two different types they cannot be coalesced.[7] Finally, the last two conditions guarantee the antisymmetry of the subtyping relation in the resulting signature module: The third condition requires the two signature modules to be mergeable. The last condition requires that no subtyping cycles be created by the attachment of parameters: If q_1 is a supertype of q_1' in S_1 and q_2 is a supertype of q_2' in S_2, then q_2' and q_2 cannot be both attached to q_1 and q_1', respectively. Notice that as in the merge operation, two signature modules can be attached only if the resulting subtyping

7 A variant of attachment can be defined in which if two typed parameters, which are attached to each other, are marked by two different types, then the type of the exported node overrides the type of the imported node.

relation is indeed a partial order, where the only obstacle can be the antisymmetry of the resulting relation. The combination of the appropriateness relations, in contrast, cannot cause the attachment operation to fail because any violation of the appropriateness conditions in signature modules can be deterministically resolved.[8]

Definition 14
Let $S_1 = \langle\langle Q_1, T_1, \preceq_1, Ap_1\rangle, Int_1, Imp_1, Exp_1\rangle$ and $S_2 = \langle\langle Q_2, T_2, \preceq_2, Ap_2, \rangle, Int_2, Imp_2, Exp_2\rangle$ be two consistent signature modules. If S_2 can be attached to S_1, then the **attachment** of S_2 to S_1, denoted $S_1(S_2)$, is: $S_1(S_2) = compact(ApCl(compact(S)))$, where $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ 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:

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

When a module S_2 is attached to a module S_1, all the exported nodes of S_2 are first attached to the imported nodes of S_1, respectively, through the equivalence relation, '$\equiv$'. In this way, for each imported node of S_1, all the information encoded by the corresponding exported node of S_2 is added. Notice that each equivalence class of '$\equiv$' contains either one or two nodes. In the former case, these nodes are either non-imported nodes of S_1 or non-exported nodes of S_2. In the latter, these are pairs of an imported node of S_1 and its corresponding exported node from S_2. Hence '$\equiv$' is trivially transitive. Then, similarly to the merge operation, pairs of nodes marked by the same type and pairs of indistinguishable anonymous nodes are coalesced. In contrast to the merge operation, in the attachment operation two distinguishable anonymous nodes, as well as an anonymous node and a typed node, can be coalesced. This is achieved by the parametric view of nodes and the view of one module as an input to another module.

The imported and exported nodes of the resulting module are the equivalence classes of the imported and exported nodes of the first module, S_1, respectively. The nodes of S_2 which are neither internal nor exported are classified as external nodes in the resulting module. This asymmetric view of nodes stems from the view of S_1 receiving S_2 as input: In this way, S_1 may import further information from other modules.

8 Relaxed variants of these conditions are conceivable; 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.

Notice that in the attachment operation internal nodes facilitate no interaction between modules, external nodes facilitate interaction only through naming, and imported and exported nodes facilitate interaction both through naming and by reference.

Example 10
Consider again S_1 and S_9, the signature modules of Figures 1 and 7, respectively. Let S_{1a} and S_{9a} be the signature modules of Figure 10 (these signature modules have the same underlying graphs as those of S_1 and S_9, respectively, with different classification of nodes). Notice that all nodes in both S_{1a} and S_{9a} are non-internal. Let $Imp_{1a} = \langle q_4, q_5 \rangle$ and let $Exp_{9a} = \langle p_9, p_{10} \rangle$. $S_{1a}(S_{9a})$ is depicted in the same figure. 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.

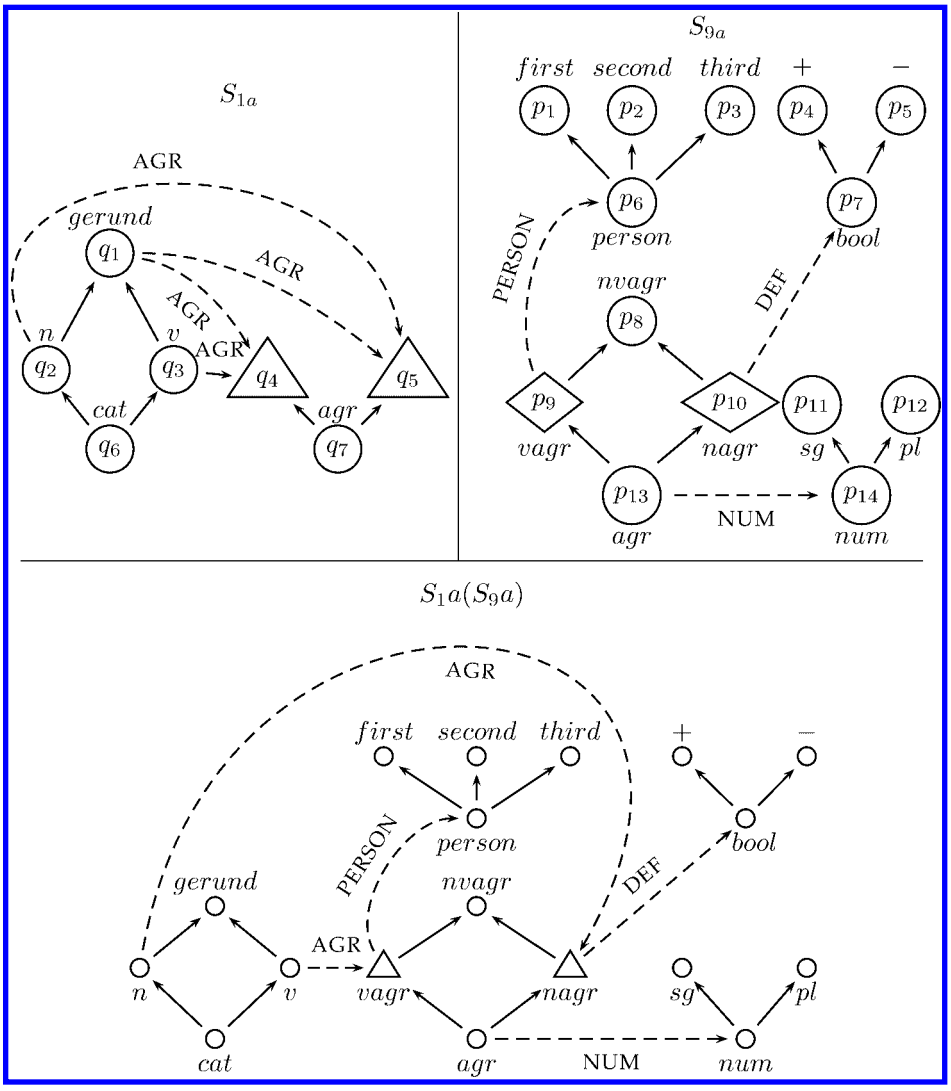

Figure 10
Attachment.

2.4.4 Example: Parametric Lists. Lists and parametric lists are extensively used in typed unification-based formalisms, for example in HPSG. The mathematical foundations for parametric lists were established by Penn (2000). As an example of the utility of signature modules and the attachment operation, we show how they can be used to construct parametric lists in a straightforward way.

Consider Figure 11. The signature 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 subsequently. Now, if *Phrase* is a simple module consisting of one exported node, of type *phrase*, then the signature module obtained by *List*(*Phrase*) is obtained by coalescing q_3, the imported node of *List* with the single exported node of *Phrase*.

Other modules can now use lists of phrases; for example, the module *Struct* 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 *Struct*(*List*(*Phrase*)). The single node of *Phrase* instantiates the imported node of *List*, thus determining a list of phrases. The entire list is then attached to the signature module *Struct*, where the root of the list instantiates the imported node typed by *phrase_list* in *Struct*.

