Modular Development of Typed Unification Grammars:

A Mathematical and Computational Infrastructure for Grammar Engineering

Yael Sygal

A THESIS SUBMITTED FOR THE DEGREE

"DOCTOR OF PHILOSOPHY"

University of Haifa

Committee of Doctoral Studies

November, 2009

Modular Development of Typed Unification Grammars:

A Mathematical and Computational Infrastructure for Grammar Engineering

By: Yael Sygal

Supervised By: Dr. Shuly Wintner

A THESIS SUBMITTED FOR THE DEGREE

“DOCTOR OF PHILOSOPHY”

University of Haifa

Committee of Doctoral Studies

November, 2009

Recommended by: ____________________________ Date: ________

(Advisor)

Approved by: ____________________________ Date: ________

(Chairman of Ph.D Committee)

Contents

1 Introduction **1**

1.1 Typed Unification Grammars 3

1.2 Motivation 7

1.3 Related Work 10

1.3.1 Modularity in programming languages 10

1.3.2 Initial approaches: modularized parsing 11

1.3.3 Modularity in typed unification grammars 12

1.3.4 Modularity in related formalisms 14

1.4 Contributions of the Thesis 19

2 Modularization of the Signature **21**

2.1 Overview 21

2.2 Signature Modules 22

2.3 Combination Operators for Signature Modules 27

2.3.1 Compactness 28

2.3.2 Merge 30

2.3.3 Attachment 38

2.3.4 Example: parametric lists 43

2.3.5 Example: the ‘addendum’ operator in LKB 45

2.4 Extending Signature Modules to Type Signatures 46

2.5 Grammar Modules 55

3 Modular Construction of the Basic HPSG Signature 58

4 MODALE: A Platform for Modular Development of Signature Modules 65

5 Implications on Other Formalisms 68

5.1 Overview 68

5.2 Tree Combination in PUG 70

5.3 Tree Combination is not Associative 76

5.3.1 Non-Polarized Tree Combination 76

5.3.2 Colors 76

5.3.3 Polarities 77

5.3.4 General Polarity Systems 79

5.3.5 Practical Consequences 81

5.4 From Trees to Forests 83

5.5 (Polarized) Forest Combination is Associative 89

5.6 Forest Combination and XMG 93

5.7 Conclusion 96

6 Discussion and conclusions 98

Bibiliography 102

A Compactness 110

B Name Resolution 119

C Grammar modules 128

C.1 Defining Grammar Modules 128

C.2 Grammar Module Combination 131

C.2.1 Compactness 131
C.2.2 Merge and Attachment 133
C.3 Extending Grammar Modules to Full Type Unification Grammars 134
C.3.1 Appropriateness Consolidation 136

D MODALE Description of the Basic HPSG Grammar of Chapter 3 140
D.1 Modular Design 140
D.2 The Resolved HPSG Signature 146

Modular Development of Typed Unification Grammars:

A Mathematical and Computational Infrastructure for Grammar Engineering

Yael Sygal

Abstract

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, etc.

This work provides the 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 (1994). We then describe the MODALE system, a platform that supports modular development of type signatures. Finally, we show that the methods we propose have an impact on the development of large-scale grammars in some other, related, formalisms.

List of Figures

2.1 A partially specified signature, P_1 24
2.2 A signature module, S_1 26
2.3 A signature module with redundant arcs 30
2.4 Compactness 31
2.5 Merge: intermediate steps 34
2.6 Merge 35
2.7 Merge 36
2.8 BCPO relaxation 37
2.9 Merge of signature modules 37
2.10 Attachment – input 42
2.11 Attachment result, $S_{1a}(S_{9a})$ 43
2.12 Implementing parametric lists with signature modules 44
2.13 Name resolution result for S_6 50
2.14 Appropriateness consolidation: result 52
2.15 Appropriateness consolidation 53

3.1 The main fragments of the signature 59
3.2 A signature module, $Sign$ 59
3.3 Phrase structure 60
3.4 A signature module, *Head* 61
3.5 Signature modules 61

3.6 A classification of nominal objects 62
3.7 Parametric signature modules 62
3.8 HPSG signature construction 63

4.1 MODALE description of S_1 66

5.1 Tree combination 72
5.2 $\{T_1\}\overset{t}{+}\{T_2\}$ 73
5.3 $\{T_2\}\overset{t}{+}\{T_3\}$ 74
5.4 Non-polarized tree combination 77
5.5 Color combination table 77
5.6 Tree combination with colors 78
5.7 PUG polarity systems 79
5.8 Tree combination with polarities 79
5.9 Candidate solutions for PUG tree combination 81
5.10 Length-3 paths solutions 82
5.11 Two forests to be combined 86
5.12 Legal and illegal combinations of F_1, F_2 86
5.13 A forest combination of F_1 and F_2 86
5.14 Intermediate results 88
5.15 Forest combination of F_9, F_{10} and F_{11}: $(\{F_9\}\overset{f}{+}\{F_{10}\})\overset{f}{+}\{F_{11}\} \cong \{F_9\}\overset{f}{+}(\{F_{10}\}\overset{f}{+}\{F_{11}\})$ 89

A.1 A signature module with indistinguishable nodes, S_1 112
A.2 Strict restriction subgraphs 113
A.3 The compacted signature module of S_1 116
A.4 A compactness example 117

B.1 Quasi-indistinguishability versus indistinguishability 122

B.2 A signature module with a non transitive '$\overset{q}{\approx}$' relation 123

B.3 Name resolution 125

B.4 Name resolution 126

C.1 Grammar appropriateness consolidation 139

Chapter 1

Introduction

Development of large-scale grammars for natural languages is an active area of research in human language technology. Such grammars are developed not only for purposes of theoretical linguistic research, but also for natural language applications such as machine translation, speech generation, etc. Wide-coverage grammars are being developed for various languages (Abeillé, Candito, and Kinyon, 2000; XTAG Research 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, e.g., 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 includes 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). Yet while 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, 1992a), 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; etc. This is the main issue that we address in this work.

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 1.2, and then survey related work in section 1.3, 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 chapter 2 we define *signature modules* and provide operators of *module combination*. Modules may specify partial information about the components of the signature and may communicate through parameters, similarly to function composition. We then show 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 2.5. In chapter 3 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 4). In chapter 5 we show that the methods we develop

are instrumental also for development of large-scale grammars in some other, related, formalisms. In particular, we use our methods to identify and correct a significant flaw in an otherwise powerful and flexible formalism, PUG. In chapter 6 we show how our solution complies with the desiderata of section 1.2, and conclude with directions for future research.

1.1 Typed Unification Grammars

We assume familiarity with theories of (typed) unification grammar, as formulated by, e.g., Carpenter (1992b) 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 y and assign 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 hence it is denoted $min(S)$.

Definition 2. *A partially ordered set $\langle P, \leq \rangle$ is a* **bounded complete partial order** *(BCPO) if 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 $\perp$), since 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 (1992b) and Penn (2000) in viewing subtypes as greater than their supertypes (hence the least element $\perp$ 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.*

In this work we restrict ourselves to standard type signatures (as defined by Carpenter (1992b) 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.

For the following discussion we assume that a type signature $\langle\textsc{Type}, \sqsubseteq, \textsc{Feat}, Approp\rangle$ has been specified.

Definition 7. *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 8. *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.*
- $\Theta : \Pi \to \textsc{Type}$ *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* $\Theta(\pi_1) = \Theta(\pi_2)$

Below, the meta-variable t ranges over types, F – over features and π, α (with or without subscripts) range over paths. A, B (with or without subscripts) range over typed feature structures and $\Pi, \Theta, \bowtie$ (with the same subscripts) over their constituents. Let TFSs be the set of all typed feature structures.

Definition 9. *A* TFS $A = \langle \Pi, \Theta, \bowtie \rangle$ *is* **well-typed** *if whenever* $\pi \in \Pi$ *and* $F \in \textsc{Feat}$ *are such that* $\pi F \in \Pi$*, then* $Approp(\Theta(\pi), F)\downarrow$*, and* $Approp(\Theta(\pi), F) \sqsubseteq \Theta(\pi F)$.

To be able to represent complex linguistic information, such as phrase structure, the notion of feature structures is extended into multi-rooted feature structures.

Definition 10. *A* **typed pre-multi-rooted structure** *(pre-*TMRS*) is a quadruple* $\sigma = \langle Ind, \Pi, \Theta, \bowtie \rangle$*, where:*

- $Ind \in \mathbb{N}$ *is the number of* ***indices*** *of* σ

- $\Pi \subseteq \{1, 2, \ldots, Ind\} \times$ PATHS *is a set of* ***indexed paths****, such that for each* i*,* $1 \leq i \leq Ind$*, there exists some* $\pi \in$ PATHS *with* $(i, \pi) \in \Pi$

- $\Theta : \Pi \to$ TYPE *is a total function, assigning a type for all paths.*

- $\bowtie \subseteq \Pi \times \Pi$ *is a relation specifying reentrancy*

A **typed multi-rooted structure** *(*TMRS*) is a pre-*TMRSσ *for which the following requirements, naturally extending those of* TFSS*, hold:*

- Π *is prefix-closed: if* $\langle i, \pi\alpha\rangle \in \Pi$ *then* $\langle i, \pi\rangle \in \Pi$

- σ *is fusion-closed: if* $\langle i, \pi\alpha\rangle \in \Pi$ *and* $\langle i, \pi\rangle \bowtie \langle i', \pi'\rangle$ *then* $\langle i', \pi'\alpha\rangle \in \Pi$ *and* $\langle i, \pi\alpha\rangle \bowtie \langle i', \pi'\alpha\rangle$

- $\bowtie$ *is an equivalence relation with a finite index (with* $[\bowtie]$ *the set of its equivalence classes) including at least the pairs* $(\langle i, \epsilon\rangle, \langle i, \epsilon\rangle)$ *for all* $1 \leq i \leq Ind_\sigma$

- Θ *respects the equivalence: if* $\langle i_1, \pi_1\rangle \bowtie \langle i_2, \pi_2\rangle$ *then* $\Theta(\langle i_1, \pi_1\rangle) = \Theta(\langle i_2, \pi_2\rangle)$

The **length** *of a* TMRS σ*, denoted* $|\sigma|$*, is* Ind_σ*.*

Meta-variables σ, ρ, ξ range over TMRSS, and $Ind, \Pi, \Theta, \bowtie$ over their constituents. Let TMRSS be the set of all typed feature structures.

Definition 11. *A* TMRSσ *is* **well-typed***, if for all* i*,* $1 \leq i \leq Ind_\sigma$*,* σ^i *is well-typed.*

Rules and grammars are defined over an additional parameter, a fixed, finite set WORDS of words (in addition to the parameters FEAT and TYPE).

Definition 12. *Let* S *be a type signature. A* **rule** *over* S *is a well-typed* TMRS*of length greater than or equal to 1 with a designated (first) element, the* **head** *of the rule. The rest of the elements form the rule's* **body** *(which may be empty, in which case the rule is*

viewed as a TFS*). A* **lexicon** *is a total function from* WORDS *to finite, possibly empty sets of well-typed* TFSS*. A* **grammar** $G = \langle \mathcal{R}, \mathcal{L}, \mathcal{A} \rangle$ *is a finite set of rules* $\mathcal{R}$*, a lexicon* $\mathcal{L}$ *and a finite set of well-typed* TFSS*,* $\mathcal{A}$*, which is the set of* **start symbols***.*

1.2 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 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, p. 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). Similarly, Bugliesi, Lamma, and Mello (1994) 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, pp. 218-230). While 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, while 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 and Oepen, 2002),[1] 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

[1] 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.[2]

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

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

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

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.[3] 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, e.g., 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.

1.3 Related Work

1.3.1 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.

[3]The examples are inspired by actual grammars but are obviously much simplified.

Modular interfaces of logic programs were first suggested by O'keefe (1985) and by 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 2.3.2 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).

1.3.2 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 (1996) 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.

1.3.3 Modularity in typed unification grammars

Keselj (2001) presents a modular 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 which 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 one which concretely defines 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 2.3.2). 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 1.2.

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 since 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 frag-

ments, 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.

1.3.4 Modularity in related formalisms

The above works emphasize the fact that existing approaches to modular grammar development in the area of unification grammars are still insufficient. The same problem has been addressed also 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 above (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 trees (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 (depending on the formalism). The set of minimal trees that satisfy the resulting descriptions are the TAG elementary trees. In this way modular construction of grammars is supported, where a module is merely a tree description and modules are combined by means of the control tree logic.

When trees are semantic objects, 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. In the meta-grammar paradigm, where grammar fragments are tree *descriptions*, Candito (1996) associates with each node in a description a name, such that nodes with the same name must denote the same entity and therefore must be identified. The names of nodes are thus the only channel of interaction between two descriptions. Furthermore, these names can only be used to identify two nodes, but not to set nodes apart. Crabbé and Duchier (2004) propose to replace node naming by a *coloring* scheme, where nodes are colored black, white or red. When two trees are combined, a black node may be unified with zero, one or more white nodes and produce a black node; a white node must be unified with a black one producing a black node; and a red node cannot be unified with any other node. Then, a satisfying model must be *saturated*, i.e., one in which all the nodes are either black or red. In this way some combinations can be forced and others prevented.

This mechanism is extended in *Interaction Grammars* (Perrier, 2000), where each node is decorated by a set of *polarity features*. A polarity feature consists of a feature, arbitrarily determined by the grammar writer, and a polarity, which can be either positive, negative or neutral. A positive value represents an available resource and a negative value represents an expected resource. Two feature-polarity pairs can combine only if their feature is identical and their polarities are opposite (i.e., one is negative and the other is positive); the result is a feature-polarity pair consisting of the same feature and the neutral polarity. Two nodes can be identified only if their polarity features can combine. A solution is a tree whose features are all neutralized.

The concept of polarities is further elaborated in *Polarized Unification Grammars* (PUG, Kahane (2006)). A PUG is defined over a *system of polarities* $(P, \cdot)$ where P is a set (of polarities) and '$\cdot$' is an associative and commutative product over P. A PUG generates a set of finite structures over objects which are determined for each grammar separately. The objects are associated with polarities, and structures are combined by identifying some of their objects. The combination is sanctioned by polarities: objects can only be identified if their polarities are unifiable; the resulting object has the unified polarity. A non-empty, strict subset of the set of polarities, called the set of *neutral* polarities, determines which of the resulting structures are valid: A polarized structure is *saturated* if all its polarities are neutral. The structures that are generated by the grammar are the saturated structures that result from combining different structures.

PUGs are more general than the mechanisms of polarity features and coloring, since they allow the grammar designer to decide on the system of polarities, whereas other systems pre-define it.

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 as we show that some earlier approaches do not adhere to this desideratum (see section 5.3).

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 (2007) 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, e.g., 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 1.2 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.

1.4 Contributions of the Thesis

The main objective of this work was to provide the foundations for modular construction of (typed) unification grammars for natural languages. To this end we first had to carefully explore existing grammars and investigate proposals for grammar modularization in unification grammars and in other, related formalisms. The main contributions of these endeavors are:

- Introduction of a set of desiderata for a beneficial solution for grammar modularization (section 1.2).
- Introduction of a thorough, well-founded solution to the problem of modular construction of typed unification grammars for natural languages (chapter 2).
- Development of a *powerset-lift* method to maintain associativity in non-associative formalisms (chapter 2).
- Presentation of a modular design of the traditional HPSG grammar of Pollard and Sag (1994) (chapter 3).
- Formalization of the main combination operation of PUG and identification and correction of a significant flaw in this formalism (chapter 5).
- Development of an extension of ALE (Carpenter, 1992a) and TRALE (Meurers, Penn, and Richter, 2002) that provides a description language with which signature modules can be specified, and the two combination operations can be applied. Expressions of the language are compiled into full TRALE signatures.

The set of desiderata (section 1.2), a definition of non-parametric signature modules (sections 2.2 and 2.3.1), a first combination operator (*merge*, section 2.3.2) and the resolution stage (section 2.4) were presented in Cohen-Sygal and Wintner (2006). An extended set of desiderata (section 1.2), parametric signature modules (sections 2.2) and a second

combination operator (*attachment*, section 2.3.3) were presented in Sygal and Wintner (2008). A more detailed and complete presentation that includes also an extension to grammar modules and the modular design of the traditional HPSG grammar (covering chapters 2 and 3) is under review for a major journal.

The formalization of the PUG combination operation (section 5.2) as well as the identification of a flaw in this formalism (section 5.3) were presented in Cohen-Sygal and Wintner (2007). The above material along with the correction of the flaw in this formalism through the powerset-lift method, and implications of these results to XMG, were presented in Sygal and Wintner (2009) (covering the material of chapter 5).

Chapter 2

Modularization of the Signature

2.1 Overview

We define *signature modules* (also referred to as *modules* below), 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[1] 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).

[1]Subtyping is sometimes referred to in the literature as *type subsumption*.

We begin by defining the basic structure of signature modules in section 2.2. We then introduce (section 2.3) 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 2.4).

2.2 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, e.g., subtyping and appropriateness relations, types etc. 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 13. *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. $Ap \subseteq Q \times$ 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 '$\stackrel{*}{\preceq}$', is antisymmetric.*

7. *(Relaxed Upward Closure) for all $q_1, q'_1, q_2 \in Q$ and $F \in$ FEAT, if $(q_1, F, q_2) \in Ap$ and $q_1 \stackrel{*}{\preceq} q'_1$, then there exists $q'_2 \in Q$ such that $q_2 \stackrel{*}{\preceq} q'_2$ 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) since 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 '$\stackrel{*}{\preceq}$' 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 2.4); any intermediate result can be a general partial order. Relaxing the BCPO requirement also helps guaranteeing the associativity of module combination (see example 8).

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). An Ap-arc (q, F, q') in a module is interpreted as if that module is saying "the appropriate value of q and F should be at least q'" and 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 interpretation of multiple Ap-arcs will be further motivated when module combination is discussed (section 2.3.2). 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 2.4). 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, results in more flexibility for the grammar designer; the condition is restored after all modules combine, see section 2.4.

Example 1. *A simple PSS* P_1 *is depicted in Figure 2.1, 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.*

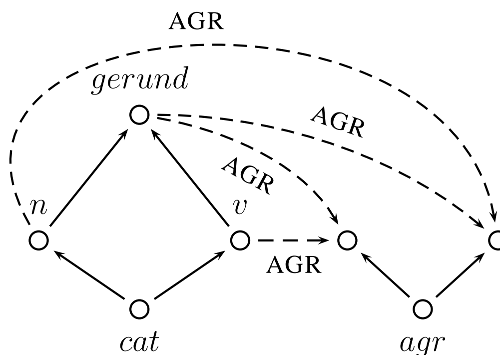


Figure 2.1: A partially specified signature, P_1

Definition 14. *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 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. Since 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 below.[2] In the examples, the classification of nodes is encoded graphically as follows:

Example 2. *Figure 2.2 depicts a module S_1, based on the PSS of Figure 2.1. $S_1 = \langle P_1, Int_1, Imp_1, Exp_1 \rangle$, where P_1 is the PSS of Figure 2.1, $Int_1 = \emptyset$, $Imp_1 = \{q_4, q_5\}$ and $Exp_1 = \emptyset$.*

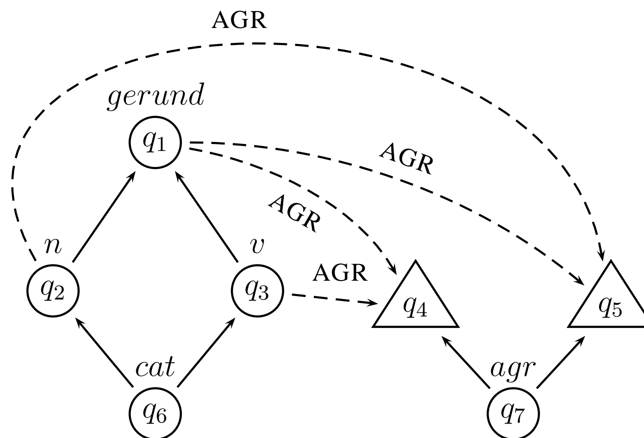


Figure 2.2: A signature module, S_1

Below, 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]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 19 and 20 .

2.3 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 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 15. *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. $\{T_1(q) \mid q \in Int_1\} \cap \{T_2(q) \mid q \in Q_2 \textit{ and } T_2(q)\downarrow\} = \emptyset$
2. $\{T_2(q) \mid q \in Int_2\} \cap \{T_1(q) \mid q \in Q_1 \textit{ and } T_1(q)\downarrow\} = \emptyset$

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.

2.3.1 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*, i.e., 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 $q_2 \overset{*}{\preceq} q_3$ 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. Since $q_2 \overset{*}{\preceq} q_3$, (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, since 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 Appendix A. We do provide two simple examples to illustrate the general idea.

Example 3. *Consider the signature module of Figure 2.3.* (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 2.4. Note that* S_2 *is not a signature module (since it includes two nodes labeled by* a*) and that compactness*

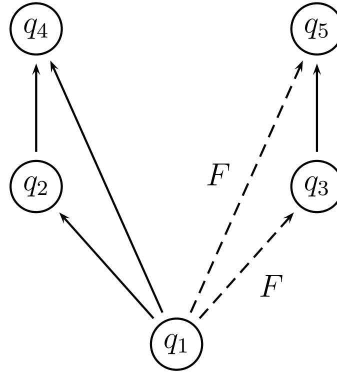


Figure 2.3: A signature module with redundant arcs

is defined over pre-signature modules rather then 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, since these are two pairs of anonymous nodes with isomorphic environments.* q_5 *is not coalesced with* q_3 *and* q_4 *since* 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 *since they are distinguishable:* q_8 *has a supertype marked by* a *while* q_6 *and* q_7 *have anonymous supertypes.*

2.3.2 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 defini-

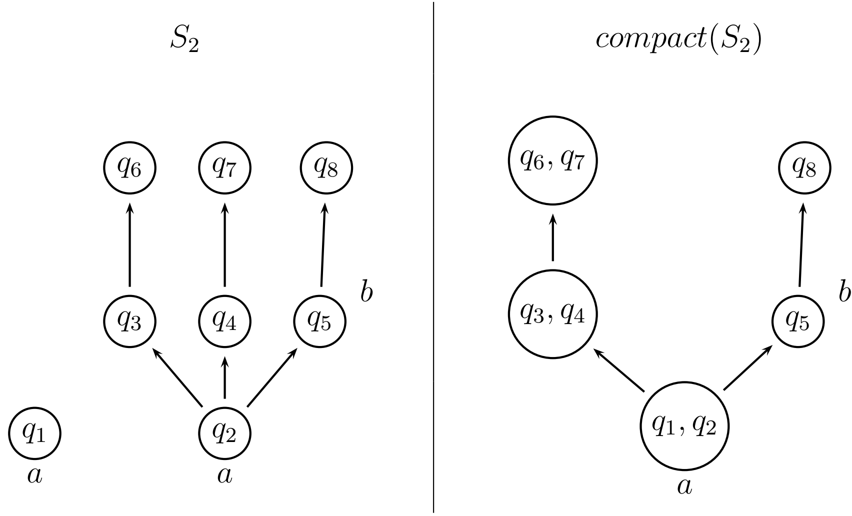


Figure 2.4: Compactness

tion 16). 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 17. 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 18).

Definition 16. *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* $S = \langle\langle Q_1 \cup Q_2, T_1 \cup T_2, \preceq_1 \cup \preceq_2, Ap_1 \cup Ap_2\rangle, Int_1 \cup Int_2, Imp_1 \cdot Imp_2, Exp_1 \cdot Exp_2\rangle$ *('$\cdot$' is the concatenation operator).*

Definition 17. *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* $q_1' \overset{*}{\preceq} q_1$ *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.

Definition 18. *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. $q_1 \overset{*}{\preceq}_1 q_2$ *and* $q_3 \overset{*}{\preceq}_2 q_4$

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, since that would result in

a non-associative combination operation. Anonymous nodes are assigned types only after all modules combine, see section 2.4.

Consider the merge of modules with respect to Ap-arcs: when two modules are merged, Ap-arcs from the two different modules are gathered; 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). Given two Ap-arcs, (q, F, q_1) and (q, F, q_2), 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 $Ap(q, F, q')$. A different approach for multiple Ap-arcs would be to take the *disjunction* of the two, thus requiring that at least one of the statements hold. However, taking the disjunction would imply that the combined module may ignore a requirement made by one of its arguments, which seems unreasonable.

Example 5. *Let S_3 and S_4 be the signature modules depicted in Figure 2.5. $S_3 \Cup S_4$ 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 2.6 depicts a naïve agreement module, S_5. Combined with S_1 of Figure 2.1, $S_1 \Cup S_5 = S_5 \Cup S_1 = S_6$. 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.

