Higher-Order Types for Grammar Engineering

by

Kenneth Richard Hoetmer

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Higher-Order Types for Grammar Engineering

Kenneth Richard Hoetmer

Master of Science

Graduate Department of Computer Science

University of Toronto

2005

Linguistically precise general-purpose grammars of natural language enable a detailed semantic analysis that is currently unavailable to corpus-based approaches. Unfortunately, the engineering of such grammars is often tedious, time-consuming, error-prone, and inaccessible to new developers. This work seeks to alleviate the engineering problem by discovering, documenting, and exploiting structural patterns of current grammar signatures. More specifically, it mines the English Resource Grammar (ERG) for evidence of intended patterns of type usage and documents those patterns within the framework of Alexandrian design patterns. The structural patterns are then exploited by way of parametric types, special higher-order type constructors, and methods for automatic type selection. The applicability of the patterns is illustrated by ICEBERG, a higher-order refactoring of the ERG.

For those who reside at SW-17-39-5-W5

Acknowledgements

First acknowledgments, of course, are due my thesis advisor, Gerald Penn. All the the work in this thesis either builds on Gerald's previous research or is indebted to his suggestions and guidance. Gerald's vast knowledge of mathematics, linguistics, and computer science have helped me appreciate the perspectives of all three fields, and his insistence on precision and meticulous attention to mathematical detail have made this work more thorough than I could have achieved otherwise.

Acknowledgments must also go out to those who, in different ways, have helped this rural Albertan settle in Toronto. Thanks go to Karen, who sees abilities in me when I can't see them myself, and to Brad, who continues to keep me focused on what's most important. Thanks also to Tom, Fred, Vlad, and the Friday afternoon hockey crew, to Bowen, Mike, and the DGP volleyball team, to Geoff, Brian, Mike, Joanna, Lauren, and the rest of the pilgrims of the Graduate Christian Fellowship, and finally, to Jane, Faye, Bob, Cosmin, Chris, and the other members of the computational linguistics group who truly enjoy studying language and who strive to create an inclusive and welcoming student environment.

Suzanne Stevenson, my second reader, provided insightful comments under strict time requirements, and her comments were invaluable in helping the document focus on its key contributions.

Lastly, and most importantly, thanks go to my parents, Garry and Marilyn, and my siblings, Kristina, Dennis, Ryan, Robynn, Reuben, Karalee, and Michael, who have always encouraged me and whose hospitality, humour, and honesty continue to fill me with pride.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The engineering of detailed grammars of natural language has been part of language research from the very beginning, and over the years many attempts have been made to construct linguistically-precise broad-coverage grammars in many different grammatical frameworks. Unfortunately, although these attempts often begin with an elegant design and consistent structure, inevitably, as Erbach and Uszkoreit [EU90] report, the intricacies and inherent rule exceptions of natural languages threaten the original elegance. As grammar writers attempt to extend coverage beyond standard examples, both the grammar formalism and original design decisions are stretched in order to accommodate the exceptional cases that do not quite fit the existing grammar. Often these stretches result in grammar patching, producing decreased modularity, loss of grammar transparency, and increased time spent fixing unpredictable errors. The grammar source becomes increasingly inaccessible to new developers and the undocumented knowledge of a few principal developers becomes as important as the grammar source code itself. The final result is reminiscent of what software engineers have labeled "spaghetti code", or, more affectionately "a big ball of mud" [FY97].

Over the years, much work has been done to alleviate the problems of grammar engineering, focusing attention on a number of specific problems. There have been research

contributions regarding:

- grammar development environments (LKB [CF00], PAGE [KS94a], ALE [CP96])
- performance profiling and coverage evaluation ([OF98, ONK97]),
- multi-lingual grammar resources ([BFO02, FB03]),
- grammar learning ([Abn97, KM01]), and
- distributed development strategies. ([OBC$^+$02]).

This work has been extremely successful. Though the focus of much computational linguistics research has shifted toward more simple language models such as probabilistic context-free grammars (PCFGs) estimated via corpus-based statistical approaches, there do exist manually engineered grammars of natural language which, though unable to compete with the coverage of a statistically estimated PCFG, are capable of much deeper semantic analysis while maintaining a high-level of coverage. Within the logic of typed feature structures and the grammatical framework of Head-Driven Phrase Structure Grammar (HPSG) [PS94], there exist broad-coverage grammars of English (83% coverage of 10,000 transcribed utterances) [FCS00], German [MK00], and Japanese [Sie00], with projects begun for Norwegian [HH03] and Greek [KN03]. The ParGram project [BKNS99] is using Lexical Functional Grammar (LFG) [KB82] to produce broad-coverage grammars of English, French, German, Japanese, Malagasy, Norwegian, Urdu, and Welsh, and there is a broad-coverage grammar of English being developed in lexicalized Tree Adjoining Grammar (XTAG) [DEH$^+$94].

The existence of these broad-coverage grammars is sufficient proof that manually-constructed broad-coverage grammars are possible, but no one involved would argue that the engineering problem is solved. Constructing detailed grammars is still tedious, still time-consuming, still error-prone, and still accessible to only a few principal developers. The English Resource Grammar (ERG), the largest and broadest-coverage HPSG grammar of English, had, as of the year 2000, consumed 11 person-years of effort, and although many people have made peripheral contributions to the grammar, the difficulty of understanding the grammar well enough to make significant changes has meant that the core grammar is the result of four developers [CF00].

This work addresses the problem of inaccessibility and unpredictability by way of higher-order types motivated by the design pattern view of software engineering. Design patterns are intended to draw on the collective experience of existing implementations to provide a catalogue of design problems and design solutions that can be used as the basis for new development and for refactoring existing implementations. More specifically, we mine the ERG for evidence of intended patterns in the type signature topology, and propose a series of evidenced structural patterns of type usage. These structural patterns are then exploited by way of parametric types, special higher-order type constructors, and methods for automatic type selection.

The goal is two-fold. First, patterns and higher-order typing can enable higher-order re-factoring of HPSG signatures so as to explicitly state patterns that were already implicitly present in the signature. This goal is illustrated by ICEBERG, our higher-order re-implementation of the ERG type hierarchy. Secondly, and more importantly, the patterns and higher-order typing techniques developed in this work can guide the future development of more intuitive and usable typed feature structure grammars.

1.1 Statement of thesis and objectives

1.1.1 Thesis

Current typed feature structure grammars exhibit evidence of underlying structural patterns that, if mined and documented, can facilitate the enumeration of higher-order constructors which can in turn lead to more intuitive, readable, and extendable type signatures.

1.1.2 Objectives

This document supports this thesis by making the following specific contributions:

1. **Design patterns for grammar engineering:** it adapts the notion of design pattern extraction, documentation, and utilization from software engineering to the problem of grammar development, providing a framework within which to discover and document the structural and linguistic patterns of typed feature structure grammars.

2. **Extension of parametric types for attribute-value logic:** it presents a formal extension of parametric types for attribute-value logic [Pen00] by introducing parametric restrictions and constrained parametric induction. Existing attribute-value grammars provide evidence that restrictions and constrained induction are necessary, but parametric types previously had no formal methods for restricting parameter values and no formal methods for automated type selection.

3. **HOPS - a higher-order parametric signature description language:** it introduces, describes, and implements HOPS, a higher-order parametric signature description language. A HOPS interpreter has been implemented as a front-end extension to ALE.

4. **ICEBERG - a higher-order ERG:** it presents ICEBERG, a re-factoring of the ERG type hierarchy using pattern-influenced higher-order type constructors. ICEBERG is the first large-scale grammar implementation to consistently employ higher-order typing.

1.2 Structure of the document

The remainder of this document is structured as follows:

Chapter 2: Typed feature structures introduces the logic of typed feature structures (LTFS) [Car92], the feature logic assumed by most typed feature structure grammar implementations. Type hierarchies, feature structures, appropriateness specifications, feature structure subsumption and unification, and well-typing are formally defined.

Chapter 3: Types in contemporary typed feature structure grammars examines the effects of using hand-crafted subtyping-based type hierarchies in typed feature structure grammars. By way of reference to the ERG, it sketches advantages to using types in grammar and discusses the drawbacks of implementing large-scale grammars with hand-crafted type hierarchies.

Chapter 4: Design patterns and grammar engineering introduces design patterns for grammar engineering and describes a number of structural design patterns currently evidenced by the ERG. A review of the history of design patterns is included, their application to software engineering is discussed, and sequence of grammar-specific patterns is documented.

Chapter 5: Exploiting design patterns via parametric typing introduces parametric polymorphism as an option for exploiting design patterns in typed feature structure grammars. Penn's [Pen00] treatment of parametric types is extended to include parametric restrictions, and a series of closure operations are introduced for the purpose of constraining the induction of parametric type hierarchies.

Chapter 6: HOPS - higher-order parametric signature descriptions introduces HOPS, a higher-order parametric signature specification language designed to facilitate intuitive, readable, and extendable grammars by implementing the patterns of §4 as higher-order type constructors and providing operators for performing the closure operations introduced in §5.

Chapter 7: ICEBERG - a higher-order English Resource Grammar presents ICEBERG, a higher-order re-implementation of ERG written in HOPS. Some implementation issues are discussed, and some portions of ICEBERG are described in detail. Finally, ICEBERG is compared to ERG both qualitatively and operationally, concluding that ICEBERG is operationally equivalent to ERG but more terse, readable, intuitive, and extendable.

Chapter 8: Conclusions is an analysis of the main contributions, some suggestions for further applications of design patterns and/or higher-order typing, and some suggestions for further testing the usability of higher-order type signatures.

Chapter 2

Typed feature structures

This chapter formally introduces the logic of typed feature structures (LTFS) developed by Carpenter [Car92], and assumed by standard HPSG. The chapter draws heavily from [Car92], with reference to Penn's [Pen00] work on algebraic properties of attributed type signatures and the work on order theory by Davey and Priestly [DP02]. Readers familiar with feature logics and LTFS may skip directly to chapter 3. For those who choose to stay, we note that this chapter will provide only the overview of typed feature structures necessary to comprehend the work of this thesis; we will not delve into the history of typed feature structures, other feature structure logics such as Rounds-Kasper logic [KR90] and SRL[Kin89], or other aspects of LTFS such as satisfiability and signature equivalence. For a more detailed discussion, the interested reader is referred to Carpenter's seminal work on typed feature structure logics [Car92], Penn's dissertation work on the algebraic properties of attributed type signatures [Pen00], Keller's survey of feature logics [Kel93], and Richter's [Ric04] dissertation work on formalizing the HPSG grammar presented in Pollard and Sag's [PS94] introduction to HPSG.

For the purposes of our discussion, we will simply recognize that LTFS is constructed on top of *attributed type signatures* which are themselves composed of three distinct pieces:

- a hierarchy of types,
- a set of features, and
- an appropriateness specification.

Such an attributed type signature provides the basic vocabulary for talking about feature structures and describing relationships between them. We will proceed incrementally, starting with type hierarchies, augmenting type hierarchies with features to form feature structures, and adding appropriateness to feature structures to obtain totally well-typed feature structures.

2.1 Type hierarchies

Typed feature structure grammars such as HPSGs assume the availability of a finite set of types ordered according to their specificity. In such orders, also known as *IS-A networks*, more specific types are seen to inherit information from more general types via a relation known as *subsumption*. This is a departure from early unification-based grammars such as Unification Grammar [Kay85], and earlier typed feature logics developed by Pollard and Moshier [PM90] and Smolka [Smo89], which, though typed, present types as incomparable sorts. The motivation for ordering types draws from [AK84] which introduced subtyping as a means of better capturing generalizations. Carpenter [Car92] and Pollard and Sag [PS94] both employ type hierarchies in order to facilitate their interpretation that types and feature structures do not represent objects themselves, but represent partial information states about objects.

2.1.1 IS-A networks

Borrowing from a tradition in knowledge representation starting with Quillian's semantic networks [Qui68] and continued in Brachman's KL-ONE system [Bra77], the specificity relationship is typically expressed by IS-A links between conceptual types. In the world of linguistics, for example, *noun* IS-A *substantive*, and *accusative* IS-A *case*. The subsumption relation is then taken as the transitive and reflexive closure over the set of IS-A links, forming what is mathematically known as a *partially ordered set*, or, equivalently, a *partial order*.

Definition 2.1.1. A *partial order* on a set, P, is a relation, $\leq \subseteq P \times P$, such that, for all $x, y, z \in P$:

- (reflexivity) $x \leq x$,
- (anti-symmetry) if $x \leq y$ and $y \leq x$, then $x = y$, and
- (transitivity) $x \leq y$ and $y \leq z$, then $x \leq z$.

Throughout mathematics, partially ordered sets are represented topologically as *Hasse* diagrams. In a Hasse diagram, elements in the order appear topologically lower than the elements they subsume. The diagram is a directed graph in which elements of the order are represented as vertices and an edge exists from element x to element y iff there is an IS-A link from x to y in the subsumption cover relation. Figure 2.1 is a Hasse diagram of a partial order representing grammatical person, number, and gender information. In this diagram *1-sing* IS-A *1st*, *3-s-fem* IS-A *3-sing*, and, because of transitivity, *3-s-neut* IS-A *sing*. Note that this notation is opposite of the notation typically used in HPSG type hierarchies where more general types are drawn topologically higher than the elements they subsume.

Carpenter specifically limits his type hierarchies to those in which the partial information can be deterministically combined through unification. That is, the combination of two consistent partial information states (types) should result in a partial information state that can be deterministically given a type. This unification is formalized in terms of upper and lower bounds.

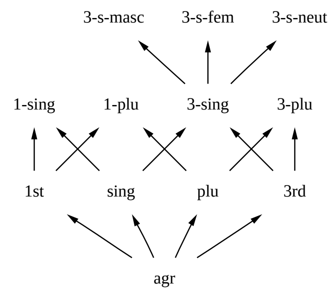

Figure 2.1: A partially ordered set (excerpted from [Car92])

Definition 2.1.2. Given a subset $S \subseteq P$, the set $S^u = \{x \in P \mid \exists y \in S, y \leq x\}$ is the set of *upper bounds* of S. The set of *lower bounds* S^l is defined dually.

Given sets of upper bounds (more specific types) and lower bounds (less specific types), we can reason about greatest lower bounds (most specific of the less specific types) and least upper bounds (least specific of the more specific types).

Definition 2.1.3. Given a subset $S \subseteq P$, if S^u has a least element x, then x is called the *least upper bound* or *join* of S. We denote the join of S as $\sqcup S$. If S^l has a greatest element x, then x is called the *greatest lower bound* or *meet* of S, denoted $\sqcap S$.

More colloquially, the join of a set of elements S is the least upper bound of S, if it exists and is unique. The meet of a set of elements S is the greatest lower bound of S, if it exists and is unique. If S has only two elements x and y, we may denote $\sqcup S$ by $x \sqcup y$ and $\sqcap S$ by $x \sqcap y$. If joins and meets exist, they can be represented by $(S^u)^{min}$ and $(S^l)^{max}$ respectively.

For example, in figure 2.1, the join of *1st* and *sing* is *1-sing*, the join of *agr* and *1-plu* is *1-plu*, and the meet of *3-s-masc* and *3-s-fem* is *3-sing*. *1-plu* and *3-plu* have no common upper bounds and are therefore declared *semantically inconsistent*. In a properly modelled world, all semantically consistent types will have a join and all semantically inconsistent types will have no join (or join to a designated top element, $\top$), and, in fact, this is the condition Carpenter places on partial orders in order for the partial order to be considered a type hierarchy. The construction that captures this intuition is a *bounded complete partial order* or *BCPO*.

Definition 2.1.4. A *bounded complete* partial order (BCPO) is a partial order $\langle P, \leq \rangle$, such that for every subset $S \subseteq P$, with $S^u \neq \emptyset$, $\sqcup S \downarrow$.

Definition 2.1.5. A *type hierarchy* is a non-empty, finite, bounded complete partial order.

Figure 2.1 is a type hierarchy since it is finite and since all subsets of elements which have a set of common upper bounds have a join. If we postulated an IS-A link between *3rd* and *1-sing* (bear with me!), figure 2.1 would cease be a type hierarchy (but remain a partial order) since *3rd* and *sing* would then have two least upper bounds (*1-sing* and *3-sing*) and therefore no join.

Penn [Pen00] provides conditions based on path lengths and branching factors under which we can relax our assumption that the underlying BCPO is finite. [Car92] gives no such conditions, however, simply assuming type hierarchies are finite. Because typical typed feature structure grammars assume finiteness, for the remainder of this thesis we will follow Carpenter and simply assume type hierarchies are finite.

2.1.2 Meet-semilattice completions

It is overly restrictive to suggest that every domain's concepts be modelled by a bounded complete partial order and it is also difficult for a grammar writer (or any ontology developer) to ensure that their large partial order is, in fact, bounded complete. Fortunately, there is a way to expand any given partial order to the smallest BCPO that contains it. The expansion rests on the equivalence between finite BCPOs and finite meet semi-

lattices and the fact that there exists a well-defined completion algorithm to restore any finite partial order to a meet semi-lattice.

Definition 2.1.6. A partial order, $\langle P, \sqsubseteq \rangle$, is a *meet semi-lattice* iff for any $x, y \in P$, $x \sqcap y \downarrow$.

A meet semi-lattice is a partial order such that every pair of elements has a meet, or greatest lower bound. In the finite case, this is exactly the condition necessary to ensure bounded completeness.

Proposition 2.1.1. A finite partial order is bounded complete iff it is a meet semi-lattice.

Meet semi-latticehood can be obtained by computing the Dedekind-MacNeille completion of any partial order, which embeds the partially ordered set into the least bounded-complete subset of its set-inclusion-ordered powerset that includes it [DP02].

Definition 2.1.7. Given a partially ordered set, P, the Dedekind-MacNielle completion of P, $\langle DM(P), \subseteq \rangle$, is given by:

$$DM(P) = \{A \subseteq P \mid A^{ul} = A\}$$

Proposition 2.1.2. If P is a partial order, $DM(P)$ is a meet semi-lattice.

Proposition 2.1.3. If P is a finite partial order, $DM(P)$ is a BCPO.

Effectively, elements in the Dedekind-MacNeille completion are the subsets S of the original partial order such that the set of least upper bounds of S is S itself. The Dedekind-MacNeille completion can be obtained by incrementally adding completion types as follows:

1. Find two elements, t_1, t_2 with minimal upper bounds, $u_1, \ldots, u_k$, such that their join $t_1 \sqcup t_2$ is undefined, i.e., $k > 1$. If no such pair exists, then stop.

2. Add an element, v, such that:

- for all $1 \leq i \leq k$, $v \sqsubseteq u_i$, and
- for all elements t, $t \sqsubseteq v$ iff for all $1 \leq i \leq k$, $t \sqsubseteq u_i$.

3. Go to (1)

This algorithm can be attributed to Penn [Pen00] with influence from Bertet et. al. [BMN97], who were the first to notice that the incremental "where there is no meet, add one" completion algorithms used by Fall [Fal96] and Aït-Kaci [AKBLN89] did not always compute the minimal meet-semilattice.

2.2 Typed feature structures

In LTFS, every type is augmented with a (possibly empty) set of features organized in a record-like structure. These features are essentially named value fields, and the resulting structures are called *feature structures*. In LTFS, the feature values are also feature structures and feature values within the same feature structure can be either identified or inequated. Because of this, feature structures can be defined by directed acyclic graphs whose nodes are types and whose edges are the features of that type.

Definition 2.2.1. A *typed feature structure* is a tuple, $F = \langle Q, \bar{q}, \Theta, \delta, \not\leftrightarrow \rangle$ where:

- Q is a finite set of *nodes*,
- $\bar{q} \in Q$ is the *root node*,
- $\Theta : Q \to T$ is a total *node typing* function,
- $\delta :$ *Feat* $\times Q \to Q$ is a partial *feature value* function, and
- $\not\leftrightarrow \subseteq Q \times Q$ is an anti-reflexive and symmetric *inequation* relation.

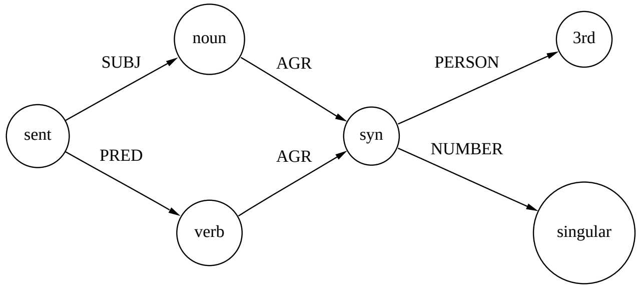

Figure 2.2: Feature structure: graph notation (excerpted from [Car92])

such that for every $q \in Q$, there is a finite sequence of features $\mathtt{F}_1, \ldots, \mathtt{F}_n \in$ *Feat* such that $q = \delta(\mathtt{F}_n, \delta(\mathtt{F}_{n-1}, \ldots, \delta(\mathtt{F}_2, \delta(\mathtt{F}_1, \bar{q}))))$, i.e., a finite sequence that connects $\bar{q}$ to q with δ.

$\mathcal{F}$ denotes the set of all feature structures relative to the (implicit) set of types, T, and features, *Feat*.

Figure 2.2 is a directed acyclic graph of a feature structure (excerpted from [Car92]) representing valence considerations for the verb *sent*. *sent* is encoded as the type of the root node and the root feature structure has two features, `SUBJ` and `PRED`. `SUBJ` and `PRED` lead to nodes labeled with types *noun* and *verb*, respectively. These nodes each have an `AGR` feature which leads to the same node (labeled with *syn*) indicating that the values of the agreement features of the verb's subject and predicate are *structure shared*.

Feature structure graphs can become frustrating and difficult to understand, hence the linguistics community has become more accustomed to representing feature structures as Attribute-Value Matrices or AVMs (figure 2.3). In AVM notation node labels are placed at the top left corner of a bracketed structure whose contents represent edges leaving that node. Structure sharing is indicated by co-indexed boxed numbers.

$$\begin{bmatrix} \text{sent} & \\ \text{SUBJ} & \begin{bmatrix} \text{noun} & \\ \text{AGR} & \begin{bmatrix} \boxed{4}\ \text{syn} & \\ \text{PERSON} & \begin{bmatrix}\text{3rd}\end{bmatrix} \\ \text{NUMBER} & \begin{bmatrix}\text{singular}\end{bmatrix} \end{bmatrix} \end{bmatrix} \\ \text{PRED} & \begin{bmatrix} \text{verb} & \\ \text{AGR} & \begin{bmatrix}\boxed{4}\end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Figure 2.3: Feature structure: AVM notation (excerpted from [Car92])

2.2.1 Feature structure subsumption

Because feature structures are constructed relative to an inheritance hierarchy of types, feature structures themselves can be arranged in a subsumption relationship. This relationship is one of the key properties of typed feature structures since it extends the idea of partial information from feature-less objects (types) to complex information states (feature structures).

Structure sharing can cause subsumption to become quite complex and actually prevents feature structure subsumption from forming a partial order since it is possible for $F_1 \sqsubseteq F_2$ and $F_2 \sqsubseteq F_1$ where $F_1 \neq F_2$ (see [Car92] and [Pen00] for more details). But, for the most part, feature structure subsumption is as straightforward as the example in figure 2.4 (extracted from [Car92]): a feature structure F subsumes another feature structure G if G contains all the information that F contains.

The formal definition of feature structure subsumption makes use of *feature paths* and *feature path values* to provide a way of reference to every feature structure node. Subsumption is then defined over these paths and path values.

$$\begin{bmatrix} \text{sign} & \\ \text{AGR} & \begin{bmatrix} \text{agr} & \\ \text{PERS} & \begin{bmatrix}\text{1st}\end{bmatrix} \end{bmatrix} \end{bmatrix} \sqsubseteq \begin{bmatrix} \text{phrase} & \\ \text{AGR} & \begin{bmatrix} \text{agr} & \\ \text{PERS} & \begin{bmatrix}\text{1st}\end{bmatrix} \\ \text{NUM} & \begin{bmatrix}\text{sing}\end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Figure 2.4: A subsumption example (assuming $\mathit{sign} \sqsubseteq \mathit{phrase}$)

Definition 2.2.2. A *feature path* is a finite sequence of features $\pi \in \mathit{Feat}^*$.

Definition 2.2.3. Given a typed feature structure, $F = \langle Q, \bar{q}, \Theta, \delta, \nleftrightarrow \rangle$, its partial *path value* function is the function, $\delta' : \mathit{Feat}^* \times Q \to Q$ induced from F such that:

- $\delta'(\epsilon, q) = q$, and
- $\delta'(\mathtt{F}\pi, q) = \delta'(\pi, \delta(\mathtt{F}, q))$.

Definition 2.2.4. Given a common signature, a typed feature structure, $F = \langle Q, \bar{q}, \Theta, \delta, \nleftrightarrow \rangle$ *subsumes* another typed feature structure, $F' = \langle Q', \bar{q}', \Theta', \delta', \nleftrightarrow' \rangle$, written $F \sqsubseteq F'$, iff there is a total function, $h : Q \to Q'$, called a *morphism*, such that:

- $h(\bar{q}) = \bar{q}'$,
- for every $q \in Q$, $\Theta(q) \sqsubseteq \Theta'(h(q))$,
- if $\delta(\mathtt{F}, q)\downarrow$, then $h(\delta(\mathtt{F}, q)) = \delta'(\mathtt{F}, h(q))$, and
- if $q_1 \nleftrightarrow q_2$, then $h(q_1) \nleftrightarrow' h(q_2)$.

If $F \sqsubseteq F'$, we can also say that F' *extends* F.

2.2.2 Feature structure unification

Feature structures also extend the idea of combining consistent information states. This combination takes the form of feature structure *unification*. Though the mathematics of unification are quite complex, the idea again is easy: the unification of two information states (feature structures) results in the most general information state (feature structure) consistent with both previous states.

The formal definition groups feature structures into equivalence classes called *alphabetic variants* and defines unification is terms of quotient sets of equivalence classes.

Definition 2.2.5. Given a set, S, an *equivalence relation* on S is a relation $\approx \subseteq S \times S$ such that, for all $s, s', s'' \in S$:

- (reflexivity) $s \approx s$,
- (symmetry) if $s \approx s'$, then $s' \approx s$, and
- (transitivity) if $s \approx s'$ and $s' \approx s''$, then $s \approx s''$.

Definition 2.2.6. Typed feature structures F_1 and F_2 are *alphabetic variants*, written $F_1 \sim F_2$, iff $F_1 \sqsubseteq F_2$ and $F_2 \sqsubseteq F_1$.

Proposition 2.2.1. $\sim$ is an equivalence relation.

Definition 2.2.7. Given a set, S, an equivalence relation, $\approx$, and an element $s \in S$, the *equivalence class* of s under $\approx$ is:

$$[s]_{\approx} = \{s' \in S \mid s \approx s'\}$$

Definition 2.2.8. Given a set, S, an equivalence relation, $\approx$, the *quotient set* of S modulo $\approx$ is:

$$S/\approx \; = \; \{[s]_{\approx} \mid s \in S\}$$

Definition 2.2.9. Given a common signature and $F \sim \langle Q, \bar{q}, \Theta, \delta, \not\leftrightarrow \rangle$ and $F' \sim \langle Q', \bar{q}', \Theta', \delta', \not\leftrightarrow' \rangle$ such that $Q \cup Q' = \emptyset$, let $\bowtie$ be the finest grained equivalence relation on $Q \cup Q'$ such that:

- $\bar{q} \bowtie \bar{q}'$, and
- if $\delta(\mathtt{F}, q)\downarrow$, $\delta'(\mathtt{F}, q')\downarrow$ and $q \bowtie q'$, then $\delta(\mathtt{F}, q) \bowtie \delta'(\mathtt{F}, q')$.

The *unification* of F and F' is then defined to be:

$$F \sqcup F' = \langle (Q \cup Q')/\bowtie, [\bar{q}]_{\bowtie}, \Theta^{\bowtie}, \delta^{\bowtie}, \not\leftrightarrow \rangle,^{\bowtie}$$

where:

- $\Theta^{\bowtie}([q]_{\bowtie}) = \bigsqcup\{\Theta(q) \sqcup \Theta'(q') \mid q' \bowtie q\}$,
- $\delta^{\bowtie}(\mathtt{F}, [q]_{\bowtie}) = \begin{cases} [\delta(\mathtt{F}, q)]_{\bowtie} & \text{if } q \in Q, \\ [\delta'(\mathtt{F}, q)]_{\bowtie} & \text{if } q \in Q' \end{cases}$
- $[q]_{\bowtie} \not\leftrightarrow^{\bowtie} [q']_{\bowtie}$ iff there exists q'' and q''' such that $q'' \not\leftrightarrow q'''$, $q'' \bowtie q$, and $q''' \bowtie q'$,

provided that the joins in the definition of $\Theta^{\bowtie}$ exist where needed and $\not\leftrightarrow^{\bowtie}$ is anti-reflexive. $F \sqcup F'$ is undefined otherwise.

Proposition 2.2.2. Given a common signature, and $F, F' \in \mathcal{F}$, if there exists an $F'' \in \mathcal{F}$ such that $F \sqsubseteq F''$ and $F' \sqsubseteq F''$, then $F \sqcup F'\downarrow$ and $F \sqcup F' \sqsubseteq F''$.

Proof. Proven by Moshier [Mos88] and extended to the typed case by Carpenter [Car92]. □

Carpenter furthermore shows that, in fact, the partial order induced by feature structure subsumption under alphabetic variants is a bounded complete partial order, thereby ensuring that if two feature structures are unifiable there is only one result of the unification.

2.3 Appropriateness

To this point we have simply described feature structures as they appear: we have not considered how to determine which features (if any) are taken by a given type, or how to determine which feature structures appear as feature values. Conceptually, it makes sense for types to know what their features are, and for features to know what their values are, and a way to encode this knowledge is via *appropriateness specifications*.

The definition of appropriateness conditions given here is excerpted from [Pen00], who attributes the theory to [Car92].

Definition 2.3.1. Given a type hierarchy $\langle T, \sqsubseteq \rangle$, and a finite set of features, Feat, an appropriateness specification is a partial function, $Approp : Feat \times T \rightarrow T$ such that, for every $F \in Feat$:

- (Feature Introduction) there is a type $Intro(F) \in T$ such that $Approp(F, Intro(F)) \downarrow$, and for every $t \in T$, if $Approp(F, t) \downarrow$, then $Intro(F) \sqsubseteq t$, and

- (Upward Closure / Right Monotonicity) if $Approp(F, s) \downarrow$ and $s \sqsubseteq t$, then $Approp(F, t) \downarrow$ and $Approp(F, s) \sqsubseteq Approp(F, t)$.

Feature introduction ensures there is a minimal type at which each feature is introduced, and upward closure / right monotonicity ensures that, if a feature A is appropriate to a type s, A is appropriate to every subtype of s.

Appropriateness specifications are the final piece of the type signature puzzle - appropriateness specifications, in combination with a type hierarchy and a set of features, comprise an attributed type signature.

Definition 2.3.2. An attributed type signature is a structure $\langle T, \sqsubseteq, Feat, Approp \rangle$, where $\langle T, \sqsubseteq \rangle$ is a type hierarchy, Feat is a finite set of features, and Approp is an appropriateness specification.

2.3.1 Well-typing

One last issue must be discussed before we move on - the issue of *existence* of features. Though appropriateness states which features are allowed to appear on each type, appropriateness does not say if we can expect those features to be present on the type or even whether those features are the only features the type can take. To enforce this expectation, signatures must be interpreted under some form of *well-typing*.

Definition 2.3.3. A typed feature structure $F = \langle Q, \bar{q}, \Theta, \delta, \nleftrightarrow \rangle$ is *totally well-typed* iff for every $q \in Q$ and $\texttt{F} \in Feat$:

- if $\delta(\texttt{F}, q) \downarrow$, then $Approp(\texttt{F}, \Theta(q)) \sqsubseteq \Theta(\delta(\texttt{F}, q))$, and,

- if $Approp(\texttt{F}, \Theta(q)) \downarrow$, then $\delta(\texttt{F}, q) \downarrow$.

$\mathcal{TTF}$ denotes the set of totally well-typed feature structures.

Totally well-typed feature structures carry computational significance since they allow us to statically pre-determine the resulting feature structure of any feature structure unification. This pre-determination allows a computational implementation to perform some forms of pre-compilation and to strategically allocate memory, thus allowing for more efficient feature structure operations.

The ability to assert all and only the appropriate attributes of modelled objects also affords totally well-typed feature structures a sort of conceptual cleanliness by allowing grammar writers to state what the attributes of certain object classes are before knowing what those values are, thus properly capturing the original intuition of using typed feature structures to model partial knowledge.

Totally well-typed feature structures were adopted by Pollard and Sag in the standard HPSG reference [PS94], and have been used as the logic of choice in the Linguistic Knowledge Builder (LKB) [Cop02], the Attribute Logic Engine (ALE) [CP96], and HPSG grammars of English (the ERG) [FCS00], German [MK00], Japanese [Sie00], Norwegian

[HH03], and Greek [KN03]. Because this is a thesis on the usage of types in typed feature structure grammars, and because we are referencing the English Resource Grammar as our data set, the remainder of this thesis will also assume that feature structures are totally well-typed.

Chapter 3

Types in contemporary typed feature structure grammars

The HPSG community seems to have converged on totally well-typed features structures as the basis for grammar design. In such grammars, types are related by inclusional polymorphism, otherwise known as the IS-A relation, and this relation is interpreted as subset inclusion. The IS-A relation is, of course, not a random choice, but a relation that has been exploited in knowledge representation systems such as KL-ONE [Bra77], object-oriented programming languages such as Smalltalk, C++, and Java, and recently in linguistically motivated ontologies such as WordNet [Fel98].

This chapter examines, from the perspective of software engineering and by way of reference to the English Resource Grammar,[1] the effects of using hand-built explicitly-specified IS-A networks as a basis for broad-coverage HPSG grammars. The ERG is a strategic choice since, as the largest and most established HPSG grammar and the basis for the Grammar Matrix [BFO02] from which other HPSG grammars are currently being developed, the ERG both represents and drives current HPSG grammar research.

[1]In particular, we refer to a near-ALE-compatible port of a May, 2000 version of the ERG generated from the CSLI test suite using scripts written for the purpose by Stephan Oepen. Though the ERG has changed since then, we feel these remarks still pertain to more recent versions.

Much of this chapter has been previously published as [PH03a].

3.1 Advantages of types in grammar

There are undoubtedly many advantages to using types and IS-A networks in unification grammars that justify their existence in grammar implementations such as the ERG. Many of the justifications would be similar to arguments in favour of types in programming languages and software engineering, and, indeed, these are the reasons types were introduced into feature logics historically and why all large HPSG grammars employ type hierarchies. We specifically suggest the following four advantages:

- intuitive reflection of linguistic knowledge,
- implicit grammar documentation,
- early error detection, and
- efficiency gains through grammar compilation.

3.1.1 Intuitive reflection of linguistic knowledge

Types, most notably in the form of ordered type hierarchies, facilitate the division of the world into conceptual classes which are themselves hierarchical. This is exactly how we understand many linguistic constructions. Grammatical case and verb forms are in different classes, and both are in different classes from kinds of phrases. These conceptual classes can themselves be divided - nominative case is a kind of case, participles are kinds of non-finite verb forms which are in turn kinds of verb forms, and phrases can be either unary or binary, headed or nonheaded, or if a phrase is both binary and headed, either head-final or head-initial. IS-A networks enable this world knowledge to be abstracted to natural classes and their inclusional relationships captured. Then, through subsumption checking, greatest lower bound, and least upper bound operations, IS-A networks provide efficient methods for inference among these natural classes.

Conversely, using IS-A networks to write grammars encourages grammar writers to think about their grammar in terms of abstractions and categorization. In general, thinking in terms of abstractions should lead to more abstract design, which, in turn, should lead to more intuitive, readable, and extendable grammars.

3.1.2 Implicit grammar documentation

Type signatures are also useful for developers reading the grammar source because they constitute an implicit form of documentation - documentation that describes both how and why the grammar works the way it does (useful for debugging and for making extensions), and the linguistic intuition behind the grammar.

The appropriateness specifications on features, for example, constitute a form of documentation by forming relationships between types which are not otherwise related by subtyping. When appropriateness states that type *cat* has a `HEAD` feature and that the appropriate values of `HEAD` features are feature structures rooted by subtypes of *head*, the grammar has documented both one of the cornerstones of HPSG (that all syntactic categories have a lexical head), and that the possible category head types are only those which are subtypes of *head*.

Similarly, when the ERG combines person and number information in the same hierarchy (figure 3.1) and arranges those types according to knowledge of English verbal morphology, the grammar documents this linguistic intuition.

3.1.3 Early error detection

Not only do types form implicit grammar documentation, they also provide a way of enforcing that documentation through compile-time and run-time error detection. Because a parsing system knows, for example, what types are appropriate for a feature value, the

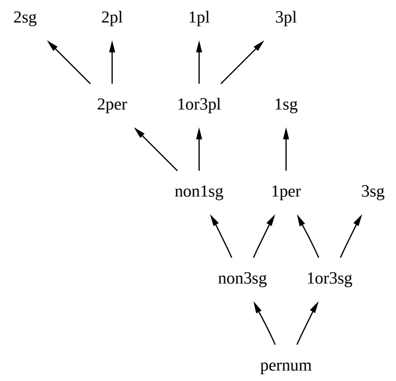

Figure 3.1: English verb morphology motivated *pernum* hierarchy

system can flag some inconsistencies before the grammar is actually executed.

This sort of error detection is absolutely critical for large typed feature structure grammars. In the ERG, parsing a seven-word sentence produces a feature structure containing approximately 400 features. If, for some reason, the grammar writer feels the parse is incorrect, tracking down the source of error can be extremely difficult. Types allow a grammar development environment to precompute greatest lower bounds and to check certain kinds of signature integrity before run-time, hence drastically reducing the amount of frustrating guess-and-test bug tracing.

3.1.4 Efficiency through grammar compilation

The first type systems in computer science, in languages such as FORTRAN [Bac81], were designed to make numerical calculations more efficient by differentiating between integral and floating point computations. By differentiating the computations, compilers were able to make more appropriate memory allocations and generate different machine instructions for each type of expression [Pie02].

Compilation of typed feature structure grammars has also resulted in efficiency gains, most notably in ALE [CP96], but even systems which do not compile grammars (such as the LKB [CF00]) perform a number of well-formedness checks before grammar execution. These pre-execution steps are enabled by IS-A networks, most notably as the backbone of totally well-typed feature structures, which facilitate offline greatest lower bound computation, partial offline computation of feature structure unification, and efficient type encodings [AKBLN89], all of which result in reduced parsing times.