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*: q_2 can be coalesced only with anonymous nodes with the exact same attributes, that is, only with nodes whose appropriate value for the feature FIRST is a node typed

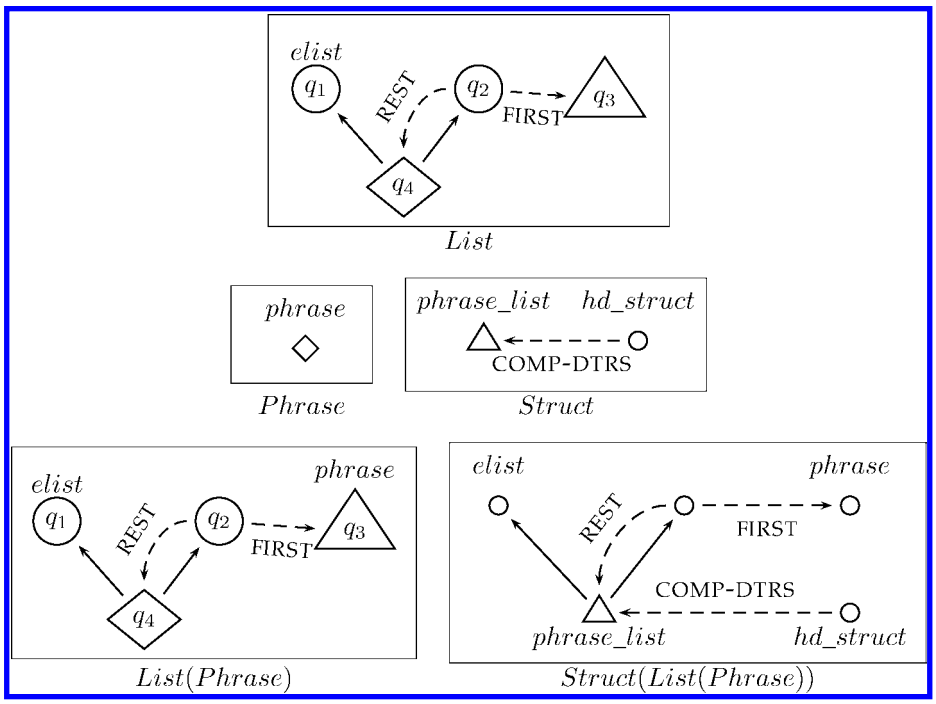

Figure 11
Implementing parametric lists with signature modules.

by *phrase*. If q_2 would have been typed by *ne_list* it could be coalesced with any other node marked by the same type, such as other such nodes from different copies of the list, resulting in a list whose members have various types. Observe that the uniqueness of each copy of the list could be achieved also by declaring q_2 an internal node, but this solution prevents other modules from referring to this node, as is reasonably desired. q_1 (of *List*) is typed by *elist*. Because only one copy of this node is required for all the list copies, there is no problem with typing this node.

Compared with the parametric type signatures of Penn (2000), our implementation of parametric lists is simple and general: It falls out directly as one application of signature modules, whereas the construction of Penn requires dedicated machinery (parametric subtyping, parametric appropriateness, coherence, etc.) We conjecture that signature modules can be used to simulate parametric type signatures in the general case, although we do not have a proof of such a result.

2.4.5 Example: The 'Addendum' Operator in LKB. The 'addendum' operator[9] was added to the type definition language of LKB (Copestake 2002) in 2005, to allow the grammar developer to add attributes to an already defined type without the need to repeat previously defined attributes of that type. The need for such an operation arose as a consequence of the development of frameworks that generate grammars from pre-written fragments (e.g., the LINGO grammar matrix, Bender, Flickinger, and Oepen 2002), since editing of framework-source files may lead to errors.

Signature modules trivially support this operator, either by the merge operation (in which case different attributes of a typed node are gathered from different modules) or by attachment, where attributes can be assigned to a specific node, even without specifying its type.

2.5 Extending Signature Modules to Type Signatures

Signature modules encode only partial information, and are therefore not required to conform with all the constraints imposed on ordinary signatures. After modules are combined, however, the resulting signature module must be extended into a bona fide signature. For that purpose we use four algorithms, each of which deals with one property:

1. **Name resolution**: This algorithm assigns types to anonymous nodes (Section 2.5.1).
2. **Appropriateness consolidation**: This algorithm determinizes Ap, converts it from a relation to a function and enforces upward closure (Section 2.5.2).
3. **Feature introduction completion**: This algorithm (whose use is optional) enforces the feature introduction condition. This is done using the algorithm of Penn (2000).
4. **BCPO completion**: This algorithm extends '$\preceq$' to a BCPO. Again, we use the algorithm of Penn (2000).

9 See `http://depts.washington.edu/uwcl/twiki/bin/view.cgi/Main/TypeAddendum`.

The input to the resolution algorithm is a signature module and its output is a bona fide type signature.

Algorithm 1 **(Resolve (S))**

1. $S := NameResolution(S)$
2. $S := BCPO{-}Completion(S)$
3. $S := ApCl(S)$
4. $S := ApConsolidate(S)$
5. $S := FeatureIntroductionCompletion(S)$
6. $S := BCPO{-}Completion(S)$
7. $S := ApCl(S)$
8. $S := ApConsolidate(S)$
9. return S

The order in which the four algorithms are executed is crucial for guaranteeing that the result is indeed a bona fide signature. First, the resolution algorithm assigns types to anonymous nodes via the name resolution algorithm (stage 1). The BCPO completion algorithm (stage 2) of Penn (2000) adds types as least upper bounds for sets of types which have upper bounds but do not have a minimal upper bound. However, the algorithm does not determine the appropriateness specification of these types. A natural solution to this problem is to use Ap-Closure (stage 3) but this may lead to a situation in which the newly added nodes have multiple outgoing Ap-arcs with the same label. To solve the problem, we execute the BCPO completion algorithm before the Ap-consolidation algorithm (stage 4), which also preserves bounded completeness. Now, the feature introduction completion algorithm (stage 5) of Penn assumes that the subtyping relation is a BCPO and that the appropriateness specification is indeed a function and hence, it is executed after the BCPO completion and Ap-consolidation algorithms. However, as Penn observes, this algorithm may disrupt bounded completeness and therefore the result must undergo another BCPO completion and therefore another Ap-consolidation (stages 6–8).

A signature module is extended to a type signature after all the information from the different modules have been gathered. Therefore, there is no need to preserve the classification of nodes and only the underlying PSS is of interest. However, because the resolution procedure uses the compactness algorithm which is defined over signature modules, we define the following algorithms over signature modules as well. In cases where the node classification needs to be adjusted, we simply take the trivial classification (i.e., $Int = Imp = Exp = \emptyset$).

2.5.1 Name Resolution. During module combination only pairs of indistinguishable anonymous nodes are coalesced. Two nodes, only one of which is anonymous, can still be otherwise indistinguishable but they are not coalesced during combination to ensure the associativity of module combination. The goal of the name resolution procedure is to assign a type to every anonymous node, by coalescing it with a typed node with an identical environment, if one exists. If no such node exists, or if there is more than one such node, the anonymous node is given an arbitrary type.

The name resolution algorithm iterates as long as there are nodes to coalesce. In each iteration, for each anonymous node the set of its typed equivalent nodes is computed (stage 1). Then, using the computation of stage 1, anonymous nodes are coalesced with their corresponding typed node, if such a node uniquely exists (stage 2.1). Coalescing all such pairs may result in a signature module that may include indistinguishable anonymous nodes and therefore the signature module is compacted (stage 2.2). Compactness can trigger more pairs that need to be coalesced, and therefore this procedure is repeated (stage 2.3). When no pairs that need to be coalesced are left, the remaining anonymous nodes are assigned arbitrary names and the algorithm halts.

We first define $NodeCoalesce(S, q, q')$: this is a signature module S' that is obtained from S by coalescing q with q'.

Definition 15
Let $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ be a signature module and let $q, q' \in Q$. Define $NodeCoalesce(S, q, q') = \langle\langle Q_1, T_1, \preceq_1, Ap_1\rangle, Int_1, Imp_1, Exp_1\rangle$ where:

- $Q_1 = Q \backslash \{q\}$
- $T_1 = T\,|_{Q_1}$
- $\preceq_1 = \{(q_1, q_2) \mid q_1 \preceq q_2 \text{ and } q_1, q_2 \neq q\} \cup \{(p, q') \mid p \preceq q\} \cup \{(q', p) \mid q \preceq p\}$
- $Ap_1 = \{(q_1, F, q_2) \mid (q_1, F, q_2) \in Ap \text{ and } q_1, q_2 \neq q\} \cup$
  $\{(p, F, q') \mid (p, F, q) \in Ap\} \cup \{(q', F, p) \mid (q, F, p) \in Ap\}$
- $Int = Imp = Exp = \emptyset$

The input to the name resolution algorithm is a signature module and its output is a signature module whose typing function, T, is total. Let $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ be a signature module, and let NAMES $\subset$ TYPE be an enumerable set of fresh types from which arbitrary names can be taken to mark nodes in Q. The following algorithm marks all the anonymous nodes in S:

Algorithm 2 **(NameResolution (**$S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$**))**

1. for all $q \in Q$ such that $T(q)\uparrow$, compute $Q_q = \{q' \in Q \mid T(q')\downarrow \text{ and } q' \text{ is equivalent to } q\}$.
2. let $\overline{Q} = \{q \in Q \mid T(q)\uparrow \text{ and } |Q_q| = 1\}$. If $\overline{Q} \neq \emptyset$ then:
   - 2.1. for all $q \in \overline{Q}$, $S := NodeCoalesce(S, q, q')$, where $Q_q = \{q'\}$
   - 2.2. $S := compact(S)$
   - 2.3. go to (1)
3. Mark remaining anonymous nodes in Q with arbitrary unique types from NAMES and halt.