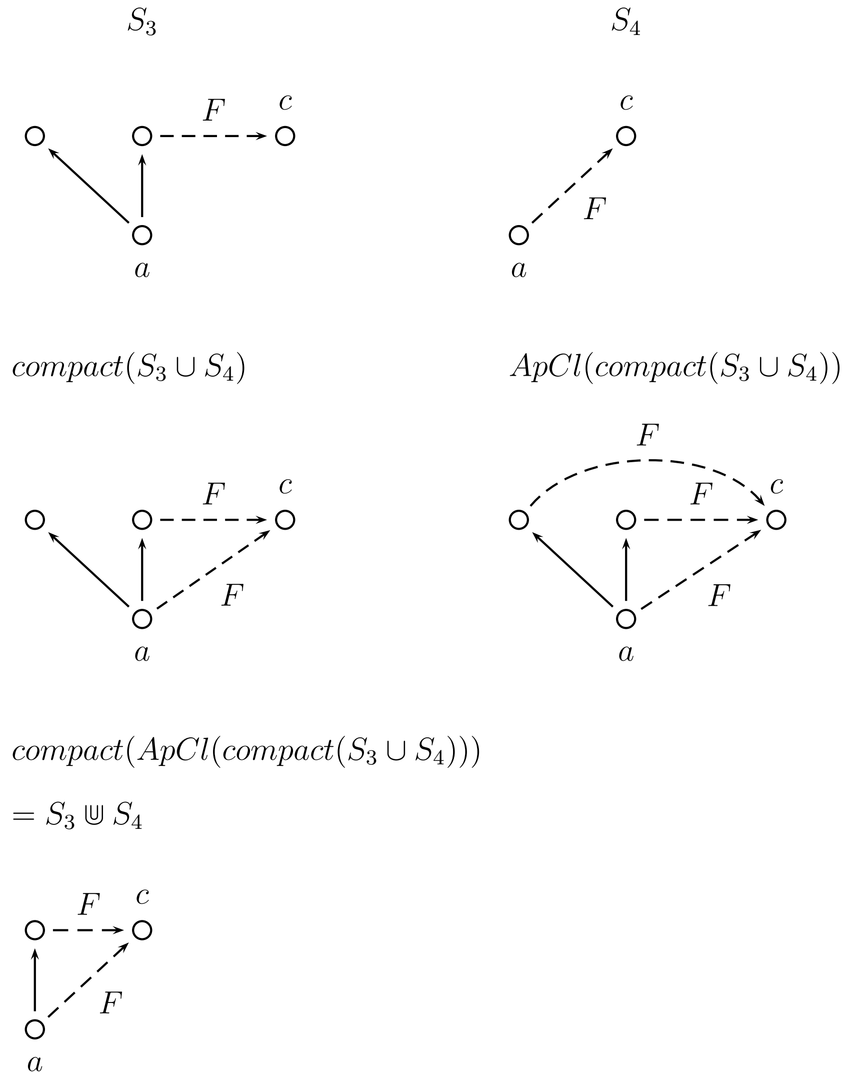


Figure 2.5: Merge: intermediate steps

Example 7. *Let S_7 and S_8 be the signature modules depicted in Figure 2.7. S_7 includes general agreement information while 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, while 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 2.4).

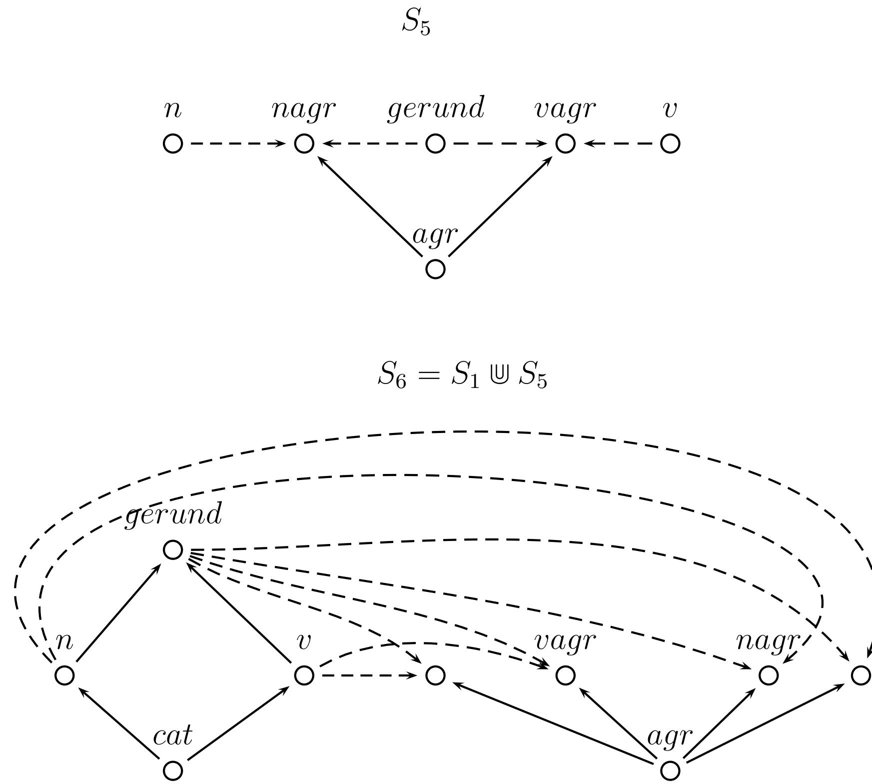


Figure 2.6: Merge

Example 8. *Let S_{10}, S_{11}, S_{12} be the signature modules depicted in Figure 2.8. 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 2.9. In S_{13} the appropriate value for a and F is b while 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). While 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 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.*

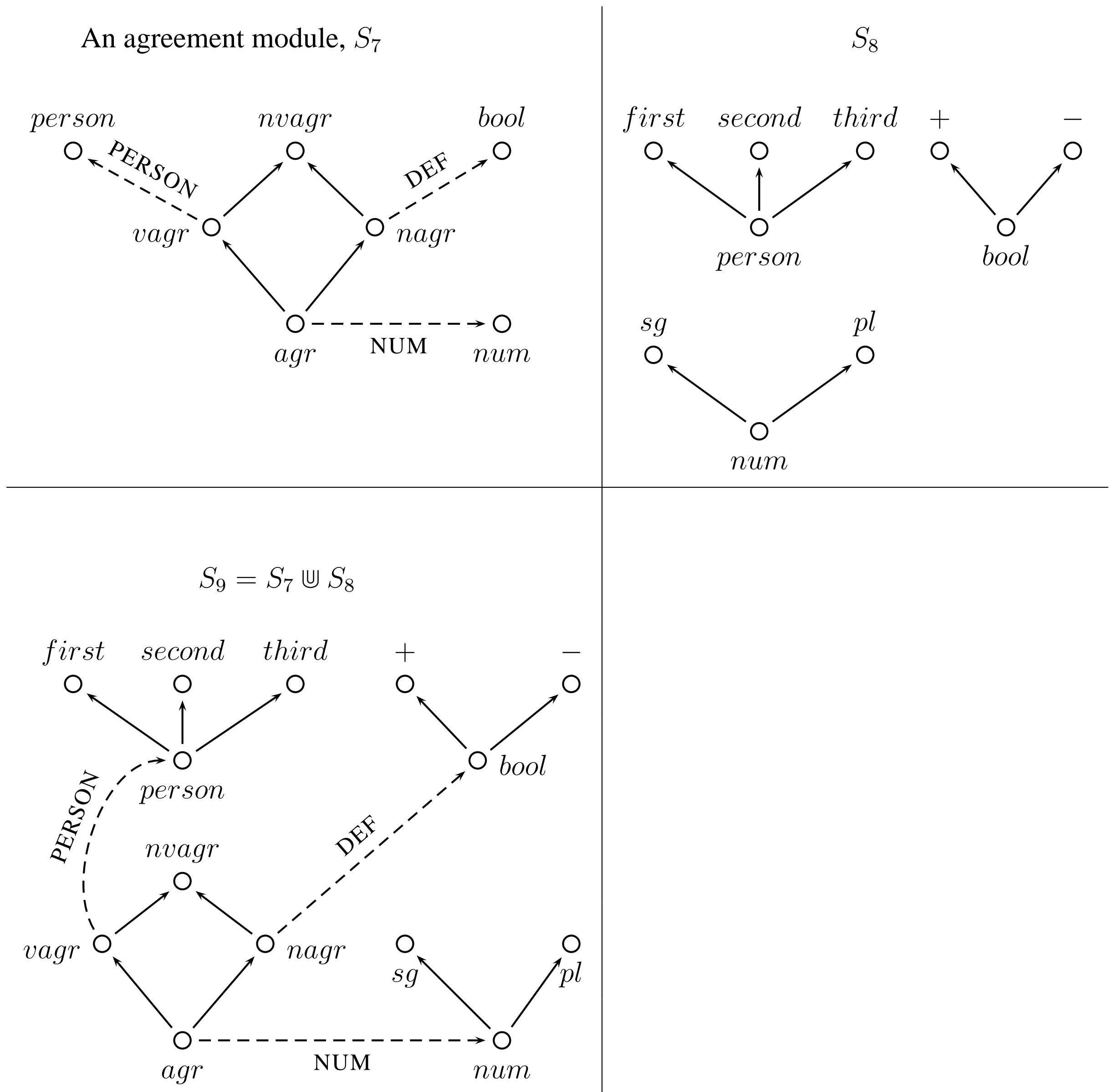


Figure 2.7: Merge

Theorem 1. *Given two mergeable signature modules S_1, S_2, $S_1 \Cup S_2$ is a signature module.*

Proof. Let S_1, S_2 be two mergeable signature modules. Evidently, $S_1 \cup S_2$ is a pre-signature module and therefore so is $S_1 \Cup S_2$. Compactness guarantees that the node-marking function of $S_1 \Cup S_2$ is one to one and that the subtyping relation maintains condition 6 of the definition of a PSS. The relaxed upward closure condition is guaranteed by

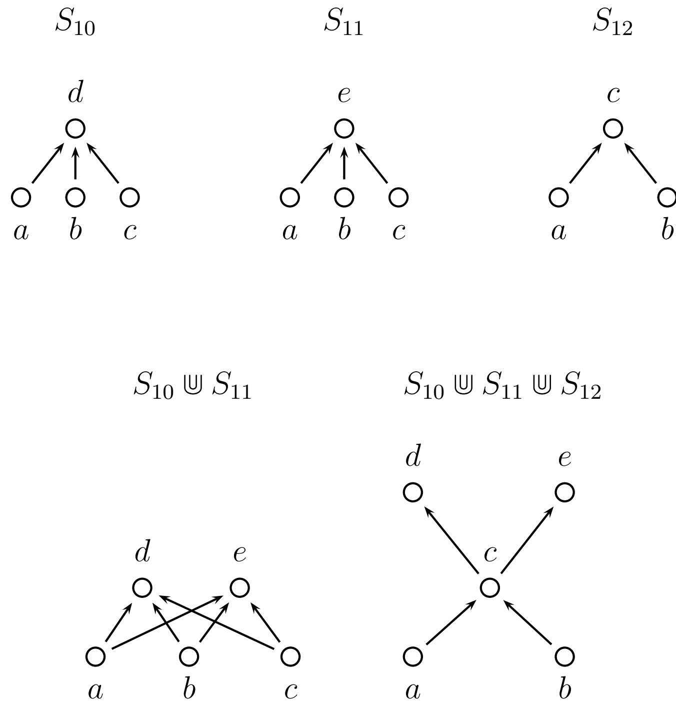


Figure 2.8: BCPO relaxation

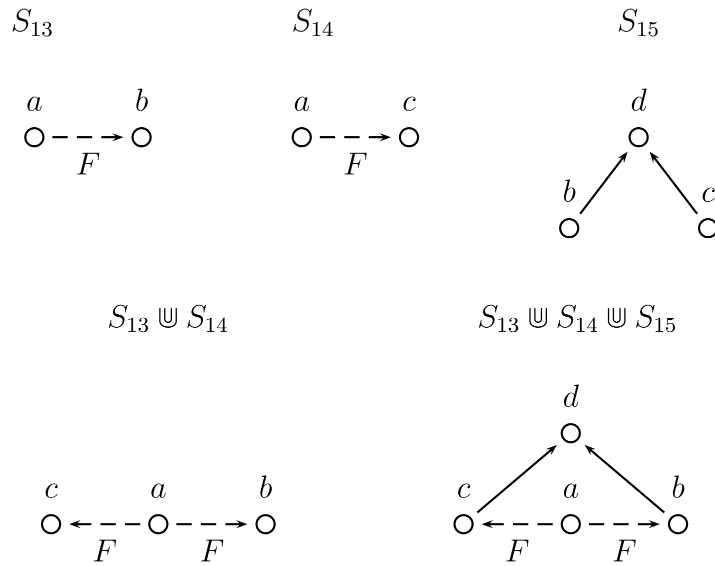


Figure 2.9: Merge of signature modules

the $ApCl$ operation. Evidently, $S_1 \Cup S_2$ is a signature module. □

Theorem 2. *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.

Theorem 3. *Merge is associative up to isomorphism:*[3] *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* $P \sim P'$*.*

The proof of associativity is similar in spirit to the proof of the associativity of (polarized) forest combination (section 5.5) and is therefore suppressed.

2.3.3 Attachment

Consider again S_1 and S_9, the signature modules of Figures 2.1 and 2.7, 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

[3]For the definition of isomorphism see definition 53, Appendix A.

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 19. *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.[4] 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'

[4]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.

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 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.[5]

Definition 20. *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:*

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

[5]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.

- *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.

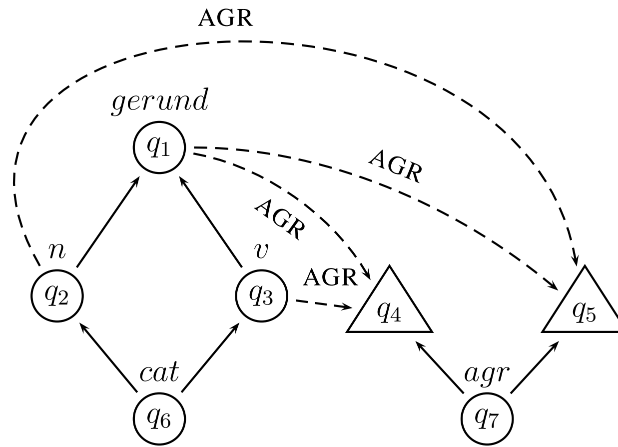
The imported and exported nodes of the resulting module are the equivalence classes of the imported and exported nodes of the first module, S_1, respectively. The nodes of S_2 which are neither internal nor exported are classified as external nodes in the resulting module. This asymmetric view of nodes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

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 2.1 and 2.7, respectively. Let S_{1a} and S_{9a} be the signature modules of Figure 2.10 (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 Figure 2.11. 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.

S_{1a} :



S_{9a} :

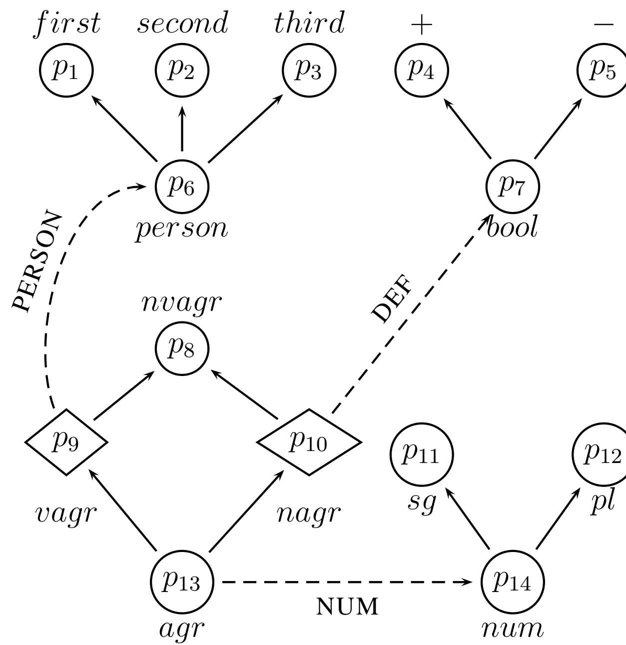


Figure 2.10: Attachment – input

Theorem 4. *Given two signature modules, S_1, S_2 such that S_2 can be attached to S_1, $S_1(S_2)$ is a signature module.*

Proof. Similar to the proof of theorem 1. □

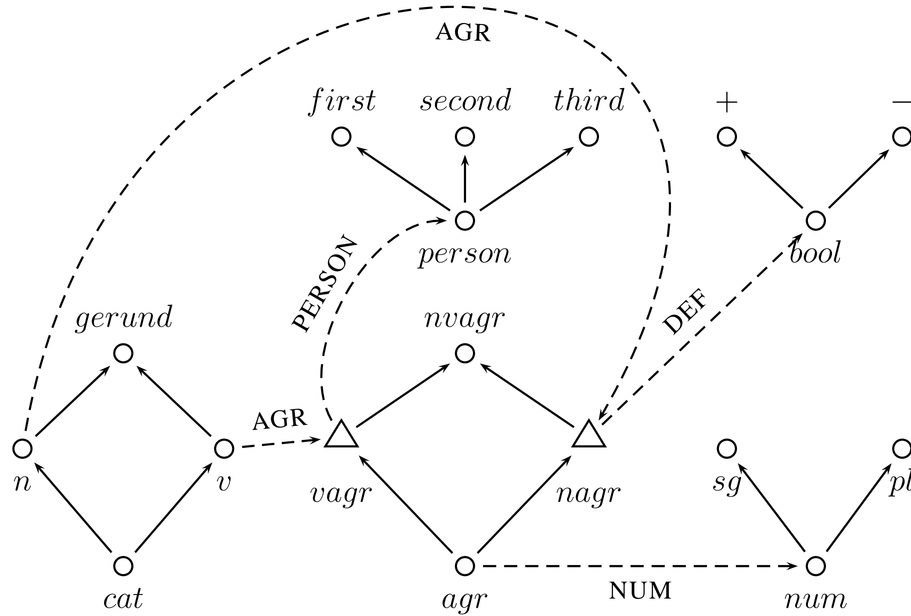


Figure 2.11: Attachment result, $S_{1a}(S_{9a})$

2.3.4 Example: parametric lists

Lists and parametric lists are extensively used in typed unification based formalisms, e.g., HPSG. The mathematical foundations for parametric lists were established by Penn (2000). 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 2.12. 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 below. 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 $Struct$.

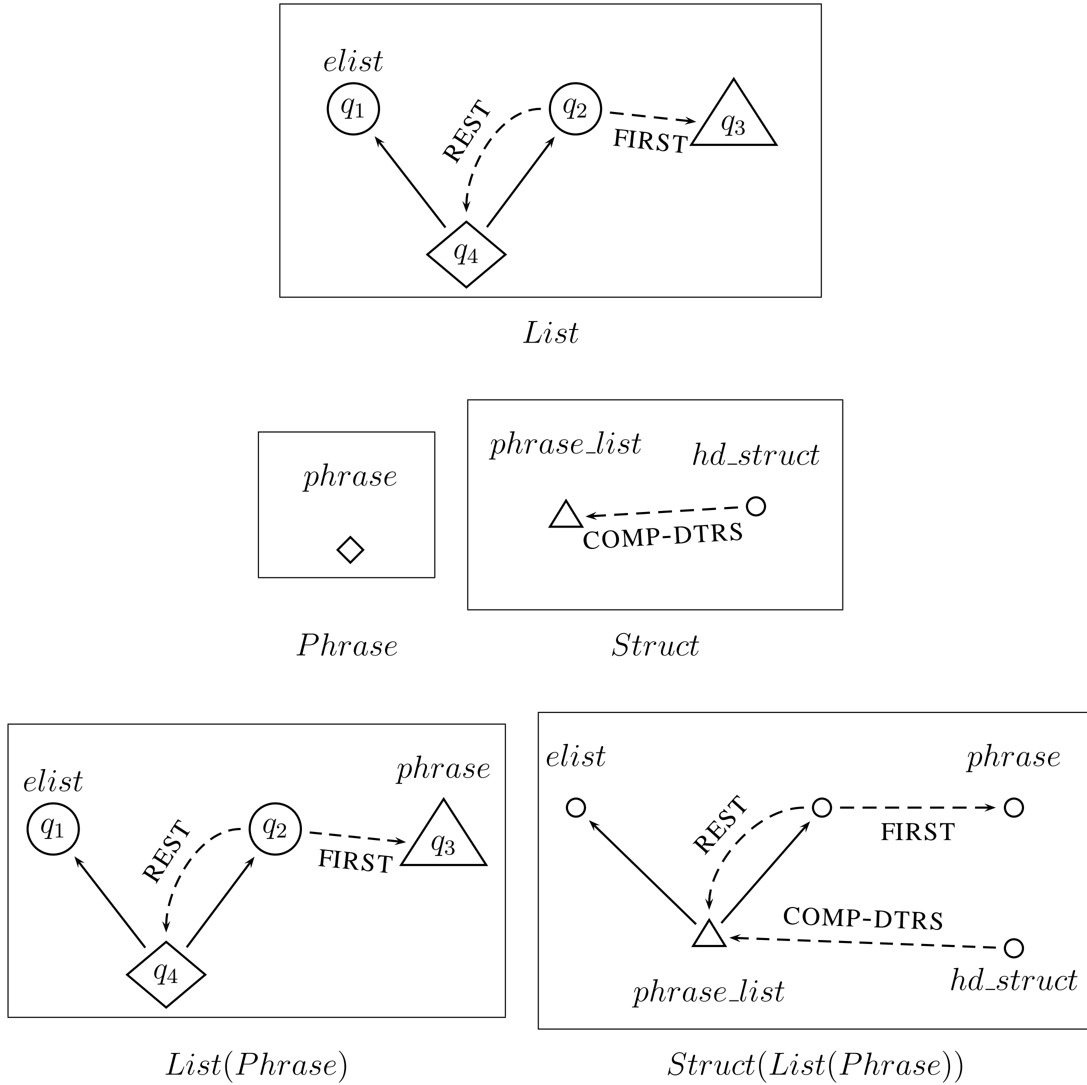


Figure 2.12: Implementing parametric lists with signature modules

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, e.g., only with nodes whose appropriate value for the feature FIRST is a node typed 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, e.g., 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 $elist$. Since 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 (2000) 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.

2.3.5 Example: the 'addendum' operator in LKB

The 'addendum' operator[6] 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 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

[6] See http://depts.washington.edu/uwcl/twiki/bin/view.cgi/Main/TypeAddendum

attachment, where attributes can be assigned to a specific node, even without specifying its type.

2.4 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 2.4).

2. *Appropriateness consolidation*: this algorithm determinizes Ap, converts it from a relation to a function and enforces upward closure (section 2.4).

3. *Feature introduction completion*: this algorithm 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).

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. $S := BCPO\text{-}Completion(S)$

3. $S := ApCl(S)$

4. $S := ApConsolidate(S)$

5. $S := FeatureIntroductionCompletion(S)$

6. $S := BCPO\text{-}Completion(S)$

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 (2000) 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 (2000) 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, since 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., $Int = Imp = Exp = \emptyset$).

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 the above 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 21. *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:*

- $Q_1 = Q\backslash\{q\}$

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

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* $Q_q = \{q' \in Q \mid T(q')\downarrow$ *and* q' *is equivalent to* $q\}$.
2. *let* $\overline{Q} = \{q \in Q \mid T(q)\uparrow$ *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 deferred to Appendix B.

Example 11. *Consider the signature module S_6 depicted in Figure 2.6. Executing the name resolution algorithm on this module results in the signature module of Figure 2.13 (*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. Cf. Figure 2.1, in particular how two anonymous nodes in S_1 are assigned types from S_5 (Figure 2.6).*

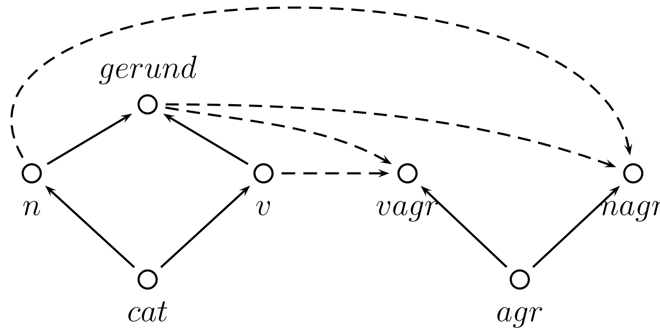


Figure 2.13: Name resolution result for S_6

A more detailed account of the name resolution algorithm is given in Appendix B (along with the technicality of the calculation of the equivalent typed node for a given anonymous node).