3.2 Disadvantages of current type usage

The major disadvantage of simple IS-A type systems used in current HPSG implementations, however, is that they communicate only simple forms of information about their types. In the case of IS-A networks there is only one form of information - subtyping. Though limited, subtyping induces a precise semantics which enables automated inference via the well-defined operation of unification and is still more desirable than ad hoc semantic networks.[2]

In practice, though, subtyping's limited paths of communication have resulted in a proliferation of types and unwieldy signatures. Depending on how one counts, the English Resource Grammar has anywhere between 2000 and 10,000 types in its network, each one specifically placed in the network by a grammar writer. Is it likely that this network still accurately reflects the designers' perspective? According to our analysis of the signature, it does not. Specifically, upon inspection of the ERG, we note the following negative side-effects that can arise in large scale grammar development on a foundation solely of hand-crafted IS-A networks:

- lack of a uniform semantics,

[2]For a classic argument in favor of semantic primitives, see [Bra77]

- erosion of dimensionality,
- inconsistent naming conventions, and
- structural inconsistencies.

3.2.1 Lack of a uniform semantics

Problems with semantic uniformity should be readily apparent to those who have attempted to construct world models using subtyping-based hierarchies. The problem centers around the following: Given that communication in IS-A networks is limited to subtyping, how does one encode relationships between objects that are not *really* in a subtyping relationship? The solution is not clear. Object oriented programming languages have attempted differing approaches, including aspect-oriented programming (AOP) and Java's interface implementation.

To observe the problem in grammar development, consider Sag's [Sag97] treatment of relative clauses to classify entities according to multiple dimensions of types (figure 3.2). Sag achieves his classification by analyzing relative clauses on two separate dimensions: clausality and headedness. The idea is that every subtype of phrase makes some claim regarding whether or not it is a clause or has a head.

The problem with Sag's treatment is that the signature in figure 3.2 is not really multi-dimensional. It implicitly uses the typographical notation of capitalization and boxing to indicate that, conceptually, type CLAUSALITY is not a kind of phrase but a dimension of phrasal classification. In grammar development environments where capitalization and boxing are not a part of the defined language, this simply looks like an IS-A link. Whether the links from CLAUSALITY and HEADEDNESS to the row of types above are IS-A links is furthermore somewhat unclear, and that the link from CLAUSALITY to *clause* is not ordinary IS-A, but reifies a particular choice of CLAUSALITY is not indicated with even a typographical convention. The semantics of CLAUSALITY and *clause* are not inclusionally

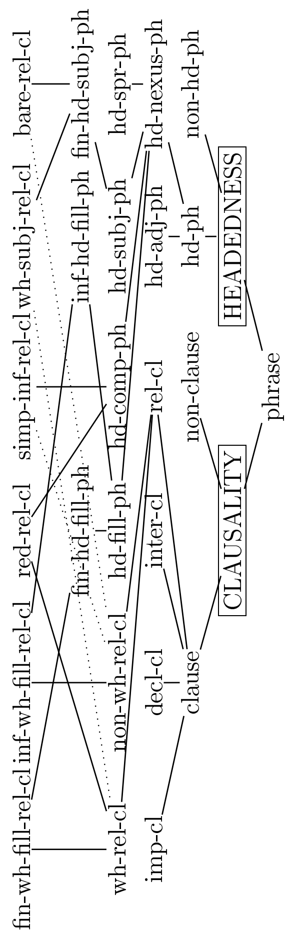

Figure 3.2: Dimensions of classification

related.

3.2.2 Erosion of dimensionality

The ERG, to its credit, has eliminated the types CLAUSALITY and HEADEDNESS, but has retained the essential problem with this analysis. These types have been replaced by types called *clause*, *hd-ph*, and *non-hd-ph*, which are three among the many immediate subtypes of *phrase* (figure 3.3). In doing so, it is simply less apparent that phrases can be analyzed along CLAUSALITY and HEADEDNESS dimensions.

3.2.3 Inconsistent naming conventions

Most HPSG linguists probably realize what a *wh-subj-rel-cl* is, and the name itself does suggest that this phrasal type is a *rel-cl* and a *hd-subj-ph* (although headedness itself is not indicated), but there are other cases in the ERG where the naming conventions are far less transparent. For example:

- Order is sometimes used rather than a compound name. The difference between a *head_adj_ph* and a *adj_head_ph*, for example, is that former is both *head_initial* and a *head_mod_phrase_simple*, while the latter is *head_final* and a *head_mod_phrase_simple*.

- The type *non1sg* does not actually refer to all non-1st-singular person-number combinations, but only to those that are also non-3rd-singular. To know this, we must observe that *non1sg* is actually a subtype of *non3g* in the ERG.

- The syntactic head type *a_or_p* refers to all heads of type *adjective* or type *preposition* and *v_or_p* refers to all heads of type *verb* or type *preposition*, while the type *v_or_g* does not refer to heads of type *verb* or type *gerund*, but to heads of type *verb*, type *comp* or type *gerund*.

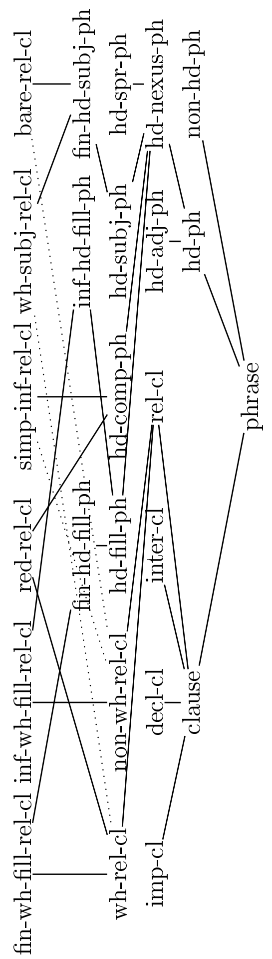

Figure 3.3: Erosion of dimensionality

- Several different kinds of connectives are employed in names, and, because these names are simply strings, it is not always clear what their operator precedence is. We thought we understood *1or3pl+2per+1per+non1sg*, for example, until we saw that it is a subtype of *1sg*+2per+1per+non1sg*.

- Other connectives are simply not clear in their intended meaning. *basic-cp-prop+ques-verb*, has only one supertype (*verb-synsem*) and is a supertype of both propositional and question verbs. This is not the same + that denotes intersection elsewhere.

As for *head_adj_ph* and *adj_head_ph*, there are at least five independent dimensions on which phrases are being classified:

1. initial vs. final,
2. binary vs. unary,
3. headed vs. non-headed,
4. intersective vs. scopal, and
5. 'h' vs. 'n' (we have not determined what these letters stand for).

These are in addition, although not unrelated, to the more familiar distinctions among complement phrases, subject phrases, etc. of HPSG. It took us at least a day to determine that these were the dimensions, but we can now say where a *n_adj_redrel_ph* stands with respect to all of them. Can you?

3.2.4 Structural inconsistencies

Manual enforcement of design decisions over a large-scale software project is a recipe for inconsistency, rendering design intentions unclear and introducing sources of error which are difficult to trace. The ERG is no exception. There are many examples where types and IS-A links seem to be either missing or misplaced, such as:

- There is a most general strict tense type (*strict_tense*), a most general strict aspectual type (*strict_aspect*), and a most general person-number strict type (*strict_pernum*), but no most general strict gender type and no most general strict mood type, although these types are added by meet-semilattice completion (appearing in the signature as *glbtype724* and *glbtype723*, respectively).

- Type *intadj9-* is a subtype of *intadj9*, *intadj6-* is a subtype of *intadj6*, and *intadj3-* is a subtype of *intadj3*, but *digit9-* is not a subtype of *digit9*, *digit6-* is not a subtype of *digit6*, and *digit3-* is not a subtype of *digit3*.

- Type *non_affix_bearing* is a direct subtype of *word_or_lexrule* but the ability to bear affixes is a dimension for classifying words, not lexical rules. Other word dimension types, such as *nonmsg* and *nonque*, are subtypes of *word*.

- Every grammar rule type is a subtype of type *phrase* except the most general grammar rule type, *lingo_rule*, which is a subtype of *sign*, but not *phrase*.

Inconsistencies like these are unavoidable in IS-A networks the size of the ERG network. Errors will always creep in, either due to human error, poorly communicated design decisions, or some other unforeseen reason - largely the same reasons error-free software is so rarely encountered.

Chapter 4

Design patterns and grammar engineering

Many of the problems afflicting IS-A networks in HPSG grammars are very similar to problems encountered in software engineering. Problems with semantic uniformity plague all large object-oriented software systems, even the mostly tightly controlled code base has naming inconsistencies, and software architectures are often ad-hoc, undocumented, and inconsistently followed.

Software engineering research has attempted to stem such problems by adapting Christopher Alexander's ideas on what he terms patterns and pattern languages in architectural form - recasting them into what are now known as design patterns and design pattern languages. The design pattern contribution draws heavily from *Notes on the Synthesis of Form* [Ale64], *The Timeless Way of Building* [Ale79], and *A Pattern Language* [AIS77], and focuses its efforts on the documentation and cataloguing of recurring successful design principles in the hopes that this catalogue will help make software more reusable, more accessible to new developers, and less error-prone. Design patterns have received a great deal of attention in both academic and industrial circles, arguably more-so than Alexander's original work has among architects.

This chapter presents an overview of design patterns and their use in software engineering, gives an argument for their applicability to grammar engineering, and presents four recurring patterns of type relationships we have mined from the English Resource Grammar. The patterns presented in §4.3 are drawn from work previously published as [PH03a].

4.1 Design patterns

In *The Timeless Way of Building* [Ale79, page 147], patterns are defined as follows:

> "Each pattern is a generic solution to some system of forces in the world."

Alexander claims that the natural state of the world can be interpreted as a system of patterns, each existing because of collective experience within its context, combining to form what we understand as events and spaces. To further develop his argument, Alexander suggests that patterns are in fact relations between contexts, problems, and solutions [Ale79, page 247]. His central claim is that architects, as creators of forms, must be aware of these patterns and the way they interact. In so doing, architects will be able to combine the patterns which "create life," thereby creating spaces which increase the quality of life of those inhabiting them.

In a move that should be of interest to linguists, Alexander furthermore proposes that his patterns come in two forms, base patterns and patterns which connect patterns, and that those patterns relate in finite combinatory systems to form *pattern languages* in a generative and principled manner, thereby allowing for infinite and creative expressions of form. The relationship is summed up in table 4.1 (excerpted from [Ale79, page 187]):

Alexander's ideas of patterns and pattern languages were adopted by the software engineering community in the early to mid 1990's as an approach for tackling problems of engineering large-scale software projects. The seminal work in this area continues to be *Design Patterns* by Gamma, Helm, Johnson, and Vlissides [GHJV95], now affectionately

Table 4.1: Christopher Alexander: natural languages vs. pattern languages

Natural Language	Pattern Language
Words	Patterns
Rules of grammar and meaning which give connections	Patterns which specify connections between patterns
Sentences	Buildings and places

called the "Gang of Four," which, giving direct credit to Alexander's work, documents twenty-three patterns for object-oriented software design. Gamma et al. take a programmer's perspective, providing patterns for the creation, structure, and behaviour of classes and objects in a object-oriented program.

A similarly very popular work, *A System of Patterns* by Buschmann, Meunier, Rohnert, Sommerlad, and Stal [BMR+96] notes that patterns can come in flavours of differing granularity, proposing three levels of patterns: system-level *architectural patterns*, component-level *design patterns*, and *idioms*. [1] Patterns of different layers combine patterns in what Buschmann et al. term, not pattern languages, but *pattern systems*. Buschmann et al. replace the term *language* with *system* to stress the notion that, in the field of software engineering, a collection of patterns may not completely describe the problem domain.

By far the most significant point that patterns adherents will stress is that, in order to be of any further use, patterns must be documented and collected into packages that

[1]Buschmann et al.'s use of the term *idiom* is misleading at best. Buschmann et al. are not referring to non-compositionality, but are using the term *idiom* to refer to translation rules from design languages to specific programming languages.

can be referenced for future design decisions. Buschmann et al. suggest documenting patterns by four descriptions: a *name*, *context*, *problem*, and *solution*. For example:

Name Model-View-Controller

Context User interface design

Problem How do we manage the interaction of user actions, data display, and (possible) data changes?

Solution Build three independent agents - a data model (Model), a presentation agent that takes the model as a parameter (View), and an intermediary (Controller) that both accepts input from the View to update the Model and updates the View if the Model changes.

Numerous other patterns and pattern languages have been proposed, facilitated by a consortium called the Hillside Group [Hil] which maintains the definitive web portal for patterns and exists to facilitate the creation of a body of literature that documents design solutions in the form of patterns and pattern languages. Already this body of literature is large and encompasses all conceivable sorts of patterns, from pedagogical patterns [Ped] to patterns for source code management [Dev] to patterns for avionics control systems [Des]. There is even a pattern to describe design gone awry - the aptly named "Big Ball of Mud" [FY97].

No matter what the problem domain, however, all patterns center around the maxim that not only is it pointless to reinvent wheels, it is also costly. To avoid reinvention, a pattern will attempt to capture the essence of a solution to a recurring problem. A proper pattern will rely on the collective experience of previous solutions to provide a design solution that is broadly applicable and customizable. Patterns should not be restrictive: there could be many valid problem solutions. Patterns simply seek to provide a customizable framework which can function as the building blocks of a specific implementation. As an example, the Model-View-Controller pattern mentioned above specifies that for the context of customizable user interface design, a valid and useful solution involves three independent agents: the data model (Model), the presentation (View), and an intermediary (Controller) which accepts input from the View and updates the Model.

Used properly, patterns add many benefits to the software engineering process. By utilizing the best recurring solutions to solve design decisions, software becomes easier to interface with other components, hence making the software more reusable. By cataloguing patterns and providing them with names, communication is increased across possibly distributed development teams. By the same token, pattern-based software facilitates accessibility to new developers by increasing system predictability. A proper pattern-influenced implementation should also be less error-prone because components have regularized behaviour. Finally, patterns can be used to guide the implementation of further abstractions such as higher-level programming languages and abstract class libraries.

4.2 A role for patterns in grammar development

Grammar engineering and software engineering bear a striking resemblance. Each requires the construction of a large interlocking system of components whose predictable and correct function is critical; each requires the management of a large source code base; and each involves development teams attempting to disseminate knowledge while working in parallel. Even the evolution of broad coverage grammars is remarkably similar to software production. There is a test-edit-debug cycle, there are stages of requirements gathering and evaluation, there is a release cycle, and the source code base must allow new extensions while maintaining previous functionality.

This similarity, though, should be expected: grammar engineering and software engineering appear to be very similar problems precisely because grammars *are* software programs. More specifically, grammars are special kinds of programs which are compiled

(or interpreted) into parsers and generators, for example, whose function is determined by the grammar source code and the logic of the underlying formalism. The logics themselves are not simply frameworks within which to explain linguistic constructs, but are actually first-class programming languages for which compilers and interpreters can be written and which can be given a declarative semantics [PS84]. Fundamentally, grammar engineering is just a special case of software engineering.

It would seem, then, that if design patterns have been successful in software engineering, one should expect patterns also to inform grammar engineering. The question is simply where the patterns are and how they can be leveraged. Unfortunately, while there has been plenty of work on grammar profiling [OF98, ONK97], development environments [Cop02, KS94a, CP96], and formalisms [Car92, Pen00, Ric04, KR90, KB82], grammar implementations are generally evaluated more for linguistic insight than for engineering quality.

One notable exception is the Grammar Matrix [BFO02, FB03] which provides a common platform for multilingual HPSG grammar development by contributing a basic type signature which includes types for list manipulation, minimal recursion semantics (MRS) [CFSP99], lexical and phrase structure rules, and semantic selection. This signature is intended to be cross-linguistically applicable - any new grammar implementation, in any language, should be able to simply make extensions to the Matrix signature. Accordingly, the Matrix signature is quite broad (many kinds of inconsistent types) and quite flat (not many subtypes), as introduced subtypes quickly cease to be cross-linguistically applicable and become language specific.

However, while the Matrix undoubtedly facilitates rapid commencement of new grammar development by using previous experience to develop a basic set of types and feature structures, its relative flatness makes it unclear how the Matrix will facilitate *better* engineered grammars and how Matrix-based grammars will avoid the engineering pitfalls that have plagued broad-coverage grammars such as the ERG. HPSG grammars with explicitly-specified IS-A networks are not flat (the ERG has a maximum subtyping depth of 21 types), but the Matrix provides no tools or suggestions for arranging subtypes as signatures are extended - one would presume there are also ways of arranging types which are cross-linguistically applicable. A library of types and features can be cross-linguistically "modular" and yet still unable to inform efficient, well-documented, easily adapted, and intuitively encapsulated (in a sense, *well-written*) grammars. Design patterns have provided a forum for discussing what constitutes good object-oriented design, and, we believe, can provide similar feedback to HPSG type signatures.

4.3 Mining the ERG for patterns

There are undoubtedly many patterns to be found in current grammar implementations. Indeed, every design decision that covers multiple portions of a grammar could be considered a pattern. The standard HPSG reference [PS94] could even be considered a set of high-level architectural patterns for grammar development using typed feature structures - sketching patterns for subcategorization, agreement, raising, and other grammatical principles. In this section we examine the English Resource Grammar in search of different sorts of patterns, namely, recurring patterns of type relationships in broad-coverage HPSG signatures. These patterns, which become apparent as HPSG grammars extend the breadth and depth of their signatures, form a set of structural primitives which we believe can succinctly express many of the linguistic principles and generalizations encoded in HPSG grammars.

4.3.1 Disjunctive types

The ERG encodes feature value disjunction by way of disjunctive types. This typed encoding is used as an alternative to what has historically been considered more computationally expensive - requiring grammar development environments to handle disjunctive

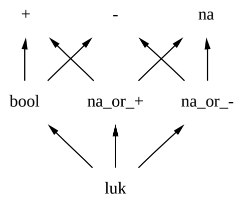

Figure 4.1: Disjunctive types in the ERG *luk* filter

feature values in descriptions [Fli00]. Hence, for each feature that requires disjunction, there is a type which subsumes the types which would otherwise appear in the disjunction, typically named using the names of non-disjunctive types and the connective '*_or_*'. The disjunctive types are ordered by set inclusion - a disjunctive type A subsumes type B iff B represents the disjunction of a subset of the types represented by A.

Types that appear with disjunctive subtypes are typically subtypes of the ERG type **sort** - types with no features and which are intended to represent indecomposable linguistic entities. A classic example is the **sort** subtype, *luk* (according to the ERG source, named *luk* in reference to Polish logician Jan Lukasiewicz), which implements a three-valued boolean logic (figure 4.1). For basic values *not applicable*, *plus*, and *minus*, *luk* has disjunctive types *not applicable or plus* (*na_or_+*), *not applicable or minus* (*na_or_-*), and *plus or minus* (*bool*). Besides *luk*, disjunctive types also appear as subtypes of *pernum* (person-number), *aspect*, *mood*, *vform* (verb form), *xmod*, *conj* (conjunction), *cat* (category), and *head* (syntactic head).

We can thus catalogue the following pattern:

Name Disjunctive types

Context Underspecification

Problem How do we represent underspecified feature values?

Solution Augment the type signature with disjunctive types representing underspecified feature values. Order these disjunctive types according to set inclusion.

4.3.2 Conjunctive types

While disjunctive types serve the purpose of underspecification, HPSG analyses of phenomena such as German case neutralization require that case features be overspecified - both nominative and accusative, both dative and accusative, or perhaps both dative and genitive. Similar problems arise in the coordination of unlike categories, in which many HPSG analyses require overspecification of category head, tense, and person-number feature values.

The ERG is consistent with proposals discussed recently by a large number of researchers ([Sag03], [LP02], [Dan02], [LHC01]) who have proposed augmenting HPSG signatures with conjunctive types drawn from a Smyth powerlattice - if a feature value needs to be both nominative and accusative at the same time, add a new type that is a subtype of both nominative and accusative. In the ERG, conjunctive types are typically named by joining the type names with either the connective '*+*' or the connective '*_and_*' (the latter usually being reserved for maximally specific conjunctive types). These conjunctive types are ordered by set inclusion in a dual way to disjunctive types - a conjunctive type A is subsumed by a conjunctive type B iff B represents a subset of the sort values represented by A.

A classic example is *tense* (figure 4.2) which, for basic values past, present, and future, has conjunctive types *past+fut* (past and future), *pres+fut* (present and future), and *pres+past* (present and past). In addition to *tense*, conjunctive types also appear as subtypes of *luk*, *pernum*, *gender*, *aspect*, *mood*, *vform*, *voice*, and *head*.

The conjunctive types pattern is thus recorded:

Name Conjunctive types

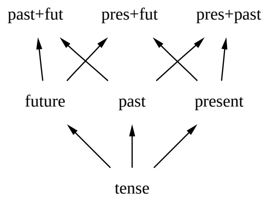

Figure 4.2: Conjunctive types in the ERG *tense* filter

Context Neutralization / Coordination of unlikes

Problem How do we represent feature values when that value must serve as multiple otherwise inconsistent values at one time?

Solution Augment the type signature with conjunctive types representing the neutralization / coordination of feature values. Order these conjunctive types according to set inclusion.

4.3.3 Strict variants

According to the logic of typed feature structures, linguistic objects must correspond to maximally specific feature structures - any feature structure that can subsume other feature structures is interpreted as an underspecification of some set of linguistic objects. Such maximally specific feature structures are called *sort resolved*, and every feature value of a sort resolved feature structure must be a leaf in the type hierarchy. Unfortunately, augmenting a signature with conjunctive types adds subtypes to many previously maximally specific types, thereby rendering feature structures taking those types non-sort resolved. This is clearly problematic, as the original intention of the types was that they could appear as feature values in linguistic objects. To counter the problem, Levine et al. [LHC01] and Daniels [Dan02] have proposed a solution in which the type hierarchy

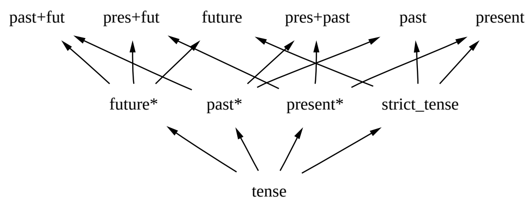

Figure 4.3: Strict variant types in ERG *tense* types

contains two copies of each conceptual type - an impure type (which has conjunctive subtypes), and a pure type (which is a subtype of the impure type and has no conjunctive subtypes). The ERG uses a variant of this solution, terming pure types *strict* types, indicating impure types with a '*' suffix, and, to provide a common supertype for pure types, usually introduces a pure disjunctive type that subsumes all the pure (but not the impure!) types. Figure 4.3 is an example of strict variants as exhibited by *tense* types – types *past**, *present**, and *future** are impure types, *past*, *present*, and *future* are the pure copies, and *strict_tense* is the pure disjunctive supertype. Strict variants also occur in *luk*, *pernum*, *gender*, *aspect*, *mood*, *vform*, and *head* types.

Name Strict Variant

Context Pure - impure distinctions

Problem How do we deal with sort-resolvedness in the presence of conjunctive types?

Solution Posit an ordinary ordered type hierarchy A. Then, produce a copy B of this hierarchy and create a subtyping link between each type in A and its copy in B.

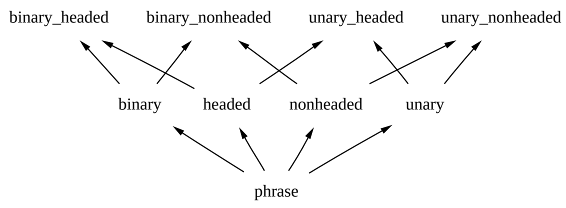

Figure 4.4: Dimensional classification

4.3.4 Dimensional encoding

Many HPSG analyses classify types along multiple dimensions. As mentioned in §3.2.1, these dimensions are often represented with boxes around the type names. The most well-known of such analyses is Sag's [Sag97] analysis of English relative clauses although the boxed notation for dimensionality is also found in Ginzburg and Sag's [GS01] analysis of English interrogatives, Malouf's [Mal00] work on verbal gerunds and mixed categories, and even in Pollard and Sag's first description of HPSG [PS87].

In all cases a most general type is introduced, subtyped by several "dimension types", after which a number of types are introduced as combinations of dimension types. Figure 4.4, for example, is a classification of phrases according to dimension types, *binary*, *unary*, *headed*, and *nonheaded*. Though it is not explicitly stated, because there are no common subtypes of *binary* and *unary* (and because of their names), we can guess that these are values of an *arity* dimension. Similarly, *headed* and *nonheaded* are values of a *headedness* dimension.

Use of dimensional classification is rampant in the ERG, especially within phrase, word, and synsem types. Phrases are classified according to (among others) head position, clausality, and argument type, in addition to headedness and arity. Word types use at least nine dimensions including affixation and participation in coordination, and synsem types are classified according to argument number, head type, and transitivity.

Name Dimensional encoding

Context Multidimensional classification

Problem How do we cross-classify types according to multiple dimensions?

Solution Posit a most general type P. Then, add subtypes to P for each dimension type. Finally, each multi-dimensionally classified type inherits from its necessary dimension types.

Chapter 5

Exploiting design patterns via parametric typing

One of the most significant advantages furnished by design patterns is the ability to guide development of higher-level abstractions. In the realm of object-oriented programming these abstractions generally take the form of abstract classes and libraries, and, in the realm of typed feature structure grammars, we suggest these abstractions can take the form of parametric typing. This chapter presents that suggestion by introducing parametric types for attribute-value logic and formalizing a number of extensions that make them usable for typed feature structure grammar engineering.

In particular, §5.1 sketches previous work in parametric typing and formally extends Penn's [Pen00] parametric types to include parametric restrictions. §5.2 then provides an alternative to induced parametric type hierarchies by motivating and formalizing several useful constrained induction operations and proving certain conditions under which such constrained induction operations can be properly computed. Previously, parametric types had no formal methods for declaring appropriate parameter values and no usable methods for determining which ground instances of parametric types are required for unification-based processing.

5.1 Parametric types for attribute-value logic

Parametric types are an adaptation of the more familiar parametric polymorphism found in many programming languages, forms of which have appeared numerous times in the feature structure literature. Smolka [Smo89] used parametric polymorphism in his typed constraint resolution languages, Klein used parametric types to encode hierarchical structures in phonology [Kle91], Pollard [Pol90], and later Pollard and Sag [PS94] used parametric types for the description of HPSG lists, and Penn [Pen98, Pen00] has provided an account of parametric types for the logic of typed feature structures. More recently, Pollard [PH03b, Pol04] has advocated Higher-Order Grammar, a formalism which obtains parametric polymorphism via indexed products drawn from category theory.

This author is currently unaware of any typed feature structure grammars consistently employing parametric typing, perhaps because their appearance in the literature has either been informal or not general enough to facilitate their use in large-scale grammar development. Penn's account appears to hold the most promise, providing a way give parameters to types, a way to think of subtyping and parameter sharing among parametric types, and a way of unfolding parametric signatures into non-parametric ones, but, as we shall see, lacks a number formal devices necessary for their application to grammar implementations.

The central problem with Penn's account is its unrestricted parameters: each parameter of each parametric type is allowed to take any type as a parameter value. Unfortunately, though these unrestricted parameters allow for a significant amount of conceptual and mathematical simplicity, we intuitively know that some types will not make sense as values for certain parameters. For a simple example, consider an agreement type with parameters for person, number, and gender. Type *masculine* should be a valid value for the gender parameter, but not for person or number, and type *3rd* should be a valid value for person, but not for number or gender.

One could simply put the onus on the grammar writer to use parameter values that make sense and, as [Pen00] described in §5.5, require grammar development environments to infer which parameter values are necessary for computation. This approach, of course, is extremely susceptible to human error and has no way of providing meaningful error reporting. A better approach, and the one taken here, is to enforce appropriate parameter values at the level of type descriptions by introducing *parametric restrictions*. Analogous to appropriateness specifications for typed feature structures, parametric restrictions assign an appropriate type t to each parameter p, thereby restricting the allowed values of p to types subsumed by t.

The rest of this section is organized as follows. §5.1.1 introduces parametric type hierarchies without restrictions, §5.1.2 describes how to induce non-parametric hierarchies from parametric ones, §5.1.3 adds restrictions to parametric type hierarchies and considers the effects of restrictions on induced non-parametric hierarchies, §5.1.4 discusses a few sensible conditions for well-behaved parametric type hierarchies, §5.1.5 shows that, under the right conditions, restricted parametric type hierarchies induce meet-semilattices, §5.1.6 considers the conditions under which restricted parametric type hierarchies induce finite non-parametric ones, and, finally, §5.1.7 considers parametric appropriateness specifications for restricted parametric type hierarchies. §5.1.1, §5.1.2, and §5.1.7 remain unchanged from [Pen00]; the remaining sections, though making reference to and adapting [Pen00] at various points, can be considered novel contributions.

5.1.1 Parametric type hierarchies

Parametric types are not types in the sense we understand types in IS-A networks. Instead, parametric types are functions that provide reference to a set of types (their *ground instances*) by way of a set of argument types (their *parameters*). For example, in figure 5.1, *list* is a parametric type whose ground instances include $list(bot)$, $list(sign)$, and $list(list(word))$. In a similar fashion, parametric type hierarchies are not simple inheritance hierarchies, but are sets of functions ordered over a subsumption relation,

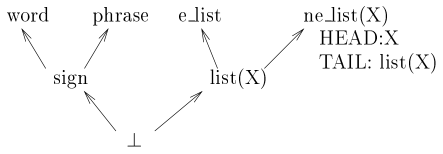

Figure 5.1: Parametric type hierarchy

$\sqsubseteq_P$, augmented with specifications stipulating which and how many parameters each parametric type takes. To compute with parametric types, $\sqsubseteq_P$ is interpreted as a relation between image sets, thus providing a method for computing meets and joins between ground instances of parametric types.

We will refer to all such functions from sets of parameters to sets of ground instances as *parametric types*. 0-ary functions will be alternatively labelled *simple types*, with the understanding that simple types are really parametric types whose image is a singleton set. The terms *type* and *ground instance type* will be reserved for types appearing in the image of some parametric type. We will reserve the term *parameter* to indicate a specific argument of a parametric type, and the term *parameter value* for the type a parameter takes in a ground instance. Following [Pen00], we will assume that parametric type hierarchies are finite bounded complete partial orders and we will also assume the presence of a most general simple type, $\perp$.

In the parametrically typed lists of figure 5.1, for example, *list* and *ne_list* are unary functions from types to types, and $\perp$, *sign*, *word*, *phrase*, and *e_list* are 0-ary functions (simple types).

Definition 5.1.1. A *parametric (type) hierarchy* is a finite BCPO, $\langle P, \sqsubseteq_P \rangle$, plus an arity function, *arity* : $P \longrightarrow \mathsf{Nat} \cup \{0\}$, and a partial *argument assignment function*, $a_P : P \times P \times \mathsf{Nat} \rightarrow \mathsf{Nat} \cup \{0\}$, in which:

- P consists of (simple and) parametric types, and includes the most general type,

$\perp$, which is simple, i.e., $arity(\perp) = 0$,

- For $p, q \in P$, $a_P(p, q, i)$, written $a_p^q(i)$, is defined iff $p \sqsubseteq_P q$ and $1 \le i \le arity(p)$,

- $0 \le a_p^q(i) \le arity(q)$, when it exists,

- if $a_p^q(i) \ne 0$ and $a_p^q(i) = a_p^q(j)$, then $i = j$,

The argument assignment function, a, tells us which parameters are shared among parametric types. The effects of shared parameters will be made clear in the next sections, but for now it suffices to note that in figure 5.1, $a_{list}^{ne\_list}(1) = 1$ and $a_{list}^{e\_list}(1) = 0$, indicating that X is a parameter of both *list* and *ne_list*, but not a parameter of *e_list*.

5.1.2 Induced non-parametric hierarchies

As noted previously, $\sqsubseteq_P$ has two interpretations. In one sense $\sqsubseteq_P$ is a relationship between functions, but we can just as easily view $\sqsubseteq_P$ as a relationship between the *images* of those functions. This duality makes it possible to induce non-parametric hierarchies from parametric ones by ordering the images such that they preserve the subtyping relationships of their domains.

Definition 5.1.2. Given a parametric type hierarchy, $\langle P, \sqsubseteq_P, arity, a\rangle$, the *unrestricted induced (type) hierarchy*, $\langle U(P), \sqsubseteq_U\rangle$, is defined such that:

- $U(P) = \bigcup_{n<\omega} U_n$, where the sequence $\{U_n\}_{n<\omega}$ is defined such that:
  - $U_0 = \{p \mid p \in P, arity(p) = 0\}$,
  - $U_{n+1} = U_n \cup \{p(t_1, \ldots, t_{arity(p)}) \mid p \in P, t_i \in U_n, 1 \le i \le arity(p)\}$, and

- $p(t_1, \ldots, t_{arity(p)}) \sqsubseteq_U q(u_1, \ldots, u_{arity(q)})$ iff $p \sqsubseteq_P q$, and, for all $1 {\le} i {\le} arity(p)$, either $a_p^q(i) = 0$ or $t_i \sqsubseteq_U u_{a_p^q(i)}$.

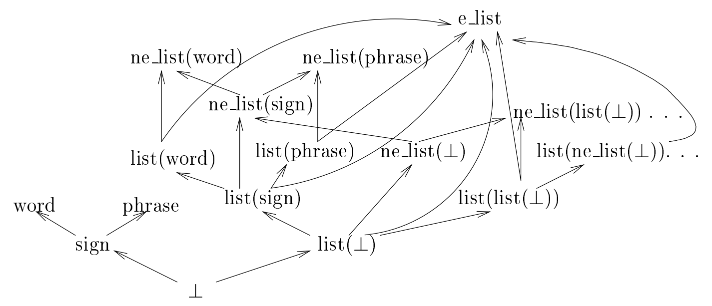

Figure 5.2: Induced non-parametric type hierarchy

In an induced hierarchy, ground instance types are formed by inductively "filling" parameter slots with other ground instances. Subtyping in the induced hierarchy is then defined such that the ground instances obey the subtyping relationships of their parameter values. As a visual example, figure 5.2 is a portion of the corresponding non-parametric unrestricted hierarchy induced by figure 5.1.

Because parametric types can take ground instances of themselves (or of their supertypes) as parameters, induction could proceed to an unbounded depth, even to include types with infinitely embedded parameters. The meanings of such types are rather unclear. For this reason $U(P)$ only considers finite sequences $\{U_n\}_{n<\omega}$, thereby excluding all would-be types with infinitely embedded parameters.

Throughout the rest of the chapter we will distinguish such types by *parametric depth*.

Definition 5.1.3. Given a parametric type hierarchy, $\langle P, \sqsubseteq_P, arity, a_P\rangle$, the *parametric depth* of a type $g = p(t_1, \ldots, t_n) \in U(P)$, $\pi(g)$, is defined such that:

$$\pi(g) = \begin{cases} 0 & if \quad n = 0, \\ 1 + \max_{1 \le i \le n} \pi(t_i) & if \quad n > 0. \end{cases}$$

5.1.3 Parametric restrictions

Parametric restrictions can be viewed as appropriateness conditions on parameters, denoting the subset of $U(P)$ which contains allowable parameter values. We formalize parametric restrictions by associating a ground instance type with each parameter:

Definition 5.1.4. Given a parametric type hierarchy, $\langle P, \sqsubseteq_P, arity, a\rangle$, a *parametric restriction function* $Restrict : P \times \mathsf{Nat} \longrightarrow U(P)$ is a function such that:

- for $p \in P$, $Restrict(p, i)$ is defined iff $1 \leq i \leq arity(p)$, and
- (parametric right monotonicity) for $p, q \in P$, if $a^q_p(i) \neq 0$, then $Restrict(p, i) \sqsubseteq_U Restrict(q, a^q_p(i))$.

$Restrict(p, i)$ is called the i^{th} *parametric restriction* of p.

Definition 5.1.5. A *restricted* parametric type hierarchy, $\langle P, \sqsubseteq_P, arity, a, Restrict\rangle$ is a parametric type hierarchy, $\langle P, \sqsubseteq_P, arity, a\rangle$, plus a parametric restriction function, $Restrict$.

Notice that parametric restrictions allow two parametric types p and q to enforce different restrictions r_1 and r_2 on a shared parameter X. Parametric right monotonicity states that, in this case, if $p \sqsubseteq_P q$, then $r_1 \sqsubseteq_U r_2$. In our running example, figure 5.3 is a parametric type hierarchy in which $Restrict(list, 1) = sign$ and $Restrict(ne\_list, 1) = sign$.

Restricted parametric type hierarchies induce non-parametric hierarchies in the same way unrestricted hierarchies do, but with an important difference – restricted induced non-parametric hierarchies include only the ground instances whose parameter values are subtypes of their corresponding parameter restrictions.

Definition 5.1.6. Given a restricted parametric type hierarchy, $\langle P, \sqsubseteq_P, arity, a, Restrict\rangle$, the *restricted induced (type) hierarchy*, $I(P)$ is defined such that:

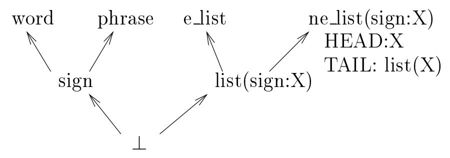

Figure 5.3: Parametric type hierarchy with restrictions

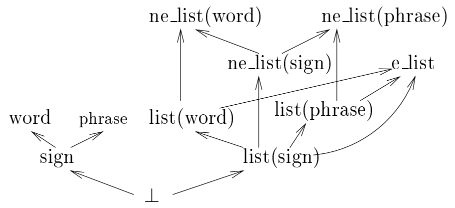

Figure 5.4: A restricted induced non-parametric type hierarchy

- $I(P) = \bigcup_{n<\omega} I_n$, where the sequence $\{I_n\}_{n<\omega}$ is defined such that:
  - $I_0 = \{p \mid p \in P, arity(p) = 0\}$,
  - $I_{n+1} = I_n \cup \{p(t_1, \ldots, t_{arity(p)}) \mid p \in P, t_i \in I_n, Restrict(p, i) \sqsubseteq_U t_i, 1 \leq i \leq arity(p)\}$, and
- $\sqsubseteq_I = \sqsubseteq_U |_{I(P)}$

Proposition 5.1.1. For any parametric type hierarchy P, $I(P) \subseteq U(P)$.

Proof. $Restrict$ only eliminates types from $U(P)$. □

Figure 5.4 is the restricted induced hierarchy of figure 5.3. Because the restriction of both *list* and *ne_list*'s parameter was *sign*, there are only lists of signs, words, and phrases.

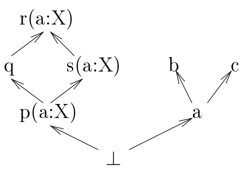

Figure 5.5: A parametric type hierarchy for which $I(P)$ is not a partial order.

5.1.4 Conditions on restricted parametric type hierarchies

There are a few outstanding issues that can cause parametric type hierarchies to behave strangely. The first involves the restriction function. Because *Restrict* must be defined before the restricted non-parametric hierarchy it induces, *Restrict* must be a function from P to $U(P)$, not from P to $I(P)$, and it is therefore quite possible to assign a restriction $q(a)$ to p where $q(a)$ does not obey the parametric restrictions on q. By definition, the construction $I(P)$ allows p to take only those parameters which obey the restrictions on q, but, for the sake of conceptual clarity, we simply avoid this situation by deliberately forcing restrictions to those types which will appear in $I(P)$.