For a given anonymous node, the calculation of its typed equivalent nodes is mostly technical and is therefore suppressed.

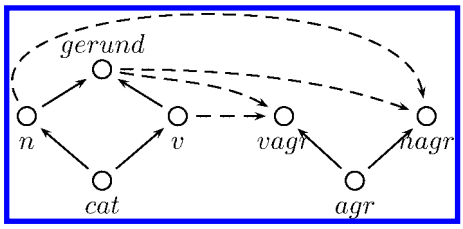

Figure 12
Name resolution result for S_{10}.

Example 11
Consider the signature module S_6 depicted in Figure 6. Executing the name resolution algorithm on this module results in the signature module of Figure 12 (AGR-labels are suppressed for readability.) The two anonymous nodes in S_6 are coalesced with the nodes marked *nagr* and *vagr*, as per their attributes. Compare to Figure 1, in particular how two anonymous nodes in S_1 are assigned types from S_5 (Figure 6).

2.5.2 Appropriateness Consolidation. For each node q, the set of outgoing appropriateness arcs with the same label F, $\{(q, F, q')\}$, is replaced by the single arc (q, F, q_l), where q_l is marked by the *lub* of the types of all q'. If no *lub* exists, a new node is added and is marked by the *lub*. The result is an appropriateness relation which is a function, and in which upward closure is preserved; feature introduction is dealt with separately.

The input to the following procedure is a signature module whose typing function, T, is total; its output is a signature module whose typing function is total and whose appropriateness relation is a function that maintains upward closure. Let $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ be a signature module. For each $q \in Q$ and $F \in$ FEAT, let

- $target(q, F) = \{q' \mid (q, F, q') \in Ap\}$
- $sup(q) = \{q' \in Q \mid q' \preceq q\}$
- $sub(q) = \{q' \in Q \mid q \preceq q'\}$

Algorithm 3 **(ApConsolidate (**$S = \langle\langle Q, T, \preceq, Ap\rangle, \mathbf{Int}, \mathbf{Imp}, \mathbf{Exp}\rangle$**))**

1. Set $Int := Imp := Exp := \emptyset$
2. Find a node q and a feature F for which $|target(q, F)| > 1$ and for all $q' \in Q$ such that $q' \stackrel{*}{\preceq} q$, $|target(q', F)| \leq 1$ (i.e., q is a minimal node with respect to a topological ordering of Q). If no such pair exists, halt
3. If $target(q, F)$ has a *lub*, p, then:
   (a) for all $q' \in target(q, F)$, remove the arc (q, F, q') from Ap
   (b) add the arc (q, F, p) to Ap
   (c) for all $q' \in target(q, F)$ and for all $q'' \in sub(q')$, if $p \neq q''$ then add the arc (p, q'') to $\preceq$

4. (a) Otherwise, If $target(q, F)$ has no *lub*, add a new node, p, to Q with:
   - $sup(p) = target(q, F)$
   - $sub(p) = \bigcup_{q' \in target(q,F)} sub(q')$
   (b) Mark p with a fresh type from NAMES
   (c) For all $q' \in target(q, F)$, remove the arc (q, F, q') from Ap
   (d) Add (q, F, p) to Ap
5. $S := ApCl(S)$
6. $S := compact(S)$
7. go to (2)

The order in which nodes are selected in step 2 of the algorithm is from supertypes to subtypes. This is done to preserve upward closure. When a set of outgoing appropriateness arcs with the same label F, $\{(q, F, q')\}$, is replaced by a single arc (q, F, q_l), all the subtypes of all q' are added as subtypes of q_l (stage 3c). This is done to maintain the upwardly closed intention of appropriateness arcs (see Example 13). Additionally, q_l is added as an appropriate value for F and all the subtypes of q. This is achieved by the Ap-Closure operation (stage 5). Again, this is done to preserve upward closure. If a new node is added (stage 3), then its subtypes are inherited from its immediate supertypes. Its appropriate features and values are also inherited from its immediate supertypes through the Ap-Closure operation (stage 5). In both stages 3 and 4, a final step is compaction of the signature module in order to remove redundant arcs.

Example 12
Consider the signature module depicted in Figure 12. Executing the appropriateness consolidation algorithm on this module results in the module depicted in Figure 13.

Example 13
Consider the signature modules depicted in Figure 14. Executing the appropriateness consolidation algorithm on S_{16}, the two outgoing arcs from a labeled with F are first replaced by a single outgoing arc to a newly added node, *new*1, which is the *lub* of b and c. During this first iteration, *new*1 is also added as a supertype of e and f. The result of these operations is S_{17}. Notice that in S_{16}, the arc (a, F, b) is interpreted as "the appropriate value of a and F is at least b." In particular, this value may be e. S_{17} maintains

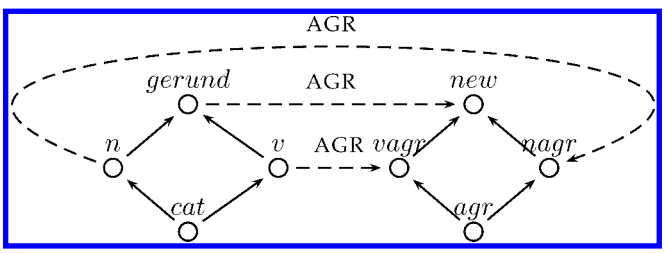

Figure 13
Appropriateness consolidation: result.

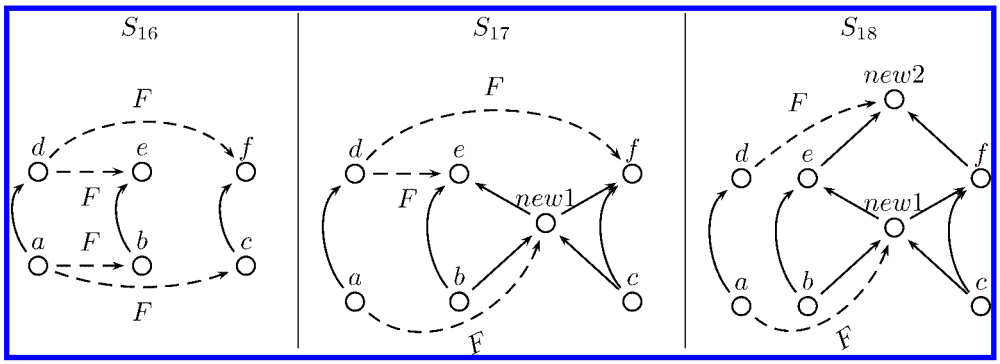

Figure 14
Appropriateness consolidation.

this interpretation by means of the subtyping arc that is added from *new*1 to *e*. Then, the two outgoing arcs from *d* labeled with *F* (to *e* and *f*) are replaced by a single outgoing arc to a newly added node, *new*2, which is the *lub* of *e* and *f*. The result of these operations is S_{18}, which is also the final result.

3. Grammar Modules

Before extending signature modules to grammar modules, we first recall basic notions of typed unification grammars. For the following definitions we assume that a type signature $\langle \textsc{Type}, \sqsubseteq, \textsc{Feat}, \mathit{Approp} \rangle$ has been specified.

Definition 16
A **path** is a finite sequence of features, and the set $\textsc{Paths} = \textsc{Feat}^*$ is the collection of paths. ϵ is the empty path.

Definition 17
A **typed pre-feature structure** (pre-TFS) is a triple $\langle \Pi, \Theta, \bowtie \rangle$ where:

- $\Pi \subseteq \textsc{Paths}$ is a non-empty set of Paths
- $\Theta : \Pi \rightarrow \textsc{Type}$ is a total function, assigning a type for all paths
- $\bowtie \subseteq \Pi \times \Pi$ is a relation specifying reentrancy

A **typed feature structure** (TFS) is a pre-TFS $A = \langle \Pi, \Theta, \bowtie \rangle$ for which the following requirements hold:

- Π is prefix-closed: if $\pi\alpha \in \Pi$ then $\pi \in \Pi$ (where $\pi, \alpha \in \textsc{Paths}$)
- A is fusion-closed: if $\pi\alpha \in \Pi$ and $\pi \bowtie \pi'$ then $\pi'\alpha \in \Pi$ and $\pi\alpha \bowtie \pi'\alpha$
- $\bowtie$ is an equivalence relation with a finite index (with $[\bowtie]$ the set of its equivalence classes) including at least the pair (ϵ, ϵ)
- Θ respects the equivalence: if $\pi_1 \bowtie \pi_2$ then $\Theta(\pi_1) = \Theta(\pi_2)$

Definition 18
A TFS $A = \langle \Pi, \Theta, \bowtie \rangle$ is **well-typed** iff whenever $\pi \in \Pi$ and $F \in \text{FEAT}$ are such that $\pi F \in \Pi$, then $Approp(\Theta(\pi), F)\downarrow$, and $Approp(\Theta(\pi), F) \sqsubseteq \Theta(\pi F)$.