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 and whose subtyping relation is a BCPO; its output is a signature module whose typing function is total, whose subtyping relation is a BCPO, and whose ap-

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

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* $q' \overset{*}{\preceq} q$, $|target(q', F)| \leq 1$ *(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. *Otherwise, if* $target(q, F)$ *has no lub, then:*
   (a) *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 below). 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 2.13. Executing the appropriateness consolidation algorithm on this module results in the module depicted in Figure 2.14.*

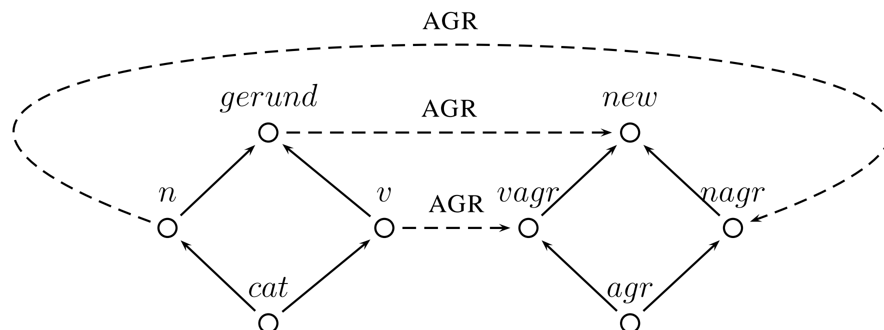


Figure 2.14: Appropriateness consolidation: result

Example 13. *Consider the signature modules depicted in figure 2.15. 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, $new1$, which is the lub of b and c. During this first iteration, $new1$ 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 this interpretation by means of the subtyping arc that is added from $new1$ 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, $new2$, which is the lub of e and f. The result of these operations is S_{18}, which is also the final result.*

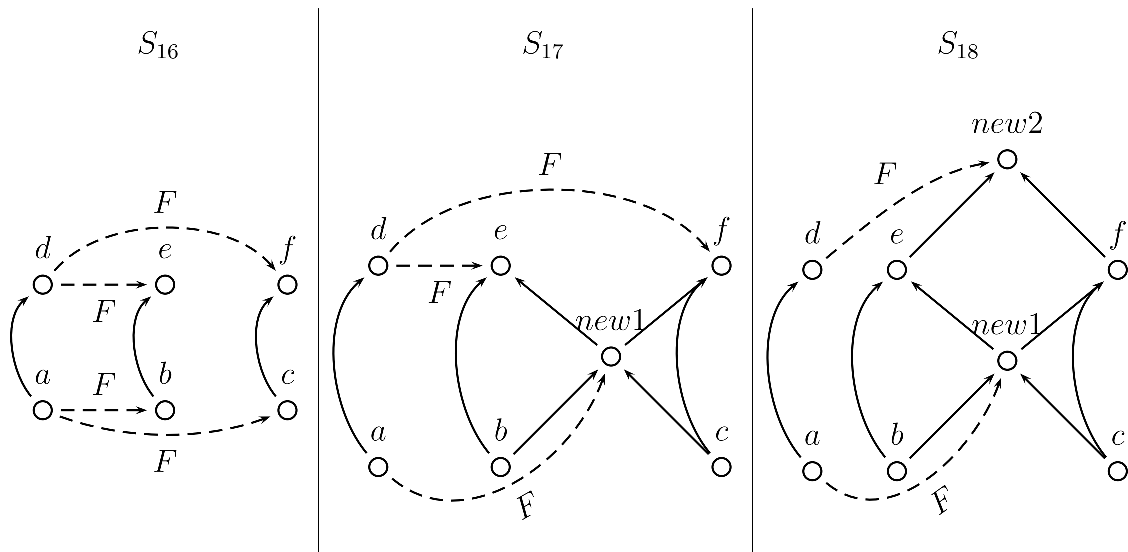


Figure 2.15: Appropriateness consolidation

A naive solution for determinizing Ap would be to simply add a new lub node to all non-empty subsets of Q (evidently there is a finite number of them). The Ap-Consolidation algorithm we present adds lubs only when they are needed. The termination of the algorithm clearly stems from this fact. Furthermore, since the algorithm is executed after BCPO-completion, it adds new elements only as lubs of subsets which have no common upper bound.

Corollary 5. *The appropriateness consolidation algorithm terminates.*

Theorem 6. *Let* $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ *be a signature module where* T *is total and* $\overset{*}{\preceq}$ *is a BCPO. Let* $S_1 = ApConsolidate(S) = \langle\langle Q_1, T_1, \preceq_1, Ap_1\rangle, Int_1, Imp_1, Exp_1\rangle$*. Then* S_1 *is a signature module where* T_1 *is total,* $\overset{*}{\preceq}_1$ *is a BCPO and* Ap_1 *is a function that maintains upward closure.*

Proof. Let $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ be a signature module whose typing function, T, is total and $\overset{*}{\preceq}$ is a BCPO. Each iteration of the appropriateness consolidation algorithm does not change the type assignment of typed nodes. If a new node is added (stage 4), it is assigned a fresh type. Hence, if T is total, so is T_1.

Now, define $Q_{done} = \{q \in Q \mid$ for every feature F, $|target(q, F)| \leq 1$ and for all $q' \in Q$ such that $q' \overset{*}{\preceq} q$, $|target(q', F)| \leq 1\}$. Since S is a signature module, it maintains the relaxed upward condition. Observe that at stage 2 of each iteration, $S|_{Q_{done}}$ (the restriction of S to Q_{done}, see definition 61, page 120) is a signature module whose appropriateness relation is a function that maintains upward closure (the technical proof is suppressed). From theorem 5 it follows that the appropriateness consolidation algorithm terminates and it terminates when $Q_{done} = Q$. When $Q_{done} = Q$, $S|_{Q_{done}} = S|_Q = S$ and therefore S_1 is a signature module whose appropriateness relation is a function that maintains upward closure.

The Ap-consolidation algorithm affects the subtyping relation only in stages 3 and 4: Ap-closure does not affect the subtyping relation and since the typing function is total and the input is a signature module, the only affect of compactness (stage 6) is removement of redundant Ap-arcs. The addition of a new type (stage 4) and addition of subtyping arcs (stages 3 and 4) are done in exactly the same way as in the BCPO completion algorithm of Penn (2000). The proof that these additions maintain bounded completeness is the same as the proof of the correctness of the BCPO completion in Penn (2000). □

Theorem 7. *If* S *is a signature module whose appropriateness relation is a function, then the underlying PSSs of* S *and* $ApConsolidate(S)$ *are equal.*

Proof. Follows immediately from the fact that if S is a signature module whose appropriateness relation is a function then the appropriateness consolidation algorithm terminates at stage 2 of the first iteration. □

To maintain the associativity of signature modules combination we used a method of *powerset-lift*: In contrast to type signatures, the Ap relation of signature modules 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. In this way, each module may specify its own appropriate values without needing to know the value specification of other modules. When two modules are combined (in either one of the two combination operations), multiple outgoing Ap-arcs are preserved and are not replaced by a single arc in order to maintain the associativity of the combination (see example 9). Only in the resolution stage is Ap determinized, converted from a relation to a function. The method we use here is a *powerset-lift* of the domain and the corresponding operation. In this way, all the possibilities are 'remembered' and a resolution stage is added to produce the desired result. This method is a general method which is also applicable to some other, related, formalisms; we show in chapter 5 that it can be used to guarantee the associativity of module combination in PUG.

2.5 Grammar Modules

A **grammar** (definition 12) is defined over a concrete type signature and is a structure including a set of rules (each a TMRS), a lexicon mapping words to sets of TFSs and a start symbol which is a TFS. 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 as-

sociated with a type. When TFSs are defined over signature modules this is not the case, since 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-typedenss 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 1.2). 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 1.2) 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.

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.[7]

Since these definitions naturally extend the basic grammar definition (definition 12) and the definitions and algorithms presented in this chapter, we suppress them here and they are given in appendix C.

[7] In practice, an extra adjustment is required in order to restore well-typedness, see appendix C.

Chapter 3

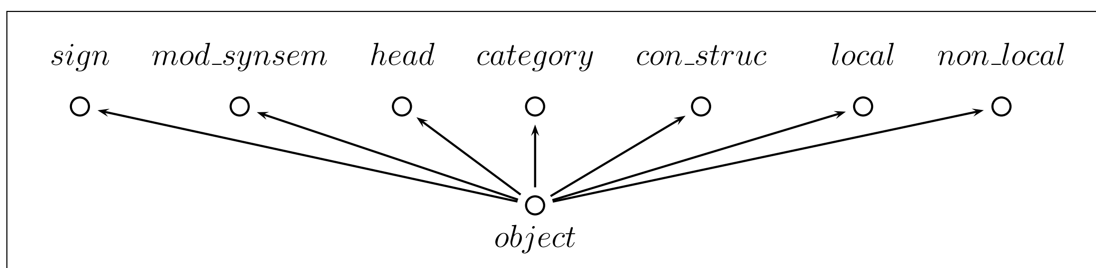
Modular Construction of the Basic HPSG Signature

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 A_1) 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'.[1] Some of the fragments make use of the signature module $List$ of Figure 2.12.

We begin with a module defining *objects* (Figure 3.1), where the type *object* is the most general type. This module defines the main fragments of the signature.

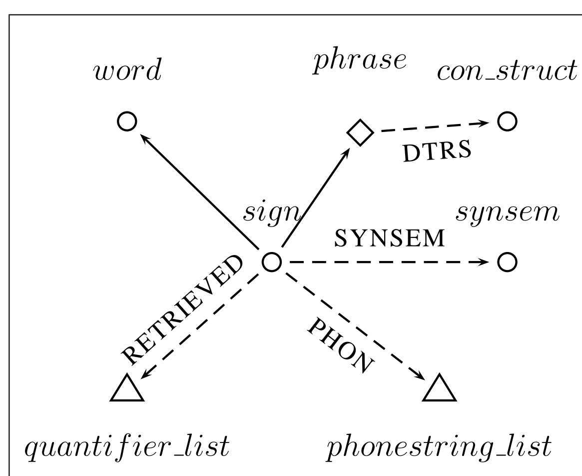
Figure 3.2 defines the module $Sign$. It consists of the type *sign*, and its two subtypes

[1] Of course, other ways to break up the given signature to modules are conceivable. In particular, the $Synsem$ module of Figure 3.5 may better be broken into two modules.



Object

Figure 3.1: The main fragments of the signature



Sign

$$Imp = \langle phonstring_list, quantifier_list \rangle$$

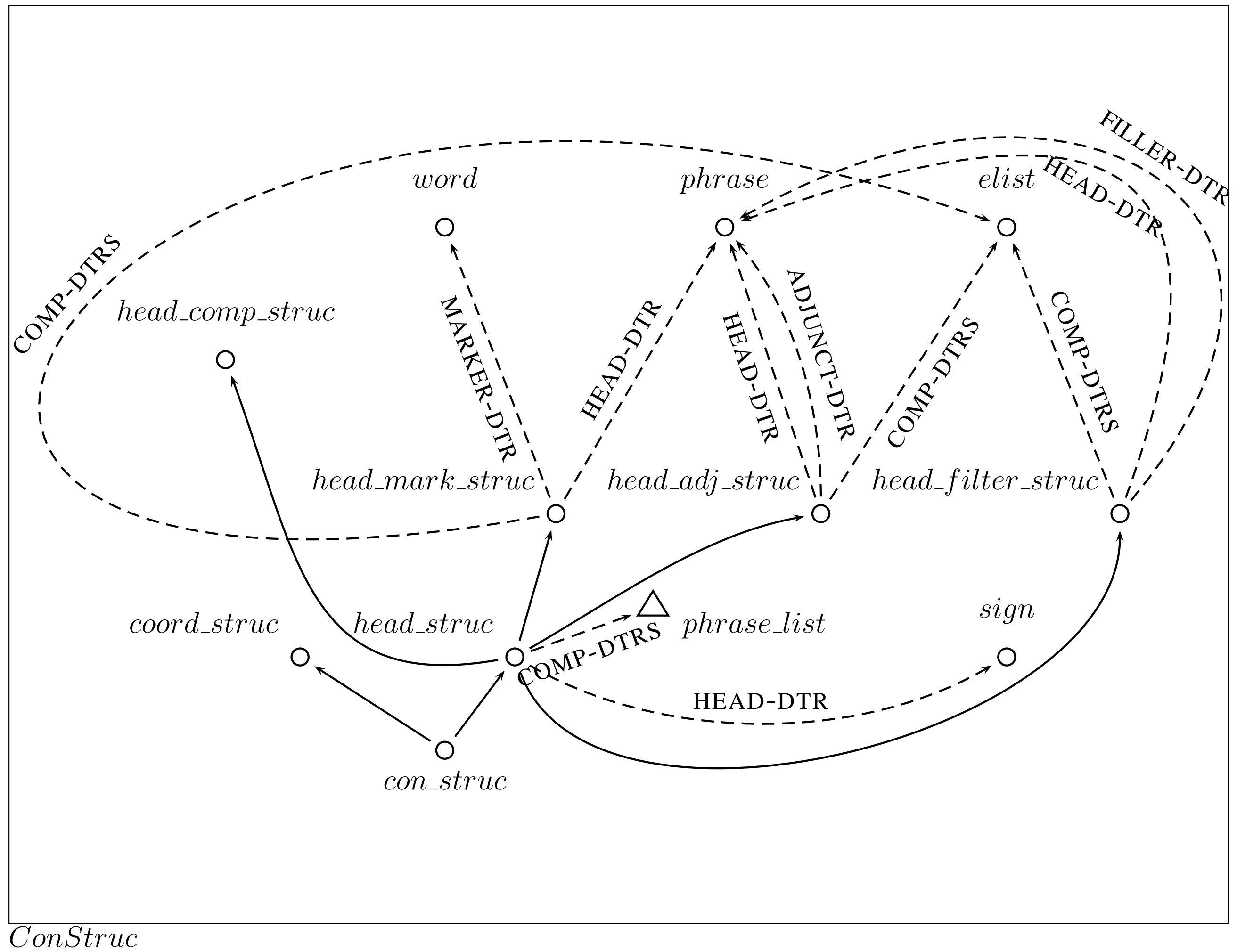
$$Exp = \langle phrase \rangle$$

Figure 3.2: A signature module, $Sign$

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 3.3. 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 2.12).



$$Imp = \langle phrase_list \rangle$$

Figure 3.3: Phrase structure

Figure 3.4 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 3.5 provides straight-forward definitions of *cat*egory and *synsem*, respectively. As another example, Figure 3.6 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

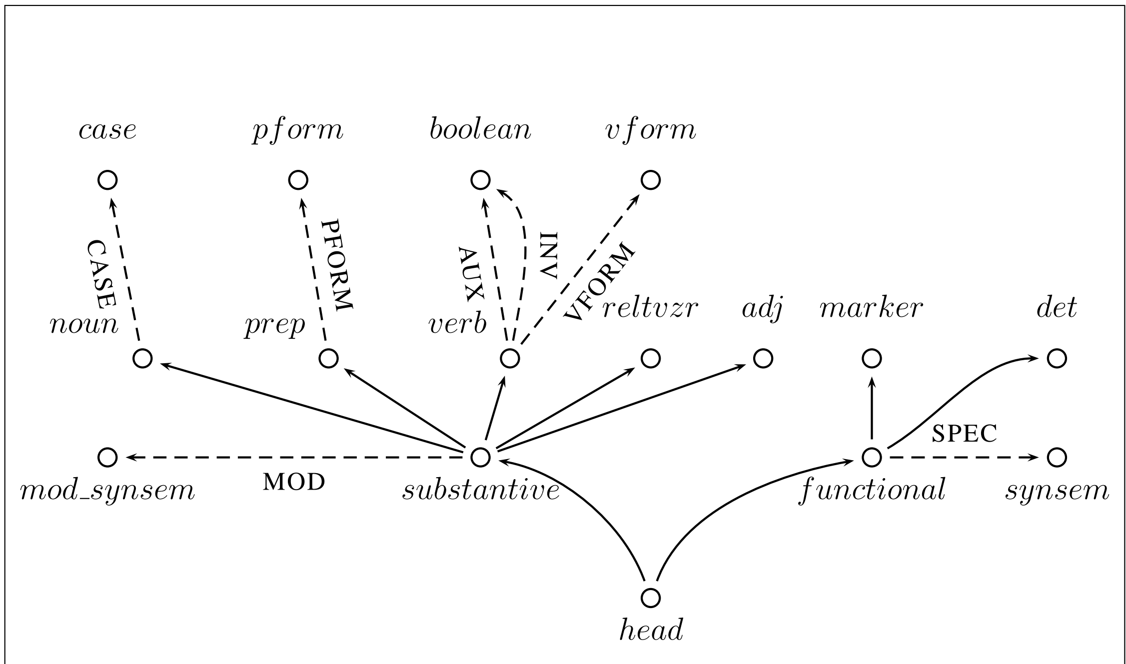


Figure 3.4: A signature module, *Head*

root). Finally, Figure 3.7 abstracts over the internal structure of *Phonstring* and *Quantifier*; these are only representatives of the actual signature modules which define these fragments.

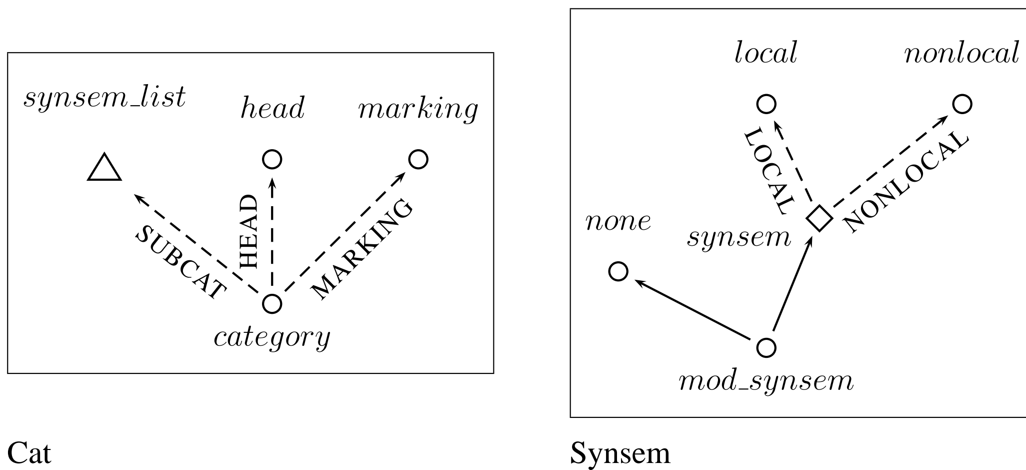


Figure 3.5: Signature modules

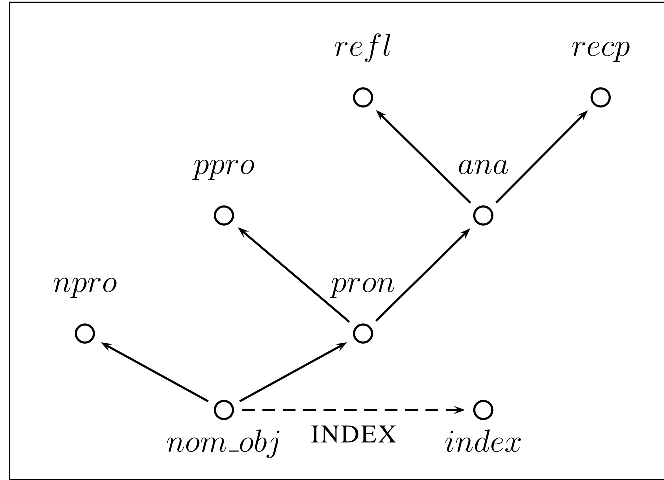


Figure 3.6: A classification of nominal objects

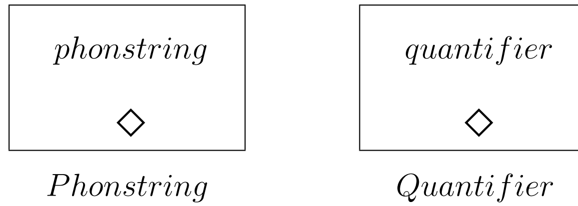


Figure 3.7: Parametric signature modules

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 above. The construction is given in Figure 3.8.

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

$$List(\textit{\textbf{Phonestring}}) \Cup List(\textit{\textbf{Quantifier}})$$

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

$$Sign(List(\textit{\textbf{Phonestring}}) \Cup List(\textit{\textbf{Quantifier}}))$$

Notice how the order of the parameters ensures the correct instantiation. Now, in the

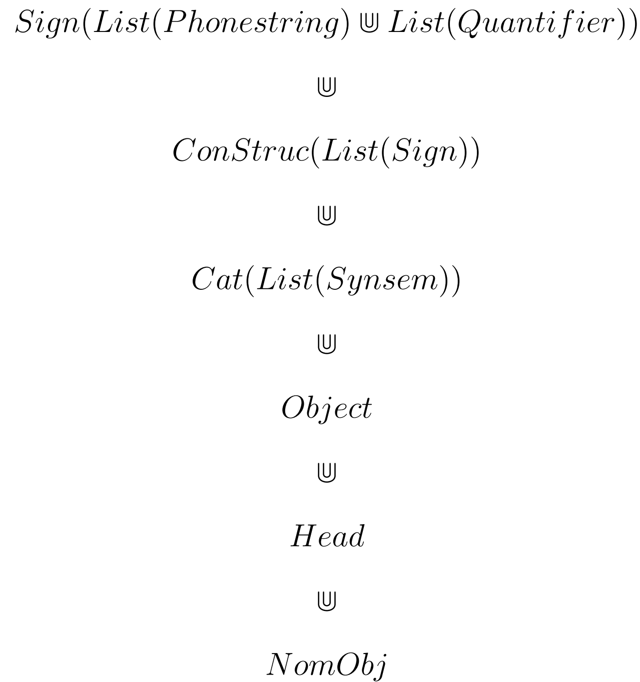


Figure 3.8: HPSG signature construction

second element, $List(Sign)$ both creates a list of *phrase* (since *phrase* is an exported 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 (1994) is hard to conceptualize

since 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, since 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 work.