Definition 5.1.7. A parametric type hierarchy, $\langle P, \sqsubseteq_P, \mathit{arity}, a, \mathit{Restrict}\rangle$, is *appropriately restricted* iff for all p, i such that $\mathit{Restrict}(p,i)\downarrow$, $\mathit{Restrict}(p,i) \in I(P)$.

There are also ways to propagate parameters which prohibit $I(P)$ from being a partial order. For a simple example, consider the parametric type hierarchy in figure 5.5 (excerpted from [Pen00]). In this case, $p(b) \sqsubseteq_I q$ and $q \sqsubseteq_I r(c)$, but $p(b) \not\sqsubseteq_I r(c)$, and so transitivity is violated. The problem is that different paths from p to r disagree on whether the parameter should be propagated. To solve this problem, a notion of *semi-coherence* is required (adapted from [Pen00]) to force all subtyping paths to agree on how parameters are propagated.

Definition 5.1.8. $\langle P, \sqsubseteq_P, \mathit{arity}, a_P, \mathit{Restrict}\rangle$ is *semi-coherent* iff, for all $p, q \in P$ such

that $p \sqsubseteq_P q$, all $1 \le i \le \mathit{arity}(p)$, $1 \le j \le \mathit{arity}(q)$:

- $a_p^p(i) = i$,
- either $a_p^q(i) = 0$ or for every chain $p = p_1 \sqsubseteq_P p_2 \sqsubseteq_P \ldots \sqsubseteq_P p_n = q$, $a_p^q(i) = a^{p_n}_{p_{n-1}}(a^{p_{n-1}}_{p_{n-2}}(\ldots a^{p_2}_{p_1}(i)\ldots))$, and
- if $p \sqcup_P q \downarrow$, then for all i and j for which there is a $k \ge 1$ such that $a_p^{p\sqcup_P q}(i) = a_q^{p\sqcup_P q}(j) = k$, the set, $\{s \mid p \sqcup_P q \sqsubseteq_P s$ and $(a_p^s(i) = 0$ or $a_q^s(j) = 0)\}$ is empty or has a least element (with respect to $\sqsubseteq_P$).

Proposition 5.1.2. If $\langle P, \sqsubseteq_P, \mathit{arity}, a_P, \mathit{Restrict}\rangle$ is semi-coherent, then $\langle I(P), \sqsubseteq_I\rangle$ is a partial order.

Proof. ([Pen00]) Transitivity can be proven by induction on the greatest parametric depth, k, of three types $g_1 = p(t_1, \ldots, t_n)$, $g_2 = q(u_1, \ldots, u_m)$, and $g_3 = r(v_1, \ldots, v_l)$ in $I(P)$ such that $g_1 \sqsubseteq_I g_2$ and $g_2 \sqsubseteq_I g_3$. It must then be that $p \sqsubseteq_P q$ and $q \sqsubseteq_P r$.

If $k = 0$, then p, q, and r are simple, and transitivity follows from the transitivity of $\sqsubseteq_P$. Suppose $k > 0$ and transitivity holds for $k-1$. Then, we know that, for all $1 \le i \le n$, either $a_p^q(i) = 0$ or not. If $a_p^q(i) = 0$, then if $a_p^r(i) \ne 0$, the chain $p \sqsubseteq_P q \sqsubseteq_P r$ would violate semi-coherence, so $a_p^r(i) = 0$. If $a_p^q(i) \ne 0$, then either $a_q^r(a_p^q(i)) = 0$ or not. If $a_q^r(a_p^q(i)) = 0$, then $a_p^r(i) = 0$ or the chain $p \sqsubseteq_P q \sqsubseteq_P r$ would again violate semi-coherence. If $a_q^r(a_p^q(i)) \ne 0$, then $a_q^r(a_p^q(i)) = a_p^r(i)$ by semi-coherence, and hence $t_i \sqsubseteq_I u_{a_p^q(i)}$ and $u_{a_p^q(i)} \sqsubseteq_I v_{a_q^r(a_p^q(i))} = v_{a_p^r(i)}$. The greatest parametric depth of t_i, $u_{a_p^q(i)}$, and $v_{a_p^r(i)}$ is at most $k-1$, and thus, by induction, $t_i \sqsubseteq_I v_{a_p^r(i)}$. This applies to all $1 \le i \le n$, so $g_1 \sqsubseteq_I g_3$.

Reflexivity and anti-symmetry follow from a similar inductive proof. $\square$

5.1.5 Induced semi-lattices and parametric well-formedness

Fortunately, a nice result arises if P is semi-coherent and appropriately restricted, namely, that the induced non-parametric hierarchy $I(P)$ is a meet semi-lattice.

Theorem 5.1. *If $\langle P, \sqsubseteq_P, arity, a_P, Restrict\rangle$ is semi-coherent then $\langle I(P), \sqsubseteq_I\rangle$ is a meet semilattice. In particular, given $g_1 = p(t_1, \ldots t_n)$, $g_2 = q(u_1, \ldots u_m) \in I(P)$, $g_1 \sqcup_I g_2 \downarrow$ iff:*

- *$p \sqcup_P q \downarrow$, and*
- *there exists an $s \sqsupseteq_P p \sqcup_P q$ such that for all i, j and all $k > 0$, if $a_p^{p\sqcup_P q}(i) = a_q^{p\sqcup_P q}(j) = k$, then $t_i \sqcup_I u_j \sqcup_I Restrict(s,k) \downarrow$, or $a_p^s(i) = 0$ or $a_q^s(j) = 0$.*

and, when it exists, $g_1 \sqcup_I g_2 = r(v_1, \ldots v_l)$, where r is the least such s as described above, and for all $1 \leq h \leq l$:

$$v_h = \begin{cases} t_i \sqcup_I u_j \sqcup_I Restrict(r,h) & \text{if there exist } i,j \text{ such that } a_p^r(i) = a_q^r(j) = h \\ t_i \sqcup_I Restrict(r,h) & \text{if such an } i, \text{ but no such } j \\ u_j \sqcup_I Restrict(r,h) & \text{if such a } j, \text{ but no such } i \\ Restrict(r,h) & \text{if no such } i \text{ or } j \end{cases}$$

Proof. (Adapted from [Pen00])

($\Rightarrow$): By contraposition using types $g_1 = p(t_1, \ldots, t_n)$ and $g_2 = q(u_1, \ldots, u_m)$. If $p \sqcup_P q \uparrow$, then by the bounded completeness of P, $\{g_1, g_2\}^u = \{p, q\}^u = \emptyset$. If $p \sqcup_P q \downarrow$, we use induction on the greatest parametric depth d of types g_1, g_2, and $g_3 = (p \sqcup_P q)(Restrict(p \sqcup_P q, 1), \ldots, Restrict(p \sqcup_P q, arity(p \sqcup_P q)))$.

First, notice that if $p \sqcup_P q \downarrow$, then $\{g_1, g_2\}^u = \emptyset$ iff $\{g_1, g_2, g_3\}^u = \emptyset$. Now, if $d = 0$, then $g_1 \sqcup_I g_2 = g_3$, which is the least ground instance of $p \sqcup_P q$. Suppose $d > 0$ and for all $s \sqsupseteq_P p \sqcup_P q$, there exist i, j and a $k > 0$ such that $a_p^{p\sqcup q}(i) = a_p^{p\sqcup q}(j) = k$ and $t_i \sqcup_I u_j \sqcup_I Restrict(p \sqcup_P q, k) \uparrow$ and $a_p^s(i) \neq 0$ and $a_q^s(j) \neq 0$. Now consider some $g_4 = s(v_1, \ldots, v_l)$ such that $g_1 \sqsubseteq_I g_4$. So $p \sqsubseteq_P s$. Either $q \sqsubseteq_p s$ or not. If not, then $g_2 \not\sqsubseteq_I g_4$ and $g_4 \notin \{g_1, g_2\}^u$. Consider the i,j, and k of s as specified above. $t_i \sqcup_I u_j \sqcup_I Restrict(p \sqcup_P q, k) \uparrow$, so, by induction, $\{t_i, u_j, Restrict(p \sqcup_P q, k)\}^u = \emptyset$. $p \sqsubseteq_P p \sqcup_P q \sqsubseteq_P s$ and $q \sqsubseteq_P p \sqcup_P q \sqsubseteq_P s$ are chains, so, by semi-coherence, $a_p^s(i) = a_{p\sqcup q}^s(a_p^{p\sqcup q}(i)) = a_{p\sqcup q}^s(k) = a_{p\sqcup q}^s(a_q^{p\sqcup q}(j)) = a_q^s(j) = h \neq 0$. Therefore, $a_{p\sqcup q}^s(k) = h$, and by parametric

right monotonicity, $Restrict(p \sqcup_P q, k) \sqsubseteq_I v_h$. Also, since $g_1 \sqsubseteq_I g_4$ and $a_p^s(i) = h \neq 0$, $t_i \sqsubseteq_I v_h$. Therefore, since $\{t_i, u_j, Restrict(p \sqcup_P q, k)\}^u = \emptyset$, $u_j \not\sqsubseteq_I v_{a_q^s(j)} = v_h$, and since $a_q^s(j) \neq 0$, $g_2 \not\sqsubseteq_I g_4$. Thus, in either case, $\{g_1, g_2, g_3\}^u = \emptyset$ and so $\{g_1, g_2\}^u = \emptyset$.

($\Leftarrow$): It is sufficient to show that when such an s exists, there is a least s, r. Given that claim, the choice of $v_1, \ldots, v_l$ above is clearly the unique least choice of parameters.

Given some not necessarily least s, consider all triples $\langle i, j, k\rangle$ for which $a_p^{p\sqcup q}(i) = a_q^{p\sqcup q}(j) = k > 0$, $t_i \sqcup_I u_j \sqcup_I Restrict(p \sqcup_P q, k) \uparrow$, and either $a_p^s(i) = 0$ or $a_q^s(j) = 0$. If there are no such $\langle i, j, k\rangle$ then $t_i \sqcup_I u_j \sqcup_I Restrict(p \sqcup_P q, k) \downarrow$ whenever $a_p^{p\sqcup q}(i) = a_q^{p\sqcup q}(j) = k > 0$ and so $r = p \sqcup_P q$. Otherwise, for each such triple, let: $R_{\langle i,j,k\rangle} = \{r \mid p \sqcup_P q \sqsubseteq_P r, (a_p^r(i) = 0 \text{ or } a_q^r(j) = 0)\}$

Clearly, for all such triples, $s \in R_{\langle i,j,k\rangle}$, so by semi-coherence, all $R_{\langle i,j,k\rangle}$ have least elements, $r_{\langle i,j,k\rangle}$. Furthermore, $s \in \{r_{\langle i,j,k\rangle}\}^u_{\langle i,j,k\rangle}$, so, by the bounded completeness of P, there exists an $r = \bigsqcup_{\langle i,j,k\rangle} r_{\langle i,j,k\rangle}$. There are chains $p \sqsubseteq_P p \sqcup_P q \sqsubseteq_P r_{\langle i,j,k\rangle} \sqsubseteq_P r$ and $q \sqsubseteq_P p \sqcup_P q \sqsubseteq_P r_{\langle i,j,k\rangle} \sqsubseteq_P r$, so if $a_p^{r_{\langle i,j,k\rangle}}(i) = 0$, then by semi-coherence, $a_p^r(i) = 0$; and likewise for q. Thus r satisfies the same conditions as s, and, by its construction, is clearly least. □

Definition 5.1.9. Parametric type hierarchy $\langle P, \sqsubseteq_P, arity, a, Restrict\rangle$ is *parametrically well-formed* iff it is semi-coherent and appropriately restricted.

From now on we will assume parametric type hierarchies are parametrically well-formed.

5.1.6 Finite induced hierarchies

Assuming the existence of a most general non-parametric type $\perp$, finite unrestricted parametric type hierarchies with at least one parametric type always generate infinite induced hierarchies (figure 5.6). Restrictions, by disallowing certain parameter values, can disallow infinite parametric depths and therefore generate finite induced hierarchies

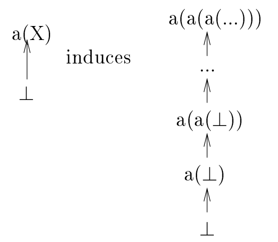

Figure 5.6: A small unrestricted type hierarchy and its non-finite induced hierarchy

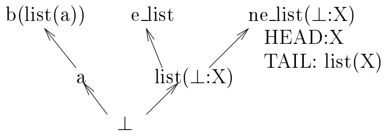

Figure 5.7: A restricted parametric type hierarchy for which $I(P)$ is not finite

as in figure 5.4, but, even in the presence of restrictions, infinite induced hierarchies are possible. Consider the parametric type hierarchy in figure 5.7. $Restrict(list, 1) = \bot$, and so for each subtype p of $\bot$ in $I(P)$, $I(P)$ must include a ground instance $list(p)$. These ground instances include, among others, $list(\bot)$, $list(list(\bot))$, $list(list(list(\bot)))$, and $list(list(list(list(\bot))))$. In fact, if $I(P)$ had not explicitly imposed a parametric depth limit ω, $I(P)$ would have included both an infinite number of types and types with infinite parametric depth. Parametric type b is also problematic. Ground instances of b include $b(list(a))$, $b(list(b(list(a))))$, $b(list(b(list(b(list(a))))))$, and so on – their number and parametric depth is limited only by the imposed parametric depth ω.

This section investigates the necessary and sufficient conditions for the finiteness of induced hierarchies. We will show that finiteness can be guaranteed by the absence of parametric restrictions $Restrict(p, i) = r$ where r subsumes the least ground instance of p or where r or any of its subtypes contains a (possibly embedded) parameter value

which subsumes the least ground instance of p. The proofs of such conditions adapt the subtype-appropriateness graphs used in [Pen00] to prove finiteness conditions on typed feature structure signatures. Accordingly, many of the proofs and definitions included in this section closely resemble those in [Pen00, §4.2].

Definition 5.1.10. Given a parametric type hierarchy, $\langle P, \sqsubseteq_P, arity, a, Restrict\rangle$, the *least ground instance* of a type $p \in P$, $min : P \to I(P)$, is defined such that:

$$min(p) = \begin{cases} p & \text{if} \quad arity(p) = 0, \\ p(Restrict(p,1), \ldots, Restrict(p, arity(p))) & \text{if} \quad arity(p) > 0. \end{cases}$$

Definition 5.1.11. Given a parametric type hierarchy $\langle P, \sqsubseteq_P, arity, a, Restrict\rangle$, let $MIN(P) = \{min(p) \mid p \in P\}$, the set of minimal ground instances of P.

Definition 5.1.12. Given a parametric type hierarchy $\langle P, \sqsubseteq_P, arity, a, Restrict\rangle$, its partial *parameter path value* function is a function $\delta : \mathsf{Nat}^* \times I(P) \to I(P)$ such that for $g = p(t_i, \ldots, t_n)$:

- $\delta(\epsilon, g) = g$, and
- $\delta(i\pi, g) = \delta(\pi, t_i)$.

Definition 5.1.13. Given ground instance type g, let $\mathcal{V}^*(g) = \{p \mid \exists\pi, \delta(\pi, g) = p\}$, the set of parameter path values of g.

Parameter path values are designed to provide reference to embedded parameter values. As an example, $\mathcal{V}^*(p(t(a), u(b(c)))) = \{p(t(a), u(b(c))), t(a), u(b(c)), b(c), a, c\}$. Notice that, in all cases, if P is appropriately restricted, $\mathcal{V}^*(g) \subseteq I(P)$. Furthermore, due to the construction of $I(P)$, $\mathcal{V}^*(g)$ is always finite.

Definition 5.1.14. Given a parametric type hierarchy $\langle P, \sqsubseteq_P, arity, a, Restrict\rangle$, the *subtype-restriction graph* of P, $SR(P)$, is a labelled directed graph $\langle \bigcup_{p \in MIN(P)} \mathcal{V}^*(p)$, $\{\langle p_1, p_2, i\rangle \mid \exists\pi\delta(i\pi, p_1) = p_2\} \cup \{\langle p_1, p_2, \mathsf{S}\rangle \mid p_1 \sqsubset_I p_2\}\rangle$, whose vertices are the parameter

path values of minimal ground instances of P and whose edges consist of edges with integer labels that map from types to parameter path values plus edges with label $\mathtt{S}$ that close $SR(P)$ under $\sqsubset_I$.
Given $p \in SR(P)$, let $SR(p)$ be the subgraph of $SR(P)$ consisting of all and only those nodes p', for which there is a path from p to p' in $SR(P)$.

We can now characterize the conditions on a finite filter of types in $I(P)$ based solely on properties observable from P and *Restrict*.

Definition 5.1.15. Given a parametric type hierarchy $\langle P, \sqsubseteq_P, arity, a, Restrict\rangle$, let $A : P \to Pow(I(P))$ be such that:

$$A(p) = \{a \in I(P) \mid min(p) \sqsubseteq_I a\}$$

Definition 5.1.16. Parametric type, p is *finite* iff $|A(p)|$ is finite.

Definition 5.1.17. Given $SR(P) = \langle V, E\rangle$, $p \in V$ is *SR-recursive* iff there is a path from p to p in $SR(P)$.

Definition 5.1.18. Parametric type p, is *provably finite* iff $SR(min(p))$ is acyclic.

Definition 5.1.19. Given a parametric type hierarchy $\langle P, \sqsubseteq_P, arity, a, Restrict\rangle$, let $\theta : P^* \times \mathtt{Nat}^* \to I(P)$ be be a partial function such that $\theta(t_1 \ldots t_k, i_1 \ldots i_k) = a_1$ where

$$\begin{aligned}
&a_1 = t_1(Restrict(t_1, 1), \ldots, Restrict(t_1, i_1 - 1), a_2, \ldots),\\
&a_2 = t_2(Restrict(t_2, 1), \ldots, Restrict(t_2, i_2 - 1), a_3, \ldots),\\
&\ldots,\\
&a_{k-1} = t_{k-1}(Restrict(t_{k-1}, 1), \ldots, Restrict(t_{k-1}, i_{k-1} - 1), a_k, \ldots), \text{ and}\\
&a_k = t_k(Restrict(t_k, 1), \ldots, Restrict(t_k, arity(t_k))
\end{aligned}$$

Lemma 5.1. *If $g = p(u_1, \ldots, u_{arity(p)})$ is SR-recursive, then p is not finite.*

Proof. Choose a cycle of length n from g to itself in $SR(P) : g \xrightarrow{x_1} g_2 \xrightarrow{x_2} \ldots g_{n-1} \xrightarrow{x_n} g$, where the x_i are either the subtype label $\mathtt{S}$, or integral parameter labels. Let $\mathtt{j} = j_1 \ldots j_k$ be the sequence of indices for which x_i is a parameter label and let $\mathtt{a} = a_1 \ldots a_k$ be the sequence of parent nodes for which x_i is a parameter label. $k \geq 1$ because $\sqsubseteq_I$ is anti-symmetric. Let $\psi : P^* \times \mathtt{Nat}^* \to Pow(I(P))$ be a partial function such that $\psi(\phi, \delta) = \theta(\mathtt{a} \circ \phi, \mathtt{j} \circ \delta)$ iff $len(\phi) = len(\delta)$. Then for all $i \geq 1$, $\psi^i(\mathtt{a}, \mathtt{j}) \in A(p)$ and p is not finite. □

Lemma 5.2. *If P is finite and p is not provably finite, then p is not finite.*

Proof. Without loss of generality, let us assume that $arity(p) = 1$. The proof proceeds by induction on the length λ of the shortest path from $min(p)$ to an SR-recursive type g. Such a path must exist because p is not provably finite and P is finite. If $\lambda = 0$, then $min(p)$ is SR-recursive and p is not finite. If $\lambda \geq 1$ and $min(p) \xrightarrow{i} g$ is on the shortest path, then g is a parameter path value of $min(p)$ and for each subtype g^i of g, there exists t^i such that $p(t^i) \in A(p)$ and $g^i \in \mathcal{V}^*(t^i)$ and p is therefore not finite. If $\lambda \geq 1$ and $min(p) \xrightarrow{S} g$, then for each subtype g^i of g, $g^i \in A(p)$ and p is not finite. □

Lemma 5.3. *If P is finite and p is provably finite, then p is finite.*

Proof. Suppose $SR(P) = \langle G, V\rangle$, p is provably finite and p is not finite. Without loss of generality, assume also that for all $q \in P$, $arity(q) = 1$. Choose any infinite set $S \subseteq A(p)$. It must be the case that for any $n > 0$, there exists a $t \in S$ and $r \in P$ such that $\mathcal{V}^*(t)$ contains n ground instances of r (if not, then there is an $N > 0$ which bounds the maximum parametric depth of any type in S and hence because P is finite then S must be finite). Let $g_1, g_2, \ldots, g_n$ be the set of n ground instances of r, ordered by decreasing parametric depth. Consider g_1 and g_2. Both are ground instances of r, and because $arity(p) = 1$, g_2 must be a parameter path value of g_1. Therefore, there must be some parameter path value x_1 of $min(r)$ such that $x_1 \sqsubseteq_I g_2$ and hence a path in $SR(P)$ from $min(r)$ to x_1. Because g_2 is an embedded parameter of g_1, this path must contain the edge $min(r) \xrightarrow{1} Restrict(r, 1)$ and so $min(r) \neq x_1$ or $SR(P)$ is cyclic. Consider g_2 and g_3. Because g_3 is a parameter path value of g_2, there must be some parameter path value x_2 of x_1 such that $x_2 \sqsubseteq_I g_3$. Again, $x_1 \neq x_2$ or $SR(P)$ is cyclic. Furthermore, since there is a path in $SR(P)$ from $min(r)$ to x_1 to x_2, $min(r) \neq x_1 \neq x_2$ or $SR(P)$ is cyclic. This holds for all $n > 0$, and hence $x_1 \neq x_2 \neq \ldots \neq x_n$ for all $n > 0$ and $SR(P)$ is not finite, a contradiction. □

Theorem 5.2. *If $\langle P, \sqsubseteq_P, arity, a, Restrict\rangle$ is finite, then $I(P)$ is finite iff $SR(P)$ is acyclic.*

Proof. Notice that $SR(P) = SR(min(\bot))$ and $I(P) = A(\bot)$. The theorem follows by lemmas 5.2 and 5.3. □

If $I(P)$ is finite, we classify $\langle P, \sqsubseteq_P, arity, a, Restrict\rangle$ as *finitely parametrically well-formed.*

Definition 5.1.20. Parametric type hierarchy $\langle P, \sqsubseteq_P, arity, a, Restrict\rangle$ is *finitely parametrically well-formed* iff it is parametrically well-formed and $I(P)$ is finite.

5.1.7 Parametric appropriateness

Currently we have only considered parametric type hierarchies, but, to make use of parametric type hierarchies in attributed type signatures, we need methods of introducing appropriateness specifications to parametric types. As noted by [Pen00], appropriateness forms an integral part of a parametric type signature because the scope of the parameter variables can be extended to include it. [Pen00] defines appropriateness for unrestricted parametric type hierarchies, and, because the induced hierarchy of a restricted parametric type hierarchy is a subset of the unrestricted case, we can simply apply the definitions to the restricted case.

Definition 5.1.21. Given a parametric type, p, for all $i > 0$, the i^{th} *parametric projection* is a partial function, $\pi_i : I(P) \to I(P)$ such that for any $g = p(t_1, \ldots t_{arity(p)})$ with $arity(p) \geq i$, $\pi_i(g) = t_i$.

Definition 5.1.22. Given a parametric type hierarchy $\langle P, \sqsubseteq_P, arity, a, Restrict\rangle$, the *ground instance function*, $T : P \to Pow(I(P))$ is defined such that:

$$T(p) = \{p(t_1, \ldots, t_{arity(p)}) \mid p(t_1, \ldots, t_{arity(p)}) \in I(P)\}$$

Definition 5.1.23. A function $f : I(P) \to I(P)$ is *parametrically determined* iff it is:

- a constant function,
- a parametric projection function, or
- a function for which there exist a $p \in P$ and functions $f_1, \ldots, f_{arity(p)}$ such that for all $g \in I(P)$, $f(g) = p(f_1(g), \ldots, f_{arity(p)}(g))$ and $f_1, \ldots, f_{arity(p)}$ are parametrically determined.

Definition 5.1.24. A parametric (type) signature is a parametrically well-formed parametric type hierarchy, $\langle P, \sqsubseteq_P, arity, a_P, Restrict\rangle$, along with a set of features, $Feat_P$, and a partial (parametric) appropriateness specification, $Approp_P : Feat_P \times P \to (I(P) \to I(P))$, such that:

- (Parametric Determination) If $Approp_P(\mathtt{F}, p)\downarrow$, then $Approp_P(\mathtt{F}, p)\downarrow$ is a parametrically determined total function from $T(p)$ to $I(P)$,
- (Feature Introduction) For every feature $\mathtt{F} \in Feat_P$, there is a most general parametric type $Intro(\mathtt{F}) \in P$ such that $Approp_P(\mathtt{F}, Intro(\mathtt{F}))\downarrow$, and
- (Parametric Upward Closure / Parametric Right Monotonicity) For any $p, q \in P$, any $\mathtt{F} \in Feat_P$, any $g_1 \in T(p)$, and any $g_2 \in T(p)$, if $Approp_P(\mathtt{F}, p)\downarrow$ and $p \sqsubseteq_P q$, then:

– $Approp_P(\mathtt{F}, q)\downarrow$, and

– if $g_1 \sqsubseteq_I g_2$, then $Approp_P(\mathtt{F}, p)(g_1) \sqsubseteq_I Approp_P(\mathtt{F}, p)(g_2)$.

The parametric appropriateness function, $Approp_P$, maps a feature and parametric type p to a function that defines appropriateness conditions on $T(p)$. Parametric determination defines what sort of sharing exists between parameters and appropriateness. The first kind of parametric determination states that parameters need not be shared with any appropriate value restriction, such as `ne_list(sign) intro hd:index`. The second states that parameters with unspecified values can be shared with an appropriate value restriction, such as `ne_list(X) intro hd:X`. The third kind of parametric determination states that a parameter can be shared with a parameter of an appropriate value restriction, such as `ne_list(X) intro tl:list(X)`.

Definition 5.1.25. The *induced appropriateness function*, $Approp_{I(P)} : Feat_P \times I(P) \rightarrow I(P)$ is a partial function defined such that, for every feature $\mathtt{F} \in Feat_P$, every ground instance $g = p(t_1, \ldots t_{arity(p)}) \in I(P)$, $Approp_{I(P)}(\mathtt{F}, g)\downarrow$ iff $Approp_P(\mathtt{F}, p)\downarrow$, and, when defined, $Approp_{I(P)}(\mathtt{F}, g) = Approp_P(\mathtt{F}, p)(g)$.

Proposition 5.1.3. If $\langle P, \sqsubseteq_P, arity, a_P, Restrict\rangle$ is a parametric type signature, then $Approp_{I(P)}$ is an appropriateness specification.

5.2 Constrained parametric unfolding

Restrictions define the domain of a parametric type, but there are many cases in which the image still contains types which should not exist in the grammar. Consider, for example, Sag's [Sag97] classification of English relative clauses (figure 3.2). Sag classifies phrases by two dimension types, CLAUSALITY and HEADEDNESS, furnishing CLAUSALITY with immediate subtypes *clause* and *non-clause*, and HEADEDNESS with immediate subtypes *hd-ph* (headed phrase) and *non-hd-ph* (non-headed phrase). All other phrasal types are classified by clausality and headedness by subtyping either *clause* or *non-clause* and either *hd-ph* or *non-hd-ph*.

It is easy to see that Sag is parameterizing phrases by two Boolean parameters: a parameter for clausality, and a parameter for headedness. It therefore seems reasonable to re-analyze Sag's classification with a parametric type *phrase(bool,bool)* whose first parameter (referring to clausality) is restricted to some Boolean type *bool*, and whose second parameter (referring to headedness) is also restricted to *bool*. Ground instances of *phrase* would include *phrase(bool,bool)* (*phrase*), *phrase(plus,bool)* (*clause*), *phrase(minus,bool)* (*non-clause*), *phrase(bool,plus)* (*hd-ph*), and *phrase(bool,minus)*, (*non-hd-ph*). There would also be also four "cross-classified" ground instances of *phrase*, *phrase(plus,plus)* (headed clauses), *phrase(plus,minus)* (non-headed clauses), *phrase(minus, plus)* (headed non-clauses), and *phrase(minus,minus)*, (non-headed non-clauses).

However, referring again to Sag's classification, we find Sag's analysis posits no joins between classifiers *non-clause* and *hd-ph*, *non-clause* and *non-hd-ph*, or *clause* and *non-hd-ph* – the analysis contains no non-headed clauses, headed non-clauses, or non-headed non-clauses. The absence of such joins is of course important to the analysis, as the absence of joins is critical for eliminating ungrammatical constructions. Should we wish to re-analyze Sag's phrases with a parametric type, we cannot posit ground instances *phrase(plus,minus)*, *phrase(minus,plus)*, and *phrase(minus,minus)*.

Unfortunately, the current formalization of parametric types insists all joins between appropriate parameter values must necessarily exist in $I(P)$ – there are no formal method for eliminating types such as *phrase(plus,minus)*, *phrase(minus,plus)*, and *phrase(minus,minus)*. Fortunately, such types may be eliminated via *closure specifications*. Effectively, closure specifications constrain the induction of $I(P)$ by determining that a certain set of ground instances (a *generator set*) must exist in the induced hierarchy and by closing this set of ground instances under some set of conditions on $I(P)$.

The closure specifications formalized in this section extend the notion of sub-algebras and generator sets mentioned in [Pen00] by associating generator sets and structural closures with each parametric type.

Definition 5.2.1. The *join closure* of p generated by $G \subseteq T(p)$, $J(p,G) \subseteq T(p)$, is the smallest subset of $T(p)$ such that:

- $G \subseteq J(p,G)$, and
- if $g_1 \in J(p,G)$, $g_2 \in J(p,G)$, $g_1 \sqcup_I g_2 \downarrow$, and $g_1 \sqcup_I g_2 \in T(p)$, then $g_1 \sqcup_I g_2 \in J(p,G)$.

$J(p,G)$ is only defined if $G \subseteq T(p)$ and contains the minimal set of ground instances of $T(p)$ required so that all joins between elements of G in $T(p)$ appear in $J(p,G)$. We can similarly enforce closure under meets, supertypes, subtypes, enforce meet-semilatticehood, or perform no structural closures.

Definition 5.2.2. The *meet closure* of p generated by $G \subseteq T(p)$, $M(p,G) \subseteq T(p)$, is the smallest subset of $T(p)$ such that:

- $G \subseteq M(p,G)$, and
- if $g_1 \in M(p,G)$, $g_2 \in M(p,G)$, $g_1 \sqcap_I g_2 \downarrow$, and $g_1 \sqcap_I g_2 \in T(p)$, then $g_1 \sqcap_I g_2 \in M(p,G)$.

Definition 5.2.3. The *supertype (subtype) closure* of p generated by $G \subseteq T(p)$, $S(p,G) \subseteq T(p)$, is the smallest subset of $T(p)$ such that:

- $G \subseteq S(p,G)$, and
- if $g_1 \in S(p,G)$, $g_2 \sqsubseteq_I g_1$ ($g_1 \sqsubseteq_I g_2$), and $g_2 \in T(p)$, then $g_2 \in S(p,G)$.

Definition 5.2.4. The *Dedekind-MacNeille closure* of p generated by $G \subseteq T(p)$, $DM(p,G) \subseteq T(p)$, is the smallest subset of $T(p)$ such that:

- $G \subseteq DM(p,G)$, and

- $DM(p,G)$ is a meet-semilattice.

Definition 5.2.5. The *trivial closure* of p generated by $G \subseteq T(p)$, $TC(p,G) \subseteq T(p)$, is G.

Definition 5.2.6. The *unconstrained closure* of p is $T(p)$.

Given a set of closure operations and a number of generator sets, we can create parametric signature closures by associating a closure operation (and, if necessary, a generator set) with each parametric type. Note that because our closures restrict themselves to the set of ground instances $T(p)$ of a single parametric type p, the parametric signature closures we compute will simply be the union of individual type closures. Hence it is possible to encounter signatures in which there are parametric types p, q, and r such that $p(a) \sqcup_I q(a) = r(a)$, $p(a) \in J(p,G_p)$, $q(a) \in J(q,G_q)$, but $r(a) \notin J(r,G_r)$ and so, even though p, q, and r all underwent join closure, $p(a) \sqcup_I q(a) \uparrow$ does not exist in the closed induced signature. One could imagine an alternative formalization in which closures are associated with sets of parametric types, or, perhaps, sets of ground instances, but we have found the current formalization is sufficient for parameterizing the English Resource Grammar (§7).

Definition 5.2.7. Given a parametric type hierarchy $\langle P, \sqsubseteq_P, arity, a, Restrict\rangle$, a *parametric closure specification* is a function $Close : P \rightarrow ClosureSpec$ such that either $Close(p) = \langle A, G\rangle$ where $G \subseteq T(p)$ and $A \in \{$ trivial, dedekind-macneille, supertype, subtype, join, meet $\}$ or $Close(p) =$ unconstrained.

The combination of a closure specification $Close$ and a parametric type hierarchy $\langle P, \sqsubseteq_P, arity, a, Restrict\rangle$ allows the closure of an induced type hierarchy to be computed from the closure of each of its parametric types (figure 5.8). The only concern is, because ground instances of parametric types are parameter values, care must be taken to ensure each parametric type is closed before any other parametric type in which its ground

1. Construct $PSR(P)$, the parametric subtype-restriction graph of P.
2. Let V = inverse of topologically sorted vertices of $PSR(P)$.
3. For $i = 1$ to $|V|$ do
   - compute $T_I(V(i))$ = the closure of $V(i)$
4. Let $C(P) = \bigcup_{0 \leq i \leq |V|)} T_I(V(i))$ closed under $\sqsubseteq_I$.
5. Return $C(P)$.

Figure 5.8: Inducing $I(P)$ with closure specifications

instances appear as parameter path values. Luckily, a variant of the subtype-restriction graph allows us to easily calculate the proper closure order.

Definition 5.2.8. Given ground instance type g, let $\mathcal{V}^+(g) = \mathcal{V}^*(g) - g$, the set of *embedded* parameter path values of g.

Definition 5.2.9. Given a parametric type hierarchy $\langle P, \sqsubseteq_P, arity, a, Restrict\rangle$, the *parametric subtype-restriction graph* of P, $PSR(P)$, is a labelled directed graph $\langle P, \{\langle t_1, t_2, \mathtt{P}\rangle \mid t_2(u_1, \ldots, u_{arity(t_2)}) \in \mathcal{V}^+(min(t_1))\} \cup \{\langle t_1, t_2, \mathtt{S}\rangle \mid t_1 \sqsubset_P t_2\}\rangle$, whose vertices are the parametric types of P and whose edges consist of edges with integer labels that map from parametric types to parametric types of restrictions plus edges with label $\mathtt{S}$ that close $PSR(P)$ under $\sqsubset_P$.

Proposition 5.2.1. If P is finite and *PSR(P)* is acyclic, then the closure algorithm ensures every parametric type q is closed before any parametric type p allowing a ground instance of q as a parameter path value.

Proof. The proposition holds if for all $p, q \in P$ where p allows ground instances of q as a parameter path value, $PSR(P)$ contains a path from p to q. Now, if p allows ground

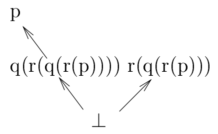

Figure 5.9: Co-referent restrictions

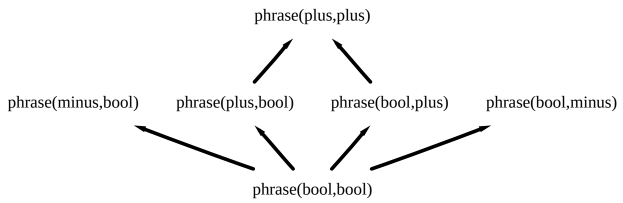

Figure 5.10: A constrained parametric unfolding of parametric type *phrase*

instances of q as a parameter path value, then either a ground instance of q is a parameter path value of $min(p)$, or a ground instance of q is a subtype of a parameter path value $r(t_1, \ldots, t_{arity(r)})$ of $min(p)$. In the first case there must be an edge $p \xrightarrow{\mathtt{P}} q$ in $PSR(P)$. In the second case, $PSR(P)$ must contain an edge $p \xrightarrow{\mathtt{P}} r$ and an edge $r \xrightarrow{\mathtt{S}} q$, which means there is a path in $PSR(P)$ from p to q. □

Calculating closure order from $SR(P)$, if it is possible, is more difficult because parametric types may take restrictions that are ground instances of each other while still inducing finite non-parametric hierarchies (figure 5.9), and, in this case, it is unclear how to order the closure operations. $PSR(P)$ is constructed so that such situations result in a cyclic parametric subtype-restriction graph and we have simply shown that closure order is well-defined for hierarchies with acyclic $PSR(P)$. Ordering closures for hierarchies such as those in figure 5.9 remains an open question.

As an quick example of closure specifications in operation, consider again our parametric type *phrase*. Supposing the translation from ground instances of *phrase* to [Sag97] types sketched at the beginning of this section, closure specification $Close(phrase) = \langle \text{meet}, G \rangle$, and generator set $G = \{phrase(bool, bool), phrase(plus, bool), phrase(minus, bool), phrase(bool, plus), phrase(bool, minus), phrase(plus, plus)\}$, the closure of *phrase* (meet closure) is the hierarchy in figure 5.10.

Chapter 6

HOPS - higher-order parametric signature descriptions

HOPS (Higher-Order Parametric Signatures) is a pattern-inspired grammar specification language we have developed based on the patterns mined in §4 and employing the parametric typing formalized in §5. Implemented on top of Prolog and intended as an extension to ALE [CP96] (which is also implemented on top of Prolog), HOPS extends ALE syntax by adding additional operators for the specification of higher-order types and closure specifications. We have implemented a HOPS interpreter which is capable of compiling HOPS specifications into valid ALE programs; HOPS can thus be understood as an add-on or plug-in to ALE. Currently, HOPS assumes finite parametrically well-founded type hierarchies and supports the specification of parametric type hierarchies plus any number of extra subtyping links between ground instances. Because extra ground instance subtyping links and certain closure specifications can break the meet-semilattice status of induced hierarchies, HOPS also provides an algorithm for restoring partial orders to meet-semilattices.

6.1 Simple type signatures

HOPS programs with no higher-order types are identical to ALE programs. For this reason, only a very brief introduction to simple type signatures will be sketched here - the interested reader is directed to the ALE user's manual [CP01].

6.1.1 Inheritance hierarchies

Inheritance hierarchies allow multiple inheritance and are declared using the `sub` (subsumes) operator. Each type must be a prolog atom. List notation is optional if the `sub` declaration includes only one RHS type.

```
bot     sub [list, atom].
list    sub [e_list, ne_list].
ne_list sub '1_list'.
```

6.1.2 Appropriateness

Appropriateness specifications are declared using the `intro` (feature introduction) operator. Each LHS must be a type and each RHS must be a list of feature:type pairs. Each feature must also be a prolog atom.

```
ne_list intro [hd:atom,tl:list].
```

6.1.3 Type-antecedent constraints

Type antecedent constraints are specified using the `cons` (constraint) operator. Each LHS must be a type, and each RHS must be a description.[1]

```
'1_list' cons tl:e_list,hd:list.
```

[1]See [CP01] for more a complete discussion of ALE descriptions.