A **grammar** is defined over a concrete type signature and is a structure including a set of rules (each constructed from a series of TFSs), a lexicon mapping words to sets of TFSs and a start symbol which is a TFS.

We are now ready to define grammar modules and the way in which they interact. A grammar module is a structure $M = \langle S, G \rangle$, where S is a signature module and G is a grammar. The grammar is defined over the signature module analogously to the way ordinary grammars are defined over type signatures, albeit with two differences:

1. TFSs are defined over type signatures, and therefore each path in the TFS is associated with a type. When TFSs are defined over signature modules this is not the case, because signature modules may include anonymous nodes. Therefore, the standard definition of TFSs is modified such that every path in a TFS is assigned a node in the signature module over which it is defined, rather than a type.
2. Enforcing all TFSs in the grammar to be well-typed is problematic for three reasons:

   (a) Well-typedness requires that $\Theta(\pi F)$ be an upper bound of all the (target) nodes which are appropriate for $\Theta(\pi)$ and F. However, each module may specify only a subset of these nodes. The whole set of target nodes is known only after all modules combine.

   (b) A module may specify several appropriate values for $\Theta(\pi)$ and F, but it may not specify any upper bound for them.

   (c) Well-typedness is not preserved under module combination. The natural way to preserve well-typedness under module combination requires addition of nodes and arcs, which would lead to a non-associative combination.

   To solve these problems, we enforce only a relaxed version of well typedness. The relaxation is similar to the way upward closure is relaxed: Whenever $\Theta(\pi) = q$, $\Theta(\pi F)$ is required to be a subtype of *one* of the values q' such that $(q, F, q') \in Ap$. This relaxation supports the partiality and associativity requirements of modular grammar development (Section 1.1). After all modules are combined, the resulting grammar is extended to maintain well-typedness.

The two combination operators, merge and attachment, are lifted from signature modules to grammar modules. In both cases, the components of the grammars are combined using simple set union. This reflects our initial observation (Section 1.1) that most of the information in typed formalisms is encoded by the signature, and therefore modularization is carried out mainly through the distribution of the signature between the different modules; the lifting of the signature combination operation to operations on full grammar modules is therefore natural and conservative.

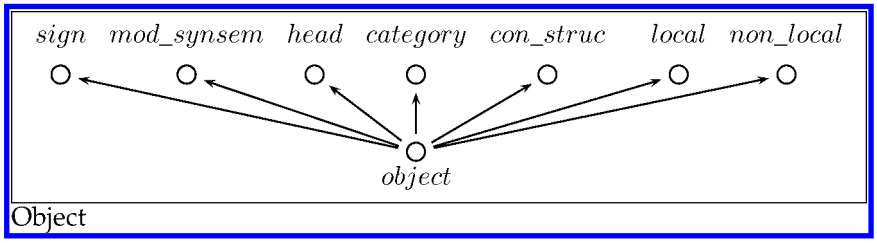

Figure 15
The main fragments of the signature.

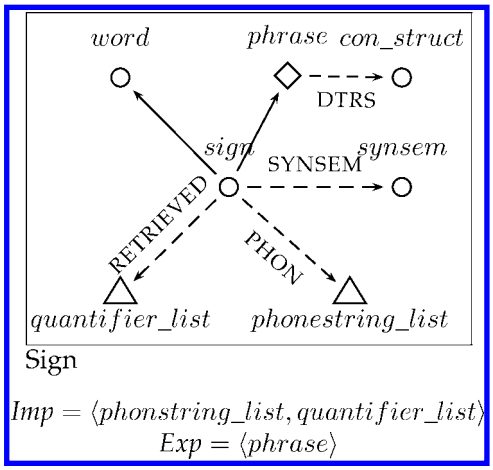

Figure 16
A signature module, *Sign*.

Finally, grammar modules are extended to bona fide typed unification grammars by extending the underlying signature module into an ordinary type signature and adjusting the grammar accordingly.[10]

4. Modular Construction of the Basic HPSG Signature

To demonstrate the utility of signature modules for practical grammar engineering we use signature modules and their combination operators in this section to work out a modular design of the HPSG grammar of Pollard and Sag (1994). This is a grammar of English whose signature, covering several aspects of syntax and semantics, is developed throughout the book. The signature is given (Pollard and Sag 1994, Appendix A1) as one unit, making it very hard to conceptualize and, therefore, to implement and maintain. We reverse-engineered this signature, breaking it up into smaller-scale modules that emphasize fragments of the theory that are more local, and the interactions among such fragments through 'merge' and 'attachment'.[11] Some of the fragments make use of the signature module *List* of Figure 11.

10 In practice, an extra adjustment is required in order to restore well-typedness, but we suppress this technicality.

11 Of course, other ways to break up the given signature to modules are conceivable. In particular, the *Synsem* module of Figure 19 may better be broken into two modules.

We begin with a module defining **objects** (Figure 15), where the type *object* is the most general type. This module defines the main fragments of the signature.

Figure 16 defines the module *Sign*. It consists of the type *sign*, and its two subtypes *word* and *phrase*. The latter is exported and will be used by other modules, as we presently show. In addition, two of the appropriate features of *sign* are lists; note that the values of PHON and RETRIEVED are imported.

Next, we consider constituent structure, and in particular headed structures, in Figure 17. Note in particular that the feature COMP-DTRS, defined at *head_struc*, takes as values a list of phrases; this is an imported type, which is obtained as a result of several attachment operations (Figure 11).

Figure 18 describes the fragment of the signature rooted by *head*. This is basically a specification of the inventory of syntactic categories defined by the theory. Note how simple it is to add, remove, or revise a category by accessing this fragment only.

Figure 19 provides straight-forward definitions of *cat*egory and *synsem*, respectively. As another example, Figure 20 depicts the type hierarchy of nominal objects, which is completely local (in the sense that it does not interact with other modules, except at the root). Finally, Figure 21 abstracts over the internal structure of *Phonstring* and

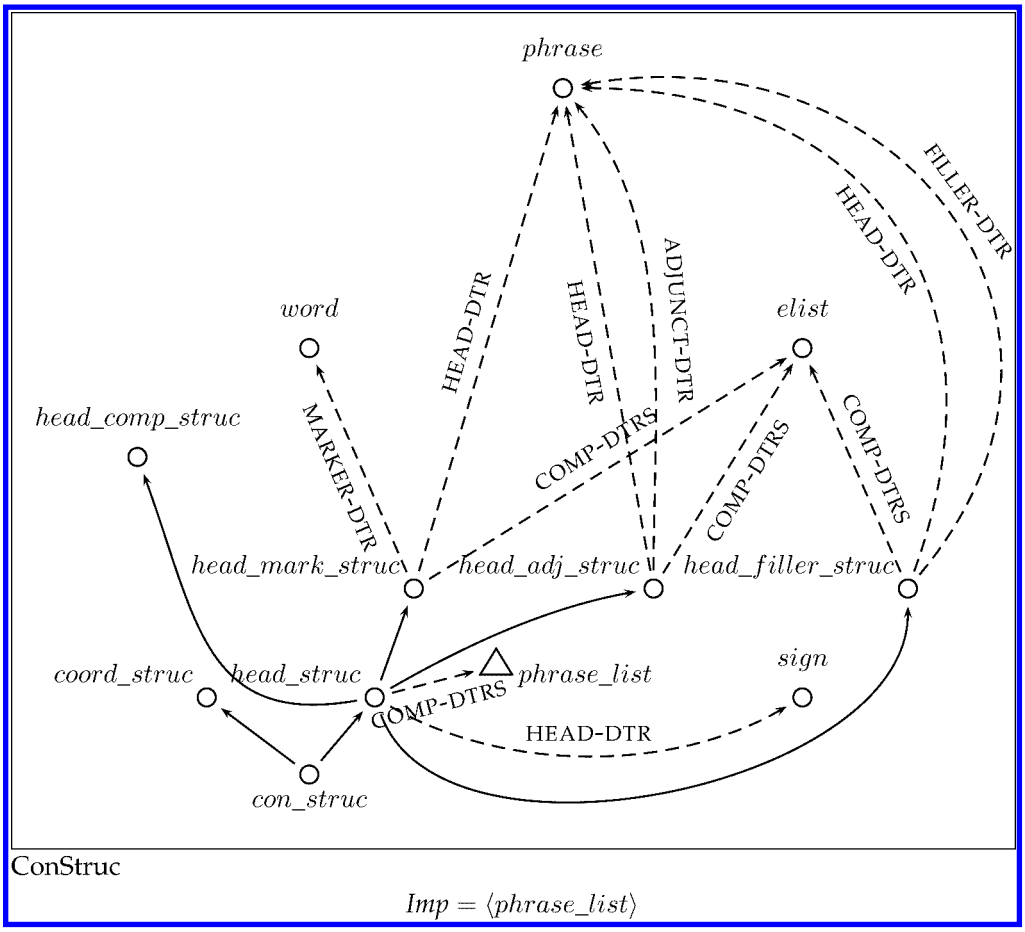

Figure 17
Phrase structure.