Chapter 4

MODALE: A Platform for Modular Development of Signature Modules

In this work we focus on typed unification grammars (TUG), and their implementation in grammar-development platforms. 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, 1992a). 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 2.2, is shown in figure 4.1.
- 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 a 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.

```
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 4.1: MODALE description of S_1

Consider again the modular design of the basic HPSG grammar (Pollard and Sag, 1994) which was presented in Chapter 3. In appendix D, sections D.1 and D.2 use the

description language to depict both the modular design and the resolved HPSG grammar, respectively. Clearly, the modular description is clearer, easier to conceptualize, and changes can be done locally and easily.

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, feature introduction and the functionality of appropriateness specification (i.e., no multiple Ap-arcs), none of which can be assumed when working with 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 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.[1] 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.

[1]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.

Chapter 5

Implications on Other Formalisms

5.1 Overview

While our main focus in this work is facilitating the necessary infrastructure for modular construction of typed unification grammars, the methods we use have an impact also on the development of large-scale grammars in some other, related, formalisms, e.g., Polarized unification grammar (PUG) (Kahane, 2006) and XMG (Duchier, Le Roux, and Parmentier, 2004; Crabbé, 2005). In this chapter we focus on PUG. We show that the grammar combination operator proposed by Kahane (2006) is not associative, and we correct it by adapting the *powerset-lift* methhod used in chapter 2.

PUG is a linguistic formalism which uses *polarities* to better control the way grammar fragments interact. A PUG is defined over a *system of polarities* $(P, \cdot)$ where P is a set (of polarities) and '$\cdot$' is an associative and commutative product over P. A PUG generates a set of finite structures (e.g., trees) over objects (e.g., nodes) which are determined for each grammar separately. The objects are associated with polarities, and structures are combined by identifying some of their objects. The combination is sanctioned by polarities: objects can only be identified if their polarities are unifiable; the resulting object has the unified polarity. A non-empty, strict subset of the set of polarities, called the set

of *neutral* polarities, determines which of the resulting structures are valid: A polarized structure is *saturated* if all its polarities are neutral, and the language generated by the grammar includes the saturated structures that result from all the possible combinations of elementary structures. PUG is a powerful and flexible formalism which was shown to be capable of simulating many grammar formalisms, including TAG, LFG, HPSG, etc.

However, unlike other tree-based formalisms and unlike our approach, PUG does not take the metagrammar approach: the basic units are grammatical objects (e.g., trees or graphs) rather than grammatical descriptions (e.g., formulas describing grammatical objects).

The grammar combination operation of PUG was conjectured to be associative (Kahane and Lareau, 2005; Kahane, 2006). We show that it is not; even attaching polarities to objects does not render grammar combination order-independent. In section 5.2 we formalize the tree combination operation of PUG and set a common notation. We limit the discussion to the case of trees, rather than the arbitrary objects of PUG, for the sake of simplicity; all our results can easily be extended to arbitrary structures and objects (e.g., graphs and their nodes and edges). In section 5.3 we show that existing polarity systems do not guarantee associativity. This is not accidental: we prove that no non-trivial polarity system can guarantee the associativity of grammar combination. We analyze the reasons for this in section 5.4 and introduce new definitions, based on a move from trees to forests, which induce an associative grammar combination operator. The immediate contribution of this chapter is thus the identification—and correction—of a significant flaw in this otherwise powerful and flexible formalism. Moreover, the method we propose is general, and therefore applicable to a variety of formalisms. In section 5.6 we show that our results can be used to define an alternative semantics for XMG (Duchier, Le Roux, and Parmentier, 2004; Crabbé, 2005).

5.2 Tree Combination in PUG

To the best of our knowledge, no formal definition of PUG was published and the formalism is only discussed informally (Kahane and Lareau, 2005; Kahane, 2006). We therefore begin by defining the formalism and its combination operation, both with and without polarities, to establish a common notation.

Definition 22. *A* ***tree*** $\langle V, E, r \rangle$ *is a connected, undirected, acyclic graph with vertices* V*, edges* E *and a unique root* $r \in V$.

Every pair of nodes in a tree is connected by a unique path, and the edges have a natural orientation, toward or away from the root. Let $\langle V, E, r \rangle$ be a tree and let $v \in V$. Any vertex u which is located on the single path from r to v is an **ancestor** of v, and v is a **descendant** of u. If the last arc on the path from r to v is (u, v) then u is the **parent** of v and v is the **child** of u. The meta-variable T ranges over trees and V, E, r over their components. The meta-variable $\mathcal{T}$ ranges over sets of trees.

Definition 23. *Two trees* T_1, T_2 *are* **disjoint** *if* $V_1 \cap V_2 = \emptyset$. *Two sets of trees* $\mathcal{T}_1, \mathcal{T}_2$ *are* **disjoint** *if for all* $T_1 \in \mathcal{T}_1$, $T_2 \in \mathcal{T}_2$, $V_1 \cap V_2 = \emptyset$.

Definition 24. *Two trees* $T_1 = \langle V_1, E_1, r_1 \rangle$, $T_2 = \langle V_2, E_2, r_2 \rangle$ *are* **isomorphic***, denoted* $T_1 \sim T_2$*, if there exists a total one to one and onto function* $i : V_1 \to V_2$ *such that* $i(r_1) = r_2$ *and for all* $u, v \in V_1$, $(u, v) \in E_1$ *iff* $(i(u), i(v)) \in E_2$. *Two sets of trees* $\mathcal{T}_1, \mathcal{T}_2$ *are* **isomorphic***, denoted* $\mathcal{T}_1 \cong \mathcal{T}_2$*, if there exist total functions* $i_1 : \mathcal{T}_1 \to \mathcal{T}_2$ *and* $i_2 : \mathcal{T}_2 \to \mathcal{T}_1$ *such that for all* $T \in \mathcal{T}_1$, $T \sim i_1(T)$ *and for all* $T \in \mathcal{T}_2$, $T \sim i_2(T)$.

Next, we define how two trees are combined. An equivalence relation over the nodes of the two trees states which nodes should be identified. In the result of the combination, nodes are equivalence classes of that relation and arcs connect nodes that are connected in their members. The equivalence relation is sanctioned in a way that guarantees that the resulting graph is indeed a tree.

Definition 25. *Let* $T_1 = \langle V_1, E_1, r_1 \rangle$, $T_2 = \langle V_2, E_2, r_2 \rangle$ *be two disjoint trees. An equivalence relation '*$\overset{t}{\approx}$*' over* $V_1 \cup V_2$ *is* **legal** *if all the following hold:*[1]

1. *for all* $v_1, v_2 \in V_1 \cup V_2$*, if* $v_1 \overset{t}{\approx} v_2$ *and* $v_1 \neq v_2$ *then either* $v_1 \in V_1$ *and* $v_2 \in V_2$ *or* $v_1 \in V_2$ *and* $v_2 \in V_1$
2. *for all* $u_1, v_1, u_2, v_2 \in V_1 \cup V_2$*, if* $v_1 \overset{t}{\approx} v_2$*,* u_1 *is the parent of* v_1 *and* u_2 *is the parent of* v_2*, then* $u_1 \overset{t}{\approx} u_2$
3. *there exists* $v \in V_1 \cup V_2$ *such that* $|[v]_{\overset{t}{\approx}}| > 1$

$Eq_t(T_1, T_2)$ *is the set of legal equivalence relations over* $V_1 \cup V_2$*.*

The first condition of definition 25 states that when two nodes are identified, they must belong to different trees. The second condition requires that when two nodes are identified, all their ancestors must identify as well. Finally, the last condition requires that at least two nodes (each from a different tree) be identified. The first two conditions guarantee that the resulting graph is acyclic and the third guarantees that it is connected.[2]

Definition 26. *Let* $T_1 = \langle V_1, E_1, r_1 \rangle$, $T_2 = \langle V_2, E_2, r_2 \rangle$ *be two disjoint trees and let '*$\overset{t}{\approx}$*' be a legal equivalence relation over* $V_1 \cup V_2$*. The* **tree combination** *of* T_1, T_2 **with respect to '**$\overset{t}{\approx}$**'**, *denoted* $T_1 +_{\overset{t}{\approx}} T_2$*, is a tree* $T = \langle V, E, r \rangle$*, where:*

- $V = \{[v]_{\overset{t}{\approx}} \mid v \in V_1 \cup V_2\}$
- $E = \{([u]_{\overset{t}{\approx}}, [v]_{\overset{t}{\approx}}) \mid (u, v) \in E_1 \cup E_2\}$
- $r = \begin{cases} [r_1]_{\overset{t}{\approx}} & \text{if } [r_1]_{\overset{t}{\approx}} = \{r_1\} \text{ or } [r_1]_{\overset{t}{\approx}} = \{r_1, r_2\} \\ [r_2]_{\overset{t}{\approx}} & \text{otherwise} \end{cases}$

[1] If '$\equiv$' is an equivalence relation then $[v]_\equiv$ is the equivalence class of v with respect to '$\equiv$'.

[2] The second condition is not an original requirement of PUG; it is added for the case in which the basic structures are trees to guarantee that the resulting graph is indeed a tree.

When two trees are combined, nodes belonging to the same equivalence class are identified. Since the equivalence relation is legal, the resulting graph is indeed a tree. Observe that since the equivalence relation is legal, either the two roots are identified; or one of them is identified with a non-root node and the other remains alone. In the former case, the root of the new tree is the node created from the identification of the two roots; in the latter case, the new root is the root whose equivalence class is a singleton. In definition 26, a systematic replacement of r_1 and r_2 in the definition of r would have yielded the same result.

Example 14. *Figure 5.1 depicts three trees, T_1, T_2, T_3. T and T' are tree combinations of T_1 and T_2. T is obtained by identifying q_1 with q_3 and q_2 with q_4. Notice that since q_2 is identified with q_4, q_1 must be identified with q_3 to maintain a tree structure (condition 2 of definition 25). T' is obtained by identifying q_2 with q_3. T'' is not a tree combination of T_2 and T_3 since it identifies q_6 with q_7, which belong to the same tree, T_3, in contradiction to condition 1 of definition 25.*

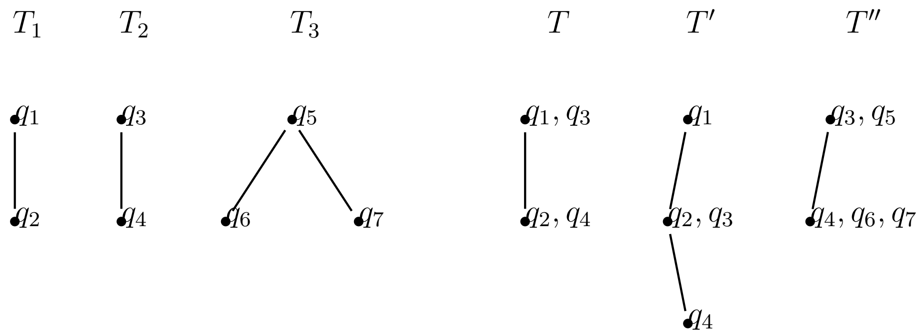


Figure 5.1: Tree combination

Definition 27. *Let $\mathcal{T}_1, \mathcal{T}_2$ be two disjoint sets of trees. The* **tree combination** *of $\mathcal{T}_1, \mathcal{T}_2$,*

denoted $\mathcal{T}_1 \overset{t}{+} \mathcal{T}_2$*, is the set of trees*

$$\mathcal{T} = \bigcup_{\substack{T_1 \in \mathcal{T}_1, T_2 \in \mathcal{T}_2 \\ \overset{t}{\approx} \in Eq_t(T_1, T_2)}} T_1 +_{\overset{t}{\approx}} T_2$$

The tree combination operation takes as input two sets of trees and yields a set of trees which includes *all* the tree combinations of any possible pair of trees belonging to the two different sets with respect to *any* possible legal equivalence relations. Notice that the definitions allows multiple isomorphic trees in the same set of trees, which may result in inefficient processing. It is assumed that the grammar designer is responsible for avoiding such inefficiency.

Example 15. *The sets of trees defined by* $\{T_1\} \overset{t}{+} \{T_2\}$ *and* $\{T_2\} \overset{t}{+} \{T_3\}$ *(Figure 5.1) are depicted in Figures 5.2 and 5.3, respectively.*

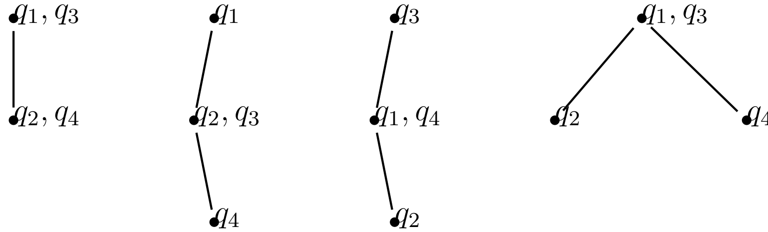


Figure 5.2: $\{T_1\} \overset{t}{+} \{T_2\}$

This combination operation is extended by attaching polarities to nodes (Crabbé and Duchier, 2004; Perrier, 2000; Kahane, 2006). In a polarized framework, an extra condition for the identification of two nodes is that their polarities combine; in this case a new node (obtained by identifying two nodes) has a polarity which is the product of the polarities of the two identified nodes.

Definition 28. *A system of polarities is a pair* $(P, \cdot)$*, where* P *is a non-empty set and '*$\cdot$*' is a commutative and associative product over* $P \times P$*.*

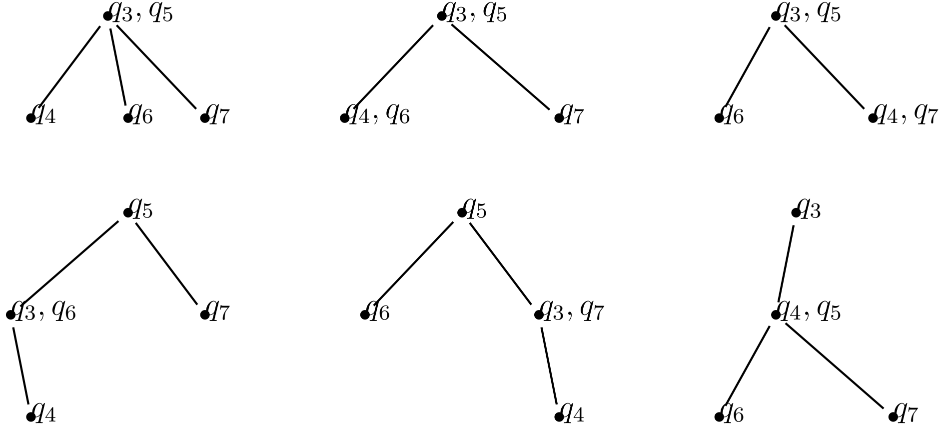


Figure 5.3: $\{T_2\}\stackrel{t}{+}\{T_3\}$

In the sequel, if $(P, \cdot)$ is a system of polarities and $a, b \in P$, $ab\downarrow$ means that the combination of a and b is defined and $ab\uparrow$ means that a and b cannot combine. For the following discussion we assume that a system of polarities $(P, \cdot)$ has been specified.

Definition 29. *A* **polarized tree** $\langle V, E, r, p\rangle$ *is a tree in which each node is assigned a polarity through a total function* $p : V \to P$*. If* $\langle V, E, r, p\rangle$ *is a polarized tree then* $\langle V, E, r\rangle$ *is its* **underlying tree***. Two polarized trees are* **disjoint** *if their underlying trees are disjoint.*

Definition 30. *Two polarized trees* $T_1 = \langle V_1, E_1, r_1, p_1\rangle$*,* $T_2 = \langle V_2, E_2, r_2, p_2\rangle$ *are* **isomorphic***, denoted* $T_1 \sim T_2$*, if their underlying trees are isomorphic and, additionally, for all* $v \in V_1$*,* $p_1(v) = p_2(i(v))$*. The definition of isomorphism of sets of trees is trivially extended to sets of polarized trees.*

Definition 31. *Let* $T_1 = \langle V_1, E_1, r_1, p_1\rangle$*,* $T_2 = \langle V_2, E_2, r_2, p_2\rangle$ *be two disjoint polarized trees. An equivalence relation '*$\stackrel{t}{\approx}$*' over* $V_1 \cup V_2$ *is* **legal** *if it is legal over the underlying trees of* T_1 *and* T_2 *and, additionally, for all* $v_1 \in V_1$ *and* $v_2 \in V_2$*, if* $v_1 \stackrel{t}{\approx} v_2$*, then* $p_1(v_1) \cdot p_2(v_2)\downarrow$*.* $Eq_t(T_1, T_2)$ *is the set of legal equivalence relations over* $V_1 \cup V_2$*.*

Definition 32. *Let* $T_1 = \langle V_1, E_1, r_1, p_1\rangle$*,* $T_2 = \langle V_2, E_2, r_2, p_2\rangle$ *be two disjoint polarized*

trees and let '$\overset{t}{\approx}$' be a legal equivalence relation over $V_1 \cup V_2$. The **polarized tree combination** *of T_1, T_2* **with respect to '$\overset{t}{\approx}$'**, *denoted $T_1 +_{\overset{t}{\approx}} T_2$, is a tree $T = \langle V, E, r, p \rangle$ where V, E and r are as in definition 26, and for all $[v]_{\overset{t}{\approx}} \in V$,*

$$p([v]_{\overset{t}{\approx}}) = \begin{cases} (p_1 \cup p_2)(v) & \text{if } [v]_{\overset{t}{\approx}} = \{v\} \\ (p_1 \cup p_2)(v) \cdot (p_1 \cup p_2)(u) & \text{if } [v]_{\overset{t}{\approx}} = \{v, u\} \text{ and } u \neq v \end{cases}$$

Notice that since '$\overset{t}{\approx}$' is legal, p is well defined. The definition of tree combination of sets of trees, denoted '$\overset{t}{+}$', is trivially extended to sets of polarized trees.

The language of a PUG consists of the neutral structures obtained by combining the initial structure and a finite number of elementary structures. In the derivation process, elementary structures combine successively, each new elementary structure combining with at least one object of the previous result.

Definition 33. *A* **Polarized Unification Grammar (PUG)** *is a structure $G = \langle T_0, \mathcal{T}, (P, \cdot) \rangle$ where $\mathcal{T}$ is a set of polarized trees, $T_0 \in \mathcal{T}$ is the initial tree and $(P, \cdot)$ is the system of polarities over which the polarized tree combination is defined.*

Let A_i be a sequence of tree sets where $A_0 = \{T_0\} \overset{t}{+} \mathcal{T}$ and for all i, $i \geq 1$, $A_i = A_{i-1} \overset{t}{+} \mathcal{T}$. The **language** *generated by G, denoted $L(G)$, is $L(G) = \bigcup_{i \in \mathbb{N}} A_i$.*

PUG is a powerful grammatical formalism that was shown to be capable of simulating various linguistic theories (Kahane, 2006). It can be instrumental for grammar engineering, and in particular for modular development of large-scale grammars, where grammar fragments are developed separately and are combined using the basic combination operation defined above. A pre-requisite for such an application is obviously that the grammar combination operation be associative: one would naturally expect that, if '$\circ$' is a grammar combination operator, then $G_1 \circ (G_2 \circ G_3) = (G_1 \circ G_2) \circ G_3$ for any three grammars (and, therefore, $L(G_1 \circ (G_2 \circ G_3)) = L((G_1 \circ G_2) \circ G_3)$).

The grammar combination operation of PUG was indeed conjectured to be associative (Kahane and Lareau, 2005; Kahane, 2006). The present paper makes two main contri-

butions: In the next section we show that the combination operation defined above is *not* associative. In section 5.4 we introduce an alternative combination operation which we prove to be associative. We thus remedy the shortcoming of the original definition, and render PUG a more suitable formalism for modular grammar development.

5.3 Tree Combination is not Associative

In this section we show that tree combination as defined above, with or without polarities, is not associative. In the examples below, the relation which determines how polarities combine *is* indeed associative; it is the tree combination operation which uses polarities that is shown to be non-associative.

5.3.1 Non-Polarized Tree Combination

Theorem 8. *(Non-polarized) tree combination is a non-associative operation: there exist sets of trees* $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$ *such that* $\mathcal{T}_1 \overset{t}{+} (\mathcal{T}_2 \overset{t}{+} \mathcal{T}_3) \not\cong (\mathcal{T}_1 \overset{t}{+} \mathcal{T}_2) \overset{t}{+} \mathcal{T}_3$.

Proof. Consider again T_1, T_2, T_3 of Figure 5.1 and the sets of trees defined by $\{T_1\} \overset{t}{+} \{T_2\}$ and $\{T_2\} \overset{t}{+} \{T_3\}$, depicted in Figures 5.2 and 5.3, respectively. T_4 of Figure 5.4 is a member of $\{T_2\} \overset{t}{+} \{T_3\}$, obtained by identifying q_3 of T_2 and q_6 of T_3. Similarly, T_5 of Figure 5.4 is a member of $\{T_1\} \overset{t}{+} \{T_4\}$. Hence $T_5 \in \{T_1\} \overset{t}{+} (\{T_2\} \overset{t}{+} \{T_3\})$. However, T_5 (or any tree isomorphic to it) is not a member of $(\{T_1\} \overset{t}{+} \{T_2\}) \overset{t}{+} \{T_3\}$. □

5.3.2 Colors

Crabbé and Duchier (2004) use *colors* to sanction tree node identification. Their color combination table is presented in Figure 5.5. W, B and R denote white, black and red, respectively, and $\perp$ represents the impossibility to combine.

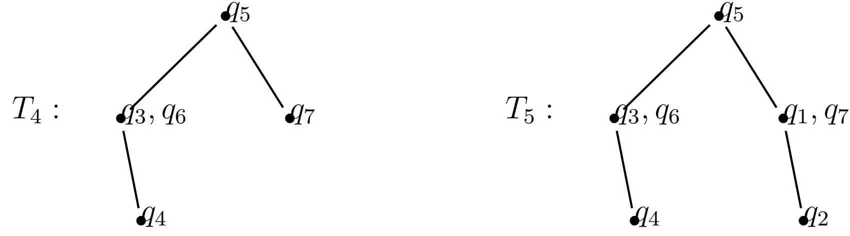


Figure 5.4: Non-polarized tree combination

$\cdot$	W	B	R
W	W	B	$\perp$
B	B	$\perp$	$\perp$
R	$\perp$	$\perp$	$\perp$

Figure 5.5: Color combination table

Theorem 9. *The color scheme of Figure 5.5 does not guarantee associativity: Let $(P, \cdot)$ be the system of Figure 5.5. Then there exist sets of trees $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$ such that $\mathcal{T}_1 \stackrel{t}{+} (\mathcal{T}_2 \stackrel{t}{+} \mathcal{T}_3) \ncong (\mathcal{T}_1 \stackrel{t}{+} \mathcal{T}_2) \stackrel{t}{+} \mathcal{T}_3$.*

Proof. Consider Figure 5.6. The results of combining $\{T_6\}, \{T_7\}, \{T_8\}$ in different orders demonstrate that $(\{T_6\} \stackrel{t}{+} \{T_7\}) \stackrel{t}{+} \{T_8\} \ncong \{T_6\} \stackrel{t}{+} (\{T_7\} \stackrel{t}{+} \{T_8\})$.

□

Notice that in Figure 5.6 all the intermediate and final solutions are saturated. Therefore, the saturation rule does not guarantee associativity.

5.3.3 Polarities

Kahane and Lareau (2005) and Kahane (2006) use two systems of polarities which are depicted in Figure 5.7. The first system includes three polarities, gray, white and black, where the neutral polarities are black and gray. A black node may be unified with 0, 1

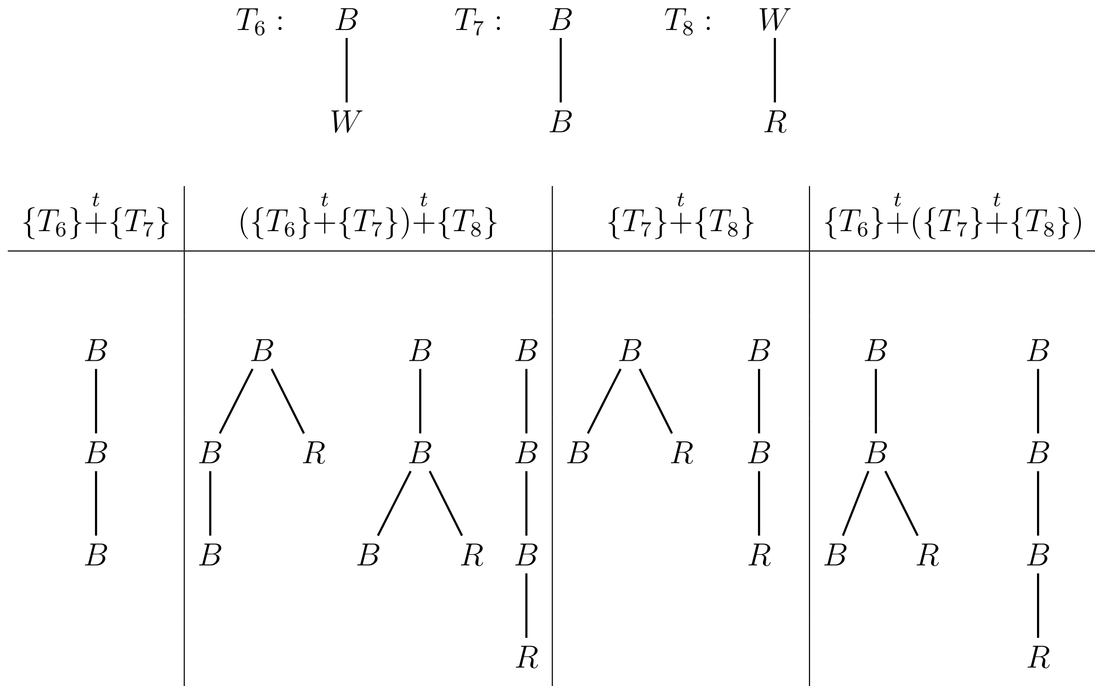


Figure 5.6: Tree combination with colors

or more gray or white nodes and produce a black node; a white node may absorb 0, 1 or more gray or white nodes but eventually must be unified with a black one producing a black node; and a gray node may be absorbed into a white or a black node. The second system extends the first by adding two more non-neutral polarities, plus and minus, which may absorb 0, 1 or more white or gray nodes but eventually a plus node must be unified with a minus node to produce a black node.

Theorem 10. *PUG combination with either of the polarity systems of Figure 5.7 is not associative.*

Proof. Consider Figure 5.8. Clearly, $\{T_9\}\overset{t}{+}(\{T_{10}\}\overset{t}{+}\{T_{11}\})\ncong(\{T_9\}\overset{t}{+}\{T_{10}\})\overset{t}{+}\{T_{11}\}$.

□

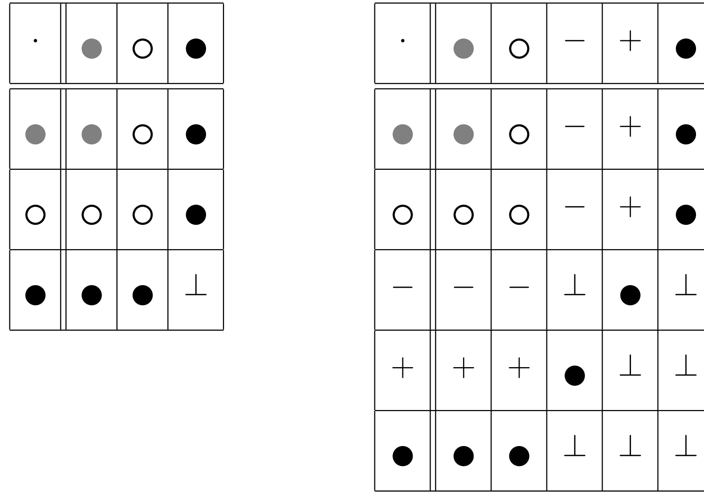


Figure 5.7: PUG polarity systems

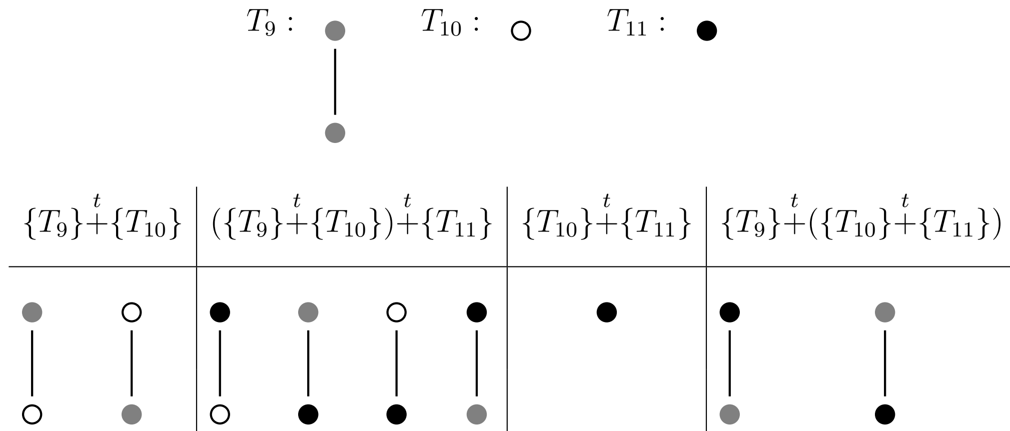


Figure 5.8: Tree combination with polarities

5.3.4 General Polarity Systems

We showed above that some existing polarity systems yield non-associative grammar combination operators. This is not accidental; in what follows we show that the only polarity scheme that induces associative tree combination is trivial: the one in which no pair of polarities are unifiable. This scheme is useless for sanctioning tree combination since it disallows any combination.

Definition 34. *A system of polarities* $(P, \cdot)$ *is trivial if for all* $a, b \in P$, $ab\uparrow$.

Theorem 11. *Let* $(P, \cdot)$ *be a system of polarities. If there exists* $a \in P$ *such that* $aa\downarrow$ *then the polarized tree combination based on* $(P, \cdot)$ *is not associative.*

Proof. Let $(P, \cdot)$ be a system of polarities and let $a \in P$ be such that $aa\downarrow$. Assume toward a contradiction that the polarized tree combination based on $(P, \cdot)$ is associative. Consider T_1, T_2, T_3 of Figure 5.1 and T_5 of Figure 5.4. Let T_1', T_2', T_3', T_5' be polarized trees obtained by attaching the polarity 'a' to all tree nodes of T_1, T_2, T_3, T_5, respectively. $T_5' \in \{T_1'\}\overset{t}{+}(\{T_2'\}\overset{t}{+}\{T_3'\})$, but T_5' (or any tree isomorphic to it) is not a member of $(\{T_1'\}\overset{t}{+}\{T_2'\})\overset{t}{+}\{T_3'\}$ (see the proof of theorem 8 for the complete details). Clearly $\{T_1'\}\overset{t}{+}(\{T_2'\}\overset{t}{+}\{T_3'\})\ncong(\{T_1'\}\overset{t}{+}\{T_2'\})\overset{t}{+}\{T_3'\}$, a contradiction. □

Theorem 12. *Let* $(P, \cdot)$ *be a non-trivial system of polarities. Then the polarized tree combination based on* $(P, \cdot)$ *is not associative.*

Proof. Let $(P, \cdot)$ be a non-trivial system of polarities. If $|P| = 1$ then let $P = \{a\}$. Since P is non-trivial, $aa = a$. Then, by theorem 11, $(P, \cdot)$ is not associative. Now assume that $|P| > 1$. Assume toward a contradiction that the polarized tree combination based on $(P, \cdot)$ is associative. There are two possible cases:

1. There exists $a \in P$ such that $aa\downarrow$: Then from theorem 11 it follows that the resulting tree combination operation is not associative, a contradiction.

2. For all $a \in P$, $aa\uparrow$: Then since $(P, \cdot)$ is non-trivial and since $|P| > 1$, there exist $b, c \in P$ such that $b \neq c$, $bb\uparrow$, $cc\uparrow$ and $bc\downarrow$. Consider T_1, T_2, T_3 of Figure 5.9. Of all the trees in $(\{T_1\}\overset{t}{+}\{T_2\})\overset{t}{+}\{T_3\}$ and $\{T_1\}\overset{t}{+}(\{T_2\}\overset{t}{+}\{T_3\})$, focus on paths of length 3. All possible instantiations of these trees are depicted in Figure 5.9 (we suppress the intermediate results). Notice that these trees are only candidate solutions; they are actually accepted only if the polarity combinations occurring in them are defined. Since $bb\uparrow$, $cc\uparrow$ and $bc\downarrow$, $(\{T_1\}\overset{t}{+}\{T_2\})\overset{t}{+}\{T_3\}$ has no solutions and $\{T_1\}\overset{t}{+}(\{T_2\}\overset{t}{+}\{T_3\})$ has one accepted solution (the rightmost tree), a contradiction.

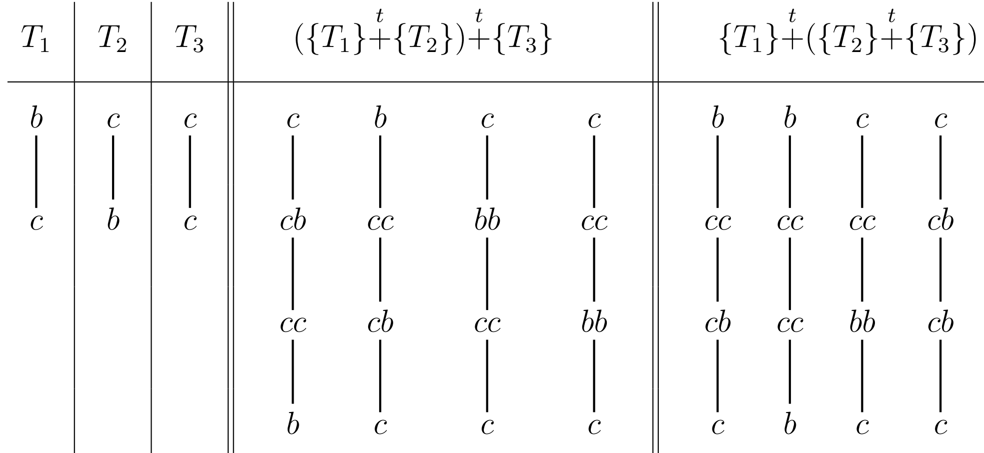


Figure 5.9: Candidate solutions for PUG tree combination

□

For the sake of completion, we also mention the reverse direction.

Theorem 13. *Let $(P, \cdot)$ be a trivial system of polarities. Then the polarized tree combination based on $(P, \cdot)$ is associative.*

Proof. If $(P, \cdot)$ is a trivial system of polarities then any combination of two sets of polarized trees results in the empty set (no solutions). Evidently, polarized tree combination based on $(P, \cdot)$ is associative. □

Corollary 14. *Let $(P, \cdot)$ be a system of polarities. Then polarized tree combination based on $(P, \cdot)$ is associative if and only if $(P, \cdot)$ is trivial.*

5.3.5 Practical Consequences