6.2 Higher-order type constructors

HOPS augments ALE with several kinds of higher-order type constructors. First and foremost are parametric products, which represent the parametric types described in §5 and implement the dimensional classification pattern discussed in §4.3.4. Other higher-order constructors encapsulate the remaining structural patterns identified in §4.

6.2.1 Parametric products

Parametric product types are declared using the `param` operator, which takes a parametric type name on the LHS and a list of parameters on the RHS. HOPS supports both positionally-indexed and named parameters - in the case of named parameters, each parameter must be provided with a name (which must be a prolog atom), and a restriction (which must be a ground instance type). For positionally indexed parameters, only the restriction is required. HOPS automatically assigns the names `1`, `2`, `3`, etc. to positionally indexed parameters. Named parameters are provided as an alternative to positionally indexed parameters in order to furnish parameters with conceptual labels and to enable resolving of ground instances without explicitly stating parameter values which are equal to the parameter restriction.

```
bool   sub   [plus,minus].
per    sub   [per1,per2,per3].
num    sub   [sg,pl].
gen    sub   [masc,fem].
phrase param [finite:bool,clausal:bool].
agr    param [per,num,gen].
```

Ground instances of parametric product types are expressed as logical terms in which the parametric type name is the functor and parameter values are name:value arguments. For example, some ground instances of *phrase* are *phrase(finite:minus,clausal:bool)* and

phrase(finite:plus,clausal:plus). Named parameter order is unimportant: HOPS considers *phrase(finite:minus,clausal:bool)* to be equivalent to *phrase(clausal:bool,finite:minus)*. Additionally, as previously mentioned, named parameters with values equal to the restriction are optional: *phrase(finite:minus)* is the same ground instance as *phrase(finite:minus, clausal:bool)*. Because *agr* has positionally indexed parameters, its ground instances can be referenced with or without parameter names: *agr(per1,sg,fem)* and *agr(1:per1,2:sg,3:fem)* are the same ground instance. Parameters with values equal to the restriction are still optional if parameter names are used: *agr(1:per1)* is the same ground instance as *agr(1:per1,2:num,3:gen)* and *agr(per1,num,gen)*.

6.2.2 Finite domains

Finite domain types are an implementation of the disjunctive types pattern (§4.3.1). Algebraically, a finite domain is a powerset of a finite set formed by enumerating all combinations of a discretely ordered set and ordering the combinations by set-inclusion. In HOPS, finite domains are declared using the `findom` operator which takes as argument a list of discretely ordered types.

```
ftense findom [past,present,future].
```

Ground instances of finite domains are expressed using the finite domain type name (in this case, *ftense*) as functor and domain types as arguments separated by the standard Prolog disjunctive operator (`;`). For example, *ftense(past;present;future)*, *ftense(past;future)*, and *ftense(past)* are ground instances of *ftense*. Additionally, "singleton" ground instances of finite domains subsume their corresponding argument type (in this case *ftense(future)* $\sqsubseteq_I$ *future*, *ftense(past)* $\sqsubseteq_I$ *past*, and *ftense(present)* $\sqsubseteq_I$ *present*).

6.2.3 Smyth powerlattices

Smyth powerlattices are an implementation of the conjunctive types pattern (§4.3.2). Algebraically, the smyth powerlattice of a partial order is the set of all sets of mutually incomparable elements, ordered by inverse set-inclusion. In HOPS, smyth lattices are formed using the `smyth` operator which takes as arguments a type t and an optional integer argument k. The image of `smyth` is the smyth powerlattice of the set of all ground instances of t and ground instances of t's subtypes minus those powerlattice members with cardinality $> k$. If the argument type is a finite domain type, finite domain "pure" types are also included in the argument poset.

```
tense sub   [past,present,future].
sm    smyth tense.
sm2   smyth tense,2.
```

Ground instances of smyth powerlattices are expressed using the smyth type name (in this case, *sm* or *sm2*) as functor, with domain types appearing as arguments separated by the standard Prolog conjunctive operator (`,`). For example, three ground instances of *sm* are *sm(past,present,future)*, *sm(past,present)*, and *sm(tense)*. Type *sm2(past,present,future)* represents a set with cardinality 3 and therefore does not exist. "Singleton" ground instances of smyth powerlattices are subsumed by their corresponding argument types (in *sm*, for example, *future* $\sqsubseteq_I$ *sm(future)*, *past* $\sqsubseteq_I$ *sm(past)*, and *present* $\sqsubseteq_I$ *sm(present)*).

6.2.4 Strict variants

Strict variant types implement the strict variant pattern (§4.3.3) and are declared using the `strictx` operator which takes as argument a type t. The image of a strict variant is a copy of the set of all ground instances of t and ground instances of t's subtypes wherein which each "copy" type is subsumed by its original type. Similar to `smyth`, if

the argument type is a finite domain type, finite domain "pure" types are also included in the argument poset.

```
tense sub    [past,present,future].
st    strictx tense.
```

Ground instances of strict variant types are expressed using the strict variant type name (in this case, *st*), with domain types appearing as arguments. In our example, *st(present)*, *st(past)*, and *st(tense)* are ground instances of *st*. Type *future* $\sqsubseteq_I$ *st(future)*, *past* $\sqsubseteq_I$ *st(past)*, and *present* $\sqsubseteq_I$ *st(present)*.

6.2.5 Synonyms

Synonyms may be declared for any type and may be used at any time in place of their referent. HOPS implements synonyms in a "call-by-macro" style, meaning whenever HOPS sees a synonym, it simply replaces the synonym with its referent. Synonyms can be used as higher-order objects in their own right; if a is a synonym of b and b has parameters, then a can take the same parameters as b.

```
phrase     param [finite:bool,clausal:bool].
fin_phr    syn   phrase(finite:plus).
fin_clause syn   phrase(finite:plus,clausal:plus).
```

In this example, *fin_clause* is a synonym for the ground instance *phrase(finite:plus, clausal:plus)*. *fin_phr* is a higher-order synonym and can take parameters:
fin_phr(clausal:minus) is a synonym for the ground instance *phrase(finite:plus,clausal:minus)*, and *fin_phr(clausal:bool)* is a synonym for the ground instance *phrase(finite:plus,clausal:bool)*.

6.3 Higher-order inheritance hierarchies

The `sub` operator is used for subtyping of both higher-order types and simple types, and subtyping declarations may be made both between types that are explicitly declared to be parametric and between those that are not defined in a higher-order typing declaration. The semantics of higher-order subtyping declarations will be described by example, but the rules of thumb are these:

1. Any kind of higher-order type may participate in subtyping declarations.
2. Only parametric product types can be defined in subtyping declarations, and only if they it appear on the RHS of some `sub` declaration.
3. If a type t is defined with a `param` declaration, its parameters and parameter restrictions are not affected by any subtyping declaration.
4. Only parametric product types may share parameters.
5. If the LHS type is a ground instance of a higher-order type, all RHS types are simple types.

The example subtyping declarations below will assume the following type declarations:

```
bool sub    [plus,minus].
f    findom [i,j,k].
sm   smyth  bool.
a    param  [x:bool,y:bool].
b    param  [x:bool,z:bool].
c    param  [t:bool,z:f(i;j)].
```

`a sub b.` Because `a` and `b` are defined with `param` declarations, `a` is a parametric product type with parameters `[x:bool,y:bool]` and `b` is a parametric product subtype of `a` with parameters `[x:bool,z:bool]`. Parameter `x` is shared.

`a sub sm.` Types `a` and `sm` are defined with higher-order constructors. Since `sm` is not a parametric product type, `a` and `sm` do not share parameters and all ground instances of `a` subsume all ground instances of `sm`.

`a(x,y) sub d.` Because `d` is not defined by a higher-order constructor, `d` is a parametric product type whose parameters must be inferred from `a`. Because `a` shows parameters `x` and `y` on the LHS which do not appear on the RHS, it is assumed `x` and `y` are dropped and `d` is a simple type.

`a sub e.` Because `e` is not defined by a higher-order constructor, `e` is a parametric product type whose parameters must be inferred. Because `a` does not show any parameters that do not appear on `e`, it is assumed that all parameters of `a` are parameters of `e`: `e` has parameters `[x:bool,y:bool]` and parameters `x` and `y` are shared.

`a(x) sub g(w:bool).` Because `g` is not defined by a higher-order constructor, `g` is a parametric product type whose parameters must be inferred. Parameter `x` appears only on the LHS and is therefore dropped. Parameter `w` is not a parameter of `a` and so is a new parameter of `g`. Therefore, `g` is a parametric subtype of `a` with parameters `[y:bool,w:bool]` and parameter `y` is shared.

`a sub h(x:plus).` Because `h` is not defined by a higher-order constructor, `h` is a parametric product type whose parameters must be inferred. Parameter `x` appears on the RHS only, but because `x` is also defined as a parameter of `a`, it is shared. Parametric type `h`, however, restricts `x` to `plus` so that the parameters of `h` are `[x:plus,y:bool]`. Parameter `y` is also shared.

`s sub c.` Type `s` is not defined by a higher-order constructor, but appears on the LHS and is therefore simple. `c` is defined by a `param` declaration and so is a parametric subtype of `s` with parameters `[t:bool,z:f(i;j)]`.

`a(x:bool,y:plus) sub h.` Because `a` is defined by a higher-order constructor and shows values for some of its parameters, `a(x:bool,y:plus)` is taken to be a ground instance of `a` and so `h` is a simple type and `a(x:bool,y:plus) sub h` is a subtyping link added to $I(P)$.

6.4 Constraining induction

Finally, HOPS provides methods for constraining induced hierarchies by way of the parametric closure specifications defined in §5.2.

6.4.1 Generator sets

Generator sets are declared using the `gen` operator, which takes a generator set identifier on the LHS and a list of generator set types on the RHS. The elements of the RHS list may be either ground instances of parametric types, the bracketed name of a file containing a list of ground instances, or the special keyword `{grammar}` which implicitly contains all ground instances attested in subtyping declarations, feature introductions, constraints, grammar rules, lexical rules, and lexical entries. There is a special reserved generator set `grammar` which is defined as `grammar gen [{grammar}].`

```
ph_gen   gen [phrase(arity:binary,head:plus), phrase(arity:unary)].
word_gen gen [['word_gen_file'], word(affixed:plus)].
head_gen gen [{grammar}, subst(noun,verb)].
```

6.4.2 Closure specifications

There are currently three kinds of closure implemented: trivial, supertype and unconstrained, all of which are declared using the `close` operator. The `close` operator takes a type on the LHS and on the RHS a closure specification of the form `trivial(generator set)`, `supertype(generator set)` or `unconstrained`. Closure specifications may appear anywhere in a grammar specification, but are applied after all type specifications have been processed. Types without closure specifications are assumed to be unconstrained, and the order of closure operations follows the algorithm in §5.2. Due to internal processing concerns, HOPS-generated induced hierarchies include the minimal ground instance of every higher-order type. Contrary to §5.2, closures may be generated by generator sets containing types which are not ground instances of the type undergoing the closure. In such cases, HOPS simply filters out the erroneous types.


```
phrase close trivial(grammar).
word   close supertype(word_gen).
tense  close unconstrained.
```

Chapter 7

ICEBERG - a higher-order English Resource Grammar

This chapter presents ICEBERG, a re-factoring of the English Resource Grammar (ERG) written in HOPS, which has replaced a significant portion of the ERG type hierarchy with higher-order types. We approach this presentation from two angles: first, §7.1 describes the topology of the ERG type signature, explores issues we have encountered in constructing a broad coverage grammar using parametric types, and describes our re-factoring of several indicative portions of the grammar; then, §7.2 compares ICEBERG to the ERG both extensionally and qualitatively. Several quantitative measures of grammar usability are reported and some engineering advantages of ICEBERG are discussed. This section further shows that ICEBERG and the ERG achieve equivalent analyses of a large development corpus (∼2000 sentences), suggesting that ICEBERG and ERG are extensionally equivalent.

7.1 Building a higher-order ERG

Broad-coverage HPSG grammars are generally large pieces of software, and the ERG is no exception. The version used in this study, an ALE-compatible port of a May 2000 release

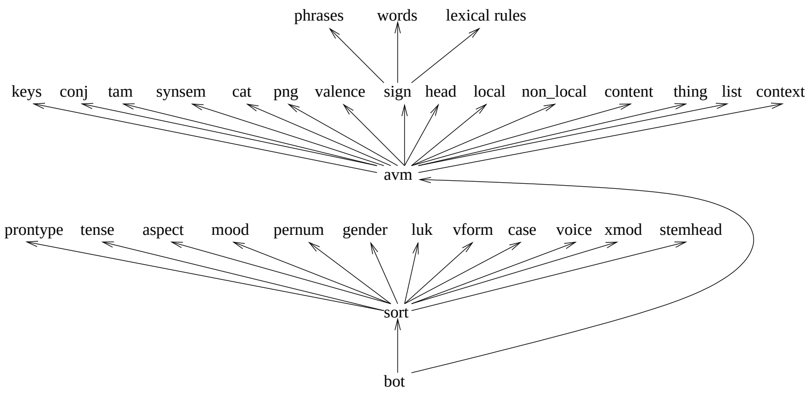

Figure 7.1: ERG IS-A topology

generated from the FUSE corpus, contains 4305 types (916 of which are GLB completion types), 155 feature introductions, 848 type-antecedent constraints, 45 grammar rules, and a 10623 word lexicon.

The topology of the ERG IS-A network (type hierarchy) is described pictorally in figure 7.1. The most general ERG type is *bot*, and types **sort** and **avm** partition the hierarchy into featureless types and feature-bearing types, respectively.[1] Each subtype of **sort** and **avm** in figure 7.1 represents a most-general type which subsumes a filter of subtypes, all of which are incomparable with types in any other filter. Figure 7.2 sketches the HAS-A (appropriateness) relation among these filters - there is an edge A-B in figure 7.2 iff A "has a" B, that is, some type in the filter subsumed by B is appropriate to some feature of some type in the filter subsumed by A. Throughout this chapter we will follow figures 7.1 and 7.2 in referencing ERG filters by their most general type. For example,

[1] **sort** is a misnomer - subtypes of **sort** are not mutually incomparable as one might expect given the standard interpretation of the term sort. Types **sort** and **avm** are legacy types imported from the PAGE system, which had, in fact, assumed sorts were incomparable. PAGE used **sort** and **avm** to direct the parser to separate handling routines for type operations and feature structure operations. More information can be found in the PAGE user manuals ([KS94b, KS94c]).

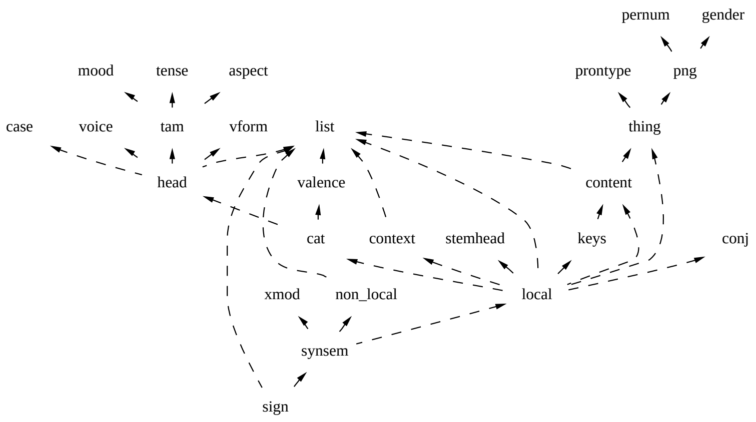

Figure 7.2: ERG HAS-A topology

when we say 'luk types' we will be referring to type *luk* and all its subtypes.

We have limited our parametric re-implementation to the type hierarchy (as opposed to appropriateness, constraints, and the grammar rules), producing a parameterized version of the ERG IS-A network. Specifically, we have produced higher-order versions of ERG three-valued boolean (*luk*), person-number (*pernum*), *gender*, *tense*, *aspect*, *mood*, external modifier (*xmod*), *stemhead*, *voice*, *verb form*, semantic index (*thing*), *head*, *category*, *local*, *word*, *phrase*, and *list* subtypes. Because *case*, pronoun type (*prontype*), *keys*, tense-aspect-mood (*tam*), person-number-gender (*png*), *valence*, *non-local*, *context*, and *lexical rule* types contain little if any parametricity, they have been left unchanged. Due to time constraints, *synsem* types and *content* have not been re-implemented, although they undoubtedly exhibit parametricity. Again due to time constraints, we have not parameterized the ERG HAS-A relation, although there are certainly many ERG constraints which could be intuitively captured by parametric appropriateness specifications.

The following sections will highlight the ERG's parametricity by way of examples

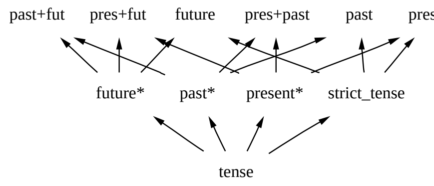

Figure 7.3: ERG tense types

from ICEBERG. These examples will be indicative of standard techniques we have employed throughout ICEBERG - strict variants and smyth powerlattices of finite domain types, parametric products for multidimensional classification, and the trivial closure over generator sets composed of grammar-attested ground instances. Not every instance of re-factoring will be described here, however–the interested reader is referred to the appendix for a comprehensive tour of ICEBERG.

7.1.1 Finite domains, smyth powerlattices, and strict variants

A number of ERG type filters contain disjunctive types, conjunctive types, and strict variants as in figure 7.3 (tense types) and figure 7.5 (luk types). In such filters, one generally finds a set of basic conceptual types (in the case of tense, *past*, *present*, and *future*; in the case of luk, $+$, $-$, and *not-applicable*), and these conceptual types appear as the components of disjunctive types, conjunctive types, and strict variant types.

A standard example is ERG tense, which contains strict variants of *past*, *present*, *future*, and *tense*, as well as conjunctive types of each pair of *past*, *present*, and *future* (which subsume the non-strict versions of their component types). To re-factor tense types, ICEBERG posits a type *tense* with subtypes *past*, *present*, and *future*. Higher-order tense is constructed with a smyth powerlattice of *tense* and a strict variant of *tense*.

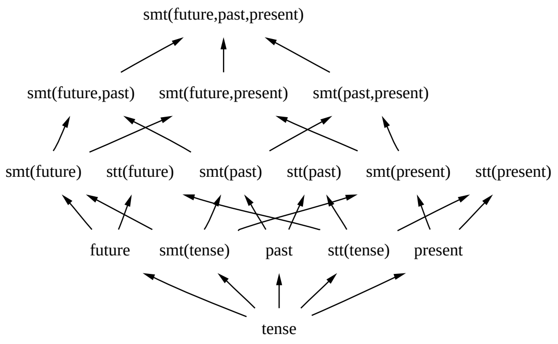

Figure 7.4: ICEBERG tense types

```
tense  sub      [future, past, present].
smt    smyth    tense.
stt    strictx  tense.
```

The induced non-parametric hierarchy of tense types is depicted in figure 7.4 and the mapping between ERG and ICEBERG names is recorded in table 7.1. ICEBERG contains four types not present in the ERG: *smt(past)*, *smt(present)*, and *smt(future)* (corresponding to singleton smyth powerlattice elements), and *smt(future,past,present)* (corresponding to a conjunction of all three conceptual types). For the corpus used in our experiments, these extra types admit no parses not admitted by the ERG.

A slightly more complicated example involves *luk* (three-valued boolean) types, which include disjunctive types in addition to both conjunctive types and strict variants (figure 7.5). To re-factor luk types, ICEBERG constructs a finite domain, *fl* over +, -, and *na*, a smyth powerlattice of *fl*, and a strict variant of *fl*.

```
fl       findom   ['+', '-', na].
sml      smyth    fl.
stl      strictx  fl.
luk      sub      fl.
```


ERG name	ERG attested	ICEBERG name
tense	✓	tense
past*	✓	past
present*	✓	present
future*	✓	future
strict_tense	✓	stt(tense)
past	✓	stt(past)
present	✓	stt(present)
future	✓	stt(future)
		smt(tense)
		smt(future)
		smt(past)
		smt(present)
past+fut	✓	smt(future,past)
pres+fut	✓	smt(future,present)
pres+past	✓	smt(past,present)
		smt(future,past,present)

Table 7.1: Tense translation table

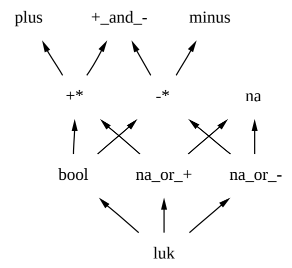

Figure 7.5: ERG luk types

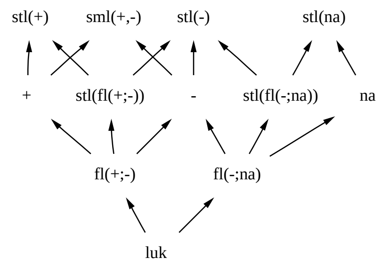

Figure 7.6: ICEBERG luk types

```
fl        close    trivial(luk_set).
sml       close    trivial(luk_set).
stl       close    unconstrained.
luk_set   gen      [{grammar}, sml('+','-')].
```

Types *fl* and *sml* are constrained by the trivial closure over a generator set composed of ground instances attested by the ERG outside the type hierarchy (ground instance types found in feature introductions, constraints, lexical rules, grammar rules, and the lexicon), and the closure of *stl* is unconstrained. Because ground instance type *sml(+,-)* is not attested outside the hierarchy, it was added manually to the generator set.

ERG name	ERG attested	ICEBERG name
luk	✓	luk
		fl(+;-;na)
bool	✓	fl(+;-)
na_or_+		fl(+;na)
na_or_-	✓	fl(-;na)
	✓	na
+*	✓	+
-*	✓	-
		stl(fl(+;-;na))
		stl(fl(-;na))
		stl(fl(+;-))
na		stl(na)
plus	✓	stl(+)
minus	✓	stl(-)
+_and_-		sml(+,-)

Table 7.2: Luk translation table

The induced luk hierarchy is depicted in figure 7.6 and the translation table in table 7.2. The induced hierarchy is equivalent to ERG *luk* except for the absence of disjunctive type *na_or_+* (*fl(+;na)*), and the presence of several strict variant types. It turns out that the ERG does not attest *na_or_+* outside the type hierarchy (no feature requires *na_or_+* as value), and since *na_or_+* is join-irreducible, it is extensionally redundant.[2] Each of the extra types is also join-irreducible and therefore make no extensional difference.

Manually adding conjunctive types to grammar-attested generator sets turns out to

[2]A short proof of this claim follows from the definition of join-irreducibility: type z is join-irreducible iff for all x,y such that $x \sqcup y = z$, $z = x$ or $z = y$. Hence since *na_or_+* is join-irreducible and is not attested by the grammar, it cannot be the join of grammar-attested types.

be ICEBERG's standard technique for closing types such as *luk* since the ERG attests very few conjunctive types in places other than the type hierarchy. This makes intuitive sense. Conjunctive types are intended to capture phenomena such as coordination and case neutralization–constructions built by the unification of feature structures labeled with non-conjunctive types. As such, conjunctive types tend not to appear as feature values in the grammar itself, and, there is therefore no way to infer from the grammar which conjunctive types should be present. In most cases (as in *luk*), we have manually added ERG conjunctive types to the generator set on the assumption that the ERG has good linguistic reasons for the existence of certain conjunctive types but not others. The only other alternative would mean including all conjunctive types in the induced hierarchy (as in *tense*), but, of course, including all conjunctive types will license sentences the ERG currently considers ungrammatical.

In addition to *tense* and *luk*, the combination of finite domains, smyth powerlattices, and strict variants is used in the re-implementation of *pernum*, *gender*, *aspect*, *mood*, *xmod*, *voice*, *verb form*, and *head* subtypes.

7.1.2 Parametric products

The most oft-stated reason for employing multiple inheritance in typed feature structure grammars is to facilitate multi-dimensional classification, and, as already noted in §4.3.4, the ERG makes use of this technique frequently. In ICEBERG, this classification is captured by parametric product types.

A standard example of multi-dimensional classification is HPSG phrase types, which were multi-dimensionally classified in [Sag97]'s analysis of English relative clauses. Continuing in that spirit, this section describes a higher-order re-factoring of a portion of the ERG phrases using parametric products. We will only give detailed descriptions of several basic phrase types–the entire phrase hierarchy re-implementation is presented in §A.7.

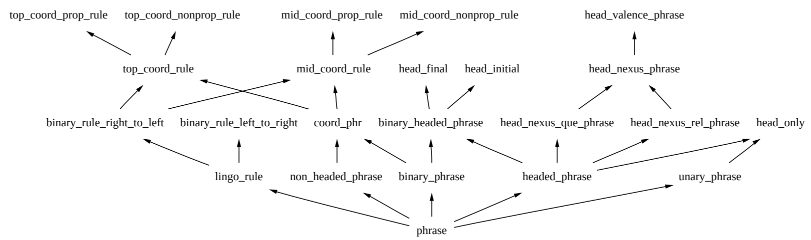

Figure 7.7: ERG phrase types (portion)

Figure 7.7 depicts basic ERG phrases, head-nexus phrases,[3] and coordinate phrases. There are five immediate subtypes of the most general type, *phrase*. By examining the type names (and by noticing they have no common subtypes), we can infer that types *headed_phrase* and *non_headed_phrase* constitute a headedness dimension, and, by the same token, types *binary_phrase* and *unary_phrase* constitute an arity dimension. Type *lingo_rule* is the most general grammar rule type (every ERG grammar rule is a subtype of *phrase*), and has two subtypes, *binary_rule_left_to_right* and *binary_rule_right_to_left* which bifurcate rules by direction from the modifier to the head.

The join of *non_headed_phrase* and *binary_phrase* is a type *coord_phr* which is the most general coordinate phrase. Coordinate phrases and rules are further classified by propositionality and "rule height" (whether the resulting coordinate structure is a member of another coordinate structure). Phrases which are both *binary* and *headed* are further classified by head position (*head_final* and *head_initial*), and *unary headed* phrases are called *head_only*. There is no join for types *unary* and *non_headed*, presumably because English does not contain any unary non-headed phrases.

Type *headed_phrase* has four other subtypes (the nexus phrases) which are type an-

[3]Use of the term *nexus* extends at least to 1924 in Otto Jespersen's *The Philosophy of Grammar* [Jes24], who used the term to describe phrases with subjects and predicates. In the ERG, all such phrases are subsumed by the head-nexus phrases.

tecedents to constraints pertaining to [Sag97]'s Slash-Inheritance Principle (SLIP) and Wh-Inheritance Principle (WHIP). Type *head_nexus_que_phrase* enforces an empty `QUE` list, *head_nexus_rel_phrase* enforces an empty `REL` list, and type *head_valence_phrase* enforces an empty `SLASH` list. *head_nexus_phrase*, the join of *head_nexus_que_phrase* and *head_nexus_rel_phrase*, implements WHIP, and its subtype *head_valence_phrase* implements SLIP.

ICEBERG parameterizes the phrases first by a parametric product type *phrase* with five parameters: headedness, arity, head position, ruleness, and rule direction. Then, we add a parametric subtype *coord_phr* which inherits all parameters from *phrase*, restricts arity and headedness to *binary* and *non-headed*, respectively, and adds parameters for propositionality and coordination height. There is another parametric subtype of phrase *nexus_phr* which also inherits all parameters from *phrase*, restricts headedness to *headed*, and adds three boolean parameters for the WHIP and SLIP principles - *que*, *rel*, and *slash*. The induced hierarchy of all three parametric types is constrained by the trivial closure over grammar-attested ground instances.

```
bool         sub    [plus,minus].
head_pos     sub    [initial, final].
arity_type   sub    [unary, binary].
rule_dir     sub    [r_l, l_r].
coord_height sub    [top, mid].

lexroot      sub    phrase(head:bool,arity:arity_type,hp:head_pos,
                           rule:bool,rdir:rule_dir).

phrase       sub    nexus_phr(head:plus,que:bool,rel:bool,slash:bool).

phrase       sub    coord_phr(head:minus,arity:binary,
                              prop:bool,ht:coord_height).

phrase       close  trivial(grammar).
nexus_phr    close  trivial(grammar).
coord_phr    close  trivial(grammar).
```

Figure 7.8 is the induced hierarchy of ICEBERG parametric phrase types, and table 7.3 is the corresponding translation table. All ERG types are attested outside the type

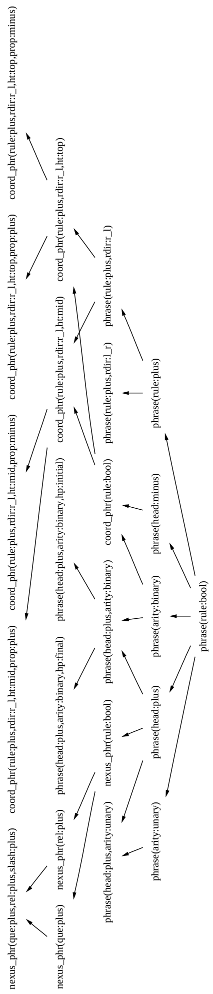

Figure 7.8: ICEBERG phrase types (portion)

ERG name	ERG attested	ICEBERG name
phrase	✓	phrase(rule:bool)
lingo_rule	✓	phrase(rule:plus)
binary_rule_left_to_right	✓	phrase(rule:plus,rdir:l_r)
binary_rule_right_to_left	✓	phrase(rule:plus,rdir:r_l)
headed_phrase	✓	phrase(head:plus)
non_headed_phrase	✓	phrase(head:minus)
binary_phrase	✓	phrase(arity:binary)
unary_phrase	✓	phrase(arity:unary)
binary_headed_phrase	✓	phrase(head:plus,arity:binary)
head_only	✓	phrase(head:plus,arity:unary)
head_initial	✓	phrase(head:plus,arity:binary,hp:initial)
head_final	✓	phrase(head:plus,arity:binary,hp:final)
		nexus_phr(rule:bool)
head_nexus_que_phrase	✓	nexus_phr(que:plus)
head_nexus_rel_phrase	✓	nexus_phr(rel:plus)
head_nexus_phrase		nexus_phr(rel:plus,que:plus)
head_valence_phrase	✓	nexus_phr(rel:plus,que:plus,slash:plus)
coord_phr	✓	coord_phr(rule:bool)
top_coord_rule	✓	coord_phr(rule:plus,rdir:r_l,ht:top)
mid_coord_rule	✓	coord_phr(rule:plus,rdir:r_l,ht:mid)
top_coord_prop_rule	✓	coord_phr(rule:plus,rdir:r_l,prop:plus,ht:top)
top_coord_nonprop_rule	✓	coord_phr(rule:plus,rdir:r_l,prop:minus,ht:top)
mid_coord_prop_rule	✓	coord_phr(rule:plus,rdir:r_l,prop:plus,ht:mid)
mid_coord_nonprop_rule	✓	coord_phr(rule:plus,rdir:r_l,prop:minus,ht:mid)

Table 7.3: Phrase translation table

hierarchy but one - *head_nexus_phrase*. Due to HOPS's convention of including the minimal ground instance of every parametric type, there is one extra type in the induced hierarchy, *nexus_phr(rule:bool)*.[4] Our experiments have shown that the missing type and extra type make no difference to the analyses licensed by the grammar.

With the exception of ERG *thing* types, ICEBERG has used trivial closures over grammar-attested ground instances for all parametric product types, finding that by constraining induced hierarchies to include only those types attested by the rest of the grammar, ICEBERG licenses the same sentences and produces the same analyses as the ERG. Because the ERG's hierarchy of *thing* types includes both conjunctive types and dimensional classification, and because conjunctive *thing* types are not attested by the grammar, ICEBERG's parametric *thing* types are closed by the trivial closure over grammar-attested ground instances augmented with explicitly added conjunctive types.

7.2 Comparing ICEBERG and ERG

This section reports some experiments and measurements undertaken to compare ICEBERG and the ERG at both the source code and extensional levels. §7.2.1 reports experiments parsing the ERG and ICEBERG over two development corpora, showing that the grammars achieve identical analyses for over 2000 sentences with length up to 50 words. §7.2.2 attempts to analyze the grammars qualitatively, looking at measures such as source code length, number of types, and number of type declarations.

[4]By definition, parameter values of minimal ground instances are equal to their parameter restriction and therefore do not need to be displayed (§6.2.1). We choose to always display one parameter value for the sake of avoiding confusion between *nexus_phr* (the minimal ground instance) and *nexus_phr* (the parametric type).

7.2.1 Extensional comparison

In an effort to determine the correctness of ICEBERG, ICEBERG was compared to the ALE-ported ERG by parsing sets of identical sentences. Because ICEBERG only re-factors the type hierarchy, and because of incompatible naming conventions, all ERG feature introductions, type-antecedent constraints, grammar rules, and lexical entries were translated to ICEBERG names (where a corresponding ICEBERG type exists) via a manually encoded translation table. These translated specifications were merged with ICEBERG's induced type hierarchy to form the new grammar (henceforth referred to as simply ICEBERG). The untouched ERG type hierarchy was also translated to ICEBERG names and merged with the rest of the translated grammar to form a translated (but not re-factored!) ERG (henceforth referred to as simply ERG).

ICEBERG and ERG were then each compiled by ALE and instructed to parse two corpora: a small test set of 40 sentences drawn from the CSLI corpus[5], and all 2363 sentences from the FUSE corpus[6]. As a preliminary test, for each sentence we compared the number of spanning edges and chart edges produced by each grammar. Once the edge counts were found to be equivalent, the feature structures associated with each spanning edge were examined. For both corpora, the ERG feature structures and ICEBERG feature structures were identical, leading us to conclude that ICEBERG and ERG are extensionally equivalent.

7.2.2 Usability measures

Because ICEBERG is intended to be a more readable, intuitive, accessible, and extendable (in a word, *usable*) grammar than the ERG, we are obliged to show this is actually

[5]The CSLI corpus was the standard test corpus used during the development of the ERG, and contains over 1000 sentences. We use only 40 sentences since only a small subset of the CSLI lexicon has been ported to ALE.

[6]The FUSE corpus is a set of sentences collected for the VERBMOBIL project by staging mock appointment scheduling conversations.

Grammar	# Declarations	# Tokens	Mean RHS arguments
ICEBERG*	349	2898	2.3
ERG*	514	4176	6.1
ICEBERG	755	6386	4.6
ERG	924	7669	6.3

Table 7.4: Source code comparison

the case. This section argues for the usability of ICEBERG over the ERG by taking some quantitative usability measures (tables 7.4 and 7.5). We will consider four "grammars:" ERG, ICEBERG, ERG*, and ICEBERG*. ERG refers to the original ALE-ported grammar, ERG* ($\subset$ ERG) refers to the portion of ERG that was re-implemented with higher-order types, ICEBERG* refers to the higher-order re-implementation of ERG*, and, finally, ICEBERG is the union of ICEBERG* and the non-re-implemented portion of ERG (i.e. ICEBERG = ICEBERG* $\cup$ (ERG $-$ ERG*)).

Table 7.4 examines the grammar source code. We have reported three measures: number of type-related declarations, number of tokens in the grammar source, and the mean number of arguments appearing on the right-hand-side (RHS) of a type-related declaration (that is, feature introductions, constraints, lexical entries, and grammar rules are excluded). The number of type-related declarations and number of tokens are intended to give a sense of grammar terseness. The mean number of arguments on the RHS of declarations also provides a measure of terseness, but provides an additional measure of complexity: grammars with more RHS arguments have longer and more complex type declarations. The most significant difference in the results, as one would expect, is between ERG* and its higher-order re-implementation in ICEBERG* - ERG* requires 514 declarations, 4176 tokens, and an average RHS length of 6.1 to describe extensionally the same hierarchy that ICEBERG* describes with only 349 declarations, 2898 tokens, and an average RHS argument length of 2.3 arguments.

Grammar	Number of types				
	P	$C(P)$	$DM(C(P))$	$JR(P)$	$JR(C(P))$
ICEBERG*	564	1177	1825	21	304
ERG*	1140	1140	1686	297	297
ICEBERG	2813	3426	4444	252	535
ERG	3389	3389	4305	528	528

Table 7.5: Type hierarchy comparison

Table 7.5 examines the ICEBERG and ERG type hierarchies. For this purpose we have taken five measurements: number of types in the user-defined (higher-order) type hierarchy (P), number of types in the induced hierarchy before GLB completion ($C(P)$),[7] number of types in the induced hierarchy after GLB completion ($DM(C(P))$), and number of join-reducible types in both the parametric and induced hierarchies ($JR(P)$ and $JR(C(P))$).

Again, we see the most significant difference between ICEBERG* and ERG*. By employing higher-order typing, ICEBERG* has reduced the number of types a grammar writer must define from 1140 to 564, a savings of just over 50%. The number of types in ICEBERG* and ERG* induced hierarchies (both before and after GLB completion) are very similar, reflecting the fact that ICEBERG*'s higher-order types and closure specifications really do re-implement ERG*.

The most telling difference between ICEBERG* and ERG*, however, is the join-reducible types. Although ICEBERG* and ERG* induced hierarchies include a nearly identical number of join-reducible types, there are only 21 join-reducible types in ICEBERG*'s parametric type hierarchy compared to 297 in ERG*, a difference of nearly 93%. In ERG*, join-reducible types account for 26% of all types in P, but account for less than

[7]Because the ERG does not have parametric types, the ERG "higher-order" and "induced" type hierarchies are simply the ERG's explicit IS-A network with GLB completion types excluded.

4% of types in ICEBERG*. This is a strong argument for modularity: partial orders with few join-reducible types are tree-like, and those with many join-reducible types are more tightly connected networks. Tree-like structures offer many advantages that come with locality: they facilite better encapsulation, looser coupling between modules, and more intuitive extensions and modifications. ICEBERG* effectively allows grammar writers to work with trees of higher-order types, deferring the complexity of managing crossing links to the constructors and closure specifications.

There is one final comment that needs to be made regarding $DM(C(P))$. As we have seen previously in §5.1.5, for unconstrained semi-coherent parametric type hierarchies, $I(P)$ is always a meet-semilattice. Now, because of HOPS's extra higher-order constructors, ground instance subtyping links, and closure operations, the proof in §5.1.5 does not hold for all type hierarchies written in HOPS, but it does hold for HOPS hierarchies consisting only of parametric product types and no extra ground instance subtyping links. In the case that $I(P)$ is a meet-semilattice, since $C(P) \subseteq I(P)$, it must also be the case that $DM(C(P)) \subseteq I(P)$, i.e., the types added by GLB completion are members of $I(P)$. HOPS does not currently attempt to locate GLB types in $I(P)$ due to the same complexities that prevent many HOPS induced type hierarchies from being meet-semilattices, but one should expect that a large number of GLB types in ICEBERG* are actually well-defined ground instances.