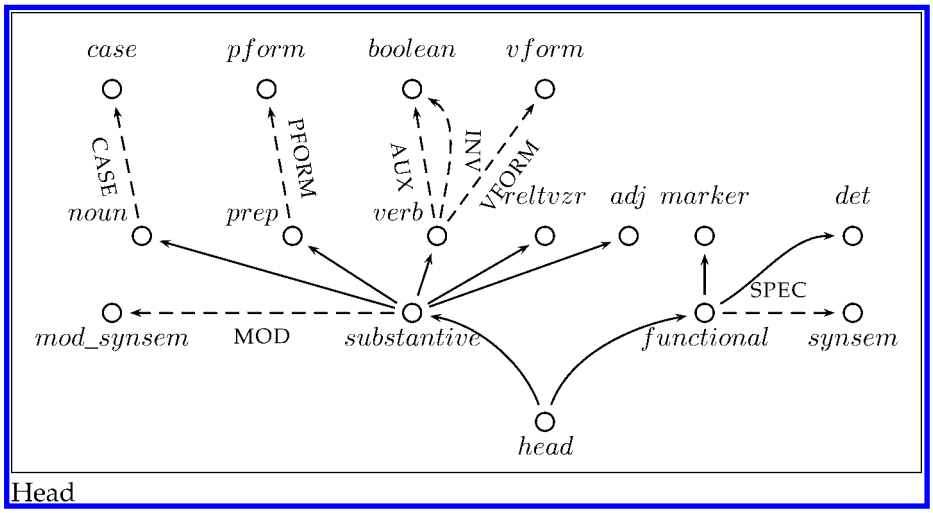

Figure 18
A signature module, *Head*.

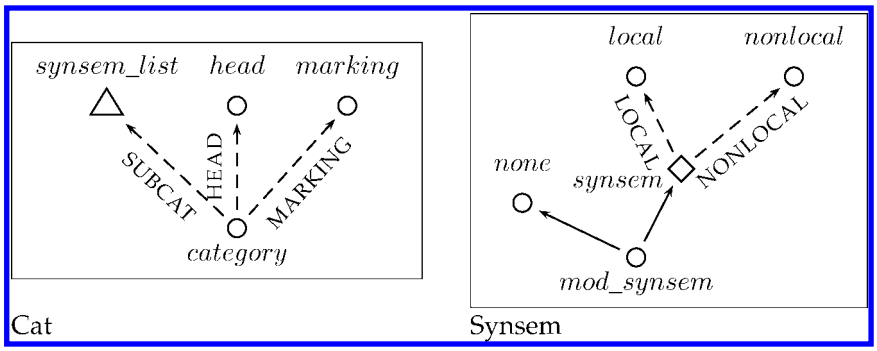

Figure 19
Signature modules.

Quantifier; these are only representatives of the actual signature modules which define these fragments.

The full HPSG signature consists of several more fragments that we do not depict here. With this in mind, the HPSG signature can now be constructed in a modular way from the fragments defined earlier. The construction is given in Figure 22.

First, we produce two lists of *phonestring* and *quantifier*, which are merged into one module through the operation

$$List(Phonestring) \uplus List(Quantifier)$$

Then, this module instantiates the two imported nodes *phonestring_list* and *quantifier_list* in the module *Sign* through the operation

$$Sign(List(Phonestring) \uplus List(Quantifier))$$

Notice how the order of the parameters ensures the correct instantiation. Now, in the second element, *List*(*Sign*) both creates a list of *phrase* (since *phrase* is an exported

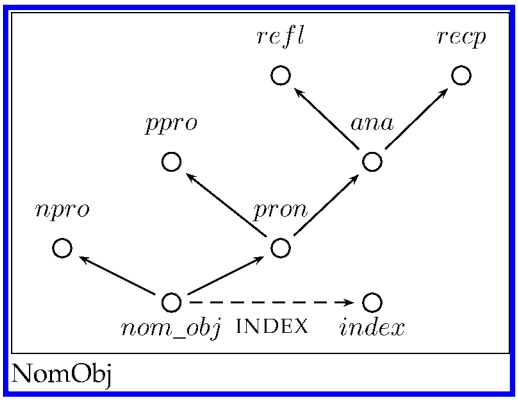

Figure 20
A classification of nominal objects.

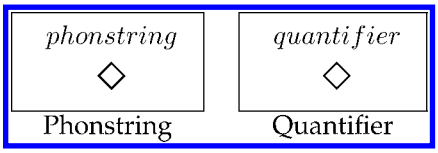

Figure 21
Parametric signature modules.

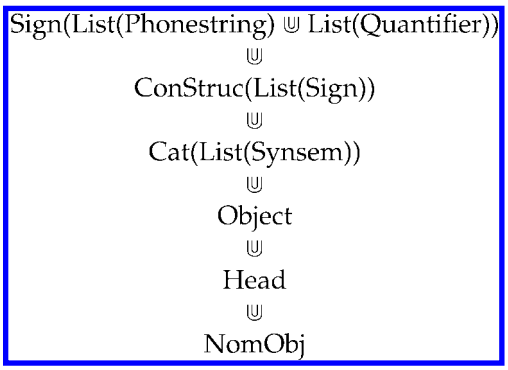

Figure 22
HPSG signature construction.

node in the module *Sign*) and unifies the information in the two modules. Similarly, *ConStruc*(*List*(*Sign*)) unifies the information in the three modules and instantiates the node *phrase_list* in the module *ConStruc*. In the same way, *List*(*Synsem*) both creates a list of *synsem* (since *synsem* is an exported node in the module *Synsem*) and unifies the information in the two modules. Then, *Cat*(*List*(*Synsem*)) unifies the information in the three modules and instantiates the node *synsem_list* in the module *Cat*. Finally, all the information from the different modules is unified through the merge operation. Other modules can be added, either by merge or by attachment. Additionally, the internal structure of each module can be locally modified. Such changes become much easier given the smaller size and theoretical focus of each of the modules.

This modular approach has significant advantages over the monolithic approach of Pollard and Sag (1994): The signature of Pollard and Sag is hard to conceptualize

because all the information is presented in a single hierarchy. In contrast, looking at each small fragment (module) separately, it is easier to understand the information encoded in the module. Contemporary type signatures are in fact much larger; working with small fragments in such grammars is instrumental for avoiding or tracking errors. Moreover, grammar maintenance is significantly simplified, because changes can be done locally, at the level of specific modules. Of course, when a new grammar is developed from scratch, modularization can be utilized in such a way as to reflect independent fragments of the linguistic theory in separate modules.

While the grammar of Pollard and Sag (1994) is not really large-scale, it is large enough to reflect the kind of knowledge organization exhibited by linguistically motivated grammars, but is at the same time modest enough so that its redesign in a modular way can be easily comprehended. It is therefore useful as a practical example of how type signatures can be constructed from smaller, simpler signature modules. Real-world grammars are not only much larger, they also tend to be more complex, and in particular express interactions in domains other than the type signature (specifically, as type constraints and as phrase-structure rules). Extending our solution to such interactions is feasible, but is beyond the scope of this preliminary work.

5. MODALE: A Platform for Modular Development of Type Signatures

Two leading implementation platforms are available for the development of typed unification grammars: The Linguistic Knowledge Building system (LKB) (Copestake 2002) and TRALE (Meurers, Penn, and Richter 2002), an extension of the Attribute Logic Engine (ALE) (Carpenter and Penn 2001). MODALE (MODular ALE) is a system that supports modular development of type signatures in both ALE and TRALE. The main features of the system are:

- The system provides a description language with which signature modules can be specified. The description language is intuitive and is built upon the description language of ALE. For example, the description of S_1, the signature module of Figure 2, is shown in Figure 23.
- Signature modules may be combined using either one of the two combination operators, merge and attachment, or by a complex combination involving several operators.
- Signature modules can be resolved to yield bona fide type signatures.
- The system compiles resolved modules into output files using either ALE or TRALE syntax; these files can be directly manipulated by one of the two systems.
- Signature modules can be printed using the syntax of the description language. This feature allows inspection of a signature module that was created as a result of several combination operators.