Evidently, (polarized) tree combination induces a non-associative grammar combination for PUG. In some cases the result of the non-associativity is plain overgeneration: For example, in Figure 5.6, $(\{T_6\}\stackrel{t}{+}\{T_7\})\stackrel{t}{+}\{T_8\}$ strictly includes (and, consequently, overgenerates with respect to) $\{T_6\}\stackrel{t}{+}(\{T_7\}\stackrel{t}{+}\{T_8\})$. In general, however, non-associativity results in two non-equal sets: For example, consider Figure 5.9 and its candidate solutions for

length-3 paths and assume that $cb = bc = bb = cc = b$. The length-3 solutions of this case are depicted in Figure 5.10. Clearly the resulting sets are not equal but none of them overgenerates with respect to the other. The non-associativity of the combination clearly compromises its usability for (modular) development of large-scale grammars: When the grammar designer wrongly assumes that the combination operation is associative, he or she can take advantage of this misconception to achieve a more efficient computation of the combination. This may lead to an incorrect result (which may sometimes over- or undergenerate with respect to the correct result). Such problems may be difficult to locate due to the size of the grammar.

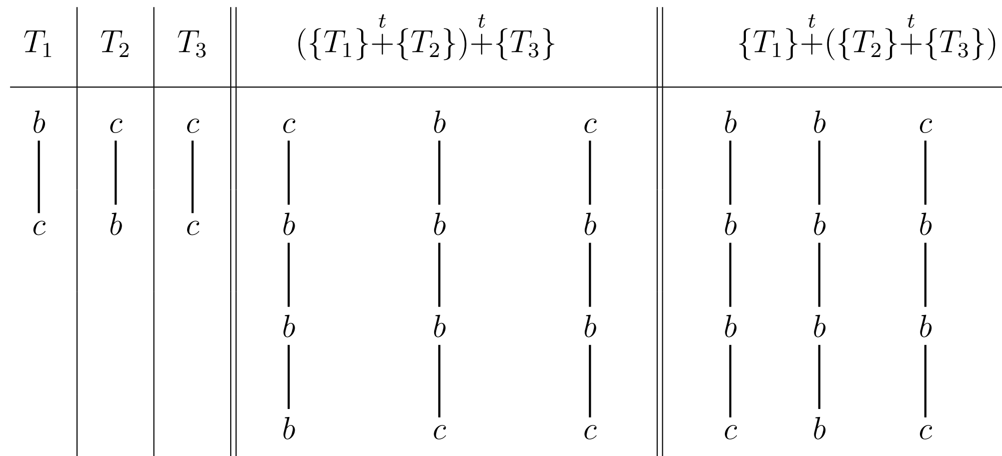


Figure 5.10: Length-3 paths solutions

When a combination *is* associative, the grammar designer is free to conceptualize about the combination of grammar fragments in any order; we trust that this makes the formalism more "friendly" to the grammar engineer, and hence easier to work with. In the next section we analyze the reasons for the non-associativity and introduce new definitions which induce an associative grammar combination operator.

5.4 From Trees to Forests

Let us now analyze the reasons for the non-associativity of tree combination. Consider again T_1, T_2, T_3 of Figure 5.1 and T_5 of Figure 5.4. T_5 is a member of $\{T_1\}\overset{t}{+}(\{T_2\}\overset{t}{+}\{T_3\})$ but not of $(\{T_1\}\overset{t}{+}\{T_2\})\overset{t}{+}\{T_3\}$. The reason is that in T_5, T_1 and T_2 are substructures separated by T_3. When T_2 and T_3 are combined first, T_2 connects to one of the nodes of T_3; then, when T_1 is added, it is connected to another node of T_3. However, when T_1 and T_2 combine first, they must be connected through a common node and cannot be separated as they are in T_5.

Similarly, considering again T_9, T_{10}, T_{11} of Figure 5.8 and their combinations, clearly

$$\{T_9\}\overset{t}{+}(\{T_{10}\}\overset{t}{+}\{T_{11}\})\ncong(\{T_9\}\overset{t}{+}\{T_{10}\})\overset{t}{+}\{T_{11}\}$$

When T_{10} and T_{11} are combined, their two single nodes must identify in order to yield a tree. However, when T_9 and T_{10} combine first, the single node of T_{10} can identify with either of the two nodes of T_9. Then, when the resulting tree is combined with T_{11}, the single node of T_{11} can be identified with the other node of T_9 (the one that was not identified with the node of T_{10}). This is why $(\{T_9\}\overset{t}{+}\{T_{10}\})\overset{t}{+}\{T_{11}\}$ overgenerates with respect to $\{T_9\}\overset{t}{+}(\{T_{10}\}\overset{t}{+}\{T_{11}\})$.

The above cases exemplify the causes for the non-associativity of tree combination: When two trees are combined, at least two nodes (each from a different tree) must identify. Hence, the two trees must be connected in the resulting tree. However, other combination orders that allow two trees to be separated (by other trees) can yield results which cannot be obtained when the two trees are first combined together.

The solution we propose is based on a move to the powerset domain in order to ensure associativity of grammar combination. Working in the powerset domain, rather than the original entities, enables the operator to 'remember' *all* the possibilities; then, after the combination, an extra stage is added (corresponding to the resolution stage in TUG) in which the original entities are restored.

In the case of tree combination, the basic units should be forests rather than trees; and *forest combination* must be defined over sets of forests rather than sets of trees. Forest combination is defined in much the same way as above: two forests are combined by identifying some of their nodes. Again, if two nodes are identified then all their ancestors must be identified as well. We allow two forests to combine even if none of their nodes are identified. Furthermore, similarly to tree combination, two different nodes in the same forest represent different entities. Therefore, when two forests are combined, two nodes can be identified only if they belong to the two different forests.

Definition 35. *A **forest** $\langle V, E, R\rangle$ is a finite set of node-disjoint trees with vertices V, edges E and roots R. If $\langle V, E, r\rangle$ is a tree, then $\langle V, E, \{r\}\rangle$ is its* **corresponding forest**.

The meta-variable F ranges over forests and V, E, R over their components. The meta-variable $\mathcal{F}$ ranges over sets of forests. The definition of disjointness is trivially extended to forests and set of forests.

Definition 36. *Two forests $F_1 = \langle V_1, E_1, R_1\rangle$, $F_2 = \langle V_2, E_2, R_2\rangle$ are* **isomorphic**, *denoted $F_1 \sim F_2$, if there exists a total one to one and onto function $i : V_1 \rightarrow V_2$ such that for all $u, v \in V_1$, $(u, v) \in E_1$ iff $(i(u), i(v)) \in E_2$; and for all $u \in V_1$, $u \in R_1$ iff $i(u) \in R_2$.*

The definition of isomorphism of sets of trees is extended to sets of forests (using the above definition of forests isomorphism).

Definition 37. *Let $F_1 = \langle V_1, E_1, R_1\rangle$, $F_2 = \langle V_2, E_2, R_2\rangle$ be two disjoint forests. An equivalence relation '$\overset{f}{\approx}$' over $V_1 \cup V_2$ is* **legal** *if both:*

1. *for all $v_1, v_2 \in V_1 \cup V_2$, if $v_1 \overset{f}{\approx} v_2$ and $v_1 \neq v_2$ then either $v_1 \in V_1$ and $v_2 \in V_2$ or $v_1 \in V_2$ and $v_2 \in V_1$; and*
2. *for all $u_1, v_1, u_2, v_2 \in V_1 \cup V_2$, if $v_1 \overset{f}{\approx} v_2$, u_1 is the parent of v_1 and u_2 is the parent of v_2, then $u_1 \overset{f}{\approx} u_2$.*

$Eq_f(F_1, F_2)$ is the set of legal equivalence relations over $V_1 \cup V_2$.

Notice that in contrast to definition 25, a legal equivalence relation over forests permits a combination in which no nodes unify. Such a grammar combination amounts to a set union of the two forests.

Definition 38. *Let $F_1 = \langle V_1, E_1, R_1\rangle$, $F_2 = \langle V_2, E_2, R_2\rangle$ be two disjoint forests and let '$\overset{f}{\approx}$' be a legal equivalence relation over $V_1 \cup V_2$. The* **forest combination** *of F_1, F_2* **with respect to '$\overset{f}{\approx}$'**, *denoted $F_1 +_{\overset{f}{\approx}} F_2$, is a forest $F = \langle V, E, R\rangle$, where V and E are as in definition 26, and $R = \{[r]_{\overset{f}{\approx}} \mid \text{for all } u \in [r]_{\overset{f}{\approx}}, u \in R_1 \cup R_2\}$.*

When two forests are combined, nodes in the same equivalence class are identified. Since the equivalence relation is legal, the resulting structure is indeed a forest.

Definition 39. *Let $\mathcal{F}_1, \mathcal{F}_2$ be two disjoint sets of forests. The* **forest combination** *of $\mathcal{F}_1, \mathcal{F}_2$, denoted $\mathcal{F}_1 \overset{f}{+} \mathcal{F}_2$, is the set of forests*

$$\mathcal{F} = \bigcup_{\substack{F_1 \in \mathcal{F}_1, F_2 \in \mathcal{F}_2 \\ \overset{f}{\approx} \in Eq_f(F_1, F_2)}} F_1 +_{\overset{f}{\approx}} F_2$$

Example 16. *Consider F_1, F_2 of Figure 5.11. Three members of $\{F_1\} \overset{f}{+} \{F_2\}$, namely F_3, F_4, F_5, are depicted in Figure 5.12. F_3 is obtained by identifying q_5 and q_6, F_4 is obtained by not identifying any of the nodes and F_5 is the result of identifying q_5 with q_6 and q_1 with q_7. Notice that in F_5, the two separated trees of F_1 are connected through the single tree of F_2. F_6 of Figure 5.12 is not a member of $\{F_1\} \overset{f}{+} \{F_2\}$ because it identifies q_1 and q_5 which belong to the same forest.*

Example 17. *Consider again T_1, T_2, T_3 of Figure 5.1 and T_5 of Figure 5.4. Let F_1, F_2, F_3, F_5 be their corresponding forests, respectively. F of Figure 5.13 is a member of $\{F_1\} \overset{f}{+} \{F_2\}$ which is obtained by not identifying any of the two forests nodes. F_5 is a member of $\{F\} \overset{f}{+} \{F_3\}$ which is obtained by identifying the two roots of F with the two leaves of F_3. Hence, F_5 is a member of both $\{F_1\} \overset{f}{+} (\{F_2\} \overset{f}{+} \{F_3\})$ and $(\{F_1\} \overset{f}{+} \{F_2\}) \overset{f}{+} \{F_3\}$.*

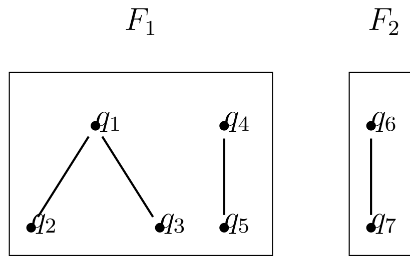


Figure 5.11: Two forests to be combined

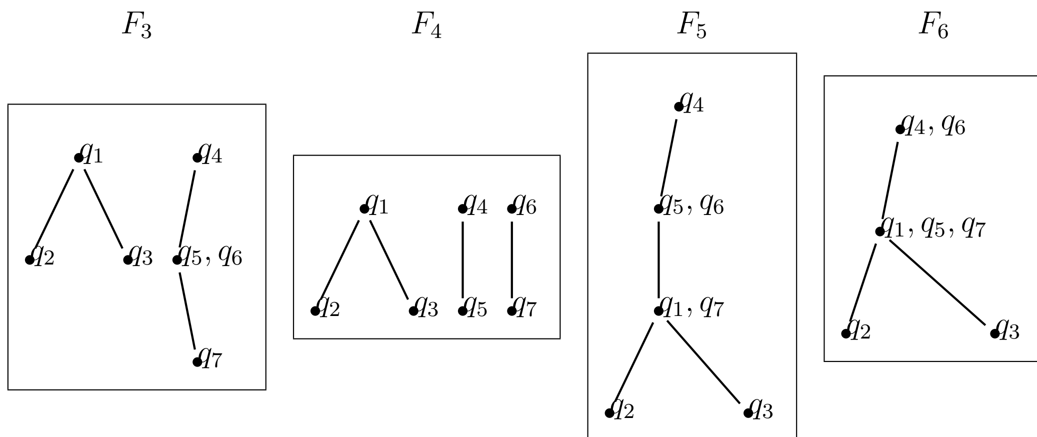


Figure 5.12: Legal and illegal combinations of F_1, F_2

The forest combination operation can be easily extended to the polarized case. This is done in the same way tree combination is extended to polarized tree combination: Polarities are attached to nodes and an extra condition for the identification of two nodes is that

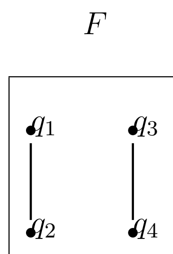


Figure 5.13: A forest combination of F_1 and F_2

their polarities combine; in that case the new node has the polarity which is the product of the two nodes polarities.

We extend the forest combination operation to the polarized case. This is done in the same way tree combination is extended to polarized tree combination: Polarities are attached to nodes and an extra condition for the identification of two nodes is that their polarities combine; in that case the new node has the polarity which is the product of the two nodes polarities. For the following discussion we assume that a system of polarities $(P, \cdot)$ is given.

Definition 40. *A* **polarized forest** $\langle V, E, R, p\rangle$ *is a forest in which each node is associated with a polarity through a total function* $p : V \to P$*. If* $\langle V, E, R, p\rangle$ *is a polarized forest then* $\langle V, E, R\rangle$ *is its* **underlying forest***.*

Definition 41. *Two polarized forests are* **disjoint** *if their underlying forests are disjoint.*

Definition 42. *Two polarized forest* $F_1 = \langle V_1, E_1, R_1, p_1\rangle$*,* $F_2 = \langle V_2, E_2, R_2, p_2\rangle$ *are* **isomorphic***, denoted* $F_1 {\sim} F_2$*, if their underlying forests are isomorphic and, additionally, for all* $v \in V_1$*,* $p_1(v) = p_2(i(v))$*.*

The definition of isomorphism of sets of forests is trivially extended to sets of polarized forests.

Definition 43. *Let* $F_1 = \langle V_1, E_1, R_1, p_1\rangle$*,* $F_2 = \langle V_2, E_2, R_2, p_2\rangle$ *be two disjoint polarized forests. An equivalence relation '*$\overset{f}{\approx}$*' over* $V_1 \cup V_2$ *is* **legal** *if it is legal over the underlying forests of* F_1 *and* F_2 *and, additionally, for all* $v_1 \in V_1$ *and* $v_2 \in V_2$*, if* $v_1 \overset{t}{\approx} v_2$*, then* $p_1(v_1) \cdot p_2(v_2)\downarrow$*.*

$Eq_f(F_1, F_2)$ *is the set of legal equivalence relations over* $V_1 \cup V_2$*.*

Definition 44. *Let* $F_1 = \langle V_1, E_1, R_1, p_1\rangle$*,* $F_2 = \langle V_2, E_2, R_2, p_2\rangle$ *be two disjoint polarized forests and let '*$\overset{f}{\approx}$*' be a legal equivalence relation over* $V_1 \cup V_2$*. The* **polarized forest combination** *of* F_1, F_2 **with respect to '**$\overset{f}{\approx}$**'***, denoted* $F_1 +_{\overset{f}{\approx}} F_2$ *is a forest* $F = \langle V, E, R, p\rangle$ *where:*

- *V, E and R are as in definition 38*

- *for all* $[v]_{\overset{f}{\approx}} \in V$, $p([v]_{\overset{f}{\approx}}) = \begin{cases} (p_1 \cup p_2)(v) & \textit{if } [v]_{\overset{f}{\approx}} = \{v\} \\ (p_1 \cup p_2)(v) \cdot (p_1 \cup p_2)(u) & \textit{if } [v]_{\overset{f}{\approx}} = \{v, u\} \textit{ and} \\ & u \neq v \end{cases}$

The definition of forest combination of sets of forests is trivially extended to sets of polarized forests.

Example 18. *Consider again the systems of polarities depicted in Figure 5.7 and T_9, T_{10}, T_{11} of Figure 5.8. Let F_9, F_{10}, F_{11} be their corresponding forests, respectively. The forest combination of $\{F_9\}, \{F_{10}\}, \{F_{11}\}$ is depicted in Figures 5.14 (intermediate results) and 5.15. Here,*

$$(\{F_9\} \overset{f}{+} \{F_{10}\}) \overset{f}{+} \{F_{11}\} \cong \{F_9\} \overset{f}{+} (\{F_{10}\} \overset{f}{+} \{F_{11}\})$$

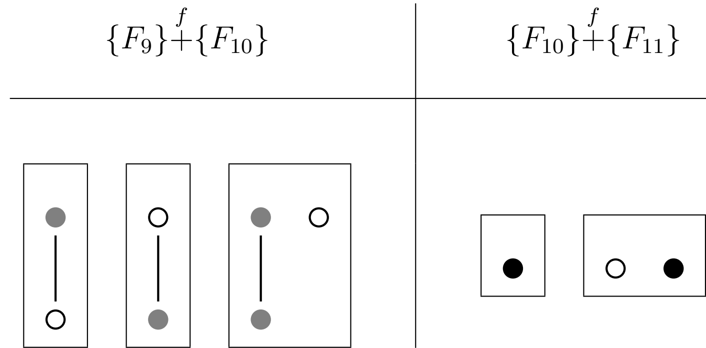


Figure 5.14: Intermediate results

In order to guarantee the associativity of tree combination we moved from trees to the powerset domain, i.e., to forests. However, our interest is in the trees rather than the forests. Therefore, after all the forests are combined, a resolution stage is required in which only desired solutions are retained. In our case, this is done by eliminating all forests which are not singletons. For example, executing the resolution stage over the forests of Figure 5.15, retains only the four forests of the upper row.

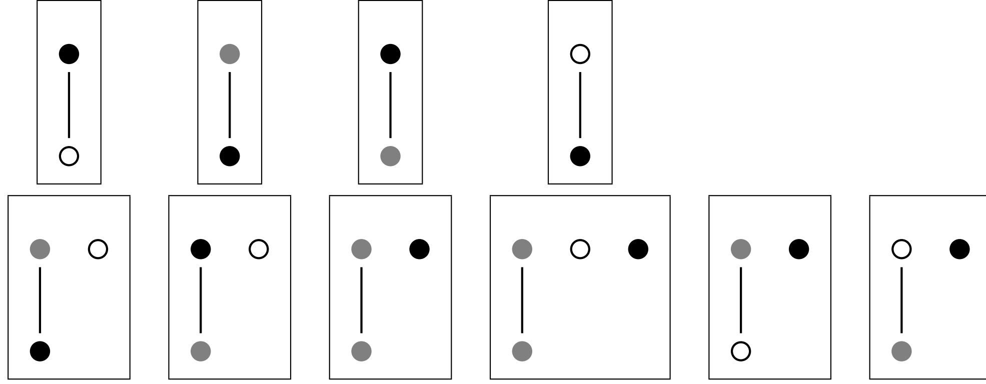


Figure 5.15: Forest combination of F_9, F_{10} and F_{11}: $(\{F_9\}\overset{f}{+}\{F_{10}\})\overset{f}{+}\{F_{11}\} \cong \{F_9\}\overset{f}{+}(\{F_{10}\}\overset{f}{+}\{F_{11}\})$

Theorem 15. *Forest combination is an associative operation: if $\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3$ are disjoint sets of forests then $((\mathcal{F}_1\overset{f}{+}\mathcal{F}_2)\overset{f}{+}\mathcal{F}_3) \cong (\mathcal{F}_1\overset{f}{+}(\mathcal{F}_2\overset{f}{+}\mathcal{F}_3))$. This holds both for non-polarized and for polarized combination, as long as $(P, \cdot)$ is commutative.*

The proof is given in the following section.

Summing up, we showed how to redefine tree combination in PUG in order to guarantee the associativity of the operation. In this way, the combination operator can be implemented more flexibly, independently of the order of the arguments, which results in more efficient computation. In particular, we showed corresponding (but associative!) computations of all the (non-associative) examples of the previous sections.

5.5 (Polarized) Forest Combination is Associative

We now show that forest combination (both with and without polarities) is an associative operation. We begin by proving the associativity of the non-polarized case. To do so, we need to show that if $F \in ((\mathcal{F}_1\overset{f}{+}\mathcal{F}_2)\overset{f}{+}\mathcal{F}_3)$ then $(\mathcal{F}_1\overset{f}{+}(\mathcal{F}_2\overset{f}{+}\mathcal{F}_3))$ includes an isomorphic forest of F. The isomorphism is required in order to ignore the irrelevant names of nodes. To be able to refer to any isomorphic tree of the combination result, we define *mutual*

combination: If F_3 is a forest combination of F_1 and F_2 with respect to some legal equivalence relation then both F_1 and F_2 are substructures of F_3, and furthermore, F_3 contains no redundant information: Every arc and node in F_3 belongs to either of the substructures that are induced by F_1 and F_2. This property is common for all the isomorphic trees of F_3. Moreover, F_1 and F_2 induce in all these isomorphic trees the exact same substructures. F_3 and all its isomorphic trees are mutual combinations of F_1 and F_2.

Definition 45. *Let F_1, F_2, F_3 be disjoint forests. F_3 is a* **mutual combination** *of F_1 and F_2, denoted $F_1 \oplus F_2 \mapsto F_3$, if there exists a total function $f : V_1 \cup V_2 \to V_3$ (***a combination function***) such that all the following hold:*

- *f is onto*
- *for all $u, v \in V_1 \cup V_2$, if u is the parent of v (in either F_1 or F_2) then $f(u)$ is the parent of $f(v)$ in F_3*
- *for all $u, v \in V_3$, if u is the parent of v in F_3 then there exist $u', v' \in V_1 \cup V_2$ such that u' is the parent of v' (in either F_1 or F_2), $f(u') = u$ and $f(v') = v$*
- *for all $u, v \in V_1 \cup V_2$, if $f(u) = f(v)$ and $u \neq v$ then either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$*

The second condition guarantees that F_1 and F_2 are substructures of F_3. The first and third conditions guarantee that F_3 contains no redundant information. The last condition guarantees that two different nodes in the same forest (representing different entities) correspond to different nodes in F_3. Lemma 16 and theorem 17 show that indeed mutual combination corresponds to forest combination.

Lemma 16. *If F_1, F_2, F_3, F_4 are disjoint forests such that $F_1 \oplus F_2 \mapsto F_3$ and $F_3 \sim F_4$, then $F_1 \oplus F_2 \mapsto F_4$.*

Proof. Let F_1, F_2, F_3, F_4 be disjoint forests such that $F_1 \oplus F_2 \mapsto F_3$ and $F_3 \sim F_4$. Then there exist a combination function $f : V_1 \cup V_2 \to V_3$ and an isomorphism $i : V_3 \to V_4$.

Define $h : V_1 \cup V_2 \to V_4$ where for all $v \in V_1 \cup V_2$, $h(v) = i(f(v))$. h is a combination function (the actual proof is suppressed) and hence, $F_1 \oplus F_2 \mapsto F_4$. $\square$

Theorem 17. *Let F_1, F_2, F_3 be disjoint forests. The following two conditions are equivalent:*

- $F_1 \oplus F_2 \mapsto F_3$
- *there exist a forest F_4 and a legal equivalence relation $\overset{f}{\approx} \in Eq_f(F_1, F_2)$ such that $F_4 = F_1 +_{\overset{f}{\approx}} F_2$ and $F_3 \sim F_4$*

Proof. Let F_1, F_2, F_3 be disjoint forests and assume that there exist a forest F_4 and a legal equivalence relation $\overset{f}{\approx} \in Eq_f(F_1, F_2)$ such that $F_4 = F_1 +_{\overset{f}{\approx}} F_2$ and $F_3 \sim F_4$. Observe that $V_4 = \{[v]_{\overset{f}{\approx}} | v \in v_1 \cup V_2\}$ and $E_4 = \{([u]_{\overset{f}{\approx}}, [v]_{\overset{f}{\approx}}) | (u, v) \in E_1 \cup E_2\}$. Define $h : V_1 \cup V_2 \to V_4$ where for all $v \in V_1 \cup V_2$, $h(v) = [v]_{\overset{f}{\approx}}$. h is a combination function and hence, $F_1 \oplus F_2 \mapsto F_4$. Since $F_4 \sim F_3$ and by lemma 16, $F_1 \oplus F_2 \mapsto F_3$.

Let F_1, F_2, F_3 be disjoint forests and assume that $F_1 \oplus F_2 \mapsto F_3$. Therefore, there exists a combination function $f : V_1 \cup V_2 \to V_3$. Define a relation '$\approx$' over $V_1 \cup V_2$ where for all $u, v \in V_1 \cup V_2$, $u \approx v$ iff $f(u) = f(v)$. Clearly, '$\approx$' is an equivalence relation. Furthermore, '$\approx$' is legal. Now, define $F_4 = F_1 +_{\approx} F_2$ and define $i : V_4 \to V_3$ where for all $[v]_{\approx} \in V_4$, $i([v]_{\approx}) = f(v)$. Notice that i is well defined because for all u, v such that $u \approx v$, $f(u) = f(v)$. i is an isomorphism of F_3 and F_4. $\square$

Notice that since forest isomorphism is reflexive and by theorem 17, if $F_1 +_{\overset{f}{\approx}} F_2 = F_3$ then $F_1 \oplus F_2 \mapsto F_3$.

Theorem 18. *Forest combination is an associative operation: if $\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3$ are disjoint sets of forests then $((\mathcal{F}_1 \overset{f}{+} \mathcal{F}_2) \overset{f}{+} \mathcal{F}_3) \cong (\mathcal{F}_1 \overset{f}{+} (\mathcal{F}_2 \overset{f}{+} \mathcal{F}_3))$*

Proof. Let $\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3$ be disjoint sets of forests and assume that $F = \langle V, E, R \rangle \in (\mathcal{F}_1 \overset{f}{+} \mathcal{F}_2) \overset{f}{+} \mathcal{F}_3$. Then there exist $F' \in \mathcal{F}_1 \overset{f}{+} \mathcal{F}_2$, $F_3 \in \mathcal{F}_3$ and $\approx_1 \in Eq_f(F', F_3)$ such that $F' +_{\approx_1} F_3 = F$. Therefore by theorem 17, $F' \oplus F_3 \mapsto F$, and hence, there exists a

combination function $f_1 : V' \cup V_3 \to V$. $F' \in \mathcal{F}_1 \overset{f}{+} \mathcal{F}_2$ and therefore there exist $F_1 \in \mathcal{F}_1$, $F_2 \in \mathcal{F}_2$ and $\approx_2 \in Eq_f(F_1, F_2)$ such that $F_1 +_{\approx_2} F_2 = F'$. Therefore by theorem 17, $F_1 \oplus F_2 \mapsto F'$ and hence, there exists a combination function $f_2 : V_1 \cup V_2 \to V'$. Define $f : V_1 \cup V_2 \cup V_3 \to V$ where:

$$f(v) = \begin{cases} f_1(v) & v \in V_3 \\ f_1(f_2(v)) & v \in V_1 \cup V_2 \end{cases}$$

Let F_4 be a graph defined by the restriction of f to $V_2 \cup V_3$, where:

- $V_4 = \{f(v) \mid v \in V_2 \cup V_3\}$
- $E_4 = \{(f(u), f(v)) \mid (u, v) \in E_2 \cup E_3\}$
- $R_4 = \{f(r) \mid r \in R_2 \cup R_3$ and for all $v \in V_2 \cup V_3$ such that $f(v) = f(r)$, $v \in R_2 \cup R_3\}$

F_4 is a forest and $f_{|V_2 \cup V_3}$ (the restriction of f to $V_2 \cup V_3$) is a combination function of F_2 and F_3 to F_4 (the actual proof is suppressed). Hence, $F_2 \oplus F_3 \mapsto F_4$ and therefore by theorem 17, there exist a forest F_5 and a legal equivalence relation $\approx_3 \in Eq_f(F_2, F_3)$ such that $F_5 = F_2 +_{\approx_3} F_3$ and $F_5 \sim F_4$. Hence, $F_5 \in \mathcal{F}_2 \overset{f}{+} \mathcal{F}_3$. Let $i : V_5 \to V_4$ be an isomorphism of F_5 and F_4. Define $h : V_5 \cup V_1 \to V$ where:

$$h(v) = \begin{cases} f(v) & v \in V_1 \\ i(v) & v \in V_5 \end{cases}$$

h is a combination function of F_1 and F_5 to F. Hence, $F_1 \oplus F_5 \mapsto F$, and therefore by theorem 17, there exists a forest F'' and a legal equivalence relation $\approx_4 \in Eq_f(F_1, F_5)$ such that $F'' = F_1 +_{\approx_4} F_5$ and $F'' \sim F'$. Hence, $F'' \in \mathcal{F}_1 \overset{f}{+} (\mathcal{F}_2 \overset{f}{+} \mathcal{F}_3)$ and $F'' \sim F'$.

The proof that if $F \in \mathcal{F}_1 \overset{f}{+} (\mathcal{F}_2 \overset{f}{+} \mathcal{F}_3)$ then there exists $F' \in (\mathcal{F}_1 \overset{f}{+} \mathcal{F}_2) \overset{f}{+} \mathcal{F}_3$ such that $F \sim F'$ is symmetric. $\square$

We now prove the associativity of polarized forest combination. The proof idea is similar to the proof of the non-polarized case. For the following discussion, assume that a system of polarities $(P, \cdot)$ has been specified.

Definition 46. *Let* F_1, F_2, F_3 *be disjoint polarized forests.* F_3 *is a* **mutual combination** *of* F_1 *and* F_2*, denoted* $F_1 \oplus F_2 \mapsto F_3$*, if there exists a total function* $f : V_1 \cup V_2 \rightarrow V_3$ *(***a combination function***) such that* f *is a combination function of the underlying forests, and, additionally, for all* $v_1, v_2 \in V_1 \cup V_2$*, if* $f(v_1) = f(v_2)$ *and* $v_1 \neq v_2$ *then* $(p_1 \cup p_2)(v_1) \cdot (p_1 \cup p_2)(v_2)\downarrow$ *and* $(p_1 \cup p_2)(v_1) \cdot (p_1 \cup p_2)(v_2) = p_3(f(v_1))$*.*

Lemma 19. *If* F_1, F_2, F_3, F_4 *are disjoint polarized forests such that* $F_1 \oplus F_2 \mapsto F_3$ *and* $F_3 \sim F_4$*, then* $F_1 \oplus F_2 \mapsto F_4$*.*

Proof. Similar to the proof of lemma 16. □

Theorem 20. *Let* F_1, F_2, F_3 *be disjoint polarized forests. The following two conditions are equivalent:*

- *there exist a forest* F_4 *and a legal equivalence relation* $\stackrel{f}{\approx} \in Eq_f(F_1, F_2)$ *such that* $F_4 = F_1 +_{\stackrel{f}{\approx}} F_2$ *and* $F_3 \sim F_4$
- $F_1 \oplus F_2 \mapsto F_3$

Proof. Similar to the proof of theorem 17. □

Theorem 21. *Let* $(P, \cdot)$ *be a system of polarities. Then polarized forest combination based on* $(P, \cdot)$ *is an associative operation: if* $\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3$ *are disjoint sets of forests then*

$$((\mathcal{F}_1 \stackrel{f}{+} \mathcal{F}_2) \stackrel{f}{+} \mathcal{F}_3) \cong (\mathcal{F}_1 \stackrel{f}{+} (\mathcal{F}_2 \stackrel{f}{+} \mathcal{F}_3))$$

Proof. Similar to the proof of theorem 18. □

5.6 Forest Combination and XMG

The results of the previous section bear relevance to the metagrammar paradigm and specifically to XMG (Duchier, Le Roux, and Parmentier, 2004; Crabbé, 2005). In particular, the forest-based grammar combination operation can be instrumental for defining an alternative semantics for XMG, which we sketch in this section.

XMG provides the grammar writer with a tree-description logic, whose semantics is based on trees. A given formula denotes an infinite set of trees, each satisfying the conditions of the formula. This denotation is restricted by considering only the finite set of *minimal* trees satisfying the description (Duchier and Gardent, 1999; Duchier and Gardent, 2001). Conceptually, computation of the minimal tree models of a given formula consists of two stages: The first computes the (infinite set of) tree models of a formula and the second extracts from these models only the minimal ones. The following definitions are based on Duchier and Gardent (1999) and Duchier and Gardent (2001).

Definition 47. *A* **formula** ϕ *is an arbitrary conjunction of dominance and labeling constrains*

$$\phi ::= \phi \wedge \phi' \mid x \lhd y \mid x = y \mid x \bot y$$

where x, y are taken from a set of variables.[3]

The semantics is given by interpretation over finite tree structures.

Definition 48. *Let V_ϕ be the set of variables occurring in a formula ϕ. A* **tree solution** *of ϕ is a pair (T, I) where $T = \langle V, E, r\rangle$ is a finite tree (a* **tree model***) and $I : V_\phi \mapsto V$ is a function (an* **interpretation***) that maps each variable in ϕ to a node in T. $x \lhd y$ means that, in the solution tree T , $I(x)$ must dominate $I(y)$; $x = y$ means that $I(x) = I(y)$; and $x \bot y$ means that $I(x) \neq I(y)$. The* **denotation** *of a formula ϕ, denoted $S_{xmg}(\phi)$ is the set of its tree solutions $\{(T, I) \mid (T, I)$ is a tree solution of $\phi\}$.*