Chapter 8

Conclusions

Though explicitly-specified IS-A networks provide a simple and intutive form of inclusional semantics, their use in contemporary typed feature structure grammars has led to a proliferation of types and unwieldly signatures that remain inaccessible to most would-be grammar developers. This thesis has attempted to provide some relief by mining signatures for evidence of intended structural patterns, encapsulating those patterns with a collection of higher-order type constructors, using the constructors as the basis of a higher-order signature specification language, and using the signature specification language to implement a higher-order version of the English Resource Grammar type hierarchy. The result is a more readable, intuitive, acessible, and extendable ERG and the first typed feature structure grammar to consistently employ higher-order typing.

In the process, a review of the effects of explicitly-specified IS-A networks in grammar design was conducted, design patterns were introduced as a framework for grammar documentation and design cataloguing, parametric types were provided with formal extensions which make them more appropriate for grammar engineering, and a higher-order description language and interpreter have been developed to facilitate the construction of other higher-order typed feature structure grammars.

8.1 Summary of contributions

Design patterns for grammar engineering

We have traced the history of design patterns, from Christopher Alexander's *Notes on the Synthesis of Form*, *A Pattern Language*, and *The Timeless Way of Building* to their adaptation for software engineering in the Gang of Four's *Design Patterns* and Buschmann et. al.'s *A System of Patterns*. We have sketched the key concepts of pattern documentation and reuse, introduced design patterns as a framework for discussing and documenting recurring patterns of design in typed feature structure grammars, and catalogued several structural patterns of type usage.

Extension of parametric types for attribute-value logic

To facilitate their use in grammar development, Penn's parametric types for attribute-value logic have been augmented with parametric restrictions and closure specifications. Previous formalizations of parametric types had no methods for stating appropriate values of parameters (restrictions) or for determining which of the appropriate values are necessary for processing (closure). Conditions for finiteness and well-formedness have been also been proved, and an algorithm for closing different parts of a type signature under different closure specifications has been developed.

HOPS - a higher-order signature description language

HOPS, a higher-order parametric signature description language has been presented, complete with several kinds of higher-order constructors and closure specifications. Futhermore, a HOPS compiler has been developed for the purposes of expanding parametric type signatures into non-parametric ones. HOPS can be used as a description language for higher-order type hierarchies and the HOPS compiler can be used in combination with ALE to compute with higher-order types.

ICEBERG - a higher-order ERG

We have presented ICEBERG, a higher-order re-implementation of a significant portion of the ERG type hierarchy. Intended as a proof of concept, ICEBERG shows that higher-order typed feature structure grammars are possible and illustrates many issues involved in designing grammars with higher-order types. In the process, ICEBERG, through explicit reference to structural patterns and parametricity, provides a sort of documentation of ERG design intentions and linguistic intuitions, illustrating how those intentions and intuitions might be carried over to future grammars.

8.2 Future directions

Investigation of cross-linguistic applicability

Our investigation of evidenced structural patterns has been limited to the English Resource Grammar type hierarchy, and we have only attempted to re-implement ERG types. One would expect other HPSG grammars, especially grammars of languages other than English, to contain their own structural patterns - patterns which may or may not be the same patterns as those encountered in the ERG. It would be interesting to discover, for example, to what extent finite domains, smyth powerlattices, and strict variants can be adapted cross-linguistically to express linguistic universals such as tense, gender, mood, and agreement.

Extending higher-order appropriateness in ICEBERG

Because variables can be shared between parameters and appropriateness specifications, appropriateness is one of the most powerful devices devices afforded by parametric types. Unfortunately, ICEBERG currently makes no use of that power, although there are plenty of places where one could imagine parametric appropriateness. The most obvious example is list types (indeed, [PS94] suggested parametrically typed lists), but one could

also imagine a *head* parameter on an HPSG *cat* type which is shared with its `HEAD` feature. Similarily, one could imagine a *synsem* parameter on ERG *word* and *phrase* types which is shared with the `SYNSEM` feature, or, for auxillary verbs, *tense* parameter shared with the `CAT:HEAD:TAM` feature. Our reluctance to use appropriateness parameters is in large part due to time considerations, but we suspect many ERG constraints could be intuitively captured with shared parameters in appropriateness specifications.

User Study

Currently our arguments for the usability of ICEBERG over ERG are limited to a few rough numerical estimations and an argument that ICEBERG exploits patterns already implicitly at work in the ERG. A more effective test, of course, would be to conduct a usability study with human subjects.

Such a usability study, we believe, would involve test subjects at least semi-familiar with linguistic theory, requiring them to extend the grammar to obtain new analyses, refactor the grammar to fix incorrect analyses, or perhaps explain why certain analyses are licensed or not licensed by the grammar. The usability study could take such measures such as quality of task completion, time for task completion, and general user preference of one grammar over the other.

We have not conducted such a study, partially due to time considerations, and partially due to a general lack of formally trained linguists with whom to conduct the study. This remains, however, an interesting avenue of further exploration.

Appendix A

ICEBERG

This appendix presents a detailed tour of ICEBERG. The tour roughly follows figure 7.1 by splitting the ERG signature into filters of incomparable types, describing, for each filter, the role of the types within the grammar, the ERG organization of types, and, if necessary, the ICEBERG re-implementation. ICEBERG has made slight modifications to the ERG topology (figure A.1) by removing types **sort** and **avm** and replacing them with more descriptive classifiers: *featureless*, *syntax*, *semantics*, *container*, and *sign structure*. The tour proceeds in order of featureless types, containers, syntax, semantics, sign structures, phrases, and finally, words. Due to the number of types in some of the filters, any types appearing in the ERG hierarchy but not appearing in a corresponding ICEBERG ground instance translation table should be assumed to be present in ICEBERG as simple types by the same name their definition can be found in the corresponding ICEBERG description.

A.1 Featureless types

ICEBERG featureless types include all the indecomposable linguistic properties encoded as subtypes of ERG type **sort**.

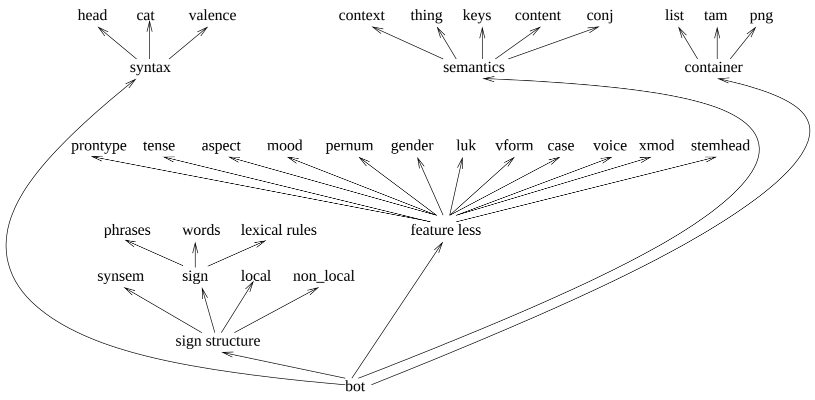

Figure A.1: ICEBERG IS-A topology

A.1.1 Aspect

Aspect is a grammatical category expressing temporal properties of verbs. The ERG attests two kinds of aspect, *progressive* and *perfective*, as well as a type *no_aspect* which is used to explicitly state aspect does not apply. The most general aspectual type, *aspect* is the appropriate value of the `ASPECT` feature of tense-aspect-mood feature structures.

Besides types for progressive and perfective, and no aspect, the ERG aspect types also include disjunctive types for progressive or no aspect (named *non-perfect*), and perfect or no aspect (named *non-progressive*). A strict variant of both basic and disjunctive types is present, as well as a number of conjunctive types. There seems to be a structural anomaly in the conjunctive types, however, as both *non-progressive* and *progressive* have a join and *non-perfect* and *perfect* have a join.

ICEBERG implements *aspect* with a finite domain over types *perf*, *progr*, and *none* (*fa*), a smyth powerlattice of *fa* (*sma*), and a strict variant of *fa* (*sta*). Types *fa* and *sma* undergo the trivial closure over grammar-attested ground instances augmented with

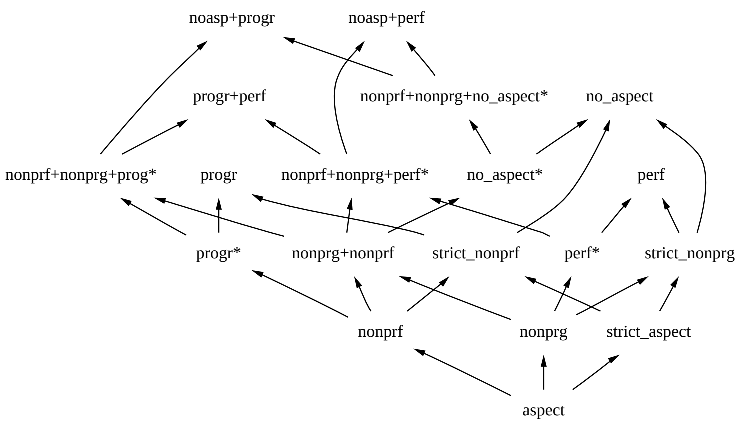

Figure A.2: ERG aspect types

conjunctive types, and the closure of *smt* is unconstrained. Although ICEBERG does not have a join between *fa(none;perf)* and *progr* or between *fa(none;progr)* and *perf*, we have found the presence of such joins makes no difference to analyses obtained by parsing our test corpora.

ICEBERG description

```
fa         findom  [perf, progr, none].
sma        smyth   fa.
sta        strictx fa.
aspect     sub     fa.

fa         close   trivial(aspect_gen).
sma        close   trivial(aspect_gen).
aspect_gen gen     [{grammar}, sma(none,perf),
                               sma(none,progr),
                               sma(perf,progr)].
```


ICEBERG ground instances	
ERG name	ICEBERG name
aspect	aspect
no_aspect*	none
perf*	perf
progr*	progr
nonprg	fa(none;perf)
nonprf	fa(none;progr)
	fa(none;perf;progr)
no_aspect	sta(none)
perf	sta(perf)
progr	sta(progr)
strict_nonprg	sta(fa(none;perf))
strict_nonprf	sta(fa(none;progr))
	sta(fa(none;perf;progr))
noasp+perf	sma(none,perf)
noasp+progr	sma(none,progr)
progr+perf	sma(perf,progr)

A.1.2 Case

Grammatical case is a linguistic category used to overtly mark nominal items for grammatical and/or semantic function. World languages include many kinds of case, including, but not limited to, accusative, dative, ergative, genitive, and nominal cases. In the ERG, the most general case type, *case*, is the appropriate value of the `CASE` feature of nominal head-type feature structures.

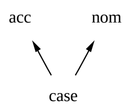

Figure A.3: ERG case types

The ERG posits only two kinds of case: accusative (*acc*) and nominative (*nom*), which are the only subtypes of *case*. There are no higher-order constructors which apply to ERG cases.

ICEBERG description

```
case sub [acc, nom].
```

A.1.3 Gender

Gender types are used to denote grammatical gender in situations such as nominals, adjectives, verb morphology, and agreement. In the ERG, the most general gender type, *gender*, appears as the appropriate value of the `GENDER` feature in person-number-gender feature structures.

The ERG posits five basic genders–*masculine*, *feminine*, *neuter*, *androgynous*, and *androgynous1*. There are strict variants of all five genders, as well as two-member conjunctive types of all pairs of *masculine*, *feminine*, *neuter*, and *androgynous*.

ICEBERG implements *gender* with a smyth powerlattice of basic gender types and a strict variant of those same basic gender types. The smyth powerlattice undergoes the trivial closure over a generator set composed of grammar-attested ground instances plus manually added conjunctive types, and the closure of the strict variant is unconstrained. One greatest lower bound type (*glbtype724*) appears in the most general satisfier of several lexical items and turns out to be equivalent to *stg(gender)*.

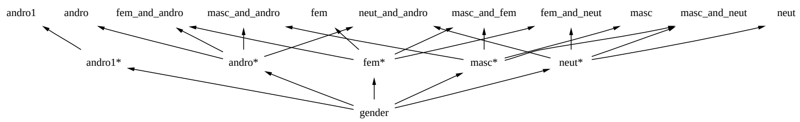

Figure A.4: ERG gender types

ICEBERG description

```
gender     sub     [masc, fem, neut, andro, andro1].
smg        smyth   gender.
stg        strictx gender.

smg        close   trivial(gender_set).
gender_set gen     [{grammar}, smg(andro,fem),
                               smg(andro,masc),
                               smg(andro,neut),
                               smg(fem,masc),
                               smg(fem,neut),
                               smg(masc,neut)].
```

ICEBERG ground instances	
ERG name	ICEBERG name
gender	gender
andro1*	andro1
andro*	andro
fem*	fem
masc*	masc
neut*	neut
glbtype724	stg(gender)
andro1	stg(andro1)
andro	stg(andro)

continued on next page

continued from previous page	
ERG name	ICEBERG name
fem	stg(fem)
masc	stg(masc)
neut	stg(neut)
fem_and_andro	smg(andro,fem)
masc_and_andro	smg(andro,masc)
neut_and_andro	smg(andro,neut)
masc_and_fem	smg(fem,masc)
fem_and_neut	smg(fem,neut)
masc_and_neut	smg(masc,neut)

A.1.4 Luk

Luk types implement a three-valued boolean logic over values + (true), - (false), and *na* (not-applicable). Luk types appear as appropriate values in many feature structure descriptions, always to denote the truth or falsity of some property. The most general boolean type, *luk*, is a supertype of +, -, and *na*, although most appropriate values are restricted to *bool*, which is a supertype of only + and -.

The ERG luk sort includes types for all disjunctions of +, -, and *na*. There is also a strict variant on the filter induced by *bool* and a conjunctive type for + and -.

ICEBERG has implemented *luk* with a finite domain over +, -, and *na* (*fl*), a smyth powerlattice of *fl* (*sml*), and a strict variant of *fl* (*stl*). The induced hierarchies of *fl* and *sml* are constrained by the trivial closure over a generator set composed of grammar-attested ground instances plus one conjunctive type (*sml('+','-')*). The closure of *stl* is unconstrained.

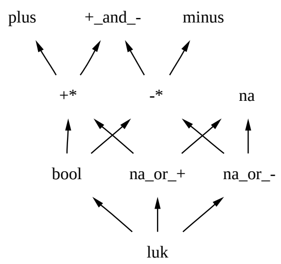

Figure A.5: ERG luk types

ICEBERG description

```
fl      findom  ['+', '-', na].
sml     smyth   fl.
stl     strictx fl.
luk     sub     fl.

fl      close   trivial(luk_set).
sml     close   trivial(luk_set).
luk_set gen     [{grammar}, sml('+','-')].
```

ICEBERG ground instances	
ERG name	ICEBERG name
luk	luk
	fl(+;-;na)
bool	fl(+;-)
na_or_+	fl(+;na)
na_or_-	fl(-;na)
na	na
+*	+
	continued on next page

continued from previous page	
ERG name	ICEBERG name
-*	-
	stl(fl(+;-;na))
	stl(fl(-;na))
	stl(fl(+;-))
	stl(na)
plus	stl(+)
minus	stl(-)
+_and_-	sml(+,-)

A.1.5 Mood

Mood types encode verbal modality. In the ERG, three types of mood are attested: indicative, subjunctive, and a subjunctive mood for modal verbs. The most general mood type, *mood*, is the appropriate value of the `MOOD` feature of tense-aspect-mood feature structures.

The ERG encodes three basic kinds of mood, *subjunctive*, *indicative*, and *modal subjunctive*. There are strict variants of all three mood types, as well as one disjunctive type (*indicative or modal subjunctive*) and one conjunctive type (*indicative or modal subjunctive*).

ICEBERG has implemented mood types with a finite domain over types *subjunctive*, *modal_subj*, and *ind* (*fm*), a smyth powerlattice of *fm* (*smm*), and a strict variant of *fm* (*stm*). Types *fm* and *smm* are closed by the trivial closure over grammar-attested ground instances plus explicitly added conjunctive types, and the closure of *stm* is unconstrained. One greatest lower bound type (*glbtype723*) appears in the most general satisfier of certain

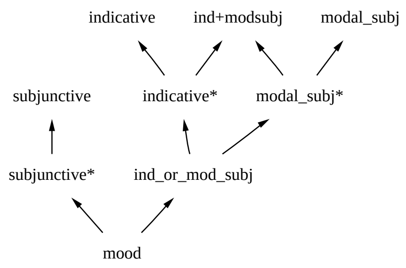

Figure A.6: ERG mood types

lexical items and has been given a place in ICEBERG as *stm(fm(ind;modal_subj))*.

ICEBERG description

```
fm       findom  [subjunctive, modal_subj, ind].
smm      smyth   fm.
stm      strictx fm.
mood     sub     fm.

sf       close   trivial(mood_set).
smm      close   trivial(mood_set).
mood_set gen     [{grammar}, smm(ind,modal_subj)].
```

ICEBERG ground instances	
ERG name	ICEBERG name
mood	mood
subjunctive*	subjunctive
indicative*	ind
modal_subj*	modal_subj
ind_or_mod_subj	fm(ind;modal_subj)
	continued on next page

continued from previous page	
ERG name	ICEBERG name
	fm(ind;modal_subj;subjunctive)
subjunctive	stm(subjunctive)
indicative	stm(ind)
modal_subj	stm(modal_subj)
glbtype723	stm(fm(ind;modal_subj))
strict_mood	stm(fm(ind;modal_subj;subjunctive))
ind+modsubj	smm(ind,modal_subj)

A.1.6 Pernum

Pernum types encode person and number values for use in various forms of agreement. The ERG combines person and number types into one filter in order to capture properties of English verb morphology. The most general person-number type, *pernum*, is the appropriate value of the `PERNUM` feature of person-number-gender feature structures.

The ERG first classifies pernum types as non-third-singular versus one-singular or third-singular, after which non-third-singular is further classified into first-singular versus non-first-singular, and, finally, non-first-singular is classified into second-person versus first-plural or third plural. There are strict variants of all these types, as well as a number of conjunctive types.

ICEBERG has explicitly stated the ERG's hierarchy of verb-morphology-influenced basic pernum types, constructing a smyth powerlattice and a strict variant of the base hierarchy. The smyth powerlattice undergoes the trivial closure over grammar-attested ground instances plus explicitly stated conjunctive types, and the strict variant closure is unconstrained. One extra pernum type (*threeper*) has been added to the signature to

facilitate the parameterization of pronominal types in §A.1.7.

ICEBERG description

```
pernum     sub     [oneor3sg, non3sg, threeper].
non3sg     sub     [non1sg, onesg].
non1sg     sub     [twoper, oneor3pl].
oneor3sg   sub     [onesg, threesg].
oneor3pl   sub     [onepl, threepl].
twoper     sub     [twosg, twopl].
threeper   sub     [threesg, threepl].

stp        strictx pernum.
smp        smyth   pernum.

smp        close   trivial(pernum_set).
pernum_set gen     [{grammar}, smp(onepl,onesg),
                               smp(onepl,twopl),
                               smp(onepl,twosg),
                               smp(onepl,threepl),
                               smp(onepl,threesg),
                               smp(onesg,twopl),
                               smp(onesg,twosg),
                               smp(onesg,threepl),
                               smp(onesg,threesg),
                               smp(twopl,twosg),
                               smp(twopl,threepl),
                               smp(twopl,threesg),
                               smp(twosg,threepl),
                               smp(twosg,threesg),
                               smp(threepl,threesg) ].
```

ICEBERG ground instances	
ERG name	ICEBERG name
pernum	pernum
1or3pl	oneor3pl
1or3sg	oneor3sg
non1sg	non1sg
	continued on next page

continued from previous page

ERG name	ICEBERG name
non3sg	non3sg
1per	
2per	twoper
	threeper
1pl*	onepl
1sg*	onesg
2pl*	twopl
2sg*	twosg
3pl*	threepl
3sg*	threesg
strict_pernum	stp(pernum)
strict_1or3pl	stp(oneor3pl)
strict_1or3sg	stp(oneor3sg)
strict_non1sg	stp(non1sg)
strict_non3sg	stp(non3sg)
strict_2per	stp(twoper)
	stp(threeper)
1pl	stp(onepl)
1sg	stp(onesg)
2pl	stp(twopl)
2sg	stp(twosg)
3pl	stp(threepl)
3sg	stp(threesg)
1sg_and_1pl	smp(onepl,onesg)

continued on next page

continued from previous page

ERG name	ICEBERG name
1pl_and_2pl	smp(onepl,twopl)
1pl_and_2sg	smp(onepl,twosg)
1pl_and_3pl	smp(onepl,threepl)
1pl_and_3sg	smp(onepl,threesg)
1sg_and_2pl	smp(onesg,twopl)
1sg_and_2sg	smp(onesg,twosg)
1sg_and_3pl	smp(onesg,threepl)
1sg_and_3sg	smp(onesg,threesg)
2sg_and_2pl	smp(twopl,twosg)
2pl_and_3pl	smp(twopl,threepl)
2pl_and_3sg	smp(twopl,threesg)
2sg_and_3pl	smp(twosg,threepl)
2sg_and_3sg	smp(twosg,threesg)
3pl_and_3sg	smp(threepl,threesg)
1or3pl+1per+non1sg	
1sg*+1per+non1sg	
3sg*+1per+non1sg	
2per+1per+non1sg	
2sg*+2per+1per+non1sg	
2pl*+2per+1per+non1sg	
2per+3sg*+1per+non1sg	
1or3pl+3sg*+1per+non1sg	
1sg*+1or3pl+1per+non1sg	
1pl*+1or3pl+1per+non1sg	

continued on next page

continued from previous page	
ERG name	ICEBERG name
3pl*+1or3pl+1per+non1sg	
2per+1or3pl	
1or3pl+2per+1per+non1sg	
1sg*+2per+1per+non1sg	
1sg*+2per+1per+1or2pl+non1sg	
1pl*+1or3pl	
2sg*+1or3pl	
2pl*+1or3pl	
3pl*+1or3pl	

A.1.7 Pronouns

Featureless pronoun types are used in combination with person-number (pernum) and gender types to label semantic indices for the purpose of agreement. The featureless pronoun types are not the type labels of pronominal lexical entries–there is another set of pronoun types which are subtypes of *word*.

The ERG attests six basic types of pronouns, demonstrative, reciprocative, reflexive, impersonal, personal, and "zero" pronouns (for constructions with no extensional subject). Personal pronouns are further classified as first-plural, first-singular, second person, and third person.

ICEBERG has implemented pronoun types with five simple types plus a parametric type (*ppro*) for personal pronouns. The restriction of *ppro* is *pernum* and *ppro* undergoes the trivial closure over grammar-attested ground instances.

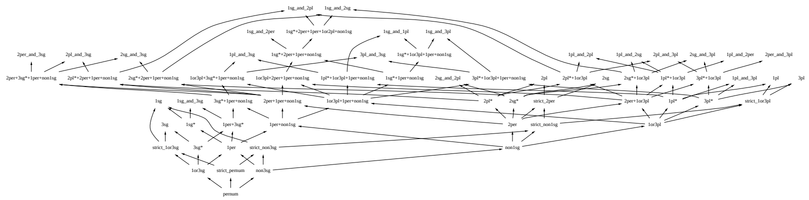

Figure A.7: ERG pernum types

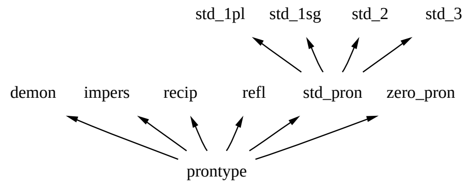

Figure A.8: ERG pronoun types

ICEBERG description

```
prontype sub  [demon, impers, recip, refl, zero_pron, ppro(pn:pernum)].

ppro     close trivial(grammar).
```

ICEBERG ground instances	
ERG name	ICEBERG name
prontype	prontype
demon	demon
impers	impers
recip	recip
refl	refl
	continued on next page

continued from previous page	
ERG name	ICEBERG name
zero_pron	zero_pron
std_pron	ppro(pn:pernum)
std_1pl	ppro(pn:onepl)
std_1sg	ppro(pn:onesg)
std_2	ppro(pn:twoper)
std_3	ppro(pn:threeper)

A.1.8 Stemhead

Stemhead types mark inflectional affixes, denoting the head constituent types of allowable stems. The ERG has separate incomparable sets of stemhead types (to mark lexemes) and ordinary head types (to mark words–§A.3.2) because the grammar does not currently distinguish between lexemes and words.[1]

The ERG classifies three basic types of stemheads: adjectival, nominal, and verbal. Nominal stemheads are further classified as count nouns and mass nouns.

ICEBERG implements stemheads as a parametric type *stemhead* with one parameter restricted to a stemhead category type for *adj*, *noun*, and *verb*. There is a parametric subtype, *nstemhead*, of *stemhead* which restricts the stemhead category to *noun* and adds a parameter for nominal stemheads. All ground instances are included in the induced signature.

ICEBERG description

```
stemhead_cat sub   [adj, noun, verb].
```

[1] ERG source code

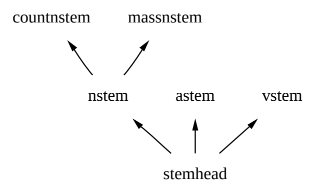

Figure A.9: ERG stemhead types

```
n_type       sub   [count, mass].

stemhead     param [cat:stemhead_cat].
nstemhead    param [cat:noun,ntype:n_type].

stemhead     sub   [nstemhead].
```

ICEBERG ground instances	
ERG name	ICEBERG name
stemhead	stemhead(cat:stemhead_cat)
	stemhead(cat:noun)
astem	stemhead(cat:adj)
vstem	stemhead(cat:verb)
nstem	nstemhead(cat:noun, ntype:n_type)
countnstem	nstemhead(cat:noun, ntype:count)
massnstem	nstemhead(cat:noun, ntype:mass)

A.1.9 Tense

ERG tense types denote grammatical tense. The ERG acknowledges three kinds of tense: past, present, and future. The most general type, *tense*, is appropriate to the

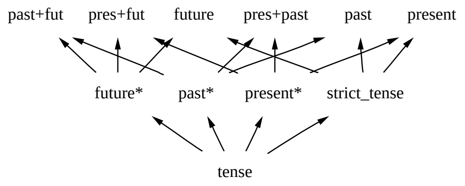

Figure A.10: ERG tense types

`TENSE` feature of tense-aspect-mood feature structures.

ERG tense types encode a basic hierarchy of three tenses, *past*, *present*, and *future*. There is a strict variant of this base hierarchy, as well as two-element conjunctions of non-strict *past*, *present*, and *future*.

ICEBERG encodes a basic tense hierarchy of three tenses, *tense*, and constructs both a smyth powerlattice of *tense* and a strict variant of *tense*. The closures of both the smyth powerlattice and strict variant are unconstrained.

ICEBERG description

```
tense sub     [future, past, present].
smt   smyth   tense.
stt   strictx tense.
```

ICEBERG ground instances	
ERG name	ICEBERG name
tense	tense
future*	future
past*	past
present*	present
strict_tense	stt(tense)

continued on next page

continued from previous page

ERG name	ICEBERG name
future	stt(future)
past	stt(past)
present	stt(present)
past+fut	smt(future,past)
pres+fut	smt(future,present)
pres+past	smt(past,present)
	smt(future,past,present)

A.1.10 Verb form

Verb-inflectional-form (*vform*) types encode English verb forms. The ERG contains, among others, verb form types for finite forms, non-finite forms, base forms, and participles. Verb form types appear as appropriate values the `VFORM` of verbal head types, which in turn appear as appropriate values for features specifying the constituent head types of HPSG words and phrases.

The ERG hierarchy of verb forms contains a number of disjunctive types of finite, infinitive, base, and imperative forms, and a conjunctive type for finite and imperative forms. There is also a type *non_fin* which subsumes a mostly incomparable hierarchy of non-finite verb forms (including gerunds, past participles, and present participles).

ICEBERG implements *vform* with five higher-order types and a number of explicitly-added ground-instance subtyping links. The first higher-order type, *fv*, is a finite domain over *inf*, *fin*, *bse*, and *imp*. Type *smv* is declared a smyth powerlattice of *fv*, and *stv* is declared a strict variant of *fv*. Both *fv* and *smv* undergo the trivial closure over grammar-attested ground instances plus an explicitly-added conjunctive type. The closure of *stv* is unconstrained. ICEBERG also implements two parametric product types, *vpresent*

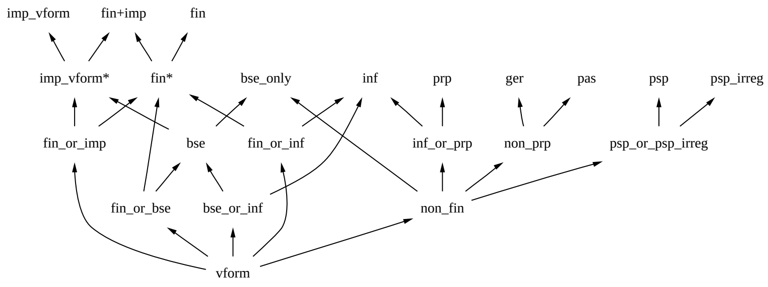

Figure A.11: ERG verb form types

and *pastpart*, in order to capture present and past participles. *vpresent* undergoes the trivial closure over grammar-attested ground instances, and the closure of *pastpart* is unconstrained. The remaining verb form types have been added by ground-instance subtyping links.

ICEBERG Description

```
fv         findom  [inf, fin, bse, imp].
smv        smyth   fv.
stv        strictx fv.
vform      sub     [fv, non_fin].

prptype    sub     [ger, pas].
non_fin    sub     [vpresent(prp:bool,type:prptype),
                    pastpart(irreg:bool)].

bse        sub     imp.
non_fin    sub     [fv(inf;prp), stv(bse)].
fv(inf;prp) sub    [vpresent(prp:plus), inf].

fv         close   trivial(vform_set).
smv        close   trivial(vform_set).
vpresent   close   trivial(vform_set).
vform_set  gen     [{grammar}, smv(fin,imp)].
```


ICEBERG ground instances	
ERG name	ICEBERG name
vform	vform
	fv(bse;fin;imp;inf)
fin_or_bse	
bse_or_inf	
fin_or_imp	fv(fin;imp)
fin_or_inf	fv(fin;inf)
inf	inf
bse	bse
fin*	fin
imp_vform*	imp
	stv(fv(bse;fin;imp;inf))
strict_vform	stv(fv(fin;imp))
	stv(fv(fin;inf))
bse_only	stv(bse)
fin	stv(fin)
imp_vform	stv(imp)
	stv(inf)
fin+imp	smv(fin,imp)
non_fin	non_fin
inf_or_prp	fv(inf;prp)
psp_or_psp_irreg	pastpart(irreg:bool)
psp_irreg	pastpart(irreg:plus)
psp	pastpart(irreg:minus)

continued on next page

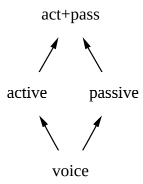

Figure A.12: ERG voice types

continued from previous page	
ERG name	ICEBERG name
	vpresent(prp:bool)
prp	vpresent(prp:plus)
non_prp	vpresent(prp:minus)
ger	
pas	vpresent(prp:minus,type:pas)

A.1.11 Voice

Grammatical voice is a property of sentential arguments which expresses semantic roles. World languages contain many kinds of voice, including active, middle, passive, and antipassive voice. The ERG posits only two kinds of voice: *active* and *passive*, which are subtypes of the most general voice type, *voice*. There is also a conjunctive type, *act+pass*. ICEBERG recognizes that *act+pass* is actually a member of the smyth powerlattice.

ICEBERG description

```
voice   sub   [active, passive].
smvoice smyth voice.
```


ICEBERG ground instances	
ERG name	ICEBERG name
active	active
passive	passive
voice	voice
act+pass	smvoice(active,passive)
	smvoice(active)
	smvoice(passive)
	smvoice(voice)

A.1.12 Xmod

Xmod (external modifier) types denote presence and directionality of modifiers external to a constituent. Xmod types are appropriate to the `MODIFIED` feature of synsem types. The ERG encodes three basic xmod types, (*lmod*, *rmod*, and *notmod*), two disjunctive types, and strict variants of *lmod* and *rmod*.

ICEBERG has implemented xmod with a finite domain over *rmod*, *lmod*, and *notmod* (*fx*) and a strict variant of *fx* (*smx*). Type *fx* undergoes trivial closure over grammar-attested ground instances and the closure of *smx* is unconstrained.

ICEBERG description

```
fx   findom  [lmod, rmod, notmod].
smx  strictx fx.
xmod sub     fx.

fx   close   trivial(grammar).
```

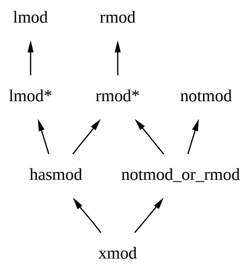

Figure A.13: ERG external modifier types

ICEBERG ground instances	
ERG name	ICEBERG name
xmod	xmod
	fx(lmod;notmod;rmod)
hasmod	fx(lmod;rmod)
notmod_or_rmod	fx(notmod;rmod)
lmod*	fx(lmod)
notmod*	fx(notmod)
rmod*	fx(rmod)
	stx(fx(lmod;notmod;rmod))
	stx(fx(lmod;rmod))
	stx(fx(notmod;rmod))
lmod	stx(lmod)
notmod	stx(notmod)
rmod	stx(rmod)

A.2 Container types

ICEBERG container types are those types which are not directly related to any linguistically-motivated component of the grammar, but whose purpose is to provide wrappers for collections of features. In ICEBERG, types *png*, *tam*, and *list* are classified as containers.

A.2.1 List

HPSG list types implement lists of feature structures, most notably for specifying and managing subcategorization frames and semantic arguments. HPSG lists have long been described parametrically,[2] but, in practice, have been implemented with simple types.

The ERG contains two kinds of lists, difference lists (**diff-list**) and linked-list-style lists (*list*). Type *list* has a number of subtypes which restrict the element types of lists to different subtypes of *synsem*.

Because difference lists contain very few subtypes, ICEBERG has focused its parametric re-implementation on linked-list-style lists. Type *list* has been re-cast as a parametric product type with a parameter for element type and a parameter for the value of the synsem `OPT` feature. ICEBERG has also introduced three parametric subtypes of *list*: *e_list*, *ne_list*, and *zero_one_list*. A number of simple types and ground instances subtyping links have been explicitly added, and all parametric types are closed by trivial closure over grammar-attested ground instances.

A number of ERG list types introduce constraints on feature values of their elements–constraints which greatly contribute to a rather unintuitive and cumbersome hierarchy of list types and which make succinct parameterization difficult. In the opinion of this author, constraints on feature values of list elements are more appropriately introduced by the element types themselves, and by eliminating those constraints from list types, one should expect to find a more intuitive parameterization. Because lists are so tightly

[2] see, for instance, [Pol90] and [PS94].

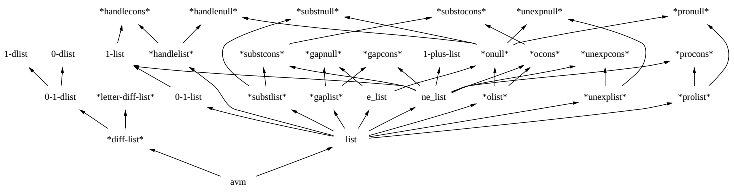

Figure A.14: ERG list types

woven into the ERG principles and constraints, we have not yet attempted such a refactoring, but it remains an obvious candidate for further investigation.

ICEBERG description

```
'*diff-list*'                  sub   ['*letter-diff-list*', '0-1-dlist'].
'0-1-dlist'                    sub   ['0-dlist', '1-dlist'].

list                           param [x:syn_sign,op:bool].

list                           sub   [e_list, ne_list, zero_one_list].
zero_one_list                  sub   [one_list].
ne_list                        sub   [one_list].

ne_list(x,op)                  sub   ['1-plus-list'].
list(x,op)                     sub   [subst_list].
ne_list(x,op)                  sub   [ne_subst_list].
subst_list                     sub   [ne_subst_list, e_subst_list].
e_list(x:unexpressed,op:plus)  sub   [e_subst_list, e_list(x:handle),
                                      e_list(x:unexpressed,op:minus),
                                      e_list(x:pro_ss)].
zero_one_list(x:handle)        sub   [e_list(x:handle)].
ne_subst_list                  sub   ['*substocons*'].
ne_list(x:unexpressed,op:plus) sub   ['*substocons*'].

list                           close trivial(grammar).
e_list                         close trivial(grammar).
ne_list                        close trivial(grammar).
zero_one_list                  close trivial(grammar).
one_list                       close trivial(grammar).
```


ICEBERG ground instances	
ERG name	ICEBERG name
e_list	e_list(x:syn_sign)
ne_list	ne_list(x:syn_sign)
1-list	one_list(x:syn_sign)
0-1-list	zero_one_list(x:syn_sign)
1-plus-list	1-plus-list
gaplist	list(x:gap)
gapcons	ne_list(x:gap)
gapnull	e_list(x:gap)
prolist	list(x:pro_ss)
procons	ne_list(x:pro_ss)
pronull	e_list(x:pro_ss)
olist	list(x:unexpressed,op:plus)
ocons	ne_list(x:unexpressed,op:plus)
onull	e_list(x:unexpressed,op:plus)
unexplist	list(x:unexpressed,op:minus)
unexpcons	ne_list(x:unexpressed,op:minus)
unexpnull	e_list(x:unexpressed,op:minus)
handlelist	list(x:handle)
handlecons	one_list(x:handle)
handlenull	e_list(x:handle)
substlist	subst_list
substcons	ne_subst_list
substnull	e_subst_list

continued on next page

continued from previous page	
ERG name	ICEBERG name
glbtype882	zero_one_list(x:handle)

A.2.2 Person-Number-Gender

Person-number-gender types express the relationship of grammatical categories person, number, and gender and are implemented as a container for pernum and gender types. There is only one person-number-gender type, *png*, which introduces two features, `PERNUM` and `GENDER`, and *png* is appropriate to the `PNG` feature of index types.

ICEBERG description

```
container sub png.
```

A.2.3 Tense-Aspect-Mood

Tense-aspect-mood types express the relationship of grammatical categories tense, aspect, and mood and are implemented as a container for tense, aspect, and mood types. The most general tense-aspect-mood type, *tam*, introduces three features, `TENSE`, `ASPECT`, and `MOOD`, and *tam* is appropriate to the `TAM` feature of head types and event types. More recent versions of the ERG introduce subtypes of *tam*, but the ported version used as the basis for ICEBERG has only one tense-aspect-mood type.

ICEBERG description

```
container sub tam.
```


A.3 Syntax

ICEBERG syntax types include the ERG types related to the syntactic component of the grammar: category, head, and valence types.

A.3.1 Category

According to [PS94], HPSG category types should encode both the syntactic category of a word and the arguments is requires. The ERG fulfills these requirements by positing a `HEAD` feature and a valency list feature `VAL`, while also positing three boolean indicator features and a number of subtypes which classify categories by constituent head. The most general ERG category type, *cat_min*, is appropriate to the `CAT` feature of ERG *local* types.