ALE and TRALE share the same underlying core, and are based on data structures and algorithms that take advantage of type signature properties such as bounded completeness, upward closure, and feature introduction, none of which can be assumed when working with a signature module. As a result, our implementation is not a direct adaption of the existing ALE/TRALE code, but a new system that was developed from

```
module(S1)
{
   cat sub [n,v].
      n sub [gerund].
      n approp [agr:{anon(q5)}].
         gerund sub [].
         gerund approp [agr:{anon(q4),anon(q5)}].
      v sub [gerund].
      v approp [agr:{anon(q4)}].
   agr sub [anon(q4),anon(q5)].
}
{
   int=<>.
   imp=<anon(q4),anon(q5)>.
   exp=<>.
}
```

Figure 23
MODALE description of S_1.

scratch. Extending the algorithms of Penn (2000) from type signatures into signature modules is left as a direction for future research.

The MODALE system provided us with an opportunity to experimentally evaluate the time efficiency of module combination. Indeed, the combination and resolution algorithms are computationally inefficient as they require repeated calculations of graph isomorphism, a problem which is neither known to be solvable in polynomial time nor NP-complete.[12] However, in the signatures we have experimented with so far, we encountered no time issues. Furthermore, it is important to note that these calculations are executed only once, in compile time, and have no impact on the run time of ALE/TRALE, which is the crucial stage in which efficiency is concerned.

6. Discussion and Conclusions

We presented a complete definition of typed unification grammar modules and their interaction. Unlike existing approaches, our solution is formally defined, mathematically proven, can be easily and efficiently implemented, and conforms to each of the desiderata listed in Section 1.1, as we now show.

Signature focus: Our solution focuses on the modularization of the signature (Section 2), and the extension to grammar modules (Section 3) is natural and conservative. We do restrict ourselves in this work to standard type signatures without type constraints. We defer the extension of type signatures to include also type constraints to future work.

Partiality: Our solution provides the grammar developer with means to specify any piece of information about the signature. A signature module may specify only partial information about the subtyping and appropriateness relations. Furthermore, the appropriateness relation is not a function as in ordinary signatures, and

12 Garey and Johnson (1979) provide a list of 12 major problems whose complexity status was open at the time of writing. Recognition of graph isomorphism is one of those, and one of the only two whose complexity remains unresolved today.

the developer may specify several appropriate nodes for the values of a feature F at a node q. The anonymity of nodes and relaxed upward closure also provide means for partiality. Another relaxation that supports partiality is not enforcing feature introduction and the BCPO conditions. Finally, the possibility of distributing the grammar between several modules and the relaxation of well-typedness also support this desideratum.

Extensibility: In Section 2.5 we show how a signature module can be deterministically extended into a bona fide signature.

Consistency: When modules are combined, either by merge or by attachment, the signature modules are required to be mergeable or attachable, respectively. In this way, contradicting information in different modules is detected prior to the combination. Notice that two signature modules can be combined only if the resulting subtyping relation is indeed a partial order.

Flexibility: The only restrictions we impose on modules are meant to prevent subtyping cycles.

(Remote) Reference: This requirement is achieved by the parametric view of nodes. Anonymity of nodes also supports this desideratum.

Parsimony: When two modules are combined, they are first unioned; thus the resulting module includes all the information encoded in each of the modules. Additional information is added in a conservative way by compaction and Ap-closure in order to guarantee that the resulting module is indeed well-defined.

Associativity: We provide two combination operations, *merge* and *attachment*. The attachment operation is an asymmetric operation, such as the function application, and therefore associativity is not germane. The merge operation, which *is* symmetric, is both commutative and associative and therefore conforms with this desideratum.

Privacy: Privacy is achieved through internal nodes which encode information that other modules cannot view or refer to.

Modular construction of grammars, and of type signatures in particular, is an essential requirement for the maintainability and sustainability of large-scale grammars. We believe that our definition of signature modules, along with the operations of *merge* and *attachment*, provide grammar developers with powerful and flexible tools for collaborative development of natural language grammars, as demonstrated in Section 4.

Modules provide *abstraction*; for example, the module *List* of Figure 11 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); 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, this work focuses on the modularity of the signature. This is not accidental, and reflects the centrality of the type signature in typed unification grammars. An extension of signature modules to include also type constraints is called for and will provide a better, fuller solution to the problem of grammar modularization. In a different track, we also believe that extra modularization capabilities can still be provided by means of the grammar itself. This direction is left for future research.

Although the present work is mainly theoretical, it has important practical implications. 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. The theoretical basis we presented in this work was implemented as a system, MODALE, that supports modular development of type signatures (Section 5). Once the theoretical basis is extended to include also type constraints, and they, as well as grammar modules, are fully integrated in a grammar development system, immediate applications of modularity are conceivable (see Section 1.1). Furthermore, although there is no general agreement among linguists on the exact form of modularity in grammar, a good modular interface will provide the necessary infrastructure for the implementation of different linguistic theories and will support their comparison in a common platform.

Finally, our proposed mechanisms clearly only fill very few lacunae of existing grammar development environments, and various other provisions will be needed in order for grammar engineering to be as well-understood a task as software engineering now is. We believe that we make a significant step in this crucial journey.

Appendix: Compactness

We provide a formal definition of the compactness algorithm in this section. For an example of the following two definitions see Example 3.

Definition 19
Let $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ be a pre-signature module. $(q_1, q_2) \in \preceq$ is a **redundant subtyping arc** if there exist $p_1, \ldots, p_n \in Q$, $n \geq 1$, such that $q_1 \preceq p_1 \preceq p_2 \preceq \ldots \preceq p_n \preceq q_2$.

Definition 20
Let $P = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ be a pre-signature module. $(q_1, F, q_2) \in Ap$ is a **redundant appropriateness arc** if there exists $q_2' \in Q$ such that $q_2 \overset{*}{\preceq} q_2'$, $q_2 \neq q_2'$ and $(q_1, F, q_2') \in Ap$.

The following definitions set the basis for determining whether two nodes are indistinguishable or not. Because signature modules are just a special case of directed, labeled graphs, we can adapt the well-defined notion of graph isomorphism to pre-signature modules. Informally, two pre-signature modules are isomorphic when their underlying PSSs have the same structure; the identities of their nodes may differ without affecting the structure. In our case, we require also that an anonymous node be mapped only to an anonymous node and that two typed nodes, mapped to each other, be marked by the same type. However, the classification of nodes as internal, imported, and/or exported has no effect on the isomorphism since it is not part of the core of the information encoded by the signature module.

Definition 21
Two pre-signature modules $S_1 = \langle\langle Q_1, T_1, \preceq_1, Ap_1\rangle, Int_1, Imp_1, Exp_1\rangle$, $S_2 = \langle\langle Q_2, T_2, \preceq_2, Ap_2\rangle, Int_2, Imp_2, Exp_2\rangle$ are **isomorphic**, denoted $S_1 \sim S_2$, if there exists a total, one-to-one and onto function i (**isomorphism**) mapping the nodes of S_1 to the nodes of S_2, such that all the following hold:

1. for all $q \in Q_1$, $T_1(q) = T_2(i(q))$
2. for all $q, q' \in Q_1$, $q \preceq_1 q'$ iff $i(q) \preceq_2 i(q')$
3. for all $q, q' \in Q_1$ and $F \in \textsc{Feat}$, $(q, F, q') \in Ap_1$ iff $(i(q), F, i(q')) \in Ap_2$

The *environment* of a node q is the set of nodes accessible from q via any sequence of arcs (subtyping or appropriateness, in any direction), up to and including the first typed node. The environment of a typed node includes itself only.

Definition 22
Let $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ be a pre-signature module. For all $q \in Q$ let the **environment** of q, denoted $env(q)$, be the smallest set such that:

- $q \in env(q)$
- If $q'' \in env(q)$ and $T(q'')\uparrow$ and for some $q' \in Q$ and $F \in \textsc{Feat}$, either $q' \preceq q''$ or $q'' \preceq q'$ or $(q', F, q'') \in Ap$ or $(q'', F, q') \in Ap$, then $q' \in env(q)$

Definition 23
Let $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ be a pre-signature module and let $Q' \subseteq Q$. The **strict restriction** of S to Q', denoted $S|^{strict}_{Q'}$, is $\langle\langle Q', T_2, \preceq_2, Ap_2\rangle, Int_2, Imp_2, Exp_2\rangle$, where:

- $T_2 = T_{|Q'}$
- $q_1 \preceq_2 q_2$ iff $q_1 \preceq q_2$, $q_1, q_2 \in Q'$ and either $T(q_1)\uparrow$ or $T(q_2)\uparrow$ (or both)
- $(q_1, F, q_2) \in Ap_2$ iff $(q_1, F, q_2) \in Ap$, $q_1, q_2 \in Q'$ and either $T(q_1)\uparrow$ or $T(q_2)\uparrow$ (or both)
- $Int_2 = Int_{|Q'}$
- $Imp_2 = Imp_{|Q'}$
- $Exp_2 = Exp_{|Q'}$

The strict restriction of a pre-signature module, S, to a set of nodes Q', is the subgraph induced by the nodes of Q' without any labeled or unlabeled arcs connecting two typed nodes in Q'.