If T is a tree model of ϕ, then every tree T' which contains T as a subtree is also a tree model of ϕ. Therefore, there are infinitely many tree models of any formula ϕ. To restrict the infinite set to desired trees, *minimal* (finite) models are considered. Any formula ϕ has only finitely many minimal tree models (up to isomorphism).

[3] Duchier and Gardent (1999) and Duchier and Gardent (2001) define several more operators (e.g., precedence and labeling). For the sake of simplicity we restrict ourselves to the list of operators presented in this definition, but all the results can be extended to the full list of operators.

Definition 49. *A tree model T is a* **minimal tree model**[4] *of ϕ if all nodes in T interpret at least one variable in ϕ. Then $extract(S_{xmg}(\phi)) = \{T \mid (T, I) \in S_{xmg}(\phi)$ and T is a minimal tree model of $\phi\}$.*

We propose an alternative semantics, denoted S_{fc}, for tree descriptions, based on the forest combination operation of section 5.4. In S_{fc} a formula denotes the set of minimal forests satisfying it. Forest combination operates directly on minimal forests in a way that corresponds to formula conjunction in the syntactic level: The denotation of a conjunction of formulas is the combination of the denotations of the conjuncts. Here, also, a resolution stage is required, to retain only forests which are singletons (i.e., trees).

Definition 50. *Let V_ϕ be the set of variables occurring in a formula ϕ. A* **forest solution** *of ϕ is a pair (F, I) where $F = \langle V, E, R\rangle$ is a finite minimal forest (a* **forest model***) and $I : V_\phi \mapsto V$ is an onto function (an* **interpretation***) that maps each variable in ϕ to a node in F such that all nodes in F interpret at least one variable in ϕ. $x \lhd y$ means that, in the solution forest F , $I(x)$ must dominate $I(y)$; $x = y$ means that $I(x) = I(y)$; and $x \perp y$ means that $I(x) \neq I(y)$. The* **denotation** *of a formula ϕ, denoted $S_{fc}(\phi)$, is the set of its forest solutions $\{(F, I) \mid (F, I)$ is a forest solution of $\phi\}$. Define $resolve(S_{fc}(\phi)) = \{F \mid (F, I) \in S_{fc}(\phi)$ and F is a singleton$\}$.*

Observe that in this semantics a formula can denote only finitely many forests (up to isomorphism). The two semantics, S_{xmg} and S_{fc}, coincide.

Theorem 22. $extract(S_{xmg}(\phi)) = resolve(S_{fc}(\phi))$

Proof. Assume $T \in extract(S_{xmg}(\phi))$. Then, there exists an interpretation I from the variables of ϕ to the nodes of T such that (T, I) is a tree solution of ϕ and T is a minimal

[4] In Duchier and Gardent (1999) and Duchier and Gardent (2001), the definition of minimal tree models is based on the notion of D-trees (Rambow, Vijay-Shanker, and Weir, 1995). For the sake of simplicity we do not use this notion, but all the results can be easily extended to D-trees.

tree model. Any tree is also a forest (a singleton) and therefore, T is also a forest model of ϕ. Hence, $(T, I) \in S_{fc}(\phi)$. Since T is a tree, it follows that $(T, I) \in resolve(S_{fc}(\phi))$.

Now assume that $F \in resolve(S_{fc}(\phi))$. Then, there exists an interpretation I from the variables of ϕ to the nodes of F such that (F, I) is a forest solution of ϕ and F is a tree. Since F is a tree, $(F, I) \in S_{xmg}(\phi)$. (F, I) is a forest solution of ϕ and therefore F is a minimal model. Hence, $(F, I) \in extract(S_{xmg}(\phi))$. □

Since the two semantics coincide, either one of them can be used in an implementation of XMG. Specifically, in the existing implementation of XMG the grammar designer is presented with finite trees only, and the infinite tree models are never explicit. S_{fc} offers the opportunity to use finite trees as the bona fide denotation of tree descriptions. This, however, comes with a cost: the number of tree fragments can grow very fast, and a sophisticated cashing mechanism will be necessary in any practical implementation.

Both approaches require a resolution stage; the resolution stage in the forest combination approach seems to be simpler, requiring only the extraction of singletons from a set. However, it could also be less efficient, due to the growth in the number of trees and the fact that resolution is deferred to the end of the computation.

To sum up, the forest combination semantics provides the grammar writer with a formally defined operation executed directly on the minimal models amounting to the conjunction operation in the syntactic level of tree descriptions. Whether or not it can be practically beneficial remains to be seen.

5.7 Conclusion

We have shown how the tree combination operation in PUG can be redefined to guarantee associativity, thus facilitating the use of this powerful and flexible formalism for grammar engineering and modular grammar development. The key to the solution is a *powerset-lift* of the domain and the corresponding operation: Rather than working with trees, manip-

ulating forests provides means to 'remember' *all* the possible combinations of grammar fragments. Then, after all fragments are combined, a *resolution* stage is added to produce the desired results. The same powerset-lift has been used to maintain the associativity of signature modules combination with respect to the Ap-relation (section 2.2). We believe that this method is sufficiently general to be applicable to a variety of formalisms. In particular, it is applicable to the general case of PUG where arbitrary objects and structures are manipulated. In this case also, the move to the powerset domain by manipulating sets of objects, rather than the objects themselves, enforces associativity.

Chapter 6

Discussion and conclusions

We presented the foundations 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 1.2 as we show below.

Signature focus: Our solution focuses on the modularization of the signature (chapter 2) and the extension to grammar modules (section 2.5) 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 partial information about the subtyping and appropriateness relations. Unlike ordinary signatures, the appropriateness relation is not a function and 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 to distribute the grammar between several mod-

ules and the relaxation of well-typedness also support this desideratum.

Extensibility: In section 2.4 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, like 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 3.

Modules provide *abstraction*; for example, the module $List$ of Figure 2.12 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.

While 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 (chapter 4). 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 1.2). Furthermore, while 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.

Bibliography

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, Italy, February.

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. Taipei, Taiwan*, pages 8–14.

Bender, Emily M. and Dan Flickinger. 2005. Rapid prototyping of scalable grammars: Towards modularity in extensions to a language-independent core. In *Proceedings of IJCNLP-05*, Jeju Island, Korea.

Bender, Emily M., Dan Flickinger, Fredrik Fouvry, and Melanie Siegel. 2005. Shared

representation in multilingual grammar engineering. *Research on Language and Computation*, 3:131–138.

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

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

Brogi, Antonio, Paolo Mancarella, Dino Pedreschi, and Franco Turini. 1994. Modular logic programming. *ACM transactions on programming languages and systems*, 16(4):1361–1398, July.

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, Denemark. Association for Computational Linguistics.

Carpenter, Bob. 1992a. ALE – the attribute logic engine: User's guide. Technical report, Laboratory for Computational Linguistics, Philosophy Department, Carnegie Mellon University, Pittsburgh, PA 15213, December.

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

Cohen-Sygal, Yael and Shuly Wintner. 2006. Partially specified signatures: A vehicle for grammar modularity. In *Proceedings of the 21st International Conference on Com-*

putational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (COLING-ACL), pages 145–152, Sydney, Australia, July.

Cohen-Sygal, Yael and Shuly Wintner. 2007. The non-associativity of polarized tree-based grammars. In *Gelbukh, A., editor, Proceedings of the Conference on Computational Linguistics and Intelligent Text Processing (CICLing-2007), volume 4394 of Lecture Notes in Computer Science (LNCS), Berlin and Heidelberg: Springer*, pages 208–217, Mexico City, Mexico, February.

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

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)*, Athens, Greece.

Crabbé, Benoit. 2005. Grammatical development with XMG. In *Proceedings of the 5th International Conference on Logical Aspects of Computational Linguistics (LACL)*, Bordeaux, France, April.

Crabbé, Benoit and Denys Duchier. 2004. Metagrammar redux. In *CSLP*, Copenhagen, Denemark.

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

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, Edinburgh, August.

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

Duchier, Denys and Claire Gardent. 2001. Tree descriptions, constraints and incrementality. In Harry Bunt, Reinhard Muskens, and Elias Thijsse, editors, *Computing Meaning, Volume 2*, volume 77 of *Studies In Linguistics And Philosophy*. Kluwer Academic Publishers, Dordrecht, December, pages 205–227.

Duchier, Denys, Joseph Le Roux, and Yannick Parmentier. 2004. The metagrammar compiler: An NLP application with a multi-paradigm architecture. In *Proceedings of the Second International Mozart/Oz Conference (MOZ 2004)*, Charleroi, Belgium, October.

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

Fodor, Jerry. 1983. *The modularity of Mind*. MIT Press, Cambridge, Mass.

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

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, UK.

Joshi, Aravind K., L. Levy, and M. Takahashi. 1975. Tree Adjunct Grammars. *Journal of Computer and System Sciences*.

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, Australia, July.

Kahane, Sylvain and Francois Lareau. 2005. Meaning-text unification grammar: modularity and polarization. In *Proceedings of the 2nd International Conference on Meaning-Text Theory*, pages 197–206, Moscow.

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, USA. Association for Computational Linguistics.

Kasper, Walter and Hans-Ulrich Krieger. 1996. Modularizing codescriptive grammars for efficient parsing. In *Proceedings of the 16th Conference on Computational Linguis tics*, pages 628–633, Kopenhagen. Also available as Verbmobil-Report 140.

Keselj, Vlado. 2001. Modular hpsg. In *Proceedings of the 2001 IEEE Systems, Man, and Cybernetics Conference*, October.

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, Mass. MIT Press.

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

Meurers, Detmar, Gerald Penn, and Frank Richter. 2002. *A web-based instructional platform for constraint-based grammar formalisms and parsing*. In Dragomir Radev and Chris Brew, editors, Effective Tools and Methodologies for Teaching NLP and CL, The Association for Computational Linguistics, New Brunswick, NJ.

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

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, 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.

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

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.

Rambow, Owen, K. Vijay-Shanker, and David Weir. 1995. D-tree grammars. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics*, pages 151–158.

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 (Ed.) Proceedings of FG 2008: The 13th conference on Formal Grammar*, pages 113–128, Hamburg, Germany, August.

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

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, Lecture Notes in Computer Science, Berlin: Springer Verlag*, 5533.

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, Italy, February.

Appendix A

Compactness

Definition 51. *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 52. *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* $(q_1, F, q'_2) \in Ap$*.*

For an example of the above two definitions see example 3.

The following definitions set the basis for determining whether two nodes are indistinguishable or not. Since 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 53. *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 54. *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 55. *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 56. *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 $S \mid^{strict}_{env(q_1)} \sim S \mid^{strict}_{env(q_2)}$ via an isomorphism i such that $i(q_1) = q_2$.*

Example 19. *Let S_1 be the signature module of Figure A.1. $env(q_4) = env(q_7) = \{q_1, q_4, q_7\}$, $env(q_2) = env(q_6) = \{q_1, q_2, q_6\}$, $env(q_5) = \{q_1, q_5, q_8\}$ and $env(q_1) = \{q_1\}$. The strict restrictions of S_1 to these environments are depicted in Figure A.2. $q_2 \approx q_4$ and $q_6 \approx q_7$, where in both cases the isomorphism is $i = \{q_1 \mapsto q_1, q_2 \mapsto q_4, q_6 \mapsto q_7\}$. 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.*

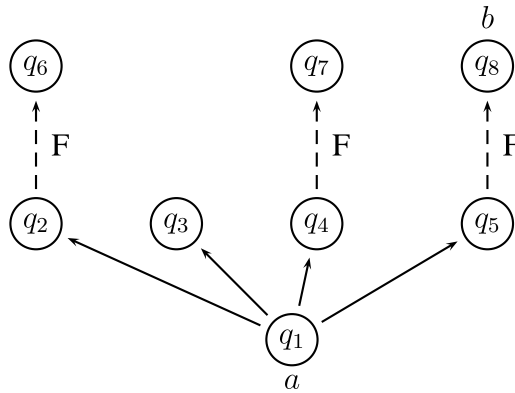


Figure A.1: A signature module with indistinguishable nodes, S_1

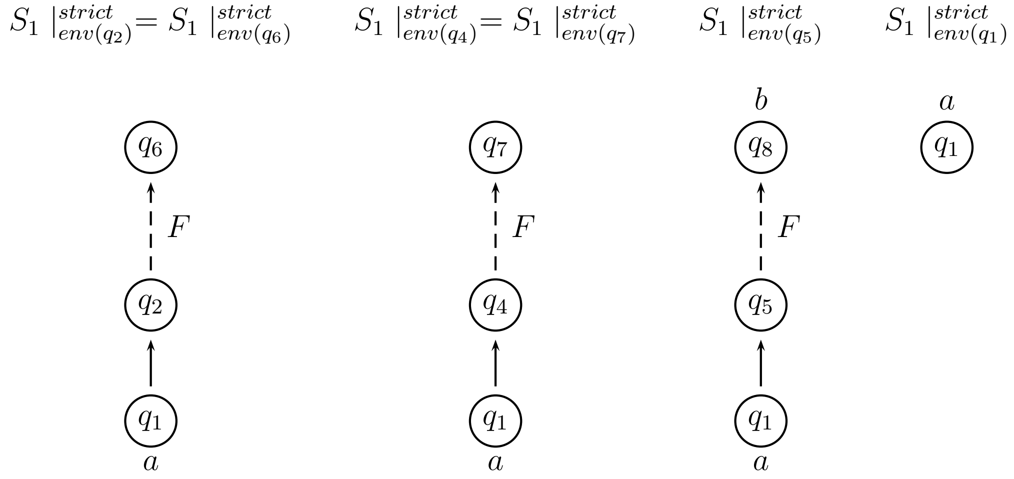


Figure A.2: Strict restriction subgraphs

Theorem 23. *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.*

Proof. Let $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ be a pre-signature module.

Reflexivity: For all $q \in Q$ $S \mid^{strict}_{env(q)} \sim S \mid^{strict}_{env(q)}$ by the identity function $i : env(q) \rightarrow env(q)$ that maps each node in $env(q)$ to itself. Evidently, i is an isomorphism and hence $q \approx q$.

Symmetry: Let $q_1, q_2 \in Q$ be such that $q_1 \approx q_2$. Therefore $S \mid^{strict}_{env(q_1)} \sim S \mid^{strict}_{env(q_2)}$ via an isomorphism $i : env(q_1) \rightarrow env(q_2)$. $S \mid^{strict}_{env(q_2)} \sim S \mid^{strict}_{env(q_1)}$ via the isomorphism $i^{-1} : env(q_2) \rightarrow env(q_1)$ (the detailed proof that i^{-1} is indeed such an isomorphism is suppressed). Hence, $q_2 \approx q_1$.

Transitivity: Let $q_1, q_2, q_3 \in Q$ be such that $q_1 \approx q_2$ and $q_2 \approx q_3$. Hence, $S \mid^{strict}_{env(q_1)} \sim S \mid^{strict}_{env(q_2)}$ via an isomorphism $i_1 : env(q_1) \rightarrow env(q_2)$ and $S \mid^{strict}_{env(q_2)} \sim S \mid^{strict}_{env(q_3)}$ via an isomorphism $i_2 : env(q_1) \rightarrow env(q_2)$. Therefore, $S \mid^{strict}_{env(q_1)} \sim S \mid^{strict}_{env(q_3)}$ via the isomorphism $i_1 \circ i_2 : env(q_1) \rightarrow env(q_3)$ defined by $i_1 \circ i_2 : (q) = i_2(i_1(q))$ for all $q \in env(q_1)$ (again, the detailed proof that $i_1 \circ i_2$ is indeed such an isomorphism is

suppressed).

□

Definition 57. *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 58. *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 ($S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$)

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:*
   - $Q_1 = Q$
   - $T_1 = T$
   - $\preceq_1 = \{(q_1, q_2) \in \preceq \mid (q_1, q_2)$ *is a non-redundant subtyping arc in* $S\}$
   - $Ap_1 = \{(q_1, F, q_2) \in Ap \mid (q_1, F, q_2)$ *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 20. *Consider again S_1, the signature module of Figure A.1. The compacted signature module of S_1 is depicted in Figure A.3. Notice that S_1 has no redundant arcs to be removed and 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.*

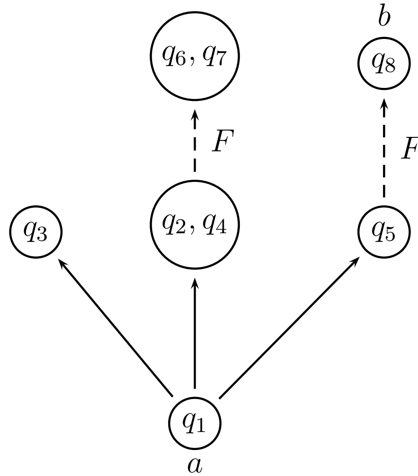


Figure A.3: The compacted signature module of S_1

Example 21. *Consider S_2, S_3, S_4, S_5, the signature modules depicted in Figure A.4. 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.*

Theorem 24. *The compactness algorithm terminates.*

Proof Idea. Stage 1 of the algorithm removes redundant arcs, if such exist. In stage 2, the signature module is coalesced. If it is non-redundant then the coalesce algorithm returns a signature module in which each new node is an equivalence class containing the previous node and then the algorithm terminates in stage 3. If the signature module includes nodes that should be identified, then the coalesce algorithm reduces the number of nodes at least

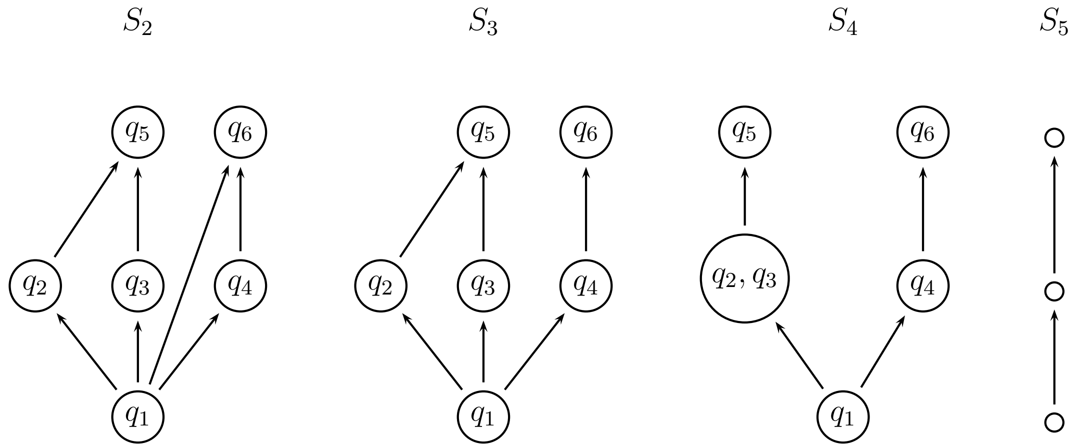


Figure A.4: A compactness example

in one node. To sum up, each iteration of stages 1 and 2 reduces the number of arcs and nodes in the signature module. Since the number of arcs and nodes in a signature module is finite, there could be only a finite number of such iterations and the algorithm terminates. □

Theorem 25. *The compactness algorithm is deterministic, i.e., always produces the same result.*

Proof. Follows immediately from the fact that there is no element of choice in any of the algorithm stages. □

Theorem 26. *If S is a signature module then $\mathit{compact}(S)$ is a non-redundant signature module.*

Proof. Follows immediately from the fact that the algorithm terminates only when a non-redundant module is obtained (stage 3). The termination of the algorithm is guaranteed by theorem 24. □

Theorem 27. *If S is a non-redundant signature module then $\mathit{compact}(S) \sim S$.*

Proof. Let S be a non-redundant signature module. In that case, stage 1 of the algorithm has no effect on S. In stage 2, S is coalesced into $coalesce(S)$. Define a function $i : Q \rightarrow Q_1$ by $i(q) = [q]_{\approx}$. i is an isomorphism between S and $coalesce(S)$ (the detailed proof is suppressed). Evidently, $coalesce(S)$ is non-redundant and hence the algorithm terminates in stage 3, returning $coalesce(S)$ as $compact(S)$. Hence, $S \sim compact(S)$. $\square$

Appendix B

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. We show how for a given anonymous node the set of its *typed* equivalent nodes is calculated. The calculation is reminiscent of the calculation of isomorphic nodes (see Appendix A) but some modifications are required to deal with the comparison of typed nodes versus anonymous nodes as will be shown below.

Definition 59. *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* **quasi-isomorphic**, *denoted* $S_1 \overset{q}{\sim} S_2$, *if there exists a total, one-to-one and onto function* i *(***quasi-isomorphism***) mapping the nodes of* S_1 *to the nodes of* S_2*, such that all the following hold:*

1. *for all* $q \in Q_1$*, if* $T_1(q)\downarrow$ *and* $T_2(i(q))\downarrow$ *then* $T_1(q) = T_2(i(q))$

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* $(i(q), F, i(q')) \in Ap_2$.

Quasi-isomorphism is a relaxed version of isomorphism (definition 53) since it allows an anonymous node to be mapped onto a typed node and vise versa, but two typed nodes, mapped to each other, still must be marked by the same type.

Definition 60. *The* **extended environment** *of* q, *denoted* $extenv(q)$, *is the smallest set such that:*

- $q \in extenv(q)$;
- *If* $q'' \in extenv(q)$ *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* $q' \in extenv(q)$.

The *extended environment* of a node q extends its environment (definition 54) by including all the nodes accessible from q via any sequence of arcs, regardless of whether these nodes are typed or not. Clearly, the environment of a node is a subset of its extended environment.

Definition 61. *Let* $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ *be a pre-signature module and let* $Q' \subseteq Q$. *The* **restriction** *of* S *to* Q', *denoted* $S|_{Q'}$, *is* $\langle\langle Q', T_1, \preceq_1, Ap_1\rangle, Int_1, Imp_1, Exp_1\rangle$, *where:*

- $T_1 = T_{|Q'}$
- $q_1 \preceq_1 q_2$ *iff* $q_1 \preceq q_2$ *and* $q_1, q_2 \in Q'$
- $(q_1, F, q_2) \in Ap_1$ *iff* $(q_1, F, q_2) \in Ap$ *and* $q_1, q_2 \in Q'$
- $Int_1 = Int_{|Q'}$
- $Imp_1 = Imp_{|Q'}$
- $Exp_1 = Exp_{|Q'}$

The strict restriction (definition 55) of a pre-signature module, S, to a set of nodes, Q', is a subgraph of the restriction of S to Q' as it does not include any labeled or unlabeled arcs connecting two typed nodes. In particular, notice that the graph that is strictly-induced by $env(q)$ is a subgraph of the graph induced by $extenv(q)$.

Definition 62. *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* **quasi-indistinguishable***, denoted $q_1 \overset{q}{\approx} q_2$, if $S \mid_{extenv(q_1)} \overset{q}{\sim} S \mid_{extenv(q_2)}$ via a quasi-isomorphism i such that $i(q_1) = q_2$.*

Compare the two notions of node similarity defined in definition 56 and definition 62. To determine if two nodes are indistinguishable, smaller sets of nodes are inspected than in determining quasi-indistinguishability. However in the latter the mapping between these two sets is more liberal, allowing a typed node to be mapped onto an anonymous one and vise versa.

In a signature module any two indistinguishable nodes are also quasi-indistinguishable, but two quasi-indistinguishable nodes can still be distinguishable as the following example shows:

Example 22. *Consider the signature module depicted in Figure B.1. $extenv(q_1) = extenv(q_2) = extenv(q_3) = \{q_1, q_2, q_3\}$ and $extenv(q_4) = extenv(q_5) = extenv(q_6) = \{q_4, q_5, q_6\}$. $q_1 \overset{q}{\approx} q_4$, $q_2 \overset{q}{\approx} q_5$ and $q_3 \overset{q}{\approx} q_6$ where in all cases the quasi-isomorphism is $\{q_1 \mapsto q_4, q_2 \mapsto q_5, q_3 \mapsto q_6\}$. Observe that $q_1 \not\approx q_4$, $q_2 \not\approx q_5$ and $q_3 \not\approx q_6$.*

Theorem 28. *Let $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ be a signature module. For all $q_1, q_2 \in Q$, if $q_1 \approx q_2$ then $q_1 \overset{q}{\approx} q_2$.*

Proof. Let $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ be a signature module and let $q_1, q_2 \in Q$ be such that $q_1 \approx q_2$. By the definition of the relation '$\approx$' it follows that $S \mid^{strict}_{env(q_1)} \sim S \mid^{strict}_{env(q_2)}$ via an isomorphism i such that $i(q_1) = q_2$. Evidently, $env(q_1) \subseteq extenv(q_1)$

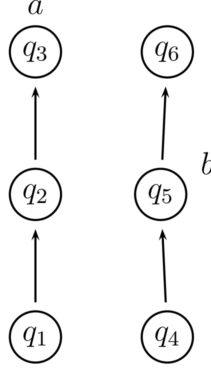


Figure B.1: Quasi-indistinguishability versus indistinguishability

and $env(q_2) \subseteq extenv(q_2)$. Let $i_1 : extenv(q_1) \rightarrow extenv(q_2)$ be a function defined by:

$$i_1(q) = \begin{cases} i(q) & q \in env(q_1) \\ q & \text{otherwise} \end{cases}$$

Clearly, $i_1(q_1) = q_2$. $S \mid_{extenv(q_1)} \sim S \mid_{extenv(q_2)}$ via the isomorphism i_1. The detailed proof that i_1 is indeed such an isomorphism is suppressed. However, notice that since S is a signature module, each typed node is unique with respect to its type and therefore all the nodes that belong to the extended environment and not to the environment are mapped to themselves. Hence, $q_1 \overset{q}{\approx} q_2$. □

Notice that theorem 28 does not hold when general pre-signature modules are concerned since they may contain multiple nodes marked by the same type.

As theorem 23 shows, indistinguishability is an equivalence relation over the nodes of a pre-signature module. However, this is not the case with the quasi-indistinguishability relation.

Theorem 29. *Let $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ be a signature module. Then:*

1. *'$\overset{q}{\approx}$' is a reflexive and symmetric relation over Q.*

2. '$\stackrel{q}{\approx}$' *is not necessarily a transitive relation over* Q.

Proof. Let $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ be a pre-signature module.

1. The proof of reflexivity and symmetry of '$\stackrel{q}{\approx}$' is the same as in theorem 23.

2. Let S be the signature module depicted in Figure B.2. $q_1 \stackrel{q}{\approx} q_2$ and $q_2 \stackrel{q}{\approx} q_3$ but $q_1 \stackrel{q}{\not\approx} q_3$ since $T(q_1) = a \neq b = T(q_3)$.

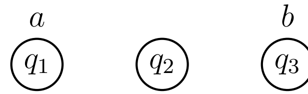


Figure B.2: A signature module with a non transitive '$\stackrel{q}{\approx}$' relation

□

The input to the name resolution algorithm is a non-redundant signature module and its output is a non-redundant 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 5. 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* $Q_q = \{q' \in Q \mid T(q')\downarrow$ *and* $q \stackrel{q}{\approx} q'\}$.

2. *let* $\overline{Q} = \{q \in Q \mid T(q)\uparrow$ *and* $|Q_q| = 1\}$. *If* $\overline{Q} \neq \emptyset$ *then:*