ICEBERG implements category types with a parametric product type, *cat*, which classifies categories by category head, minimality, and whether a specifier is required. We posit two parametric subtypes of *cat*, a parametric product for verbal categories, *cat_v*, which introduces both a verb form parameter and a parameter indicating main-clausness, and a parametric product for nominal categories, *cat_n*, which introduces a parameter for case. One simple type, *prd_cat*, is posited as a subtype of *cat(min:minus)*. All three parametric product types undergo the trivial closure over grammar-attested ground instances.

ICEBERG description

```
cat            param [head:head,min:bool,spec:bool].

syntax         sub   [cat].
cat(min:minus) sub   [prd_cat].

cat            sub   cat_v( head  : fh(comp;verb),
                            min   : minus,
                            mc    : luk,
                            vform : vform ).
```

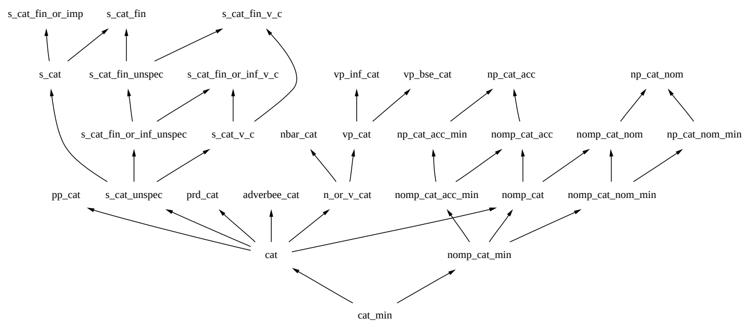

Figure A.15: ERG category types

```
cat            sub   cat_n( head  : sth(fh(comp;gerund;modnp;noun)),
                            case  : case ).

cat            close trivial(grammar).
cat_n          close trivial(grammar).
cat_v          close trivial(grammar).
```

ICEBERG ground instances	
ERG name	ICEBERG name
cat	cat(min:minus)
cat_min	cat(min:bool)
nbar_cat	cat(head:noun,min:minus,spec:plus)
pp_cat	cat(head:prep,min:minus)
nomp_cat_min	cat_n(min:bool)
nomp_cat	cat_n(min:minus)
nomp_cat_acc	cat_n(min:minus,case:acc)
nomp_cat_acc_min	cat_n(case:acc)

continued on next page

continued from previous page

ERG name	ICEBERG name
nomp_cat_nom	cat_n(head:mobile,min:minus,case:nom)
nomp_cat_nom_min	cat_n(head:mobile,case:nom)
np_cat_acc	cat_n(head:noun,min:minus,case:acc)
np_cat_acc_min	cat_n(head:noun,case:acc)
np_cat_nom	cat_n(head:noun,min:minus,case:nom)
np_cat_nom_min	cat_n(head:noun,case:nom)
vp_cat	cat_v(spec:plus)
vp_bse_cat	cat_v(head:verb,vform:stv(bse),spec:plus)
vp_inf_cat	cat_v(vform:stv(inf),spec:plus)
s_cat	cat_v(head:verb,mc:plus)
s_cat_fin	cat_v(head:verb,mc:plus,vform:stv(fin))

A.3.2 Head

Head types encode the kinds of constituent heads supported by the ERG. The most general ERG head type, *head_min* is the appropriate value for the `HEAD` feature of category types.

The ERG head types filter contains 90 types, and, after greatest lower bound completion, contains 247. The central portion of the hierarchy is a number of disjunctive, conjunctive, and strict variant types of basic head types for verbs, nouns, adjectives, gerunds, complementizers, prepositions, and modifier noun phrases. There are other mostly independent types for specifiers, punctuation, adverbs, and numbers.

ICEBERG has implemented head types with a finite domain type, *fh*, over types *adj*, *comp*, *gerund*, *modnp*, *noun*, *prep*, and *verb*, a smyth powerlattice (*smh*) of *fh*, and a strict variant (*sth*)of *fh*. Both *fh* and *smh* are closed by trivial closure over grammar-

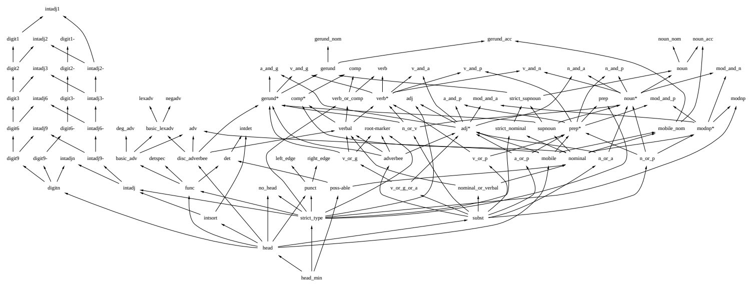

Figure A.16: ERG head types

attested ground instances plus explicitly added conjunctive types, and the closure *sth* is unconstrained. The remaining head types have been recorded with simple types and two parametric types which classify nouns and gerunds by case.

ICEBERG description

```
subst_gen    gen     [{grammar},
                      smh(adj,gerund),
                      smh(adj,modnp),
                      smh(adj,noun),
                      smh(adj,prep),
                      smh(adj,verb),
                      smh(gerund,verb),
                      smh(modnp,noun),
                      smh(modnp,prep),
                      smh(noun,prep),
                      smh(noun,verb),
                      smh(prep,verb),
                      smh(adj, fh(comp;gerund;verb)),
                      smh(adj, fh(prep;verb))].

fh           close   trivial(subst_gen).
smh          close   trivial(subst_gen).
sth          close   unconstrained.

fh           findom  [adj, comp, gerund, modnp, noun, prep, verb].
smh          smyth   fh.
```



```
smt          strictx fh.
subst        sub     fh.

adv_type     sub     [deg, lex, norm].
adv_lex_type sub     [normlex, neg].
adv          sub     [lex_adv(type:lex,ltype:adv_lex_type)].
adverbee     sub     [st_adverbee, gerund, fh(comp;verb)].
st_adverbee  sub     [adv(type:lex), adv(type:norm),
                      sth(gerund), sth(fh(comp;verb))].

sth(noun)    sub     [pnoun(case:case)].
sth(gerund)  sub     [pgerund(case:case)].

mobile       sub     [mobile_nom, adj, prep, adv(type:norm)].
mobile_nom   sub     [pnoun(case:acc), pgerund(case:acc)].

func         sub     [adv(type:adv_type), det, detspec].
'poss-able'  sub     [det, fh(comp;gerund;modnp,noun)].
det          sub     [intdet].
punct        sub     [left_edge, right_edge].

intsort      sub     [intadj, intdet].
intadj       sub     [intadjn, 'intadj9-'].
digitn       sub     [intadjn, digit9, 'digit9-'].

digit9       sub     [intadj9, digit6].
digit6       sub     [intadj6, digit3].
digit3       sub     [intadj3, digit2].
digit2       sub     [intadj2, digit1].
digit1       sub     [intadj1].

'digit9-'    sub     ['digit6-'].
'digit6-'    sub     ['digit3-'].
'digit3-'    sub     ['digit2-'].
'digit2-'    sub     ['digit1-'].

'intadj9-'   sub     [intadj9, 'intadj6-'].
'intadj6-'   sub     [intadj6, 'intadj3-'].
'intadj3-'   sub     [intadj3, 'intadj2-'].
'intadj2-'   sub     [intadj2, intadj1].

real_head    sub     [subst, intsort, disc_adverbee, digitn, mobile].
head         sub     [real_head, 'poss-able', strict_head].
strict_head  sub     [sth(fh(adj;comp;gerund;modnp;noun;prep;verb)),
                      no_head, 'root-marker', func, punct, intadj].
```

```
fh(comp;gerund;modnp;noun) sub [mobile_nom].
fh(comp;gerund;verb)       sub ['root-marker'].
```

ICEBERG ground instances	
ERG name	ICEBERG name
head	real_head
head_min	head
subst	real_subst
glbtype737	strict_head
adj*	adj
comp*	comp
gerund*	gerund
modnp*	modnp
noun*	noun
prep*	prep
verb*	verb
adj	sth(adj)
comp	sth(comp)
gerund	sth(gerund)
modnp	sth(modnp)
prep	sth(prep)
verb	sth(prep)
	fh(adj;comp;gerund;modnp;noun;prep;verb)
v_or_g	fh(comp;gerund;verb)
n_or_v	fh(gerund;noun;verb)
verb_or_comp	fh(comp;verb)

continued on next page

continued from previous page

ERG name	ICEBERG name
a_or_p	fh(adj;prep)
n_or_p	fh(noun;prep)
v_or_p	fh(prep;verb)
	sth(fh(adj;comp;gerund;modnp;noun;prep;verb))
	sth(fh(comp;gerund;verb))
	sth(fh(gerund;noun;verb))
	sth(fh(comp;verb))
	sth(fh(adj;prep))
	sth(fh(noun;prep))
	sth(fh(prep;verb))
a_and_g	smh(adj,gerund)
a_and_p	smh(adj,prep)
mod_and_a	smh(adj,modnp)
mod_and_n	smh(modnp,noun)
mod_and_p	smh(modnp,prep)
v_and_a	smh(adj,verb)
v_and_g	smh(gerund,verb)
v_and_n	smh(noun,verb)
v_and_p	smh(prep,verb)
n_and_a	smh(adj,noun)
n_and_p	smh(noun,prep)
glbtype740	smh(fh(comp;gerund;verb),adj)
glbtype777	smh(adj,fh(prep;verb))
adv	adv(type:norm)

continued on next page

continued from previous page

ERG name	ICEBERG name
basic_adv	adv
basic_lexadv	adv(type:lex)
deg_adv	adv(type:adeg)
lexadv	lex_adv(ltype:normlex)
negadv	lex_adv(ltype:neg)
glbtype742	st_adverbee
gerund_acc	pgerund(case:acc)
gerund_nom	pgerund(case:nom)
noun_acc	pnoun(case:acc)
noun_nom	pnoun(case:nom)

A.3.3 Valence

ERG *valence* types manage subcategorization requirements and are appropriate to the `VALENCE` feature of category types. In [PS94], valence types did not exist; category types simply took a subcategorization list feature. The ERG has since decomposed subcategorization lists into four lists: `COMP`, `SUBJ`, `SPR`, and `SPEC`, introducing *valence* types as a holder for these subcategorization lists.

There are only two *valence* types in the ERG, one of which is simply used for delayed feature introduction. Because of this, there is no opportunity for parametric refactorization of *valence* subtypes.

ICEBERG description

```
valence_min sub [valence].
```


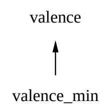

Figure A.17: ERG valence types

A.4 Semantics

ICEBERG semantics types include all ERG types which contribute to the minimal recursion semantics (MRS) [CFSP99] component of the grammar: conjunction, content, context, keys, and thing types.

A.4.1 Conjunction

Conjunction types collect semantic indices and handles of conjuncts for the purposes of building larger coordinate structures. The most general conjunction type, *conj*, is appropriate to the `CONJ` feature of local types.

The ERG contains five basic conjunction types: *atomic*, *complex*, *phrasal*, *numeric*, and *nil*. Atomic conjunctions are further classified as *both*, *either*, *neither*, and *nor*. There are a number of disjunctive types, and a type named *strict-conj* which, at first glance, appears to denote a strict variant. Further inspection shows, however, that *strict-conj* is a misnomer; *strict-conj* is effectively denoting the disjunctive type *atomic or complex or num*.

ICEBERG implements conjunction types as a finite domain over *atomic*, *complex*, *phrasal*, *numeric*, and *nil* conjunctions. Type *atomic* is further classified into lexically specific type, *both*, *either*, *neither*, and *nor*. The finite domain undergoes trivial closure over grammar-attested ground instances.

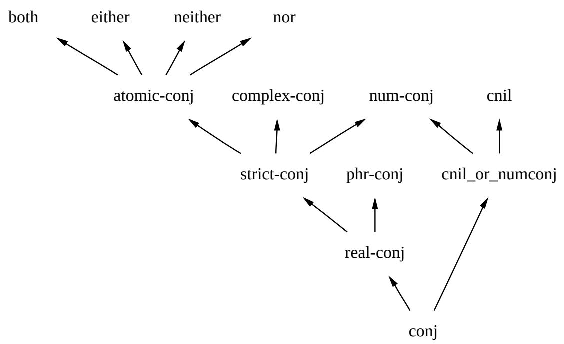

Figure A.18: ERG conjunction types

ICEBERG description

```
conj   findom [atomic, complex, phr, num, nil].
atomic sub    [both, either, neither, nor].

conj   close  trivial(grammar).
```

ICEBERG ground instances	
ERG name	ICEBERG name
conj	conj(atomic;cnil;complex;num;phr)
real-conj	conj(atomic;complex;num;phr)
strict-conj	conj(atomic;complex;num)
cnil_or_numconj	conj(cnil;num)
phr-conj	phr
cnil	cnil
num-conj	num
complex-conj	complex

continued on next page

continued from previous page

ERG name	ICEBERG name
atomic-conj	atomic
nor	nor
neither	neither
either	either
both	both

A.4.2 Content

Content types include the relations and rules required for minimal-recursion semantics, all of which are implemented as subtypes of type *cont*. Our ported version of the ERG included numerous semantic relation types and several MRS rule types, however, because the ERG has since eliminated many relation types (the ERG now uses lexical item orthography to mark relations [FB04]), and because of time constraints, ICEBERG has not performed any re-factoring of content types.

A.4.3 Context

Context types encode contextual information via a list of presuppositions. These presuppositions are used as prerequisites for the satisfaction of semantic relations.

The context filter contains two types, one of which is used for delayed feature introduction. ICEBERG has simply kept both types.

ICEBERG description

```
ctxt_min sub ctxt.
```

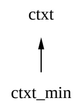

Figure A.19: ERG context types

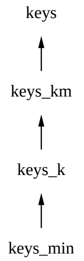

Figure A.20: ERG keys types

A.4.4 Keys

ERG keys types store MRS “key” relations and are appropriate to the KEYS feature of local types. The ERG contains a chain of four keys types, introducing features at each type. ICEBERG has simply adopted the four ERG types.

ICEBERG description

```
keys_min sub keys_k.
keys_k   sub keys_km.
keys_km  sub keys.
```


A.4.5 Thing

Feature structures of ERG type *mod-thing* are the structures to which thematic roles are assigned, as sketched in [PS94]. In addition, subtype *individual* introduces a person-number-gender feature for use in agreement. ERG things are classified into events, indices, and whether the index is part of a conjunction.

ICEBERG has followed the ERG by classifying *mod-thing* into types *thing* and *nothing* and classifying type *thing* as *handle*, *eventtime*, *hole*, or *individual*. *individual* has been implemented as a parametric type with parameters for semantic type, explicitness, fullness, and conjunctiveness, and has a parametric subtype for indices (which in turn has a parametric subtype for explicit indices). All three parametric types undergo trivial closure over a generator set of grammar-attested types plus explicitly added conjunction (*conj:plus*) types.

ICEBERG Description

```
semtype        sub   [(event;index)].
(event;index)  sub   [event, index].
indextype      sub   [deg, (noind;ref)].
(noind;ref)    sub   [ref, noind].
explindtype    sub   [it, ithere].

'mod-thing'    sub   [thing, nothing].
thing          sub   [eventtime, handle, hole].

thing          sub   ind      ( sem     : semtype,
                                expl    : bool,
                                full    : bool,
                                conj    : bool ).

ind            sub   index_ind( sem     : index,
                                indtype : indextype ).

index_ind(sem) sub   expl_ind ( expl    : plus,
                                explind : explindtype).

ind            close trivial(ind_set).
```

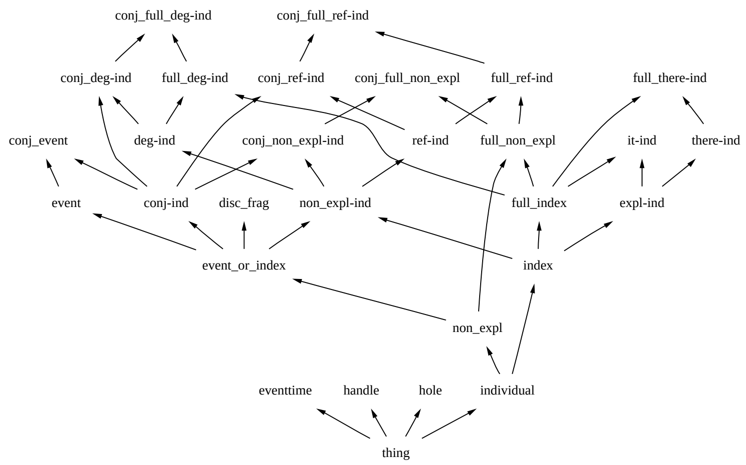

Figure A.21: ERG thing types

```
index_ind       close trivial(ind_set).
expl_ind        close trivial(ind_set).

ind:sem:sevent_or_sindex,expl:minus) sub disc_frag.

ind_set         gen   [{grammar},
                       ind((event;index),minus,bool,plus),
                       ind(event,minus,bool,plus),
                       index_ind(index,minus,plus,plus,noind),
                       index_ind(index,minus,bool,plus,ref),
                       index_ind(index,minus,bool,plus,deg),
                       index_ind(index,minus,plus,plus,ref),
                       index_ind(index,minus,plus,plus,deg),
                       index_ind(index,minus,bool,plus,noind),
                       index_ind(index,minus,bool,plus,indextype)].
```


ICEBERG ground instances

ERG name	ICEBERG name
individual	ind(expl:bool)
non_expl	ind(expl:minus)
event	ind(sem:event,expl:minus)
event_or_index	ind(sem:(event;index),expl:minus)
conj-ind	ind(sem:(event;index),expl:minus,conj:plus)
conj_event	ind(sem:event,expl:minus,conj:plus)
index	index_ind(expl:bool)
full_index	index_ind(full:plus)
non_expl-ind	index_ind(expl:minus)
full_non_expl	index_ind(expl:minus,full:plus,indtype:(noind;ref))
ref-ind	index_ind(expl:minus,indtype:ref)
deg-ind	index_ind(expl:minus,indtype:deg)
full_ref-ind	index_ind(expl:minus,full:plus,indtype:ref)
full_deg-ind	index_ind(expl:minus,full:plus,indtype:deg)
conj_full_non_expl	index_ind(expl:minus,full:plus,conj:plus,indtype:noind)
conj_ref-ind	index_ind(expl:minus,conj:plus,indtype:ref)
conj_deg-ind	index_ind(expl:minus,conj:plus,indtype:deg)
conj_full_ref-ind	index_ind(expl:minus,full:plus,conj:plus,indtype:ref)
conj_full_deg-ind	index_ind(expl:minus,full:plus,conj:plus,indtype:deg)
conj_non_expl-ind	index_ind(expl:minus,conj:plus,indtype:noind)
expl-ind	expl_ind(full:bool)
it-ind	expl_ind(full:plus,explind:it)
there-ind	expl_ind(explind:ithere)

continued on next page

continued from previous page	
ERG name	ICEBERG name
full_there-ind	expl_ind(full:plus,explind:ithere)
glbtype888	index_ind(expl:minus,full:plus)
glbtype889	index_ind(expl:minus,conj:plus)
glbtype890	index_ind(expl:minus,full:plus,conj:plus)

A.5 Sign structure

Sign structure types are those which combine syntax and semantics and argument structure to form ERG signs: local, non-local, synsem, and sign types (figure A.22).

$$\begin{bmatrix} \text{sign} & \\ \text{SYNSEM} & \begin{bmatrix} \text{synsem} & \\ \text{LOCAL} & \begin{bmatrix}\text{local}\end{bmatrix} \\ \text{NON-LOCAL} & \begin{bmatrix}\text{non-local}\end{bmatrix} \end{bmatrix} \\ \text{ARGS} & \begin{bmatrix}\text{list(sign)}\end{bmatrix} \end{bmatrix}$$

Figure A.22: ERG sign feature structure

A.5.1 Local

HPSG local types encode local syntactic and semantic information. According to [PS94], local types should contain features for grammatical category, content, and context, and should be appropriate to the `LOCAL` feature of synsem types. In the ERG, local types also include features required for coordination. The ERG filter of local types is quite

flat; there are a sequence of types used for delaying feature introduction, after which the filter fans out into a number of maximally specific sister types.

ICEBERG has implemented a parametric type *local* which captures delayed feature introduction, after which the remaining simple types have been added by ground-instance subtyping links. ICEBERG closes *local* by trivial closure over grammar-attested ground instances.

ICEBERG description

```
mod_type                   sub   [isect, scopal, disc, none].
min                        sub   [basic].
basic                      sub   [real].

avm                        sub   [local(mod:mod_type, expr:min)].

local(expr:real)           sub   [non_fin_verb, verb_participle_affix,
                                  plur_noun, pos_adj, sing_noun, non_perf,
                                  '-ly', er_comp_adj, est_super_adj,
                                  mass_noun].
non_fin_verb               sub   [psp_verb, prp_verb, bse_verb].
local(mod:none,expr:real)  sub   [psp_verb].
verb_participle_affix      sub   [prp_verb].
non_perf                   sub   [fin_verb].
fin_verb                   sub   [past_verb, pres_verb, subjunctive_verb].
pres_verb                  sub   [non_third_sg_fin_verb, third_sg_fin_verb].

local                      close trivial(grammar).
```

ICEBERG ground instances	
ERG name	ICEBERG name
int_mod_local	local(mod:isect, expr:real)
intersective_mod	local(mod:isect, expr:mintype)
local_min	local(mod:mod_type, expr:mintype)
local	local(mod:mod_type, expr:real)
	continued on next page

continued from previous page

ERG name	ICEBERG name
local_basic	local(mod:mod_type, expr:basic)
scopal_mod	local(mod:scopal, expr:mintype)
scopal_mod_local	local(mod:scopal, expr:real)
nomod_local	local(mod:none, expr:real)
-ly	-ly
bse_verb	bse_verb
er_comp_adj	er_comp_adj
est_super_adj	est_super_adj
fin_verb	fin_verb
mass_noun	mass_noun
non_fin_verb	non_fin_verb
non_perf	non_perf
non_third_sg_fin_verb	non_third_sg_fin_verb
past_verb	past_verb
plur_noun	plur_noun
pos_adj	pos_adj
pres_verb	pres_verb
prp_verb	prp_verb
psp_verb	psp_verb
sing_noun	sing_noun
subjunctive_verb	subjunctive_verb
third_sg_fin_verb	third_sg_fin_verb
verb_participle_affix	verb_participle_affix

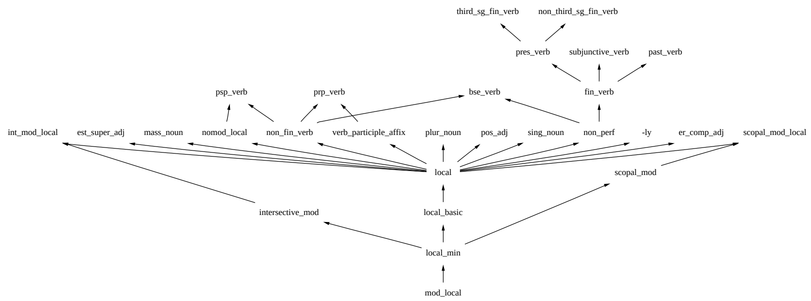

Figure A.23: ERG local types

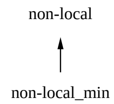

Figure A.24: ERG non-local types

A.5.2 Non-local

HPSG non-local types manage unbounded dependencies. Non-local types are appropriate to the `NON-LOCAL` feature of synsem types and introduce three features: `SLASH`, `QUE`, and `REL`. The ERG contains only two *non-local* types, one of which is used for delayed feature introduction. We have not performed any higher-order re-factoring of *non-local* types.

ICEBERG description

```
'non-local_min' sub 'non-local'.
```

A.5.3 Synsem

HPSG syntax-semantics (*synsem*) types encode all information that can be subcategorized for. Local information is encoded via a `LOCAL` feature, and non-local information is encoded via a `NON-LOCAL` feature. Although HPSG is a "system of signs", HPSG sign types simply a contain a `SYNSEM` feature, a phonology feature, and an argument list.

The ERG sysem type hierarchy is complex, containing 612 types, 266 of which are GLB completion types. The hierarchy is most succinctly described as containing mostly independent filters for each kind of part of speech and each kind of phrase. There are also a number of more general synsem types which encode number of arguments and valency considerations and which are subsumed by maximally (or near maximally) specific lexical and phrasal synsems.

Figure A.25 is a modulated sketch of the synsem filter. Each circled node refers to a set of synsem types and each uncircled node refers to a single synsem type. A link between nodes denotes an IS-A relationship between at least one element in each module. Due to time considerations, ICEBERG has not re-factored synsem types–the current ICEBERG grammar simply contains the ERG synsem hierarchy.

A.5.4 Sign

HPSG signs, as defined in [PS94], are "structured complexes of phonological, syntactic, semantic, discourse, and phrase-structural information." Signs represent independent linguistic entities–sentences, words, and phrases, and all HPSG grammar rules are defined over sign types. Every node in an HPSG parse tree is a sign.

The ERG filter of *sign* types contains four major kinds of signs: words, phrases, lexical rules, and grammar rules. As evidenced in figure A.26, phrases, words, and lexical rules are mutually independent, and each grammar rule is a speciation of some phrase type. Lexical rule types are used for derivational and inflectional morphology in order

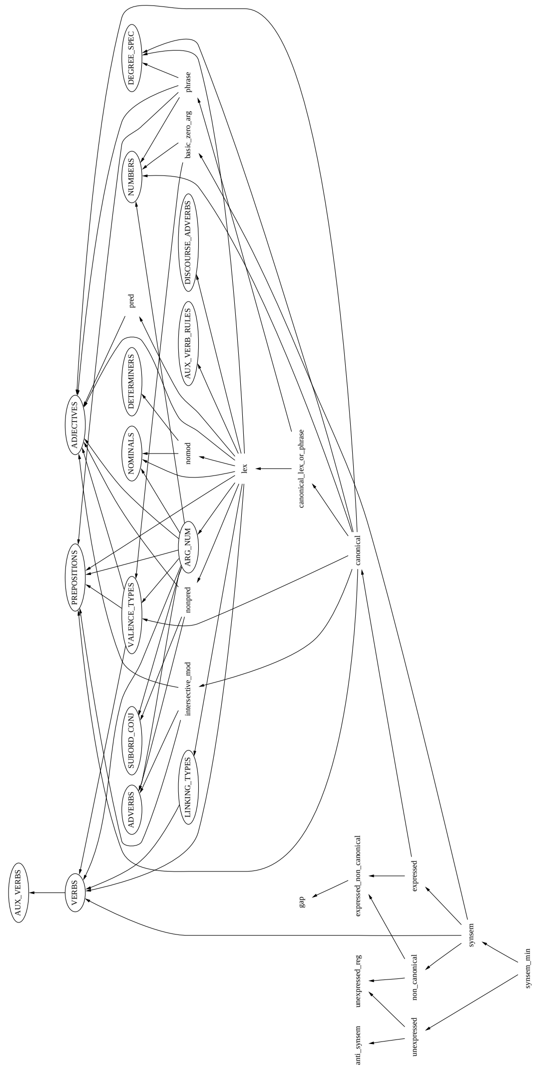

Figure A.25: ERG synsem hierarchy sketch

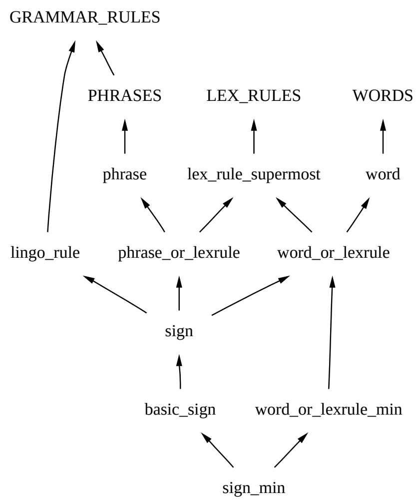

Figure A.26: ERG sign types

to capture morphological patterns and to avoid storing all inflected and derived forms of lexical entries.

In the ERG there are lexical rules for, among others, passives, gerunds, dative shifts. The lexical rule hierarchy is quite flat, implementing a very small hierarchical base structure, after which most lexical rules are immediate subtypes of *lex_rule*. For this reason ICEBERG has simply imported lexical rules from the ERG and has not attempted any higher-order re-factoring.

Phrase and word types have been re-implemented with parametric types and will be described in §A.6 and §A.7.

A.6 Phrases

Phrasal signs encode all the syntactic constructions licensed by the HPSG grammar. In particular, phrases are used by gramamr rules to determine how daughters combine in a parent structure. The ERG derives its hierarchy of phrase types from [Sag97]'s classification of English relative clauses, extending Sag's classification to handle other constructions and to provide a more fine-grained analysis of English phrases.

The phrase hierarchy is an obvious candidate for parameterization. Sag explicitly classified phrases by CLAUSALITY and HEADEDNESS dimensions, but there are numerous other dimensions of classification at work in the ERG, including arity, head position, finiteness, and argument type.

ICEBERG has implemented phrases with a hierarchy of parametric product types. At the basic level, there is a phrase type which contains six parameters and which is subtyped by 18 other parameteric types and a small number of simple types. All parametric phrase types undergo trivial closure over the set of grammar-attested ground instances. Because there are many different kinds of phrases which are not clauses, ICEBERG has broken ranks with [Sag97] by replacing the clausality dimension with a parametric type *clause* and adding other parametric types for non-clausal phrases.

A.6.1 Basic phrase classification

At the base of ICEBERG's phrase hierarchy is a parametric type, *phrase*, with parameters for arity (binary or unary), headedness, head position (initial or final), phrasality (whether the mother constituent is actually a phrase or is a lexical construction like "twenty-two"), rule status, and direction of the "key" rule argument (left or right). General cross-classified phrase types such as *headed_phrase* and *unary_phrase* are ground instances of *phrase*.

Type *phrase* is subtyped by an additional foundational phrase type, *nexus_phr*, which

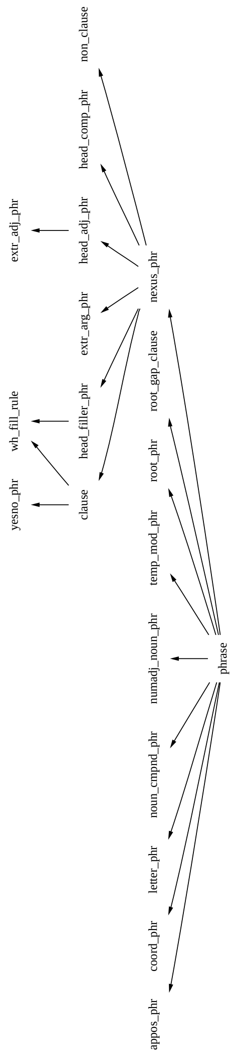

Figure A.27: ICEBERG parametric phrase types

serves as the common supertype of all phrasal constructions which support unbounded dependencies–head-filler phrases, head-complement phrases, head-subject phrases, head-specifier phrases, and clauses. Head-nexus phrases implement the wh-inheritance and slash-inheritance principles developed in [Sag97] to handle extraction and pied-piping. Though phrases supporting Sag's wh-inheritance principle must propagate both their lists of indices and lists of quantifiers, the ERG factors the wh-principle into two constraints to allow phrases to propagate either list independently. ICEBERG implements nexus phrases with a parametric type, *nexus_phr*, which subtypes phrase and introduces parameters for quantifier passing (que), index passing (rel), and slash category propagation (slash).

ICEBERG description

```
head_pos   sub   [initial, final].
arity_type sub   [unary, binary].
rule_dir   sub   [r_l, l_r].

lexroot    sub   phrase  ( headed  : bool,
                           arity   : arity_type,
                           head    : head_pos,
                           phrasal : bool,
                           rule    : bool,
                           rdir    : rule_dir ).

phrase    sub   nexus_phr( headed  : plus,
                           que     : bool,
                           rel     : bool,
                           slash   : bool).

phrase    close trivial(grammar).
nexus_phr close trivial(grammar).
```

ICEBERG ground instances	
ERG name	ICEBERG name
phrase	phrase(headed:bool)
lingo_rule	phrase(rule:plus)
binary_rule_left_to_right	phrase(rule:plus,rdir:l_r)
binary_rule_right_to_left	phrase(rule:plus,rdir:r_l)
headed_phrase	phrase(headed:plus)
non_headed_phrase	phrase(headed:minus)
binary_phrase	phrase(arity:binary)
unary_phrase	phrase(arity:unary)
binary_headed_phrase	phrase(headed:plus,arity:binary)
head_only	phrase(headed:plus,arity:unary)
head_initial	phrase(headed:plus,arity:binary,head:initial)
head_final	phrase(headed:plus,arity:binary,head:final)
phrasal	phrase(phrasal:plus)
head_nexus_que_phrase	nexus_phr(que:plus)
head_nexus_rel_phrase	nexus_phr(rel:plus)
head_nexus_phrase	nexus_phr(rel:plus,que:plus)
head_valence_phrase	nexus_phr(rel:plus,que:plus,slash:plus)

A.6.2 Appositional phrases

Appositional phrases are constructions consisting of adjacent units having identical referents. The ERG considers the first unit to be the phrasal head and classifies appositional phrases as binary head-initial phrases. ICEBERG posits a parametric subtype, *appos_phr*, of phrase which refines the *arity*, *headed*, and *head* parameters.

ICEBERG description

```
phrase    sub   appos_phr( arity  : binary,
                           headed : plus,
                           head   : initial ).

appos_phr close trivial(grammar).
```

ICEBERG ground instances	
ERG name	ICEBERG name
appos_phr	appos_phr(rule:bool)
appos_rule	appos_phr(rule:plus,rdir:r_l)

A.6.3 Clauses

Clauses are syntactic units which generally contain a subject and predicate, and which express a proposition. The ERG clauses follow the classification in [Sag97] wherein maximally specific clause types inherit from CLAUSALITY and HEADEDNESS dimensions. ICEBERG has implemented clauses as a parametric subtype of *nexus_phr* with additional parameters for relativeness, clausal type (declarative, imperative, or interrogative), and finiteness. Headedness (as well as arity, wh-inheritance, and slash-inheritance) classification is achieved via parameters inherited from *nexus_phr*. Relative clauses include a simple subtype for a head filler rule.

ICEBERG description

```
cl_type   sub   [cl_decl, cl_imp, cl_interrog].

nexus_phr sub   clause( phrasal : plus,
                        clrel   : bool,
                        cltype  : cl_type,
                        finite  : bool ).
```

```
clause   close trivial(grammar).

clause(clrel:plus,rule:plus,rdir:l_r) sub filler_head_rule_rel.
```

ICEBERG ground instances	
ERG name	ICEBERG name
clause	clause(clrel:bool)
rel_cl	clause(clrel:plus)
non_rel_clause	clause(headed:plus,rel:plus,clrel:minus)
decl	clause(headed:plus,rel:plus,clrel:minus,cltype:cl_decl)
imp	clause(headed:plus,arity:unary,que:plus,rel:plus, slash:plus,clrel:minus,cltype:cl_imp)
imperative_rule	clause(headed:plus,arity:unary,rule:plus,que:plus,rel:plus, slash:plus,clrel:minus,cltype:cl_imp)
interrog	clause(headed:plus,rel:plus,clrel:minus,cltype:cl_interrog)
wh_interrog	clause(headed:plus,arity:binary,rel:plus,clrel:minus, cltype:cl_interrog)
wh_interrog_fin	clause(headed:plus,arity:binary,rel:plus,clrel:minus, cltype:cl_interrog,finite:plus)
non_wh_rel_cl	clause(arity:unary,clrel:plus)
fin_non_wh_rel_cl	clause(arity:unary,clrel:plus,finite:plus)
inf_non_wh_rel_cl	clause(arity:unary,clrel:plus,finite:minus)
fin_non_wh_rel_rule	clause(arity:unary,rule:plus,clrel:plus,finite:plus)
inf_non_wh_rel_rule	clause(arity:unary,rule:plus,clrel:plus,finite:minus)

A.6.4 Coordinate phrases

Coordinate phrases are classified as binary non-headed phrases. ICEBERG implements coordinate phrases with a parametric type, *coord_phr*, which subtypes *phrase*, refines the

headed and *arity* parameters inherited from *phrase*, and adds parameters for coordination height and propositionality.

ICEBERG description

```
coord_height sub   [top, mid].

phrase       sub   coord_phr( arity  : binary,
                              headed : minus,
                              prop   : bool,
                              height : coord_height ).

coord_phr    close trivial(grammar).
```

ICEBERG ground instances	
ERG name	ICEBERG name
coord_phr	coord_phr(rule:bool)
top_coord_rule	coord_phr(rule:plus,rdir:r_l,height:top)
mid_coord_rule	coord_phr(rule:plus,rdir:r_l,height:mid)
top_coord_prop_rule	coord_phr(rule:plus,rdir:r_l,prop:plus,height:top)
top_coord_nonprop_rule	coord_phr(rule:plus,rdir:r_l,prop:minus,height:top)
mid_coord_prop_rule	coord_phr(rule:plus,rdir:r_l,prop:plus,height:mid)
mid_coord_nonprop_rule	coord_phr(rule:plus,rdir:r_l,prop:minus,height:mid)

A.6.5 Extracted argument phrases

Extracted argument phrases can be bifurcated into two distinct classes: extracted subject phrases and extracted complement phrases, both of which facilitate extraction of subjects and complements in unbounded dependency structures via the slash-inheritance and wh-inheritance principles. The ERG classifies extracted argument phrases as unary headed phrases and posits parameters for classification by argument type (subject or complement) and finiteness. ICEBERG has implemented extracted argument phrases as a parametric type, *extr_arg_phr*, which subtypes *nexus_phr* and introduces parameters for argument type and finiteness. In addition, finite extracted subject phrases are classified as declarative clauses.

ICEBERG description

```
arg_type     sub   [arg_subj, arg_comp].

nexus_phr    sub   extr_arg_phr( headed  : plus,
                                 arity   : unary,
                                 que     : plus,
                                 rel     : plus,
                                 slash   : plus,
                                 argtype : arg_type,
                                 finite  : bool ).

extr_arg_phr close trivial(grammar).

clause(headed:plus,rel:plus,clrel:minus,cltype:cl_decl)
             sub extr_arg_phr(argtype:arg_subj,finite:plus).
```

ICEBERG ground instances	
ERG name	ICEBERG name
extracted_arg_phrase	extr_arg_phr(argtype:arg_type)
extracted_comp_phrase	extr_arg_phr(argtype:arg_comp)
extracomp_rule	extr_arg_phr(rule:plus,argtype:arg_comp)
extracted_subj_phrase	extr_arg_phr(argtype:arg_subj)
extracted_subj_phrase_fin	extr_arg_phr(argtype:arg_subj,finite:plus)
extrasubj_fin_rule	extr_arg_phr(rule:plus,argtype:arg_subj,finite:plus)
extracted_subj_phrase_inf	extr_arg_phr(argtype:arg_subj,finite:minus)
extrasubj_inf_rule	extr_arg_phr(rule:plus,argtype:arg_subj,finite:minus)

A.6.6 Head adjunct phrases

Head-adjunct phrases implement [Sag97]'s *head-adj-phr*–for phrases with modifiers that aren't specified by their valency requirements. The ERG classifies head-adjunct phrases as headed phrases which support wh-inheritance and classifies types of head-adjuncts by head-position (initial or final), modifier type, and others. ICEBERG has implemented head-adjunct phrases as a parametric subtype, *head_adj_phr*, of *nexus_phr* which adds parameters for modifier type, 'h' versus 'n' distinctions, and to distinguish relative clause adjuncts. *head_adj_phr* also includes a parametric subtype, *extr_adj_phr*, for extracted adjuncts.