Definition 24
Let $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ be a pre-signature module. Two nodes $q_1, q_2 \in Q$ are **indistinguishable**, denoted $q_1 \approx q_2$, if $S\,|^{strict}_{env(q_1)} \sim S\,|^{strict}_{env(q_2)}$ via an isomorphism i such that $i(q_1) = q_2$.

Example 14
Let S_1 be the signature module of Figure A1

$$env(q_4) = env(q_7) = \{q_1, q_4, q_7\},\ env(q_2) = env(q_6) = \{q_1, q_2, q_6\},$$
$$env(q_5) = \{q_1, q_5, q_8\} \text{ and } env(q_1) = \{q_1\}$$

The strict restrictions of S_1 to these environments are depicted in Figure A2. $q_2 \approx q_4$ and $q_6 \approx q_7$, where in both cases the isomorphism is

$$i = \{q_1 \mapsto q_1, q_2 \mapsto q_4, q_6 \mapsto q_7\}$$

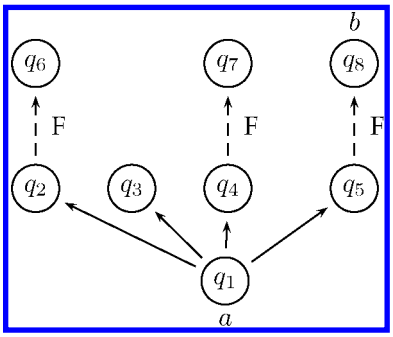

Figure A1
A signature module with indistinguishable nodes, S_1.

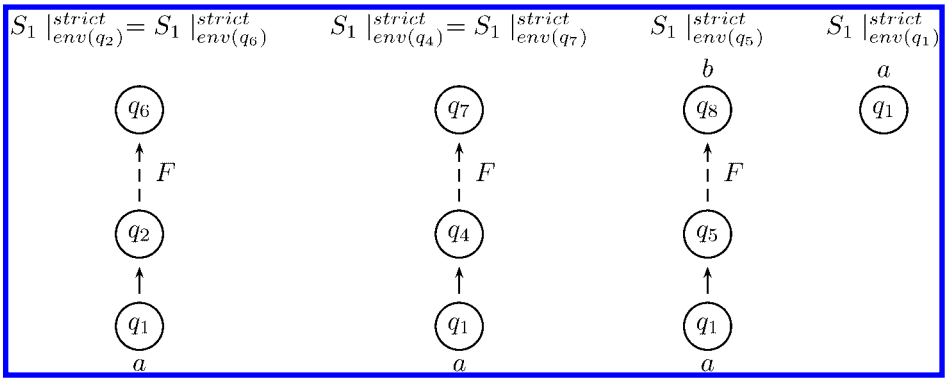

Figure A2
Strict restriction subgraphs.

However, q_5 is distinguishable from q_2 and q_4 because $T(q_8) \neq T(q_6)$ and $T(q_8) \neq T(q_7)$. Notice also that q_3 is distinguishable from q_2, q_4 and q_5 because it has no outgoing appropriateness arcs.

Proposition 3
Let $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ be a pre-signature module. Then '$\approx$' is an equivalence relation over Q.

Definition 25
A pre-signature module $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ is **non-redundant** if it includes no redundant subtyping and appropriateness arcs and for all $q_1, q_2 \in Q$, $q_1 \approx q_2$ implies $q_1 = q_2$.

Definition 26
Let $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ be a pre-signature module. The **coalesced pre-signature module**, denoted $coalesce(S)$, is $\langle\langle Q_1, T_1, \preceq_1, Ap_1\rangle, Int_1, Imp_1, Exp_1\rangle$ where:

- $Q_1 = \{[q]_\approx \mid q \in Q\}$ (Q_1 is the set of equivalence classes with respect to $\approx$)
- $T_1([q]_\approx) = T(q')$ for some $q' \in [q]_\approx$
- $\preceq_1 = \{([q_1]_\approx, [q_2]_\approx) \mid (q_1, q_2) \in \preceq\}$

- $Ap_1 = \{([q_1]_{\approx}, F, [q_2]_{\approx}) \mid (q_1, F, q_2) \in Ap\}$
- $Int_1 = \{[q]_{\approx} \mid q \in Int\}$
- $Imp_1 = \{[q]_{\approx} \mid q \in Imp \text{ and } [q]_{\approx} \notin Int\}$
- $Exp_1 = \{[q]_{\approx} \mid q \in Exp \text{ and } [q]_{\approx} \notin Int\}$
- the order of Imp_1 and Exp_1 is induced by the order of Imp and Exp, respectively, with recurring elements removed

When a pre-signature module is coalesced, indistinguishable nodes are identified. Additionally, the parameters and arities are induced from those of the input pre-signature module. All parameters may be coalesced with each other, as long as they are otherwise indistinguishable. If (at least) one of the coalesced nodes is an internal node, then the result is an internal node. Otherwise, if one of the nodes is imported then the resulting parameter is imported as well. Similarly, if one of the nodes is exported then the resulting parameter is exported.

The input to the compactness algorithm is a pre-signature module and its output is a non-redundant signature module which encodes the same information.

Algorithm 4 **(compact ($S = \langle\langle Q, T, \preceq, Ap\rangle, \boldsymbol{Int}, \boldsymbol{Imp}, \boldsymbol{Exp}\rangle$))**

1. Let $S_1 = \langle\langle Q_1, T_1, \preceq_1, Ap_1\rangle, Int_1, Imp_1, Exp_1\rangle$ be such that:
   - $Q_1 = Q$
   - $T_1 = T$
   - $\preceq_1 = \{(q_1, q_2) \in \preceq \mid (q_1, q_2) \text{ is a non-redundant subtyping arc in } S\}$
   - $Ap_1 = \{(q_1, F, q_2) \in Ap \mid (q_1, F, q_2) \text{ is a non-redundant appropriateness arc in } S\}$
   - $Int_1 = Int$
   - $Imp_1 = Imp$
   - $Exp_1 = Exp$
2. $S' = coalesce(S_1)$
3. If S' is non-redundant, return S', otherwise return $compact(S')$

The compactness algorithm iterates as long as the resulting pre-signature module includes redundant arcs or nodes. In each iteration, all the redundant arcs are first removed and then all indistinguishable nodes are coalesced. However, the identification of nodes can result in redundant arcs or can trigger more nodes to be coalesced. Therefore, the process is repeated until a non-redundant signature module is obtained. Notice that the compactness algorithm coalesces pairs of nodes marked by the same type regardless of their incoming and outgoing arcs. Such pairs of nodes may exist in a pre-signature module (but not in a signature module).

Example 15
Consider again S_1, the signature module of Figure A1. The compacted signature module of S_1 is depicted in Figure A3. Notice that S_1 has no redundant arcs to be removed and

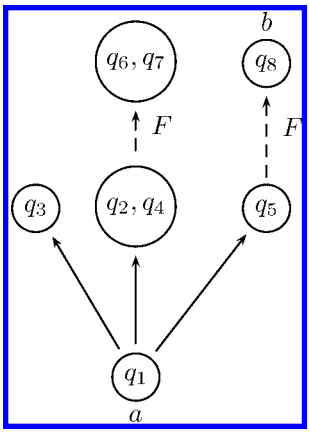

Figure A3
The compacted signature module of S_1.

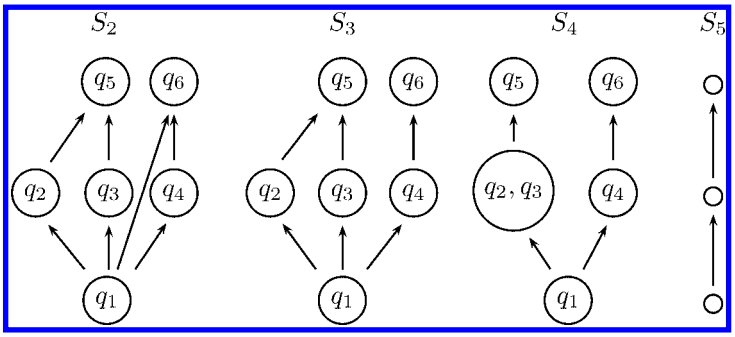

Figure A4
A compactness example.

that q_2 and q_6 were coalesced with q_4 and q_7, respectively. All nodes in $compact(S_1)$ are pairwise distinguishable and no arc is redundant.