   2.1. *for all* $q \in \overline{Q}$, $S := NodeCoalesce(S, q, q')$ *where* $Q_q = \{q'\}$ *(for the definition of NodeCoalesce, see definition 21)*

   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.*

The name resolution algorithm is reminiscent of the compactness algorithm, but some changes are required to deal with comparison of typed nodes versus anonymous nodes. Most significantly, the quasi-indistinguishability relation ('$\overset{q}{\approx}$', stage 1) is used to compare nodes instead of indistinguishability (definition 56). Further more, the subroutine $NodeCoalesce$ is used to directly coalesce two nodes instead of coalescing equivalent classes as in the compactness algorithm. The reasons for these changes are exemplified in the following example:

Example 23. *Consider the signature modules depicted in Figure B.3. In S_6, $Q_{q_2} = Q_{q_5} = \{q_8\}$, $Q_{q_7} = \{q_1, q_4\}$ and $Q_{q_3} = Q_{q_6} = Q_{q_9} = \emptyset$. The result of executing stage 2.1 of the name resolution algorithm on S_6 is S_7. In S_7, q_3, q_6 and q_9 are all indistinguishable from each other, and therefore stage 2.2 results in S_8. Then, returning to stage 1, there is no anonymous node, q, for which Q_q is a singleton, and hence stage 3 is applied, resulting in S_9 which is the final result.*

The result after executing stage 2.1 is S_7 which is obtained by coalescing q_2 and q_5 with q_8. Notice that the environment (definition 54) of q_8 includes only q_8 and the environment of q_5 includes q_4, q_5 and q_6. If we want to compare q_8 with q_5 we must add to the environment of q_8 also q_7 and q_9. More generally, the environment of a node must include also the nodes which are connected to typed nodes and addition of nodes to the environment cannot be stopped when a typed node is encountered. This is why the name resolution algorithm, through the '$\overset{q}{\approx}$' relation, compares extended environments instead of environments . Furthermore, when the extended environments of q_8 and q_5 are compared, evidently they are not isomorphic and in particular there is no isomorphism that maps q_5 to q_8 (since the former is anonymous and the later is typed). Quasi-isomorphism relaxes isomorphism by allowing a typed node to be mapped to an anonymous node and vise versa, thus the extended environment of q_5 is quasi-isomorphic to the extended envi-

ronment of q_8 via a quasi-isomorphism that maps q_5 to q_8. This is why the name resolution algorithm uses quasi-isomorphism and not isomorphism. Again, this is done through the '$\overset{q}{\approx}$' relation (stage 1). Finally, observe that $q_1 \overset{q}{\approx} q_7$ and $q_7 \overset{q}{\approx} q_4$ but $q_1 \overset{q}{\not\approx} q_4$ since q_1 and q_4 are marked by different types. Evidently, '$\overset{q}{\approx}$' is not transitive and hence not an equivalence relation. This is why we use the subroutine NodeCoalesce and do not use coalescion of equivalent classes as in the compactness algorithm (see definition 58).

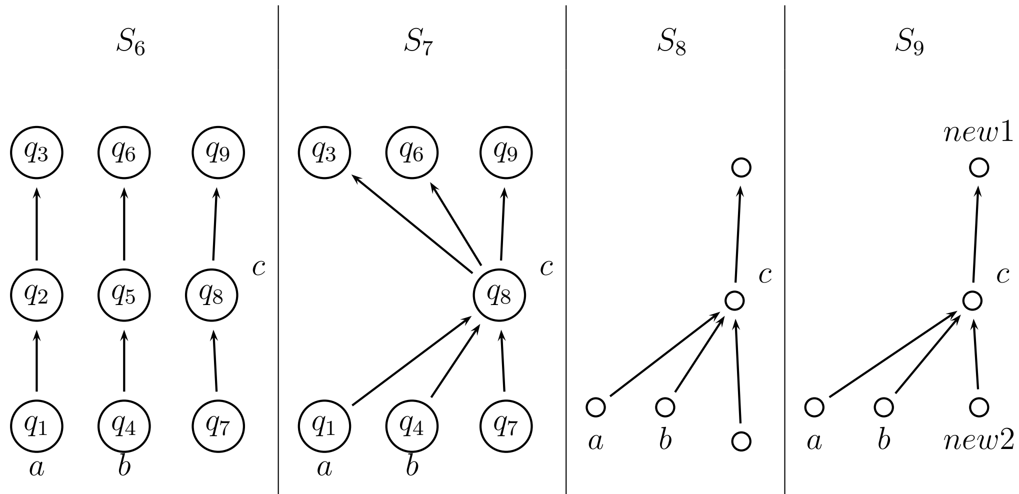


Figure B.3: Name resolution

Example 24. *Consider the signature modules S_{10}, S_{11}, S_{12} depicted in Figure B.4. In S_{10}, $Q_{q_3} = \{q_2\}$ and $Q_{q_4} = Q_{q_6} = \emptyset$. The result of executing stage 2.1 of the name resolution algorithm over S_{10} is S_{11}. S_{11} is non-redundant and therefore stage 2.2 has no effect. However, in S_{11}, $Q_{q_4} = \{q_2\}$ and $Q_{q_6} = \{q_5\}$ and therefore another iteration of stages 1-2 is required (stage 2.3). This second iteration results in S_{12} which is also the final result.*

Theorem 30. *The name resolution algorithm terminates.*

Proof Idea. Stage 1 of the algorithm computes for each anonymous node the set of its typed equivalent nodes. In stage 2, first, pairs of an anonymous node and its unique typed equivalent node (if such exist) are coalesced. If no such pair exists, the algorithm

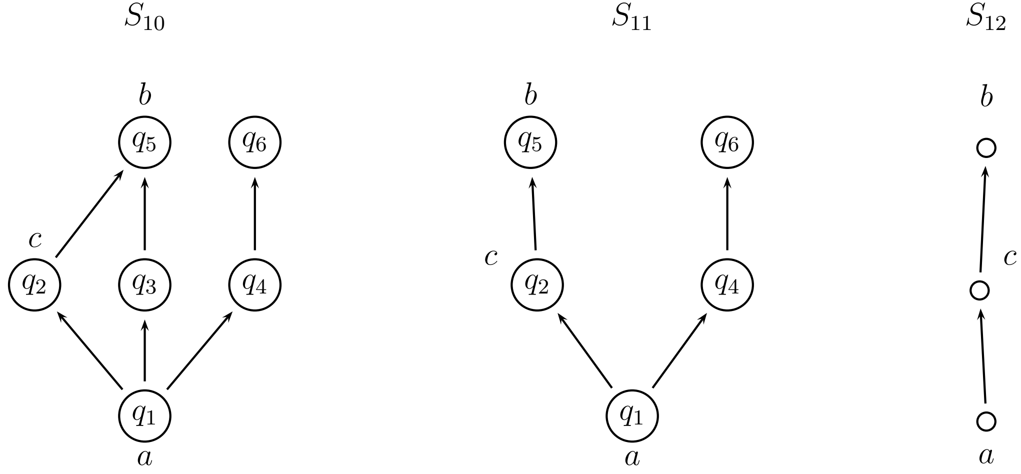


Figure B.4: Name resolution

advances to stage 3 and terminates. If such pairs exist, they are coalesced, reducing the number of anonymous nodes by at least one node. Then, the module is compacted, and by theorem 24 (Appendix A), the compactness algorithm terminates (notice that compaction cannot turn a typed node into an anonymous node and therefore, cannot increase the number of anonymous nodes). To sum up, each iteration of stages 1 and 2 terminates and reduces the number of anonymous nodes in the signature module. Since the number of nodes in a signature module is finite and therefore so is the number of anonymous nodes, there could be only a finite number of such iterations and then the algorithm advances to stage 3 where it terminates. □

Theorem 31. *If S is a signature module then $NameResolution(S)$ is a signature module whose typing function is total.*

Proof. If S is a signature module and $q \overset{q}{\approx} q'$, then $NodeCoaelesce(S, q, q')$ is also a signature module (the technical proof is suppressed). From theorem 26 it follows that the compactness of stage 2.2 also returns a signature module. Hence, if S is a signature module, each iteration of stages 1 and 2 of the algorithm produces a signature module. From theorem 30 it follows that the number of iterations of stages 1 and 2 is finite. Then,

the algorithm advances to stage 3 where all remaining anonymous nodes are assigned types and the algorithm terminates. Hence, $NameResolution(S)$ is a signature module whose typing function is total. $\square$

Theorem 32. *If S is a signature module whose typing function is total, then $NameResolution(S) = S$.*

Proof. If S is a signature module whose typing function is total, $\overline{Q} = \emptyset$ at stage 2 of the first iteration. Therefore, the algorithm moves to stage 3, which has no effect since T is total. Hence, $NameResolution(S) = S$. $\square$

Appendix C

Grammar modules

C.1 Defining Grammar Modules

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, since signature modules may include anonymous nodes. Therefore, we modify the standard definition of TFSs such that every path in a TFS is assigned a node in the signature module over which it is defined, rather than a type.

Definition 63. *Let* $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ *be a signature module. A* **typed feature structure** *(*TFS*) over* S *is a pre-*TFS$A = \langle \Pi, \Theta, \bowtie\rangle$ *for which the following requirements hold:*

- Π *is prefix-closed,* A *is fusion-closed and* $\bowtie$ *is an equivalence relation with a finite index (as in definition 8)*
- $\Theta : \Pi \rightarrow Q$ *is a total function, assigning a node to each path that respects the equivalence: if* $\pi_1 \bowtie \pi_2$ *then* $\Theta(\pi_1) \approx \Theta(\pi_2)$

Notice that for Θ to respect the equivalence, it is required to assign to equivalent paths indistinguishable nodes and not necessarily the exact same node. By the definition

of signature modules, indistinguishable nodes will eventually be coalesced and therefore denote the same type.

Definition 64. *Let* $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ *be a signature module. A* **typed multi-rooted structure** *(*TMRS*) over* S *is a pre-*TMRS $\sigma = \langle Ind, \Pi, \Theta, \bowtie\rangle$*, where:*

- Π *is prefix-closed,* σ *is fusion-closed and* $\bowtie$ *is an equivalence relation with a finite index (as in definition 10)*
- $\Theta : \Pi \rightarrow Q$ *is a total function, assigning a node for all paths that respects the equivalence: if* $\langle i_1, \pi_1\rangle \bowtie \langle i_2, \pi_2\rangle$ *then* $\Theta(\langle i_1, \pi_1\rangle) \approx \Theta(\langle i_2, \pi_2\rangle)$

While the above definitions assign each path in a TFS a node rather than a type, in cases where all nodes in the signature module are typed, we depict TFSs using the standard convention where paths are assigned types.

Well-typedness is extended in the natural way: The first condition requires that F be an appropriate value for $\Theta(\pi)$; the second requires that $\Theta(\pi F)$ be an upper bound of all those nodes which are appropriate for $\Theta(\pi)$ and F (recall that in signature modules, the appropriateness relation may specify several appropriate nodes for the values of a feature F at a node q; only after all modules combine is a unique appropriate value determined).

Definition 65. *Let* $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ *be a signature module. A* TFS$A = \langle\Pi, \Theta, \bowtie\rangle$ *is* **well-typed** *if whenever* $\pi \in \Pi$ *and* $F \in$ FEAT *are such that* $\pi F \in \Pi$*, then all the following hold:*

- *there exists* $q \in Q$ *such that* $(\Theta(\pi), F, q) \in Ap$
- *for all* $q \in Q$ *such that* $(\Theta(\pi), F, q) \in Ap$*,* $q \stackrel{*}{\preceq} \Theta(\pi F)$

Enforcing all TFSs in the grammar to be well-typed is problematic for three reasons:

1. 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.

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

3. Well-typedness is not preserved under module combination. The natural way to preserve well-typedenss 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 1.2). After all modules are combined, the resulting grammar is extended to maintain well-typedness, see section C.3.

Definition 66. *Let* $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ *be a signature module and let* $A = \langle \Pi, \Theta, \bowtie\rangle$ *be a* TFS *over* S. A *is* **weakly well-typed** *if whenever* $\pi \in \Pi$ *and* $F \in$ FEAT *are such that* $\pi F \in \Pi$*, there exists* $q \in Q$ *such that* $(\Theta(\pi), F, q) \in Ap$ *and* $q \overset{*}{\preceq} \Theta(\pi F)$.

Definition 67. *Let* $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ *be a signature module and let* $\sigma = \langle Ind, \Pi, \Theta, \bowtie \rangle$ *be a* TMRS *over* S. σ *is* **weakly well-typed** *if whenever* $\langle i, \pi\rangle \in \Pi$ *and* $F \in$ FEAT *are such that* $\langle i, \pi F\rangle \in \Pi$*, there exists* $q \in Q$ *such that* $(\Theta(\langle i, \pi\rangle), F, q) \in Ap$ *and* $q \overset{*}{\preceq} \Theta(\langle i, \pi F\rangle)$.

Definition 68. *Let* $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ *be a signature module. A* **rule** *over* S *is a weakly well-typed* TMRS *of length greater than or equal to 1 with a designated (first) element, the* **head** *of the rule. The rest of the elements form the rule's* **body** *(which may be empty, in which case the rule is depicted as a* TFS*). A* **lexicon** *is a total function from* WORDS *to finite, possibly empty sets of weakly well-typed* TFSS. *A*

grammar $G = \langle \mathcal{R}, \mathcal{L}, \mathcal{A} \rangle$ *is a finite set of rules* $\mathcal{R}$*, a lexicon* $\mathcal{L}$ *and a finite set of weakly well-typed* TFSs*,* $\mathcal{A}$*, which is the set of* **start symbols**.

Finally, we define grammar modules:

Definition 69. *A grammar module is a structure* $\langle S, G \rangle$ *where* S *is a signature module and* G *is a grammar over* S*.*

C.2 Grammar Module Combination

We extend the combination operators defined for signature modules (section 2.3) to grammar modules. In both cases, the grammars are combined using a simple set union and the only adjustment is of the Θ function according to the combination of the signature module. We begin, however, by extending the *compactness* algorithm (which is used as a mechanism to coalesce corresponding nodes in the two modules) to grammar modules.

C.2.1 Compactness

Definition 70. *A grammar module* $\langle S, G \rangle$ *is* **non-redundant** *if* S *is non-redundant.*

In Appendix A we showed how a signature module can be compacted into a non-redundant signature module that encodes the same information. We extend this process to grammar modules. The only effect of signature module compaction on the grammar is that the Θ function must be adjusted to assign to each path, π, a new node which is the equivalence class of $\Theta(\pi)$.

Definition 71. *Let* $S = \langle \langle Q, T, \preceq, Ap \rangle, Int, Imp, Exp \rangle$ *be a signature module and let* $A = \langle \Pi, \Theta, \bowtie \rangle$ *be a* TFS *over* S*.* $coalesce(A) = \langle \Pi, \Theta', \bowtie \rangle$*, where for all* $\pi \in \Pi$*,* $\Theta'(\pi) = [\Theta(\pi)]_{\approx}$ *('$\approx$' is the indistinguishability equivalence relation over signature module nodes).*

Definition 72. *Let* $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$ *be a signature module and let* $\sigma = \langle Ind, \Pi, \Theta, \bowtie\ \rangle$ *be a* TMRS *over* S. $coalesce(\sigma) = \langle Ind, \Pi, \Theta', \bowtie\rangle$*, where for all* $\langle i, \pi\rangle \in \Pi$, $\Theta'(\langle i, \pi\rangle) = [\Theta(\langle i, \pi\rangle)]_{\approx}$.

Definition 73. *Let* $M = \langle S, G\rangle$ *be a grammar module where* $G = \langle \mathcal{R}, \mathcal{L}, \mathcal{A}\rangle$. *The* **coalesced module***, denoted* $coalesce(M)$*, is* $\langle S_1, G_1\rangle$ *where:*

- $S_1 = coalesce(P)$ *(see definition 58)*
- $\mathcal{R}_1 = \{coalesce(\sigma) \mid \sigma \in \mathcal{R}\}$
- *for all* $w \in$ WORDS, $\mathcal{L}(w) = \{coalesce(A) \mid A \in \mathcal{L}(w)\}$
- $\mathcal{A}_1 = \{coalesce(A) \mid A \in \mathcal{A}\}$

Notice that the coalesce algorithm has three versions, for signature modules, for TFSs and TMRSs and for grammar modules. The version is decided according to the input.

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

Algorithm 6. compact ($M = \langle S, G\rangle$**)** *where* $S = \langle\langle Q, T, \preceq, Ap\rangle, Int, Imp, Exp\rangle$

1. *Let* $S_1 = \langle Q_1, T_1, \preceq_1, Ap_1\rangle$ *be such that:*
   - $Q_1 = Q$
   - $T_1 = T$
   - $\preceq_1 = \{(q_1, q_2) \in \preceq \mid (q_1, q_2)$ *is a non-redundant subtyping arc in* $S\}$
   - $Ap_1 = \{(q_1, F, q_2) \in Ap \mid (q_1, F, q_2)$ *is a non-redundant appropriateness arc in* $S\}$
   - $Int_1 = Int$
   - $Imp_1 = Imp$
   - $Exp_1 = Exp$

2. $M' = coalesce(\langle S_1, G\rangle)$

3. *If* M' *is non-redundant, return* M'*, otherwise return* $compact(M')$*.*

C.2.2 Merge and Attachment

Definition 74. *Let* $M = \langle S, G\rangle$ *be a grammar module. The* **Ap-Closure** *of* M*, denoted* $ApCl(M)$*, is* $\langle ApCl(S), G\rangle$*.*

Definition 75. *Let* $M_1 = \langle S_1, G_1\rangle$*,* $M_2 = \langle S_2, G_2\rangle$ *be two grammar modules such that* S_1 *and* S_2 *are consistent and disjoint.* M_1, M_2 *are* **mergeable** *if* S_1, S_2 *are mergeable, in which case, their* **merge***, denoted* $M_1 \Cup M_2$ *is:*

$$M_1 \Cup M_2 = compact(ApCl(compact(\langle S, G\rangle)))$$

where:

- $S = S_1 \cup S_2$
- $G = \langle \mathcal{R}, \mathcal{L}, \mathcal{A}\rangle$ *where:*
  - $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$
  - *for all* $w \in$ WORDS*,* $\mathcal{L}(w) = \mathcal{L}_1(w) \cup \mathcal{L}_2(w)$
  - $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$

When two modules are merged, their components are first combined using simple set union. This results in $\langle S, G\rangle$. Then, using compactness and Ap-closure, anonymous nodes and nodes marked by the same type in the signature are coalesced. Compactness and Ap-closure adjust the Θ function accordingly. This process is similar to the merge operation defined on signatures (definition 18). The only difference is that here the grammar has to be slightly addapted.

The attachment operation is similar to signature attachment (definion 20), with the natural adjustment of the grammar components. The components of the grammar are

unioned by set union, and the Θ function is adjusted according to the changes in the signature.

Definition 76. *Let $M_1 = \langle S_1, G_1\rangle$, $M_2 = \langle S_2, G_2\rangle$ be two grammar modules such that S_1 and S_2 are consistent and disjoint. M_2 can be attached to M_1 if S_2 can be attached to S_1, in that case, the* **attachment** *of M_2 to M_1, denoted $M_1(M_2)$, is:*

$$M_1(M_2) = compact(ApCl(compact(\langle S, G\rangle)))$$

where:

- *S is defined as in definition 20*
- *$G = \langle \mathcal{R}, \mathcal{L}, \mathcal{A}\rangle$ where:*
  - *$\mathcal{R} = \{\langle Ind, \Pi, \Theta', \bowtie\rangle | \langle Ind, \Pi, \Theta, \bowtie\rangle \in \mathcal{R}_1 \cup \mathcal{R}_2$ and for all $\langle i, \pi\rangle \in \Pi$, $\Theta'(\langle i, \pi\rangle) = [\Theta(\langle i, \pi\rangle)]_\equiv\}$ ($\equiv$ is the equivalence relation of definition 20)*
  - *for all $w \in$ WORDS, $\mathcal{L}(w) = \{\langle \Pi, \Theta', \bowtie\rangle | \langle \Pi, \Theta, \bowtie\rangle \in \mathcal{L}_1(w) \cup \mathcal{L}_2(w)$ and for all $\pi \in \Pi$, $\Theta'(\pi) = [\Theta(\pi)]_\equiv\}$*
  - *$\mathcal{A} = \{\langle \Pi, \Theta', \bowtie\rangle | \langle \Pi, \Theta, \bowtie\rangle \in \mathcal{A}_1 \cup \mathcal{A}_2$ and for all $\pi \in \Pi$, $\Theta'(\pi) = [\Theta(\pi)]_\equiv\}$*

C.3 Extending Grammar Modules to Full Type Unification Grammars

We now show how to extend a grammar module into a bona fide typed unification grammar. Such extension must deal with two aspects: Extending the underlying signature module into a full type signature and enforcing well-typedness.

In section 2.4 we showed how to extend a signature module into a full type signature. We use the same process to extend the grammar module into a full type unification grammar, but some modifications are required to deal with the grammar as well. For that

purpose we use the four algorithms of section 2.4, in the same order as in the resolution algorithm (algorithm 1):

1. *Name resolution*: assigning types to anonymous nodes.

2. *Appropriateness consolidation*: determinizingAp, converting it from a relation to a function and enforcing upward closure and well-typedness.

3. *Feature introduction completion*: enforcing the feature introduction condition.

4. *BCPO completion*.

The name resolution procedure (section 2.4) assigns a type to every anonymous node whether by coalescing it with a typed node with a similar environment, if one uniquely exists, or by giving it an arbitrary type. The only effect this process has on a grammar module is that when two nodes are coalesced, the Θ function must be adjusted as well: if a node q is coalesced with a node q', then every path in the grammar that was assigned q needs to be assigned q'. Appropriateness consolidation both converts the Ap relation into a function and enforces upward closure and well-typedness. Here, a more careful modification is required as will be shown below. Finally, feature introduction completion and BCPO completion have no affect on the grammar. Notice that the classification of nodes is ignored during the extension, since resolution is executed after all the information from the different modules have been gathered, and hence this classification is no longer needed.

The input to the resolution algorithm is a grammar module and its output is a bona fide typed unification grammar.

Algorithm 7. Resolve($M = \langle S, G \rangle$)

1. $M := NameResolution(M)$

2. $S := BCPO\text{-}Completion(S)$

3. $S := ApCl(S)$

4. $M := ApConsolidate(M)$

5. $S := FeatureIntroductionCompletion(S)$

6. $S := BCPO\text{-}Completion(S)$

7. $S := ApCl(S)$

8. $M := ApConsolidate(M)$

9. *return* M

C.3.1 Appropriateness Consolidation

In section 2.4 we showed how to convert the Ap relation into a function and enforce upward closure. We extend the algorithm from signature modules to grammar modules. Here, the Θ function of the TFSs of the grammar must be adjusted as well: Consider a node q and its set of outgoing appropriateness arcs with the same label F, $Out = \{(q, F, q') \mid (q, F, q') \in Ap\}$. The appropriateness consolidation algorithm replaces all these arcs by a single arc (q, F, q_l), where q_l is the *lub* of the types of all q' (q_l may exist in the signature or may be added as a fresh new node by the algorithm). Then, for each path πF in any of the TFSs in the grammar where $\Theta(\pi) = q$, if $\Theta(\pi F)$ is a node q_1 such that $(\Theta(\pi), F, q_1)$ is a member of the set Out, then Θ is adjusted to assign πF the node q_l. However, if the node q_1 is not a member of Out, then it is a subtype of a node in Out and hence it is also a subtype of q_l (by the definition of the algorithm), and there is no need to adjust the value of $\Theta(\pi F)$.

The input to the following procedure is a grammar module whose signature module typing function, T, is total and whose signature module subtyping relation is a BCPO; its output is a grammar module whose signature module typing function, T, is total, whose signature module subtyping relation is a BCPO, and its signature module appropriateness

relation is a function that maintains upward closure and all TFSs and TMRSs of the grammar are well-typed. Let $M = \langle S, G \rangle$ where $S = \langle\langle Q, T, \preceq, Ap \rangle, Int, Imp, Exp \rangle$ and $G = \langle \mathcal{R}, \mathcal{L}, \mathcal{A} \rangle$ be a grammar 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 8. ApConsolidate ($M = \langle S, G \rangle$**),** *where* $S = \langle\langle Q, T, \preceq, Ap \rangle, Int, Imp, Exp \rangle$ *and* $G = \langle \mathcal{R}, \mathcal{L}, \mathcal{A} \rangle$*:*

1. *Set* $Int := Imp := Exp := \emptyset$
2. *Find a node* $q \in Q$ *and a feature* F *for which* $|target(q, F)| > 1$ *and for all* $q' \in Q$ *such that* $q' \overset{*}{\preceq} q$, $|target(q', F)| \leq 1$. *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* $\preceq$
   (d) *for all* $A \in \mathcal{A} \cup \bigcup_{w \in \text{WORDS}} \mathcal{L}(w)$ *where* $A = \langle \Pi, \Theta, \bowtie \rangle$*, if there exists* $\pi F \in \Pi$ *such that* $\Theta(\pi) = q$ *and* $\Theta(\pi F) \in target(q, F)$*, then set* $\Theta(\pi F) = p$
   (e) *for all* $\sigma \in \mathcal{R}$ *where* $\sigma = \langle Ind, \Pi, \Theta, \bowtie \rangle$*, if there exists* $\langle i, \pi F \rangle \in \Pi$ *such that* $\Theta(\langle i, \pi \rangle) = q$ *and* $\Theta(\langle i, \pi F \rangle) \in target(q, F)$*, then set* $\Theta(\langle i, \pi F \rangle) = p$
4. *Otherwise, if* $target(q, F)$ *has no lub, then:*
   (a) *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

(e) for all $A \in \mathcal{A} \cup \bigcup_{w \in \text{WORDS}} \mathcal{L}(w)$ where $A = \langle \Pi, \Theta, \bowtie \rangle$, if there exists $\pi F \in \Pi$ such that $\Theta(\pi) = q$ and $\Theta(\pi F) \in target(q, F)$, then set $\Theta(\pi F) = p$

(f) for all $\sigma \in \mathcal{R}$ where $\sigma = \langle Ind, \Pi, \Theta, \bowtie \rangle$, if there exists $\langle i, \pi F \rangle \in \Pi$ such that $\Theta(\langle i, \pi \rangle) = q$ and $\Theta(\langle i, \pi F \rangle) \in target(q, F)$, then set $\Theta(\langle i, \pi F \rangle) = p$

5. $M := ApCl(M)$

6. $M := compact(M)$

7. *go to (2).*

Example 25. *Let $M_1 = \langle S_1, G_1 \rangle$ be a grammar module whose signature module is depicted in Figure C.1 and $G_1 = \langle \mathcal{A}_1, \mathcal{R}_1, \mathcal{L}_1 \rangle$, where $\mathcal{A}_1 = \emptyset$, $\mathcal{R}_1 = \left\{ \begin{bmatrix} a \\ F : \begin{bmatrix} c \end{bmatrix} \end{bmatrix} \rightarrow \begin{bmatrix} a \\ F : \begin{bmatrix} d \end{bmatrix} \end{bmatrix} \right\}$ and for all $w \in$ WORDS, $\mathcal{L}_1(w) = \emptyset$. In the first iteration of the algorithm, in stage 1, q_1, the node typed a, is selected since $target(q_1, F) = \{q_2, q_3\}$. Since these nodes have no lub, a fresh new node, q_5, is added to the signature (stages 3a–3d), resulting in S_2. Stage 3e has no effect but stage 3f effects only the left side* TFSs*of the single rule of the grammar. In the left side* TFS*, the node typed c, q_3, is replaced by the node typed new, q_5, since $q_3 \in target(q_1, F)$. However the node typed d, q_4, is not a member of $target(q_1, F)$ but a subtype of q_2 which is a member of $target(q_1, F)$. Therefore it is not changed in order to maintain the desired interpretation of this* TFS*. The first iteration of the algorithm results in $M_2 = \langle S_2, G_2 \rangle$ whose signature module is*

S_2 and $G_2 = \langle \mathcal{A}_2, \mathcal{R}_2, \mathcal{L}_2 \rangle$ where $\mathcal{A}_2 = \emptyset$, $\mathcal{R}_2 = \left\{ \begin{bmatrix} a \\ F: \begin{bmatrix} new \end{bmatrix} \end{bmatrix} \rightarrow \begin{bmatrix} a \\ F: \begin{bmatrix} d \end{bmatrix} \end{bmatrix} \right\}$ and and for all $w \in$ WORDS $\mathcal{L}_2(w) = \emptyset$. This is also the final result of the algorithm. Notice that the appropriateness relation of S_2 is a function that maintains upward closure and all TFSS *and* TMRSS *in G_2 are well-typed.*

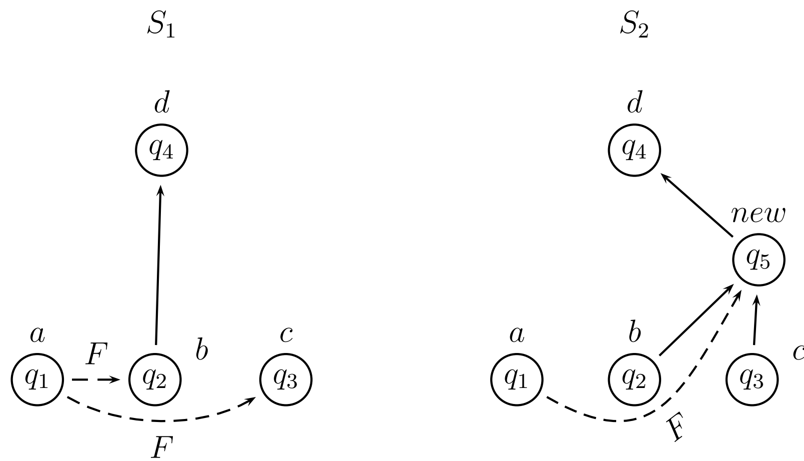


Figure C.1: Grammar appropriateness consolidation

Appendix D

MODALE Description of the Basic HPSG Grammar of Chapter 3

D.1 Modular Design