ICEBERG description

```
mod_type    sub    [isect, scopal, disc, none].
hn_type     sub    [h, n].

nexus_phr   sub    head_adj_phr( phrasal : plus,
                                 headed  : plus,
                                 rel     : plus,
                                 que     : plus,
                                 hmod    : mod_type,
                                 hn      : hn_type,
                                 relcl   : bool ).

head_adj_phr sub   extr_adj_phr( arity   : unary ).

head_adj_phr close trivial(grammar).
extr_adj_phr close trivial(grammar).
```

ICEBERG ground instances	
ERG name	ICEBERG name
head_mod_phrase	head_adj_phr(arity:arity_type)
head_mod_phrase_simple	head_adj_phr(arity:binary)
scopal_mod_phrase	head_adj_phr(arity:binary,hmod:scopal)

continued on next page

continued from previous page	
ERG name	ICEBERG name
isect_mod_phrase	head_adj_phr(arity:binary,hmod:isect)
adj_head_phrase	head_adj_phr(arity:binary,head:final)
head_adj_phrase	head_adj_phr(arity:binary,head:initial)
adj_head_int_phrase	head_adj_phr(arity:binary,head:final,hmod:isect)
adj_head_scop_phrase	head_adj_phr(arity:binary,head:final,hmod:scopal)
head_adj_int_phrase	head_adj_phr(arity:binary,head:initial,hmod:isect)
head_adj_scop_phrase	head_adj_phr(arity:binary,head:initial,hmod:scopal)
h_adj_int_phrase	head_adj_phr(arity:binary,head:initial,hmod:isect,hn:h)
n_adj_int_phrase	head_adj_phr(arity:binary,head:initial,hmod:isect,hn:n)
n_adj_relcl_phrase	head_adj_phr(arity:binary,head:initial,hmod:isect,hn:n, relcl:plus)
n_adj_redrel_phrase	head_adj_phr(arity:binary,head:initial,hmod:isect,hn:n, relcl:minus)
hadj_i_relcl_rule	head_adj_phr(arity:binary,head:initial,rule:plus,rdir:r_l, hmod:isect,hn:n,relcl:plus)
hadj_i_redrel_rule	head_adj_phr(arity:binary,head:initial,rule:plus,rdir:r_l, hmod:isect,hn:n,relcl:minus)
adjh_s_rule	head_adj_phr(arity:binary,head:final,rule:plus,rdir:l_r, hmod:scopal)
hadj_s_rule	head_adj_phr(arity:binary,head:initial,rule:plus,rdir:r_l, hmod:scopal)
hadj_i_h_rule	head_adj_phr(arity:binary,head:initial,rule:plus,rdir:r_l, hmod:isect,hn:h)
adjh_i_rule	head_adj_phr(arity:binary,head:final,rule:plus,rdir:l_r, hmod:isect)
extracted_adj_phrase	extr_adj_phr(hmod:mod_type)
extracted_adj_int_phrase	extr_adj_phr(hmod:isect)
extradj_i_rule	extr_adj_phr(rule:plus,hmod:isect)

A.6.7 Head complement phrases

Head complement phrases implement [Sag97]'s *head-comp-phr*–for phrases that have satisfied the valency requirements of their complements list. In the ERG, head-complement phrases are a specialization of nexus phrases which are head-initial and implement the slash-inheritance principle. ICEBERG implements head-complements as a parametric subtype of *nexus_phr* which refines the *head* and *slash* parameters and which adds parameters for free relatives, propositionality, and complement type. A number of ground-instance subtyping links are also added.

ICEBERG description

```
complement_type sub   [comp, marker].

nexus_phr       sub   head_comp_phr( headed   : plus,
                                     head     : initial,
                                     slash    : plus,
                                     freerel  : bool,
                                     prop     : bool,
                                     comptype : complement_type ).

head_comp_phr   close trivial(grammar).

head_comp_phr(rule:plus,rdir:l_r) sub filler_head_rule_non_wh.
head_comp_phr(headed:plus)        sub extr_arg_phr(argtype:arg_comp).
```

ICEBERG ground instances	
ERG name	ICEBERG name
head_compositional	head_comp_phr(arity:arity_type)
free_rel_phrase	head_comp_phr(arity:binary,freerel:plus)
free_rel_rule	head_comp_phr(arity:binary,rule:plus,rdir:l_r,freerel:plus)
head_comp_or_marker_phrase	head_comp_phr(arity:binary,que:plus,rel:plus, freerel:minus)
head_opt_comp_phrase	head_comp_phr(arity:unary,que:plus,rel:plus)
	continued on next page

continued from previous page	
ERG name	ICEBERG name
hoptcomp_rule	head_comp_phr(arity:unary,rule:plus,que:plus,rel:plus)
head_comp_phrase	head_comp_phr(arity:binary,que:plus,rel:plus, comptype:comp,freerel:minus)
hcomp_rule	head_comp_phr(arity:binary,rule:plus,rdir:l_r,que:plus, rel:plus,comptype:comp,freerel:minus)
head_marker_phrase	head_comp_phr(arity:binary,que:plus,rel:plus, comptype:marker,freerel:minus)
head_marker_phrase_prop	head_comp_phr(arity:binary,que:plus,rel:plus, comptype:marker,freerel:minus,prop:plus)
hmark_prop_rule	head_comp_phr(arity:binary,rule:plus,rdir:l_r,que:plus, rel:plus,comptype:marker,freerel:minus,prop:plus)
head_marker_phrase_nonprop	head_comp_phr(arity:binary,que:plus,rel:plus, comptype:marker,freerel:minus,prop:minus)
hmark_nonprop_rule	head_comp_phr(arity:binary,rule:plus,rdir:l_r,que:plus, rel:plus,comptype:marker,freerel:minus,prop:minus)

A.6.8 Head filler phrases

Head-filler phrases ground the propagation of non-local dependencies by filling the propagated empty category with a constituent. In the ERG, head-filler phrases are binary nexus phrases and, following [Sag97], come in two distinct categories, finite and non-finite. Head-filler phrases additionally serve as classifiers for interrogative-clause-filling rules, which the ERG implements for both subject, non-subject, root, and non-root clauses.

ICEBERG implements head-filler phrases as a parametric subtype of *nexus_phr* which refines the *arity* parameter and introduces a parameter for finiteness. ICEBERG then posits a parametric type *wh_fill_rule* which is a subtype of both *head_filler_phr* and *clause* and whose ground instances are non-relative interrogative-clause-filler rules. The most general head-filler type also serves as a supertype for relative clause fillers and non-wh head-filler rules are classified as finite head-final head-filler phrases. Finally, a main clause type is introduced to ensure finite wh-filler rules result in a main clause.

ICEBERG description

```
nexus_phr       sub   head_filler_phr( arity   : binary,
                                       finite  : bool ).

head_filler_phr sub   wh_fill_rule   ( headed  : plus,
                                       arity   : binary,
                                       head    : final,
                                       phrasal : plus,
                                       rule    : plus,
                                       rdir    : l_r,
                                       que     : plus,
                                       rel     : plus,
                                       clrel   : minus,
                                       cltype  : cl_interrog,
                                       finite  : bool,
                                       root    : bool,
                                       subj    : bool ).

clause          sub   wh_fill_rule.

head_filler_phr close trivial(grammar).
wh_fill_rule    close trivial(grammar).

head_filler_phr(headed:plus,head:final,que:plus,finite:plus)
                sub   filler_head_rule_non_wh.

head_filler_phr(headed:bool) sub filler_head_rule_rel.

phrase(headed:bool) sub mc_phrase.
mc_phrase           sub [wh_fill_rule(finite:plus,root:minus,subj:plus),
                         wh_fill_rule(finite:plus,root:plus)].
```

ICEBERG ground instances	
ERG name	ICEBERG name
basic_head_filler_phrase	head_filler_phr(headed:bool)
head_filler_phrase	head_filler_phr(headed:plus,head:final,que:plus)
head_filler_phrase_fin	head_filler_phr(headed:plus,head:final,que:plus,finite:plus)
head_filler_phrase_inf	head_filler_phr(headed:plus,head:final,que:plus,finite:minus)
filler_head_rule_wh_subj	wh_fill_rule(finite:plus,root:minus,subj:plus)
filler_head_rule_wh_root	wh_fill_rule(finite:plus,root:plus)
filler_head_rule_wh_nr_fin	wh_fill_rule(finite:plus,root:minus,subj:minus)
filler_head_rule_wh_nr_inf	wh_fill_rule(finite:minus,root:minus,subj:minus)

A.6.9 Head subject phrases

Head subject phrases implement [Sag97]'s *head-subj-phr*–phrases which have satisfied their subject requirements. In the ERG, head-subject phrases are a specialization of nexus phrases which are head-initial and implement the slash-inheritance principle. ICEBERG implements head-complements as a parametric subtype of *nexus_phr* which refines the *head* and *slash* parameters and which adds parameters for free relatives, propositionality, and complement type. In addition, the head-subject rule is classified as the same rule which constructs declarative clauses.

ICEBERG description

```
nexus_phr(headed:plus,arity:binary,head:final,rel:plus,que:plus,val:plus)
                 sub head_subj_phrase.

head_subj_phrase sub subjh_rule_decl.

clause(headed:plus,rel:plus,clrel:minus,cltype:cl_decl,rule:plus,rdir:r_l)
                 sub subjh_rule_decl.
```


A.6.10 Letter phrases

Letter phrases represent spelled-out words. The ERG contains both binary and unary letter phrases, as well as corresponding rules for each kind of letter phrase. ICEBERG implements letter phrases as an immediate parametric subtype of *phrase*.

ICEBERG description

```
phrase     sub   letter_phr.

letter_phr close trivial(grammar).
```

ICEBERG ground instances	
ERG name	ICEBERG name
letter_phr	letter_phr(arity:binary)
letter_rule	letter_phr(arity:binary, rule:plus)
letter_phr2	letter_phr(arity:unary)
letter_rule2	letter_phr(arity:unary,rule:plus)

A.6.11 Nominal compound phrases

Nominal compound phrases encode noun-noun compounds such as 'ceiling tile' and 'yard line'. The ERG posits several types of nominal compounds, each of which handles compounds between different kinds of nominal constructions. All nominal compounds are classified as binary head-final phrases. ICEBERG implements nominal compounds with a parametric type, *noun_cmpnd_phr* which subtypes *phrase*, adds a parameter to distinguish noun-noun compounds from np-noun compounds, and adds a parameter for optional specifiers.

ICEBERG description

```
phrase         sub   noun_cmpnd_phr( headed   : plus,
                                     arity    : binary,
                                     head     : final,
                                     np       : bool,
                                     optspec  : bool ).

noun_cmpnd_phr close trivial(grammar).
```

ICEBERG ground instances	
ERG name	ICEBERG name
n_n_cmpnd_phr	noun_cmpnd_phr(np:bool)
noun_n_cmpnd_phr	noun_cmpnd_phr(np:minus)
np_n_cmpnd_phr_2	noun_cmpnd_phr(np:plus,optspec:plus)
np_n_cmpnd_phr	noun_cmpnd_phr(np:plus,optspec:minus)
noun_n_cmpnd_rule	noun_cmpnd_phr(rule:plus,rdir:r_l,np:minus)
np_n_cmpnd_rule_2	noun_cmpnd_phr(rule:plus,rdir:r_l,np:plus,optspec:plus)
np_n_cmpnd_rule	noun_cmpnd_phr(rule:plus,rdir:r_l,np:plus,optspec:minus)

A.6.12 Non-clausal phrases

Non-clausal phrases include bare specifier phrases and head-specifier phrases. Following [Sag97], ERG bare specifiers are subtypes of the dimensional classification type *non-clause* and are classified as unary headed phrases which support wh-extraction. Head-specifier phrases support both wh-extraction and slash categories. Because ICEBERG does not have a clausality dimension, we implement non-clauses as a parametric subtype of *nexus_phr* which refines several inherited parameters and introduces parameters for existence of specifiers and to distinguish nominal heads from verbal gerunds.

ICEBERG description

```
bare_spec_type sub   [np, vger].

nexus_phr      sub   non_clause( headed    : plus,
                                 arity     : unary,
                                 phrasal   : plus,
                                 que       : plus,
                                 rel       : plus,
                                 spec      : bool,
                                 bare_spec : bare_spec_type ).

non_clause     close trivial(grammar).
```

ICEBERG ground instances	
ERG name	ICEBERG name
non_clause	non_clause(bare_spec:bare_spec_type)
bare_np_phrase	non_clause(bare_spec:np)
bare_np_rule	non_clause(rule:plus,bare_spec:np)
bare_vger_phrase	non_clause(bare_spec:vger)
bare_vger_rule	non_clause(rule:plus,bare_spec:vger)
head_spec_phrase	non_clause(arity:binary,head:final,slash:plus,spec:plus)
hspec_rule	non_clause(arity:binary,head:final,rule:plus,rdir:r_l, slash:plus,spec:plus)

A.6.13 Numerical adjective noun phrases

Numerical adjective noun phrases are unary phrases with nominal heads and a quantitative modifier. In the ERG, there is only one kind of numerical adjective noun phrase which is classified as unary and phrasal. ICEBERG implements numerical adjective phrases as a parametric subtype of *phrase* which refines the *arity* and *phrasal* parameters.

ICEBERG description

```
phrase             sub   numadj_noun_phrase( arity   : unary,
                                             phrasal : plus ).

numadj_noun_phrase close trivial(grammar).
```

ICEBERG ground instances	
ERG name	ICEBERG name
numadj_noun_phrase	numadj_noun_phrase(rule:bool)
numadj_noun_rule	numadj_noun_phrase(rule:plus)

A.6.14 Root phrases

Root phrases and root clauses are main (sentential) phrases and clauses which are not embedded in another construction. Roots always appear as the top node in a parse tree. The ERG posits a type, *root_phrase* with a subtype *root_clause*, both of which are also grammar rules. There are additional types, called root gap clauses, which encode elliptical constructions consisting of an adverb and an NP which would have normally functioned as the argument of an elided VP, such as 'not Kim' and 'maybe Sandy.'[3]

ICEBERG description

```
hpos_type       sub   [pre, post].

phrase          sub   root_phrase    ( phrasal : plus,
                                       rule    : plus,
                                       clause  : bool ).

phrase          sub   root_gap_clause( arity   : binary,
                                       phrasal : plus,
                                       mod     : hpos_type ).
```

[3]ERG source code


```
root_phrase     close trivial(grammar).
root_gap_clause close trivial(grammar).
```

ICEBERG ground instances	
ERG name	ICEBERG name
root_phrase	root_phrase(clause:bool)
root_clause	root_phrase(clause:plus)
root_gap_clause	root_gap_clause(rule:bool)
root_gap_rule_postmod	root_gap_clause(rule:plus,rdir:r_l,mod:post)
root_gap_rule_premod	root_gap_clause(rule:plus,rdir:l_r,mod:pre)

A.6.15 Temporal modifier phrases

Temporal modifier phrases are unary phrases whose argument is a constituent that adds temporal information to the head. ICEBERG implements temporal modifier phrases as a parametric subtype of *phrase* which restricts the arity and phrasal parameters.

ICEBERG description

```
phrase       sub   temp_mod_phrase( arity   : unary,
                                    phrasal : plus ).

temp_mod_phr close trivial(grammar).
```

ICEBERG ground instances	
ERG name	ICEBERG name
temp_mod_phrase	temp_mod_phrase(rule:bool)
temp_mod_rule	temp_mod_phrase(rule:plus)

A.6.16 Yes-no phrases

Yes-no phrases implement interrogatives which require a yes or no response. In the ERG, yes-no phrases are a specialization of interrogative non-relative clauses which are further cross-classified as unary headed phrases and which implement the wh-inheritance and slash-inheritance principles. ICEBERG implements yes-no phrases as a parametric type, *yesno_phrase*, which is a subtype of *clause* and which refines the necessary inherited parameters.

ICEBERG description

```
clause    sub   yesno_phr( headed : plus,
                           arity  : nary,
                           que    : plus,
                           rel    : plus,
                           val    : plus,
                           clrel  : minus,
                           cltype : cl_interrog ).

yesno_phr close trivial(grammar).
```

ICEBERG ground instances	
ERG name	ICEBERG name
yesno_phrase	yesno_phr(rule:bool)
yesno_rule	yesno_phr(rule:plus)

A.7 Words

Word signs encode the kinds of lexical entries licenced by HPSG grammars. In particular, HPSG word types are used as labels for lexicon entries, as arguments to lexical rules, and as the teminal nodes in parse trees. The ERG contains, among others, word types for

adjectives, nouns, adverbs, main verbs, auxiliary verbs, prepositions, determiners, and relativizers.

The ERG hierarchy of word types classifies all words according to nine dimension types (*nonmsg*, *nonque*, etc ...) which determine properties such as affixing, MRS messages, MRS handle location, and ability to participate in pied-piping, extraction, and coordination constructions. Each maximally specific word type subsumes some subset of these nine types.

ICEBERG has implemented the nine dimension types as parameters of a parametric product type, *word*. *word* is subtyped by 37 other parametric types (figure A.28) which refine the values of the original nine parameters and generally introduce additional parameters for further classification. *word* and all its parametric subtypes undergo the trivial closure over grammar-attested ground instances.

A.7.1 Basic word classification

Basic words implements the nine basic dimensional classifiers with the following parameters:

- slash : Is the `SLASH` list empty?
- que : Is the `QUE` list empty?
- rel : Is the `REL` list empty?
- msg : Is the `MSG` list empty?
- conj : Can the lexical item participate in coordinate structures?
- mc : Must the lexical item appear in a main clause?
- tkey : Is the local main relation identified with the relation in the top MRS handle?
- hcphr : Do head-complement structures build a phrase or another lexical item?

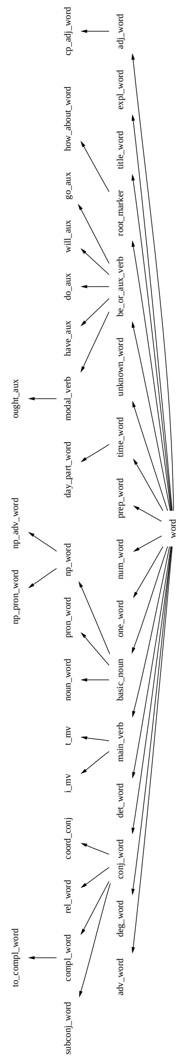

Figure A.28: ICEBERG parametric word types

- aff : Can this word type bear affixes?

ICEBERG adds two additional parameters–one parameter to encode properties of the "key" relation's label, and one parameter to handle unary branches to maximally specific word types.

ICEBERG description

```
word_or_lexrule sub   word( slash : bool,
                            que   : bool,
                            rel   : bool,
                            conj  : bool,
                            mc    : luk,
                            msg   : bool,
                            tkey  : bool,
                            hcphr : bool,
                            aff   : bool,
                            label : list,
                            le    : bool ).
word            close trivial(grammar).
```

ICEBERG ground instances	
ERG name	ICEBERG name
word	word(slash:bool)
hc-to-phr	word(hcphr:plus)
nonconj	word(conj:minus)
nonmsg	word(msg:minus)
norm_mod_no_affix_word	word(msg:plus)
nonque	word(que:minus)
nonrel	word(rel:minus)
nonslash	word(slash:minus)
mcna	word(mc:na)
topkey	word(tkey:plus)

continued on next page

continued from previous page	
ERG name	ICEBERG name
non_affix_bearing	word(aff:minus)
affix_bearing	word(aff:plus,label:list(x:syn_sign))
key_label_ne	word(label:list(x:syn_sign))

A.7.2 Adjectives

Adjectives generally modify nominals by specifying attributes or properties of a noun. The ERG classifies adjectives by, among other classifiers, transitivity, predicativity, abstractness, and as comparative or superlative. There is also a set of adjectives which modify clauses.

ICEBERG has implemented adjectives with a parametric type, *adj_word*, which inherits all parameters from *word*, refines six inherited parameters, and adds six additional parameters: transitivity, abstractness, predicativity, attributivity, adjective type (comparative or superlative), and whether the adjective is part of a multi-word expression. ICEBERG has also posited a parametric subtype, *cp_adj_word*, of *adj_word* which encodes cp-taking adjectives, and four additional simple types are introduced via ground-instance subtyping links.

ICEBERG description

```
trans_type  sub   [atrans, intrans, trans, ditrans].
adj_type    sub   [compar, superl].

word        sub   adj_word   ( tkey     : plus,
                               msg      : minus,
                               conj     : minus,
                               mc       : na,
                               hcphr    : plus,
```



```
                               label    : list(x:sign_struct),
                               multi    : bool,
                               trans    : trans_type,
                               abstract : bool,
                               adjtype  : adj_type,
                               pred     : bool,
                               attr     : bool ).

adj_word    sub   cp_adj_word( multi    : minus,
                               abstract : minus,
                               that     : bool ).

adj_word    close trivial(grammar).
cp_adj_word close trivial(grammar).

adj_word(aff:minus,abstract:minus,adjtype:compar)   sub more_adj_le.
adj_word(aff:minus,slash:minus,que:minus,rel:minus) sub meas_adj_le.
word(msg:plus,tkey:plus,conj:minus,slash:minus,
     mc:na,rel:minus,que:minus,aff:minus)           sub comparthan_adj_le.
word(aff:minus,conj:minus,mc:na,msg:minus,
     rel:minus,slash:minus)                         sub wh_adjective_le.
```

ICEBERG ground instances	
ERG name	ICEBERG name
	adj_word(abstract:bool)
adj_word	adj_word(abstract:plus)
reg_adj_word	adj_word(abstract:plus,multi:minus)
irreg_adj_word	adj_word(aff:minus,abstract:plus,multi:plus)
hcons_amalg_non_affixed_word	adj_word(aff:minus,abstract:minus)
basic_compar_adj_word	adj_word(aff:minus,abstract:minus,adjtype:compar)
basic_superl_adj_word	adj_word(aff:minus,abstract:minus,adjtype:superl)
superl_adj_le	adj_word(aff:minus,le:plus,abstract:minus, adjtype:superl)
comp_trans_adj_le	adj_word(aff:minus,le:plus,abstract:minus, adjtype:compar,trans:trans)
	continued on next page

continued from previous page	
ERG name	ICEBERG name
comp_adj_le	adj_word(aff:minus,le:plus,abstract:minus, adjtype:compar,trans:intrans)
reg_adj_equi_le	adj_word(slash:minus,que:minus,rel:minus,le:plus, attr:minus)
intrans_adj_oddsem	adj_word(slash:minus,que:minus,rel:minus,aff:minus, attr:plus)
compound_adj_le	adj_word(slash:minus,que:minus,rel:minus,aff:minus, le:plus,attr:plus)
reg_intrans_adj	adj_word(abstract:plus,attr:minus,multi:minus, trans:intrans)
attr_intrans_adj_le	adj_word(le:plus,abstract:plus,attr:plus,pred:minus, multi:minus,trans:intrans)
intrans_adj_le	adj_word(le:plus,abstract:plus,attr:minus,pred:minus, multi:minus,trans:intrans)
pred_intrans_adj_le	adj_word(le:plus,abstract:plus,attr:minus,pred:plus, multi:minus,trans:intrans)
trans_adj_le	adj_word(le:plus,abstract:plus,multi:minus,trans:trans)
irreg_trans_adj_le	adj_word(aff:minus,abstract:plus,pred:plus,multi:plus, trans:trans)
irreg_np_trans_adj_le	adj_word(aff:minus,abstract:plus,pred:minus,multi:plus, trans:trans)
irreg_pred_intrans_adj_le	adj_word(aff:minus,abstract:plus,pred:plus,multi:plus, trans:intrans)
irreg_attr_adj_le	adj_word(aff:minus,abstract:plus,attr:plus,pred:minus, multi:plus)
reg_adj_atrans_le	adj_word(aff:plus,le:plus,abstract:minus,multi:minus, trans:atrans)
	cp_adj_word(that:bool)
reg_adj_atrans_cp_le	cp_adj_word(aff:plus,le:plus,trans:atrans)
reg_adj_atrans_cp_word	cp_adj_word(aff:plus,trans:atrans)
reg_adj_atrans_that_cp_le	cp_adj_word(aff:plus,trans:atrans,that:plus)
	continued on next page

continued from previous page	
ERG name	ICEBERG name
reg_adj_cp_word	cp_adj_word(aff:plus,trans:intrans)
reg_adj_cp_le	cp_adj_word(aff:plus,le:plus,trans:intrans)
reg_adj_that_cp_le	cp_adj_word(aff:plus,trans:intrans,that:plus)

A.7.3 Adverbs

Adverbs are broadly defined as words which modify constituents other than nouns. In the ERG there are vp-modifying adverbs, sentential-modifying adverbs, discourse adverbs, and degree specifiers. Adverbs are additionally classified as intersective or scopal, and by direction from the adverb to the constituent head.

ICEBERG has implemented adverbs with two parametric subtypes of *word*: *adv_word* and *deg_word*. *adv_word* inherits all parameters from *word*, refines three word parameters, and introduces parameters for modified constituent type, position relative to the head, modifier type (intersective, scopal, or discourse), complement optionality, and presence of an adverb specifier. *deg_word* also inherits all parameters from word, refines seven word parameters, and introduces parameters for wh-adverbs and whether the degree specifier can modify prepositions.

ICEBERG description

```
hpos_type sub  [pre, post].
mod_type  sub  [isect, scopal, disc, none].
comp_type sub  [pp, vp, cp, np, s, vp_aux, nomp, nbar, none].

word      sub  adv_word( rel   : minus,
                         conj  : minus,
                         aff   : minus,
                         pos   : hpos_type,
```

```
                          type  : mod_type,
                          comp  : comp_type,
                          copt  : bool,
                          spec  : bool ).

word     sub   deg_word( slash : minus,
                          rel   : minus,
                          conj  : minus,
                          aff   : minus,
                          mc    : na,
                          msg   : minus,
                          hcphr : plus,
                          wh    : bool,
                          prep  : bool ).

deg_word close trivial(grammar).
adv_word close trivial(grammar).

nsl_adv  syn   adv_word(que:minus,slash:minus,msg:plus,mc:na).
s_adv    syn   adv_word(que:minus,slash:minus,msg:plus).

adv_word(mc:na,slash:minus,tkey:plus)                    sub wh_adverb_le.
adv_word(msg:plus,mc:na,slash:minus,tkey:plus,que:minus) sub not_le.
```

ICEBERG ground instances

ERG name	ICEBERG name
	adv_word(pos:hpos_type)
int_vp_adverb_word	nsl_adv(tkey:plus,comp:vp,type:isect)
int_vp_adv_le	nsl_adv(tkey:plus,le:plus,comp:vp,type:isect)
int_vp_adv_post_le	nsl_adv(tkey:plus,comp:vp,pos:post,type:isect)
vp_adverb_word	nsl_adv(tkey:plus,comp:vp,type:scopal)
vp_adv_le	nsl_adv(tkey:plus,le:plus,comp:vp,type:scopal)
vp_adv_pre_le	nsl_adv(tkey:plus,comp:vp,pos:pre,type:scopal)
vp_adv_post_le	nsl_adv(tkey:plus,comp:vp,pos:post,type:scopal)
adverb_word	nsl_adv(tkey:plus,copt:plus,type:scopal)

continued on next page

continued from previous page

ERG name	ICEBERG name
adv_le	nsl_adv(tkey:plus,le:plus,copt:plus,type:scopal)
adv_pre_le	nsl_adv(tkey:plus,copt:plus,pos:pre,type:scopal)
adv_post_le	nsl_adv(tkey:plus,copt:plus,pos:post,type:scopal)
norm_mod_no_affix_notopkey	s_adv(tkey:minus,type:scopal)
s_adv_pre_le	s_adv(tkey:minus,comp:s,pos:pre,spec:plus,type:scopal)
s_adv_le	s_adv(tkey:minus,le:plus,comp:s,spec:plus,type:scopal)
s_adv_pre_word_nospec_le	s_adv(tkey:minus,comp:s,pos:pre,spec:minus, type:scopal)
s_adv_nospec_le	s_adv(tkey:minus,le:plus,comp:s,spec:minus,type:scopal)
comp_vp_adv_le	nsl_adv(tkey:plus,le:plus,comp:vp_aux,pos:post)
vp_aux_adv_le	nsl_adv(tkey:plus,le:plus,comp:vp_aux,pos:pre)
int_pp_adv_le	nsl_adv(tkey:plus,le:plus,comp:pp,type:isect)
comparison_spec_le	adv_word(slash:minus,que:minus,mc:na,msg:minus, tkey:plus,hcphr:plus,le:plus)
disc_adv_word	adv_word(que:minus,tkey:minus,hcphr:plus,comp:s, type:disc)
disc_adv_sat_le	adv_word(que:minus,tkey:minus,hcphr:plus,le:plus, comp:s,spec:minus,type:disc)
disc_adv_le	adv_word(que:minus,tkey:minus,hcphr:plus,le:plus, comp:s,spec:plus,type:disc)
disc_adv_like_le	adv_word(que:minus,tkey:plus,hcphr:plus,le:plus, type:disc)
basic_degree_spec_word	deg_word(wh:bool)
wh_degree_spec_le	deg_word(le:plus,wh:plus)
degree_spec_word	deg_word(que:minus,wh:minus)
degree_spec_le	deg_word(que:minus,le:plus,prep:plus,wh:minus)
degree_spec_noprep_le	deg_word(que:minus,le:plus,prep:minus,wh:minus)

A.7.4 Auxiliary verbs and the verb 'to be'

Auxiliary verbs accompany a main verb and generally express grammatical properties such as person, number, gender, tense, and aspect. Because the verb 'to be' may function both as an auxiliary and as a main verb, the ERG word types for 'to be' verbs and for auxiliaries are intrinsically linked, and for this reason they are presented together here.

ICEBERG implements auxiliaries and 'to be' verbs with seven parametric types: *be_or_aux_verb*, *modal_verb*, *go_aux*, *ought_aux*, *have_aux*, *do_aux*, and *will_aux*. *be_or_aux_verb* is the supertype of the other six parametric auxiliary verb types and is an immediate subtype of *word*. *be_or_aux_verb* refines three parameters inherited from *word*, as well as introducing thirteen new parameters. These thirteen parameters classify auxiliaries (and 'to be' verbs) by auxiliary type ('to be' or true auxiliary), tense, aspect, mood, pernum, contraction, negation, local predicativeness and verb form, complement (main verb) predicativeness and verb form, copula type, and the `HEAD:AUX` feature value.

The auxiliary verb 'to have' is encoded by *have_aux*, which subtypes *be_or_aux_verb* and refines five inherited parameters, auxiliary verb 'to do' is encoded by *do_aux*, which subtypes *be_or_aux_verb* and refines seven inherited parameters, auxiliary verb 'will' is encoded by *will_aux*, which subtypes *be_or_aux_verb* and refines eight inherited parameters, and auxiliary verb 'to go' is encoded by *go_aux*, which subtypes *be_or_aux_verb* and refines three inherited parameters.

Lastly, modal auxiliaries are encoded by *modal_verb*, which is also a subtype of *be_or_aux_verb* and refines four inherited parameters. *modal_verb* posits one paraemtric subtype, *ought_aux*, which encodes various forms of the verb 'ought'.

ICEBERG description

```
auxtype         sub   [aux, be].
smaux           smyth auxtype.
cop_type        sub   [id, there, cop].

word            sub   be_or_aux_verb( msg    : minus,
```



```
                                       conj   : minus,
                                       mc     : na,
                                       type   : auxtype,
                                       aux    : luk,
                                       neg    : bool,
                                       cntr   : bool,
                                       pn     : pernum,
                                       tense  : tense,
                                       aspect : aspect,
                                       mood   : mood,
                                       cop    : cop_type,
                                       lvform : vform,
                                       lprd   : bool,
                                       cvform : vform,
                                       cprd   : bool ).

be_or_aux_verb sub   modal_verb     ( type   : aux,
                                       hcphr  : plus,
                                       cprd   : minus,
                                       cvform : stv(bse) ).

modal_verb     sub   ought_aux      ( lprd   : minus,
                                       lvform : fin,
                                       aspect : none,
                                       aux    : stl(-) ).

be_or_aux_verb sub   have_aux       ( type   : aux,
                                       aux    : +,
                                       hcphr  : plus,
                                       aspect : perf,
                                       cvform : pastpart(irreg:minus) ).


be_or_aux_verb sub   do_aux         ( type   : aux,
                                       hcphr  : plus,
                                       aux    : +,
                                       lprd   : minus,
                                       lvform : fin,
                                       cprd   : minus,
                                       cvform : stv(bse) ).

be_or_aux_verb sub   will_aux       ( type   : aux,
                                       hcphr  : plus,
                                       aux    : +,
                                       lprd   : minus,
```

```
                                      lvform : fin,
                                      tense  : future,
                                      cprd   : minus,
                                      cvform : stv(bse) ).

be_or_aux_verb sub   go_aux          ( hcphr  : plus,
                                      type   : aux,
                                      aux    : stl(-) ).

be_or_aux_verb close trivial(grammar).
modal_verb     close trivial(grammar).
ought_aux      close trivial(grammar).
have_aux_word  close trivial(grammar).
do_aux_word    close trivial(grammar).
will_aux       close trivial(grammar).
go_aux         close trivial(grammar).

aux_verb       syn   be_or_aux_verb(type:aux, hcphr:plus).
be_aux         syn   be_or_aux_verb(type:aux,hcphr:plus,cprd:plus,cop:cop).
be_verb        syn   be_or_aux_verb(type:be).

be_past        syn   be_verb(lprd:minus,lvform:fin,aspect:fa(none;progr),
                             mood:ind,tense:past).
be_pres        syn   be_verb(lprd:minus,lvform:fin,aspect:fa(none;progr),
                             mood:ind,tense:present).
be_cop         syn   be_verb(aux:+,type:smaux(aux,be),cop:cop,hcphr:plus,
                             cprd:plus).

be_cop_past    syn   be_cop(lprd:minus,lvform:fin,aspect:fa(none;progr),
                            mood:ind,tense:past).
be_cop_pres    syn   be_cop(lprd:minus,lvform:fin,aspect:fa(none;progr),
                            mood:ind,tense:present).

modal_pres     syn   modal_verb(aspect:none,lprd:minus,lvform:fin,
                                tense:present).

have_pres      syn   have_aux(lvform:fin,mood:ind,tense:present).
have_past      syn   have_aux(lvform:fin,mood:ind,tense:past).

do_pres        syn   do_aux(aspect:none,mood:ind,tense:present).
do_past        syn   do_aux(aspect:none,mood:ind,tense:past).
```