Example 16
Consider S_2, S_3, S_4, S_5, the signature modules depicted in Figure A4. Executing the compactness algorithm on S_2, first the redundant subtyping arc from q_1 to q_6 is removed, resulting in S_3 which has no redundant arcs. Then, q_2 and q_3 are coalesced, resulting in S_4. In S_4, $\{q_2, q_3\} \approx \{q_4\}$ and $\{q_5\} \approx \{q_6\}$, and after coalescing these two pairs, the result is S_5 which is non-redundant.

Proposition 4
The compactness algorithm terminates.

Proposition 5
The compactness algorithm is deterministic: it always produces the same result.

Proposition 6
If S is a signature module then $compact(S)$ is a non-redundant signature module.

Proposition 7
If S is a non-redundant signature module then $compact(S) \sim S$.

Acknowledgments

This research was supported by the Israel Science Foundation (grants 136/01, 137/06). We are grateful to Nurit Melnik and Gerald Penn for extensive discussions and constructive feedback, and to the CL reviewers for detailed, useful comments. All remaining errors are, of course, our own.

References

Abeillé, Anne, Marie-Hélène Candito, and Alexandra Kinyon. 2000. FTAG: developing and maintaining a wide-coverage grammar for French. In Erhard Hinrichs, Detmar Meurers, and Shuly Wintner, editors, *Proceedings of the ESSLLI-2000 Workshop on Linguistic Theory and Grammar Implementation*, pages 21–32.

Basili, R., M. T. Pazienza, and F. M. Zanzotto. 2000. Customizable modular lexicalized parsing. In *Proceedings of the Sixth International Workshop on Parsing Technologies (IWPT 2000)*, pages 41–52, Trento.

Bender, Emily M., Dan Flickinger, and Stephan Oepen. 2002. The grammar matrix: An open-source starter-kit for the rapid development of cross-linguistically consistent broad-coverage precision grammars. In *Proceedings of the Workshop on Grammar Engineering and Evaluation at the 19th International Conference on Computational Linguistics*, pages 8–14, Taipei.

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*, pages 203–208, Jeju Island.

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.

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.

Brogi, Antonio, Paolo Mancarella, Dino Pedreschi, and Franco Turini. 1994. Modular logic programming. *ACM Transactions on Programming Languages and Systems*, 16(4):1361–1398.

Brogi, Antonio and Franco Turini. 1995. Fully abstract compositional semantics for an algebra of logic programs. *Theoretical Computer Science*, 149:201–229.

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

Candito, Marie-Hélène. 1996. A principle-based hierarchical representation of LTAGs. In *COLING-96*, pages 194–199, Copenhagen.

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

Carpenter, Bob and Gerald Penn. 2001. ALE—the attribute logic engine: User's guide. Technical report, Department of computer science, University of Toronto and SpeechWorks Research.

Cohen-Sygal, Yael and Shuly Wintner. 2006. Partially specified signatures: A vehicle for grammar modularity. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 145–152, Sydney.

Copestake, Ann. 2002. *Implementing typed feature structures grammars*. CSLI Publications, Stanford, CA.

Copestake, Ann and Dan Flickinger. 2000. An open-source grammar development environment and broad-coverage English grammar using HPSG. In *Proceedings of the Second Conference on Language Resources and Evaluation (LREC-2000)*, pages 591–600.

Crabbé, Benoit and Denys Duchier. 2004. Metagrammar redux. In *Proceedings of The International Workshop on Constraint Solving and Language Processing (CSLP)*, pages 32–47.

Dalrymple, Mary. 2001. *Lexical Functional Grammar*, volume 34 of *Syntax and Semantics*. Academic Press, Oxford.

Debusmann, Ralph. 2006. *Extensible Dependency Grammar: A Modular Grammar Formalism Based On Multigraph Description*. Ph.D. thesis, University of Saarlandes.

Debusmann, Ralph, Denys Duchier, and Andreas Rossberg. 2005. Modular grammar design with typed parametric principles. In *Proceedings of FG-MOL 2005: The 10th Conference on Formal Grammar and The 9th Meeting on Mathematics of Language*, pages 113–122.

Duchier, Denys and Claire Gardent. 1999. A constraint-based treatment of descriptions. In *Third International Workshop on Computational Semantics (IWCS-3)*, pages 71–85.

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.

Fodor, Jerry. 1983. *The Modularity of Mind*. MIT Press, Cambridge, MA.

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, TX.

Garey, Michael R. and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, New York.

Hinrichs, Erhard W., W. Detmar Meurers, and Shuly Wintner. 2004. Linguistic theory and grammar implementation. *Research on Language and Computation*, 2:155–163.

Jackendoff, Ray. 2002. *Foundations of Language*. Oxford University Press, Oxford.

Joshi, Aravind K., Leon S. Levy, and Masako Takahashi. 1975. Tree adjunct grammars. *Journal of Computer and System Sciences*, 10(1):136–163.

Kahane, Sylvain. 2006. Polarized unification grammars. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (COLING-ACL 2006)*, pages 137–144, Sydney.

Kallmeyer, Laura. 2001. Local tree description grammars. *Grammars*, 4(2):85–137.

Kaplan, Ronald M., Tracy Holloway King, and John T. Maxwell. 2002. Adapting existing grammars: The XLE experience. In *COLING-02 Workshop on Grammar Engineering and Evaluation*, pages 1–7, Morristown, NJ.

Kasper, Walter and Hans-Ulrich Krieger. 1996. Modularizing codescriptive grammars for efficient parsing. In *Proceedings of the 16th Conference on Computational Linguistics*, pages 628–633, Copenhagen.

Keselj, Vlado. 2001. Modular HPSG. In *Proceedings of the 2001 IEEE Systems, Man, and Cybernetics Conference*, pages 2867–2872, Tucson, AZ.

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.

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, MA.

Melnik, Nurit. 2006. A constructional approach to verb-initial constructions in modern Hebrew. *Cognitive Linguistics*, 17(2):153–198.

Meurers, W. Detmar, Gerald Penn, and Frank Richter. 2002. A Web-based instructional platform for constraint-based grammar formalisms and parsing. In *Proceedings of the ACL Workshop on Effective Tools and Methodologies for Teaching NLP and CL*, pages 18–25.

Mitchell, John C. 2003. *Concepts in Programming Languages*. Cambridge University Press, Cambridge.

Müller, Stefan. 2007. The Grammix CD ROM. A software collection for developing typed feature structure grammars. In Tracy Holloway King and Emily M. Bender, editors, *Grammar Engineering across Frameworks 2007*, Studies in Computational Linguistics ONLINE. CSLI Publications, Stanford, CA, pages 259–266.

Oepen, Stephan, Dan Flickinger, Hans Uszkoreit, and Jun-Ichi Tsujii. 2000. Introduction to this special issue. *Natural Language Engineering*, 6(1):1–14.

Oepen, Stephan, Daniel Flickinger, J. Tsujii, and Hans Uszkoreit, editors. 2002. *Collaborative Language Engineering: A Case Study in Efficient Grammar-Based Processing*. CSLI Publications, Stanford, CA.

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.

Penn, Gerald B. 2000. *The Algebraic Structure of Attributed Type Signatures*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Perrier, Guy. 2000. Interaction grammars. In *Proceedings of the 18th Conference on Computational Linguistics (COLING 2000)*, pages 600–606.

Pollard, Carl and Ivan A. Sag. 1994. *Head-Driven Phrase Structure Grammar*. University of Chicago Press and CSLI Publications, Stanford, CA.

Ranta, Aarne. 2007. Modular grammar engineering in GF. *Research on Language and Computation*, 5(2):133–158.

Sygal, Yael and Shuly Wintner. 2008. Type signature modules. In Philippe de Groote, editor, *Proceedings of FG 2008: The 13th Conference on Formal Grammar*, pages 113–128.

Sygal, Yael and Shuly Wintner. 2009. Associative grammar combination operators for tree-based grammars. *Journal of Logic, Language and Information*, 18(3):293–316.

Vijay-Shanker, K. 1992. Using descriptions of trees in a tree adjoining grammar. *Computational Linguistics*, 18(4):481–517.

Wintner, Shuly. 2002. Modular context-free grammars. *Grammars*, 5(1):41–63.

Wintner, Shuly, Alon Lavie, and Brian MacWhinney. 2009. Formal grammars of early language. In Orna Grumberg, Michael Kaminski, Shmuel Katz, and Shuly Wintner, editors, *Languages: From Formal to Natural*, volume 5533 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin and Heidelberg, pages 204–227.

XTAG Research Group. 2001. A lexicalized tree adjoining grammar for English. Technical Report IRCS-01-03, IRCS, University of Pennsylvania.

Zajac, Rémi and Jan W. Amtrup. 2000. Modular unification-based parsers. In *Proceedings of the Sixth International Workshop on Parsing Technologies (IWPT 2000)*, pages 278–288, Trento.