```
% General guidelines:
% 1. The character '%' is used to insert comment.
%    Everything from '%' till '\n' is ignored.
% 2. Extra spaces and tabs are ignored but can be used for clarity
% 3. The general description of a module is:
%    module(moduleName)
%    {
%       description of subsumpion and appropriateness relations
%    }
%    {
%       node classification
%    }
% 4. nodes are specifyied by their types unless they are anonymous
%    in which case they are reffered to as anon(nodeName).
```

```
% 5. As in ALE\TRALE, types and features may consist of the letters
%    a-z, digits and the character '_' but must start with a letter

module(List)
{
   anon(q4) sub [elist,anon(q2)].
      anon(q2) approp [first:{anon(q3)},
                       rest:{anon(q4)}].
      elist sub [].
}
{
   int=<>.
   imp=<anon(q3)>.
   exp=<anon(q4)>.
}

module(Object)
{
   object sub [sign,mod_synsem,head,category,con_struc,
               local,non_local].
}
{
   int=<>.
   imp=<>.
   exp=<>.
}

module(Sign)
{
```

```
   sign sub [word,phrase].
   sign approp [retrieved:{quantifier_list},
                phon:{phonestring_list},
                synsem:{synsem}].
      phrase approp [dtrs:{con_struc}].
}
{
   int=<>.
   imp=<phonestring_list,quantifier_list>.
   exp=<phrase>.
}

module(ConStruc)
{
   con_struc sub [coord_struc,head_struc].
      head_struc sub [head_comp_struc,head_mark_struc,
                      head_adj_struc,head_filter_struc].
      head_struc approp [comp_dtrs:{phrase_list},
                         head_dtr:{sign}].
         head_mark_struc approp [marker_dtr:{word},
                                 head_dtr:{phrase},
                                 comp_dtrs:{elist}].
         head_adj_struc approp [adjunct_dtr:{phrase},
                                head_dtr:{phrase},
                                comp_dtrs:{elist}].
         head_filter_struc approp [filter_dtr:{phrase},
                                   head_dtr:{phrase},
                                   comp_dtrs:{elist}].
}
```

```
{
   int=<>.
   imp=<phrase_list>.
   exp=<>.
}

module(Head)
{
   head sub [substantive,functional].
      substantive sub [noun,prep,verb,reltvzr,adj].
      substantive approp [prd:{boolean},
                          mod:{mod_synsem}].
         noun approp [case:{case}].
         prep approp [pform:{pform}].
         verb approp [aux:{boolean},
                      iav:{boolean},
                      vform:{vform}].
      functional sub [marker,det].
      functional approp [spec:{synsem}].
}
{
   int=<>.
   imp=<>.
   exp=<>.
}

module(Cat)
{
   category approp [subcat:{synsem_list},
```

```
                    head:{head},
                    marking:{marking}].
}
{
   int=<>.
   imp=<synsem_list>.
   exp=<>.
}

module(Synsem)
{
   mod_synsem sub [none,synsem].
      synsem approp [local:{local},
                     nonlocal:{non_local}].
}
{
   int=<>.
   imp=<>.
   exp=<synsem>.
}

module(NomObj)
{
   nom_obj sub [npro,pron].
   nom_obj approp [index:{index}].
      pron sub [ppro,ana].
         ana sub [refl,recp].
}
{
```

```
   int=<>.
   imp=<>.
   exp=<>.
}

module(Phonestring)
{
   phonestring sub [].
}
{
   int=<>.
   imp=<>.
   exp=<phonestring>.
}

module(Quantifier)
{
   quantifier sub [].
}
{
   int=<>.
   imp=<>.
   exp=<quantifier>.
}
```

D.2 The Resolved HPSG Signature

```
module(hpsg_res)
{
   bot sub [quantifier_list, phonestring_list, phonestring,
            quantifier, phrase_list, synsem_list, marking,
            object, boolean, case, pform, vform, nom_obj,
            index, new_node_5].
      quantifier_list sub [elist, new_node_2].
         elist sub [].
         new_node_2 sub [].
         new_node_2 approp [rest:{quantifier_list},
                            first:{quantifier}].
      phonestring_list sub [elist, new_node_1].
         new_node_1 sub [].
         new_node_1 approp [rest:{phonestring_list},
                            first:{phonestring}].
      phonestring sub [].
      quantifier sub [].
      phrase_list sub [elist, new_node_3].
         new_node_3 sub [].
         new_node_3 approp [first:{phrase},
                            rest:{phrase_list}].
      synsem_list sub [elist, new_node_4].
         new_node_4 sub [].
         new_node_4 approp [rest:{synsem_list},
                            first:{synsem}].
      marking sub [].
      object sub [sign, con_struc, category, head, mod_synsem,
```

```
         local, non_local].
sign sub [word, phrase].
sign approp [synsem:{synsem},
             phon:{phonestring_list},
             retrieved:{quantifier_list}].
   word sub [].
   word approp [retrieved:{quantifier_list},
                phon:{phonestring_list},
                synsem:{synsem}].
   phrase sub [].
   phrase approp [retrieved:{quantifier_list},
                  phon:{phonestring_list},
                  synsem:{synsem},
                  dtrs:{con_struc}].
con_struc sub [coord_struc, head_struc].
   coord_struc sub [].
   head_struc sub [head_comp_struc, head_mark_struc,
                   head_adj_struc, head_filter_struc].
   head_struc approp [comp_dtrs:{phrase_list},
                      head_dtr:{sign}].
      head_comp_struc sub [].
      head_comp_struc approp [head_dtr:{sign},
                              comp_dtrs:{phrase_list}].
      head_mark_struc sub [].
      head_mark_struc approp [marker_dtr:{word},
                              head_dtr:{phrase},
                              comp_dtrs:{elist}].
      head_adj_struc sub [].
      head_adj_struc approp [adjunct_dtr:{phrase},
```

```
                              head_dtr:{phrase},
                              comp_dtrs:{elist}].
      head_filter_struc sub [].
      head_filter_struc approp [filter_dtr:{phrase},
                                 head_dtr:{phrase},
                                 comp_dtrs:{elist}].
category sub [].
category approp [marking:{marking},
                 head:{head},
                 subcat:{synsem_list}].
head sub [substantive, functional].
   substantive sub [noun, prep, verb, reltvzr, adj].
   substantive approp [mod:{mod_synsem},
                       prd:{boolean}].
      noun sub [].
      noun approp [prd:{boolean},
                   mod:{mod_synsem},
                   case:{case}].
      prep sub [].
      prep approp [prd:{boolean},
                   mod:{mod_synsem},
                   pform:{pform}].
      verb sub [].
      verb approp [prd:{boolean},
                   mod:{mod_synsem},
                   vform:{vform},
                   iav:{boolean},
                   aux:{boolean}].
      reltvzr sub [].
```

```
         reltvzr approp [prd:{boolean},
                         mod:{mod_synsem}].
         adj sub [].
         adj approp [prd:{boolean},
                     mod:{mod_synsem}].
      functional sub [marker, det].
      functional approp [spec:{synsem}].
         marker sub [].
         marker approp [spec:{synsem}].
         det sub [].
         det approp [spec:{synsem}].
   mod_synsem sub [synsem, none].
      synsem sub [].
      synsem approp [nonlocal:{non_local},
                     local:{local}].
      none sub [].
   local sub [].
   non_local sub [].
boolean sub [].
case sub [].
pform sub [].
vform sub [].
nom_obj sub [npro, pron].
nom_obj approp [index:{index}].
   npro sub [].
   npro approp [index:{index}].
   pron sub [ppro, ana].
   pron approp [index:{index}].
      ppro sub [].
```

```
            ppro approp [index:{index}].
            ana sub [refl, recp].
            ana approp [index:{index}].
               refl sub [].
               refl approp [index:{index}].
               recp sub [].
               recp approp [index:{index}].
      index sub [].
      new_node_5 sub [new_node_6].
      new_node_5 approp [first:{bot}].
         new_node_6 sub [new_node_1, new_node_2,
                         new_node_3, new_node_4].
         new_node_6 approp [first:{bot},
                            rest:{bot}].
}
{
  int=<>.
  imp=<>.
  exp=<>.
}
```