ICEBERG ground instances

ERG name	ICEBERG name
	be_or_aux_verb(cntr:bool)
contracted_aux_word	be_or_aux_verb(cntr:plus)
aux_verb_word_super	aux_verb(aux:luk)
aux_verb_word	aux_verb(aux:+)
inf_aux_verb_word	aux_verb(aux:stl(-))
bse_aux_verb_word	aux_verb(aux:+,cprd:minus,cvform:stv(bse), lprd:minus,lvform:fin)
prd_aux_verb_word	aux_verb(aux:+,cprd:plus)
psp_aux_verb_word	aux_verb(aux:+,cvform:pastpart(irreg:minus))
	modal_verb(lvform:vform)
fin_modal_verb_word	modal_verb(lprd:minus,lvform:fin)
pres_modal_verb_word	modal_verb(lprd:minus,lvform:fin,tense:present)
generic_modal_neg_super	modal_verb(neg:plus)
modal_verb_word	modal_pres(aux:+)
pos_modal_verb_word	modal_pres(tkey:plus,aux:+)
modal_pos_cx_le	modal_pres(tkey:plus,le:plus,aux:+,cntr:plus)
modal_pos_le	modal_pres(tkey:plus,le:plus,aux:+,cntr:minus)
modal_subj_pos_lex_ent	modal_pres(tkey:plus,aux:+,mood:modal_subj)
modal_subj_pos_cx_le	modal_pres(tkey:plus,aux:+,cntr:plus, mood:modal_subj)
modal_subj_pos_le	modal_pres(tkey:plus,aux:+,cntr:minus, mood:modal_subj)
modal_subj_neg_le	modal_pres(aux:+,mood:modal_subj,neg:plus)
modal_neg_le	modal_pres(le:plus,aux:+,neg:plus)
	go_aux(tkey:bool)

continued on next page

continued from previous page

ERG name	ICEBERG name
quasimodal_word	go_aux(tkey:minus)
quasimodal_le	go_aux(tkey:minus,le:plus)
quasimodal_psp_word	go_aux(tkey:plus,lprd:minus, lvform:pastpart(irreg:minus),tense:present)
quasimodal_psp_le	go_aux(tkey:plus,le:plus,lprd:minus, lvform:pastpart(irreg:minus),tense:present)
	ought_aux(tense:tense)
ought_verb_word	ought_aux(tense:present)
past_ought_verb_word	ought_aux(tkey:plus,tense:past)
pos_ought_verb_word	ought_aux(tkey:plus,tense:present)
ought_neg_le	ought_aux(neg:plus,tense:present)
past_ought_pos_le	ought_aux(tkey:plus,le:plus,tense:past)
ought_pos_le	ought_aux(tkey:plus,le:plus,tense:present)
have_aux_word	have_aux(lvform:vform)
have_fin	have_aux(lvform:fin)
have_aux_neg_lex_entry	have_aux(lprd:minus,lvform:fin,neg:plus)
have_aux_pos_lex_entry	have_aux(tkey:plus)
have_pres	have_pres
have_past	have_past
have_subj	have_aux(lvform:fin,mood:stm(subjunctive))
had_aux_lex_ent	have_past(tkey:plus)
had_aux_cx_le	have_past(tkey:plus,cntr:plus)
had_aux_le	have_past(tkey:plus,cntr:minus)
has_aux_lex_ent	have_pres(tkey:plus,pn:stp(threesg))
has_aux_cx_le	have_pres(tkey:plus,cntr:plus,pn:stp(threesg))

continued on next page

continued from previous page

ERG name	ICEBERG name
has_aux_le	have_pres(tkey:plus,cntr:minus,pn:stp(threesg))
have_fin_aux_lex_ent	have_pres(tkey:plus,pn:stp(non3sg))
have_fin_aux_cx_le	have_pres(tkey:plus,cntr:plus,pn:stp(non3sg))
have_fin_aux_le	have_pres(tkey:plus,cntr:minus,pn:stp(non3sg))
had_aux_subj_le	have_aux(tkey:plus,le:plus,lvform:fin, mood:stm(subjunctive))
have_bse_aux_le	have_aux(tkey:plus,le:plus,lprd:minus,lvform:stv(bse))
have_prespart_le	have_aux(tkey:plus,le:plus,lprd:minus, lvform:vpresent(prp:plus))
had_aux_subj_neg_le	have_aux(le:plus,lprd:minus,lvform:fin, mood:subjunctive,neg:plus)
had_aux_neg_le	have_past(le:plus,lprd:minus,neg:plus)
has_aux_neg_le	have_pres(le:plus,lprd:minus,neg:plus,pn:stp(threesg))
have_fin_aux_neg_le	have_pres(le:plus,lprd:minus,neg:plus,pn:stp(non3sg))
do_aux_word	do_aux(neg:bool)
do_aux_neg_word	do_aux(neg:plus)
do_fin	do_aux(neg:minus)
do_aux_neg_pres	do_pres(neg:plus)
dont_aux_neg_pres_le	do_pres(le:plus,neg:plus,pn:stp(non3sg))
doesnt_aux_neg_pres_le	do_pres(le:plus,neg:plus,pn:stp(threesg))
do_aux_neg_past_le	do_past(le:plus,neg:plus)
do_aux_neg_subj_le	do_aux(le:plus,mood:stm(subjunctive),neg:plus)
do_pres	do_pres(neg:minus)
did_aux_le	do_past(le:plus,neg:minus)
do_fin_aux_le	do_pres(le:plus,neg:minus,pn:stp(non3sg))

continued on next page

continued from previous page

ERG name	ICEBERG name
does_aux_le	do_pres(le:plus,neg:minus,pn:stp(threesg))
	will_aux(tkey:bool)
will_verb_word	will_aux(tkey:plus)
pos_will_verb_word	will_aux(tkey:plus,le:plus)
will_aux_word	will_aux(tkey:minus)
will_aux_pos_lex_e	will_aux(tkey:minus,neg:minus)
will_neg_le	will_aux(tkey:minus,le:plus,mood:ind,neg:plus)
would_aux_neg_le	will_aux(tkey:minus,le:plus,mood:modal_subj,neg:plus)
will_pos_cx_le	will_aux(tkey:minus,le:plus,cntr:plus,mood:ind, neg:minus)
will_pos_le	will_aux(tkey:minus,le:plus,cntr:minus,mood:ind, neg:minus)
would_aux_pos_cx_le	will_aux(tkey:minus,le:plus,cntr:plus,mood:modal_subj, neg:minus)
would_aux_pos_le	will_aux(tkey:minus,le:plus,cntr:minus, mood:modal_subj,neg:minus)
be_verb	be_or_aux_verb(type:be)
be_copula	be_aux(aux:+)
be_cop_neg	be_aux(aux:+,neg:plus)
be_cop_pos_generic	be_aux(aux:+,neg:minus)
be_cop_verb	be_aux(type:smaux(aux,be))
be_neg	be_verb(neg:plus)
be_id	be_verb(cop:id)
be_id_pos	be_verb(tkey:plus,cop:id,neg:minus)
be_th_cop_pos	be_verb(tkey:plus,cop:there,neg:minus)
be_id_neg	be_verb(cop:id,neg:plus)

continued on next page

continued from previous page

ERG name	ICEBERG name
be_th_cop	be_verb(cop:there)
be_th_cop_neg	be_verb(cop:there,neg:plus)
be_fin	be_verb(aspect:fa(none;progr),lprd:minus,lvform:fin)
be_past	be_past
be_pres	be_pres
be_subj	be_verb(aspect:fa(none;progr),lprd:minus,lvform:fin, mood:stm(subjunctive))
be_be_lex_entry	be_verb(aspect:fa(none;progr),lprd:minus,lvform:bse, mood:fm(ind,modal_subj))
be_id_be_le	be_verb(tkey:plus,le:plus,aspect:fa(none;progr),cop:id, lprd:minus,lvform:bse,mood:fm(ind,modal_subj))
be_th_cop_be_le	be_verb(le:plus,aspect:fa(none;progr),cop:there, lprd:minus,lvform:stv(bse),mood:fm(ind,modal_subj))
be_was_lex_entry	be_past(neg:minus,pn:stp(oneor3sg))
be_were_lex_entry	be_past(neg:minus,pn:stp(non1sg))
be_id_were_le	be_past(tkey:plus,le:plus,cop:id,neg:minus, pn:stp(non1sg))
be_th_cop_were_le	be_past(tkey:plus,le:plus,cop:there,neg:minus, pn:stp(non1sg))
be_id_was_le	be_past(tkey:plus,le:plus,cop:id,neg:minus, pn:stp(oneor3sg))
be_th_cop_was_le	be_past(tkey:plus,le:plus,cop:there,neg:minus, pn:stp(oneor3sg))
be_past_neg_lex_entry	be_past(neg:plus)
be_was_neg_contr_lex_entry	be_past(neg:plus,pn:stp(oneor3sg))
be_were_neg_contr_lex_entry	be_past(neg:plus,pn:stp(non1sg))
be_id_were_neg_le	be_past(le:plus,cop:id,neg:plus,pn:stp(non1sg))
be_th_cop_were_neg_le	be_past(le:plus,cop:there,neg:plus,pn:stp(non1sg))

continued on next page

continued from previous page

ERG name	ICEBERG name
be_id_was_neg_le	be_past(le:plus,cop:id,neg:plus,pn:stp(oneor3sg))
be_th_cop_was_neg_le	be_past(le:plus,cop:there,neg:plus,pn:stp(oneor3sg))
be_am_lex_entry	be_pres(neg:minus,pn:stp(onesg))
be_are_lex_entry	be_pres(neg:minus,pn:stp(non1sg))
be_is_lex_entry	be_pres(neg:minus,pn:stp(threesg))
be_id_am_le	be_pres(tkey:plus,cop:id,neg:minus,pn:stp(onesg))
be_id_are_le	be_pres(tkey:plus,cop:id,neg:minus,pn:stp(non1sg))
be_th_cop_are_le	be_pres(tkey:plus,cop:there,neg:minus,pn:stp(non1sg))
be_id_is_le	be_pres(tkey:plus,cntr:minus,cop:id,neg:minus, pn:stp(threesg))
be_th_cop_is_le	be_pres(tkey:plus,cntr:minus,cop:there,neg:minus, pn:stp(threesg))
be_id_is_cx_le	be_pres(tkey:plus,cntr:plus,cop:id,neg:minus, pn:stp(threesg))
be_th_cop_s_cx_le	be_pres(tkey:plus,le:plus,cntr:plus,cop:there,neg:minus)
be_am_neg_contr_lex_entry	be_pres(neg:plus,pn:stp(onesg))
be_is_neg_contr_lex_entry	be_pres(neg:plus,pn:stp(threesg))
be_are_neg_contr_lex_entry	be_pres(neg:plus,pn:stp(non1sg))
be_id_are_neg_le	be_pres(le:plus,cop:id,neg:plus,pn:stp(non1sg))
be_th_cop_are_neg_le	be_pres(le:plus,cop:there,neg:plus,pn:stp(non1sg))
be_id_is_neg_le	be_pres(le:plus,cop:id,neg:plus,pn:stp(threesg))
be_th_cop_is_neg_le	be_pres(le:plus,cop:there,neg:plus,pn:stp(threesg))
be_id_am_neg_le	be_pres(le:plus,cop:id,neg:plus,pn:stp(onesg))
be_pastpart	be_verb(lprd:minus,lvform:pastpart(irreg:minus))
be_been_lex_entry	be_verb(le:plus,lprd:minus,lvform:pastpart(irreg:minus))

continued on next page

continued from previous page

ERG name	ICEBERG name
be_id_been_le	be_verb(tkey:plus,le:plus,cop:id,lprd:minus, lvform:pastpart(irreg:minus),neg:minus)
be_th_cop_been_le	be_verb(tkey:plus,le:plus,cop:there,lprd:minus, lvform:pastpart(irreg:minus),neg:minus)
be_prespart	be_verb(lvform:vpresent(prp:plus))
be_being_lex_entry	be_verb(le:plus,lvform:vpresent(prp:plus))
be_id_being_le	be_verb(tkey:plus,le:plus,cop:id, lvform:vpresent(prp:plus),neg:minus)
be_th_cop_being_le	be_verb(tkey:plus,le:plus,cop:there, lvform:vpresent(prp:plus),neg:minus)
be_c_be_le	be_cop(aspect:fa(none;progr),lprd:minus,lvform:bse, mood:fm(ind,modal_subj),neg:minus)
be_c_were_le	be_cop_past(le:plus,neg:minus,pn:stp(non1sg))
be_c_was_le	be_cop_past(le:plus,neg:minus,pn:stp(oneor3sg))
be_c_were_neg_le	be_cop_past(le:plus,neg:plus,pn:stp(non1sg))
be_c_was_neg_le	be_cop_past(le:plus,neg:plus,pn:stp(oneor3sg))
be_c_been_le	be_cop(le:plus,lprd:minus,lvform:pastpart(irreg:minus), neg:minus)
be_c_am_le	be_cop_pres(neg:minus,pn:stp(onesg))
be_c_are_le	be_cop_pres(neg:minus,pn:stp(non1sg))
be_c_is_le	be_cop_pres(le:plus,cntr:minus,neg:minus, pn:stp(threesg))
be_c_is_cx_le	be_cop_pres(le:plus,cntr:plus,neg:minus,pn:stp(threesg))
be_c_are_neg_le	be_cop_pres(le:plus,neg:plus,pn:stp(non1sg))
be_c_is_neg_le	be_cop_pres(le:plus,neg:plus,pn:stp(threesg))
be_c_am_neg_le	be_cop_pres(le:plus,neg:plus,pn:stp(onesg))
be_c_being_le	be_cop(le:plus,lvform:vpresent(prp:plus),neg:minus)

A.7.5 Conjunctions

Conjunctions link words and constituents, generally expressing a semantic relationship between them. The ERG contains many kinds of conjunctions, including coordinating conjunctions, subordinating conjunctions such as complementizers and relativizers, and other kinds of subordinators.

ICEBERG has implemented conjunctions with six parametric types: *conj_word*, *subconj_word*, *compl_word*, *to_compl_word*, *rel_word*, and *coord_conj*. *conj_word* serves simply as a common supertype of all conjunctions and merely inherits all parameters from *word*. *subconj_word* encodes subordinating conjunctions such as *although*, *since*, and *as* by subtyping *conj_word*, refining six inherited parameters, and adding a parameter for the verb form of subordinated constituents and a parameter to distinguish *if* conjunctions. Complementizers are encoded with *compl_word* and *to_compl_word*. *compl_word* subtypes *conj_word*, refines five inherited parameters, and adds parameters for complementizer type, subordinated constituent verb form, raising, and eliding. *to_compl_word* subtypes *compl_word*, refines the complementizer type parameter *to*, and adds a parameter for propositionality. *rel_word* encodes relativizers by subtyping *conj_word*, refining five inherited parameters, and adding parameters for determiner type, freeness of the relativized clause, and adverbial relativizers. Finally, coordinating conjunctions are implemented by a parmetric type, *coord_conj*, which subtypes *conj_word* and introduces parameters for coordinating conjunction type (see §A.4.1), and x-to-y coordinations.

ICEBERG description

```
compl_type               sub   [to, for, whether, like, that].
det_type                 sub   [(simple;part), gen].
(simple;part)            sub   [part, simple].

word                     sub   conj_word.

conj_word                sub   subconj_word ( slash : minus,
                                              rel   : minus,
```



```
                                              que   : minus,
                                              conj  : minus,
                                              aff   : minus,
                                              tkey  : minus,
                                              vform : vform,
                                              if    : bool ).

conj_word                sub   compl_word   ( rel    : minus,
                                              que    : minus,
                                              conj   : minus,
                                              aff    : minus,
                                              hcphr  : plus,
                                              elided : bool,
                                              raise  : bool,
                                              type   : compl_type,
                                              vform  : vform ).

compl_word               sub   to_compl_word( type   : to,
                                              prop   : bool ).

conj_word                sub   rel_word     ( msg   : minus,
                                              conj  : minus,
                                              aff   : minus,
                                              mc    : na,
                                              hcphr : plus,
                                              type  : det_type,
                                              free  : bool,
                                              adv   : bool ).

conj_word                sub   coord_conj   ( type   : conj,
                                              xy     : bool ).

conj_word                close trivial(grammar).
subconj_word             close trivial(grammar).
compl_word               close trivial(grammar).
to_compl_word            close trivial(grammar).
coord_conj               close trivial(grammar).
rel_word                 close trivial(grammar).

compl_word(elided:bool)  sub   [compl_phrase_le].
```

ICEBERG ground instances	
ERG name	ICEBERG name
	conj_word(slash:bool)
	subconj_word(vform:vform)
subconj_word	subconj_word(vform:fv(fin;imp))
subconj_inf_le	subconj_word(mc:na,msg:plus,tkey:plus,le:plus, vform:inf)
subconj_if_le	subconj_word(le:plus,if:plus,vform:fv(fin;imp))
subconj_le	subconj_word(le:plus,if:minus,vform:fv(fin;imp))
	compl_word(elided:bool)
complementizer_word	compl_word(elided:minus)
plain_compl_word	compl_word(elided:minus,vform:inf)
whether_compl_word	compl_word(elided:minus,type:whether)
whether_c_inf_le	compl_word(le:plus,elided:minus,type:whether, vform:inf)
whether_c_fin_le	compl_word(le:plus,elided:minus,raise:plus, type:whether,vform:fin)
sor_compl_word	compl_word(elided:minus,raise:plus)
for_c_le	compl_word(le:plus,elided:minus,raise:plus,type:for, vform:fin)
like_c_le	compl_word(le:plus,elided:minus,raise:plus,type:like, vform:fin)
that_c_le	compl_word(le:plus,elided:minus,raise:plus,type:that, vform:fin)
	to_compl_word(elided:bool)
to_compl_elided_word	to_compl_word(elided:plus)
to_c_prop_elided_le	to_compl_word(le:plus,elided:plus,prop:plus)
to_c_nonprop_elided_le	to_compl_word(tkey:plus,le:plus,elided:plus,prop:minus)
to_c_prop_le	to_compl_word(le:plus,elided:minus,vform:inf,prop:plus)

continued on next page

continued from previous page

ERG name	ICEBERG name
to_c_nonprop_le	to_compl_word(le:plus,elided:minus,vform:inf, prop:minus)
to_compl_word	to_compl_word(elided:minus,vform:inf)
	coord_conj(type:conj)
	rel_word(free:bool)
free_rel_det_word_gen	rel_word(rel:minus,free:plus,type:(simple;part))
free_rel_pro_word	rel_word(rel:minus,free:plus,type:gen)
freerel_pdet_le	rel_word(rel:minus,le:plus,free:plus,type:partitive)
freerel_det_le	rel_word(rel:minus,le:plus,free:plus,type:simple)
freerel_pro_np_adv_le	rel_word(rel:minus,le:plus,adv:plus,free:plus,type:gen)
freerel_pro_np_le	rel_word(rel:minus,le:plus,adv:minus,free:plus,type:gen)
rel_word	rel_word(free:minus)
rel_adverb_le	rel_word(slash:minus,que:minus,tkey:plus,adv:plus, free:minus)
rel_pro_le	rel_word(slash:minus,que:minus,tkey:plus,free:minus, type:gen)
conj_word	coord_conj(type:conj(atomic;cnil;complex;num;phr)))
and_num_le	coord_conj(le:plus,type:num)
coord_a_le	coord_conj(le:plus,type:atomic)
coord_c_le	coord_conj(le:plus,type:complex)
x_to_y_le	coord_conj(slash:minus,que:minus,rel:minus,conj:minus, mc:na,tkey:plus,hcphr:plus,le:plus,xy:plus)

A.7.6 Determiners

Determiners are modifers of nouns which express the determination and/or quantity of the noun. Examples of English determiners include *a*, *the*, *some*, and *all*. The ERG encodes definite and indefinite determiners, partitive and genitive determiners, and determiners for mass and count nouns.

ICEBERG implements determiners with a parametric type, *det_word*, which subtypes *word*, refines five inherited parameters, and adds seven additional parameters. The additional parameters classify ICEBERG determiners by determiner type (simple, partitive, or genitive), presence of external modifiers, divisibility, person-number of modified nouns, wh-determiners, noun type (count or mass), and whether the determiner passes on the value of its head's `MODIFIED` feature.

ICEBERG description

```
det_type      sub   [(simple;part), gen].
(simple;part) sub   [part, simple].
n_type        sub   [count, mass].

word          sub   det_word( slash : minus,
                              msg   : minus,
                              conj  : minus,
                              aff   : minus,
                              mc    : na,
                              type  : det_type,
                              mod   : xmod,
                              div   : luk,
                              pn    : pernum,
                              wh    : bool,
                              ntype : n_type,
                              mpass : bool ).
det_word      close trivial(grammar).

simple_det    syn   det_word(rel:minus,hcphr:plus,type:(simple;part)).

det_word(hcphr:plus,rel:minus,type:part)         sub [pdet_unsp_le,
                                                      pdet_one_le].
det_word(hcphr:plus,rel:minus,le:plus,que:minus) sub [next_last_det_le].
```


ICEBERG ground instances

ERG name	ICEBERG name
	det_word(type:det_type)
det_word	simple_det(type:simple)
det_plm_le	simple_det(le:plus,div:+,mod:fx(lmod;rmod), type:simple)
det_word_modable	simple_det(mod:fx(lmod;rmod),type:simple)
det_wh_le	simple_det(le:plus,mod:fx(lmod;rmod),type:simple, wh:plus)
det_modable_le	simple_det(que:minus,le:plus,mod:fx(lmod;rmod), type:simple)
det_word_nonque	simple_det(que:minus,type:simple)
det_le	simple_det(que:minus,le:plus,mpass:plus,type:simple)
det_pl_le	simple_det(que:minus,le:plus,pn:stp(threepl), type:simple)
det_sm_le	simple_det(que:minus,le:plus,mod:fx(lmod;rmod), pn:stp(threesg),type:simple)
det_word_sing	simple_det(div:-,pn:stp(threesg),type:simple)
det_sing_nonque	simple_det(que:minus,div:-,pn:stp(threesg), type:simple)
det_sg_le	simple_det(que:minus,div:-,mod:fx(lmod;rmod), pn:stp(threesg),type:simple)
det_sg_nomod_le	simple_det(que:minus,div:-,mod:fx(notmod), pn:stp(threesg),type:simple)
pdet_word	simple_det(mod:fx(lmod;rmod),type:part)
pdet_word_nonque	simple_det(que:minus,mod:fx(lmod;rmod), type:part)
pdet_le	simple_det(que:minus,le:plus,mod:fx(lmod;rmod), type:part)
pdet_ms_le	simple_det(que:minus,div:+,mod:fx(lmod;rmod), pn:stp(threesg),type:part)
pdet_sg_le	simple_det(que:minus,div:-,mod:fx(lmod;rmod), pn:stp(threesg),type:part)

continued on next page

continued from previous page	
ERG name	ICEBERG name
pdet_pl_le	simple_det(que:minus,mod:fx(lmod;rmod), pn:stp(threepl),type:part)
pdet_pl_mass_le	simple_det(que:minus,div:+,mod:fx(lmod;rmod), ntype:mass,type:part)
pdet_word_pl_mass_wh	det_word(rel:minus,hcphr:plus,ntype:mass,type:part, wh:plus)
pdet_ms_wh_le	simple_det(le:plus,div:+,ntype:mass,pn:stp(threesg), type:part,wh:plus)
pdet_pl_wh_le	simple_det(le:plus,ntype:mass,pn:stp(threepl),type:part, wh:plus)
poss_word	det_word(type:gen)
nonwh_poss_word	det_word(que:minus,type:gen,wh:minus)
wh_poss_le	det_word(rel:minus,le:plus,type:gen,wh:plus)
poss_le	det_word(que:minus,rel:minus,le:plus,type:gen,wh:minus)
rel_poss_le	det_word(que:minus,rel:plus,le:plus,type:gen,wh:minus)

A.7.7 Main verbs

Main verbs carry the semantic brunt of a clause and may or may not appear with auxiliary verbs. In the ERG, main verbs are classified by their subcategorization requirements–each main verb type is the type-antecedent of a constraint specifying the value of the `SYNSEM` feature and/or the optionality of members of the `COMPS` list.

ICEBERG implements main verbs with three parametric types, *main_verb*, *i_mv*, and *t_mv*. *main_verb* subtypes *word*, refines four inherited parameters, and introduces parameters for synsem type, optionality of the first complement, and inflection. *i_mv* encodes intransitive main verbs by subtyping *main_verb* and setting inflection to negative. *t_mv* encodes transitive main verbs by subtyping *main_verb*, refining two parameters

(including setting inflection to negative), and adding a parameter for optionality of the second complement. Several simple types are introduced by ground-instance subtyping links.

ICEBERG description

```
word      sub   main_verb( msg    : minus,
                           conj   : minus,
                           mc     : na,
                           hcphr  : plus,
                           infl   : bool,
                           synsem : synsem_min,
                           hopt   : bool ).

main_verb sub   i_mv     ( infl   : minus ).
main_verb sub   t_mv     ( tkey   : plus,
                           infl   : minus,
                           topt   : bool ).

main_verb close trivial(grammar).
i_mv      close trivial(grammar).
t_mv      close trivial(grammar).

main_verb(tkey:plus,aff:minus,hopt:minus,
          ss:poss_verb)                   sub mv_poss_got_le.
main_verb(tkey:plus,aff:minus,hopt:minus,
         ss:obj_equi_non_trans_prd_verb)  sub mv_poss_got_prd_le.
t_mv(hopt:plus,topt:minus,
     ss:expl_it_subj_verb)                sub mv_expl_it_subj_like_le.
```

ICEBERG ground instances	
ERG name	ICEBERG name
	main_verb(synsem:synsem_min)
main_verb_sans_key	main_verb(infl:minus)
main_verb	main_verb(tkey:plus,infl:minus)
mv_sorb_pass_le	main_verb(le:plus,hopt:minus,infl:minus, ss:sor_verb)
	continued on next page

continued from previous page

ERG name	ICEBERG name
mv_atrans_le	main_verb(tkey:plus,infl:minus,ss:atrans_verb)
mv_unacc_le	main_verb(tkey:plus,infl:minus,ss:unacc_verb)
mv_unerg_le	main_verb(tkey:plus,infl:minus,ss:unerg_verb)
	i_mv(synsem:synsem_min)
mv_adv_le	i_mv(hopt:minus,ss:adv_verb)
mv_cp_prop_raise_key_le	i_mv(hopt:minus,ss:cp_prop_raise_key_verb)
mv_anom_equi_le	i_mv(tkey:plus,hopt:plus,ss:anom_equi_verb)
mv_cp_fin_inf_non_trans_le	i_mv(tkey:plus,hopt:minus,ss:cp_fin_inf_intrans_verb)
mv_empty_prep*_intrans_le	i_mv(tkey:plus,hopt:plus,ss:empty_prep_intrans_verb)
mv_cp_prop*_non_trans_le	i_mv(tkey:plus,hopt:plus,ss:cp_prop_intrans_verb)
mv_cp_prop_non_trans_le	i_mv(tkey:plus,hopt:minus,ss:cp_prop_intrans_verb)
mv_cp_ques_non_trans_le	i_mv(tkey:plus,hopt:minus,ss:cp_ques_intrans_verb)
mv_cp_non_trans_le	i_mv(tkey:plus,le:plus,hopt:minus,ss:cp_intrans_verb)
mv_empty_prep_intrans_le	i_mv(tkey:plus,hopt:minus,ss:empty_prep_intrans_verb)
mv_np_non_trans_unacc_le	i_mv(tkey:plus,hopt:minus,ss:np_non_trans_unacc_verb)
mv_np_non_trans_le	i_mv(tkey:plus,hopt:minus,ss:np_non_trans_verb)
mv_np_trans_le	i_mv(tkey:plus,hopt:minus,ss:np_trans_verb)
mv_particle_le	i_mv(tkey:plus,hopt:minus,ss:particle_verb)
mv_poss_le	i_mv(tkey:plus,le:plus,hopt:minus,ss:poss_verb)
mv_ssr_le	i_mv(tkey:plus,hopt:minus,ss:ssr_verb)
mv_subj_equi_prp_le	i_mv(tkey:plus,hopt:minus,ss:subj_equi_prp_verb)
mv_subj_equi_le	i_mv(tkey:plus,hopt:minus,ss:subj_equi_verb)
mv_np*_non_trans_le	i_mv(tkey:plus,hopt:plus,ss:np_non_trans_verb)
mv_np*_trans_le	i_mv(tkey:plus,hopt:plus,ss:np_trans_verb)

continued on next page

continued from previous page

ERG name	ICEBERG name
mv_prd_ssr_le	i_mv(tkey:plus,hopt:minus,ss:ssr_prd_verb)
mv_subj_equi_prd_le	i_mv(tkey:plus,hopt:minus,ss:subj_equi_prd_verb)
mv_prep_intrans_le	i_mv(tkey:plus,hopt:minus,ss:prep_intrans_verb)
mv_prep_intrans_event_le	i_mv(tkey:plus,hopt:minus,ss:prep_intrans_event_verb)
mv_prep*_intrans_le	i_mv(tkey:plus,hopt:plus,ss:prep_intrans_verb)
	t_mv(synsem:synsem_min)
mv_ditrans*_only_le	t_mv(hopt:plus,ss:ditrans_only_verb,topt:plus)
mv_ditrans_only_le	t_mv(hopt:minus,ss:ditrans_only_verb,topt:minus)
mv_ditrans_le	t_mv(hopt:minus,ss:ditrans_verb,topt:plus)
mv_ditrans_opt_le	t_mv(hopt:plus,ss:ditrans_verb,topt:plus)
mv_empty_prep*_trans*_le	t_mv(hopt:plus,ss:empty_prep_trans_verb,topt:plus)
mv_empty_prep*_trans_le	t_mv(le:plus,hopt:minus,ss:empty_prep_trans_verb, topt:plus)
mv_empty_prep_trans*_le	t_mv(hopt:plus,ss:empty_prep_trans_verb,topt:minus)
mv_empty_prep_trans_le	t_mv(hopt:minus,ss:empty_prep_trans_verb,topt:minus)
mv_empty_prep_non_trans_le	t_mv(hopt:minus,ss:empty_prep_non_trans_verb, topt:minus)
mv_expl_it_subj_le	t_mv(le:plus,hopt:plus,ss:expl_it_subj_verb,topt:minus)
mv_expl_obj_cp_le	t_mv(hopt:minus,ss:expl_obj_cp_verb,topt:minus)
mv_expl_pp_inf_oeq_le	t_mv(hopt:minus,ss:expl_pp_inf_oeq_verb,topt:minus)
mv_expl_pp_inf_seq_le	t_mv(hopt:minus,ss:expl_pp_inf_seq_verb,topt:minus)
mv_np_trans_cp_le	t_mv(hopt:minus,ss:np_trans_cp_verb,topt:minus)
mv_np_trans_cp_ques_le	t_mv(hopt:plus,ss:np_trans_cp_ques_verb,topt:minus)
mv_np_prep_particle_only_le	t_mv(hopt:minus,ss:np_prep_particle_verb,topt:minus)
mv_np_comp_le	t_mv(hopt:minus,ss:np_comp_verb,topt:minus)

continued on next page

continued from previous page	
ERG name	ICEBERG name
mv_np_np_cp_le	t_mv(hopt:plus,ss:np_np_cp_verb,topt:plus)
mv_obj_equi_le	t_mv(hopt:minus,ss:obj_equi_verb,topt:minus)
mv_oeq_pp_inf_le	t_mv(hopt:minus,ss:oeq_pp_inf_verb,topt:minus)
mv_particle_np_le	t_mv(le:plus,hopt:minus,ss:particle_np_verb,topt:minus)
mv_prep_particle_np_le	t_mv(hopt:minus,ss:prep_particle_np_verb,topt:minus)
mv_particle_np_pp_to_le	t_mv(hopt:minus,ss:particle_np_pp_verb,topt:minus)
mv_particle_inf_le	t_mv(hopt:minus,ss:particle_inf_verb,topt:minus)
mv_particle_pp_le	t_mv(hopt:minus,ss:particle_pp_verb,topt:minus)
mv_particle_cp_le	t_mv(hopt:minus,ss:particle_cp_verb,topt:minus)
mv_pp_inf_ssr_le	t_mv(hopt:plus,ss:ssr_pp_inf_verb,topt:minus)
mv_pp_inf_seq_le	t_mv(hopt:plus,ss:pp_inf_seq_verb,topt:minus)
mv_pp_cp_le	t_mv(hopt:plus,ss:pp_cp_verb,topt:minus)
mv_sor_non_trans_le	t_mv(hopt:minus,ss:sor_non_trans_verb,topt:minus)
mv_sorb_le	t_mv(hopt:minus,ss:sorb_verb,topt:minus)
mv_sor_le	t_mv(hopt:minus,ss:sor_verb,topt:minus)
mv_np*_prep_trans_le	t_mv(hopt:plus,ss:prep_trans_verb,topt:minus)
mv_to*_trans_le	t_mv(hopt:minus,ss:to_trans_verb,topt:plus)
mv_particle_pp*_le	t_mv(hopt:minus,ss:particle_pp_verb,topt:plus)
mv_expl_prep_trans_le	t_mv(hopt:minus,ss:expl_prep_trans_verb,topt:minus)
mv_prep_trans_le	t_mv(hopt:minus,ss:prep_trans_verb,topt:minus)
mv_particle_prd_le	t_mv(hopt:minus,ss:particle_prd_verb,topt:minus)
mv_prdp_pp_ssr_le	t_mv(hopt:minus,ss:ssr_prdp_pp_verb,topt:plus)
mv_obj_equi_non_trans_prd_le	t_mv(le:plus,hopt:minus,ss:obj_equi_non_trans_prd_verb, topt:minus)
	continued on next page

continued from previous page	
ERG name	ICEBERG name
mv_obj_equi_prd_le	t_mv(hopt:minus,ss:obj_equi_prd_verb,topt:minus)
mv_prep*_trans_le	t_mv(hopt:minus,ss:prep_trans_verb,topt:plus)
mv_np*_prep*_trans_le	t_mv(hopt:plus,ss:prep_trans_verb,topt:plus)

A.7.8 Nominals

Nominal words generally represent persons, places, things, and ideas. Syntactically, nominals function as the subjects of verbs and as the objects of both verbs and prepositions. The ERG contains nominal word types for mass and count nouns, affixed and non-affixed nouns, nouns with and without specifiers, and both singular and plural nouns.

ICEBERG classifies nominals with six parametric types: *basic_noun*, *noun_word*, *one_word*, *np_word*, *np_pron_word*, and *np_adv_word*. *basic_noun* subtypes *word*, refines two inherited parameters, and serves as the common supertype of all ICEBERG nominal types. Type *noun_word* encodes nouns which require specifiers by subtyping *basic_noun*, refining one inherited parameter, and introducing subtypes for complement type, transitivity, noun type (count or mass), and deverbalization. Type *np_word* encodes nouns which do not require specifiers by subtyping *basic_noun*, refining four inherited parameters, and introducing parameters for number, external modifiers, and noun-phrase type (adverb or pronoun). Pronouns are encoded by *pron_word* and *np_pron_word*. *pron_word* subtypes *basic_word*, refines one inherited parameters, and introduces a parameter for unspecified pronoun types and a parameter for external modifiers. *np_pron_word* subtypes *np_word*, refines four inherited parameters, and introduces a parameter for agreement and a parameter for pronoun type. *one_word* specifically encodes various forms of the word 'one' by subtyping *word*, refining two inherited parameters, and introducing a pernum parameter. Finally, *np_adv_word* encodes adverbial nominals by subtyping *np_word*, refining three inherited parameters, and introducing a parameter for comparative adverbs. A number of simple types are added by ground instance subtyping links.

ICEBERG description

```
n_type        sub   [count, mass].
np_type       sub   [adv, pron].
num           sub   [sg, pl].
pronoun_type  sub   [deictic, poss, adv, clitic].
trans_type    sub   [atrans, intrans, trans, ditrans].
comp_type     sub   [pp, vp, cp, np, s, vp_aux, nomp, nbar, none].

word          sub   basic_noun  ( conj   : minus,
                                  mc     : na ).

basic_noun    sub   noun_word   ( msg    : minus,
                                  comp   : comp_type,
                                  trans  : trans_type,
                                  ntype  : n_type,
                                  dv     : bool ).

basic_noun    sub   pron_word   ( aff    : minus,
                                  type   : pronoun_type,
                                  xmod   : xmod ).

word          sub   one_word    ( tkey   : plus,
                                  aff    : minus,
                                  pn     : pernum ).

basic_noun    sub   np_word     ( slash  : minus,
                                  rel    : minus,
                                  aff    : minus,
                                  hcphr  : plus,
                                  num    : num,
                                  xmod   : xmod,
                                  pos    : np_type ).

np_word       sub   np_pron_word( que    : minus,
                                  msg    : minus,
                                  pos    : pron,
                                  xmod   : fx(notmod),
                                  agr    : bool,
```



```
                                  type   : prontype ).

np_word       sub   np_adv_word ( num    : sg,
                                  pos    : adv,
                                  msg    : minus,
                                  compar : bool ).

basic_noun    close trivial(grammar).
noun_word     close trivial(grammar).
pron_word     close trivial(grammar).
one_word      close trivial(grammar).
np_word       close trivial(grammar).
np_pron_word  close trivial(grammar).
np_adv_word   close trivial(grammar).

aff_noun      syn   noun_word(tkey:plus,hcphr:plus,aff:plus,
                              label:list(x:syn_sign)).
noaff_noun    syn   noun_word(tkey:plus,hcphr:plus,aff:minus).
reg_pron      syn   pron_word(slash:minus,rel:minus,hcphr:plus).

reg_pron(msg:minus)                            sub [wh_pro_le].
reg_pron(que:minus,type:poss)                  sub [poss_of_le].

aff_noun(comp:pp,ntype:count,trans:trans)      sub [noun_ppin_word,
                                                    noun_ppof_le].
aff_noun(dv:minus,trans:intrans)               sub [intr_temp_noun_le,
                                                    irreg_intr_noun_le].
noun_word(aff:minus,slash:minus,
          rel:minus,hcphr:plus)                sub [part_noun_le].

np_adv_word(le:plus)                           sub [wh_np_adv_le].
np_word(que:minus,msg:minus,xmod:fx(notmod))   sub [year_le].
np_word(num:sg,que:minus,msg:minus,le:plus)    sub [holiday_le,proper_le,
                                                    season_le].
```

ICEBERG ground instances	
ERG name	ICEBERG name
	basic_noun(slash:bool)
	noun_word(comp:comp_type)
noun_ppcomp_word	aff_noun(comp:pp,ntype:count,trans:trans)
	continued on next page

continued from previous page	
ERG name	ICEBERG name
noun_ppcomp_le	aff_noun(le:plus,comp:pp,ntype:count,trans:trans)
noun_cpcomp_le	aff_noun(le:plus,comp:cp,ntype:count,trans:trans)
noun_vpcomp_le	aff_noun(le:plus,comp:vp,ntype:count,trans:trans)
noun_npcomp_le	noun_word(slash:minus,que:minus,rel:minus,tkey:plus, le:plus,comp:np,ntype:count,trans:trans)
plurn_le	noaff_noun(le:plus,dv:minus,ntype:count)
deverbal_noun_trans_word	aff_noun(dv:plus,trans:trans)
deverbal_noun_intr_le	aff_noun(le:plus,dv:plus,ntype:count,trans:intrans)
basic_intr_noun_word	aff_noun(dv:minus,trans:intrans)
intr_noun_le	aff_noun(le:plus,dv:minus,trans:intrans)
massn_le	aff_noun(le:plus,dv:minus,ntype:mass)
massn_ppcomp_le	aff_noun(comp:pp,dv:minus,ntype:mass,trans:trans)
deverbal_massn_le	noaff_noun(dv:plus,ntype:mass,trans:intrans)
deverbal_massn_pp_le	noaff_noun(le:plus,comp:pp,dv:plus,ntype:mass, trans:trans)
deverbal_noun_intr_plural_le	noaff_noun(dv:plus,ntype:count,trans:intrans)
	pron_word(type:pronoun_type)
poss_clitic_le	pron_word(msg:minus,le:plus,type:clitic)
deictic_pro_le	pron_word(slash:minus,que:minus,rel:minus,msg:minus, le:plus,type:deictic)
generic_pro_adv_word	reg_pron(msg:minus,type:adv)
generic_pro_adv_le	reg_pron(msg:minus,le:plus,type:adv, xmod:fx(lmod;rmod))
generic_pro_le	reg_pron(msg:minus,le:plus,type:adv,xmod:fx(notmod))
poss_pro_le	reg_pron(que:minus,msg:minus,tkey:plus,le:plus, type:poss)
one_word	one_word(pn:pernum)

continued on next page

continued from previous page	
ERG name	ICEBERG name
one_sing_le	one_word(le:plus,pn:threesg)
one_plur_le	one_word(le:plus,pn:threepl)
basic_np_word	np_word(pos:np_type)
np_word	np_word(que:minus,msg:minus,xmod:fx(notmod))
basic_np_sing_word	np_word(num:sg)
np_pl_word	np_word(que:minus,msg:minus,num:pl, xmod:fx(notmod))
np_sing_word	np_word(que:minus,msg:minus,le:plus,num:sg)
personal_pro	np_pron_word(pron:prontype)
pers_pro_le	np_pron_word(le:plus,agr:plus,type:ppro(pn:pernum))
pers_pro_noagr_le	np_pron_word(le:plus,agr:minus,type:ppro(pn:pernum))
refl_pro_le	np_pron_word(le:plus,type:refl)
recip_pro_le	np_pron_word(le:plus,type:recip)
basic_np_adv_word	np_adv_word(compar:bool)
np_adv_word	np_adv_word(que:minus)
np_comp_adv_le	np_adv_word(que:minus,compar:plus)
np_adv_le	np_adv_word(que:minus,compar:minus)

A.7.9 Numerals

Numerals appear most often as adjectives which quantitatively modify a noun. The ERG contains numeric word types for cardinal numbers, ordinal numbers, approximate numbers, and numbers either requiring or not requiring specifiers and/or complements.

ICEBERG implements numbers with a parametric type, *num_word*, which subtypes *word*, refines seven inherited parameters, and adds parameters for specifier presence, complement presence, whether the numeral is approximate, and numeral type (cardinal or ordinal).

ICEBERG description

```
num_type sub   [card, ord].

word     sub   num_word( slash  : minus,
                         rel    : minus,
                         conj   : minus,
                         aff    : minus,
                         mc     : na,
                         msg    : minus,
                         que    : minus,
                         spec   : bool,
                         comp   : bool,
                         approx : bool,
                         type   : num_type ).

num_word close trivial(grammar).
```

ICEBERG ground instances

ERG name	ICEBERG name
norm_num_word	num_word(type:num_type)
approx_unspecified_num_without_complements	num_word(tkey:plus,approx:plus, comp:minus,spec:minus)
complement_free_number	num_word(comp:minus)
complemented_number	num_word(comp:plus)
specified_num_with_complements	num_word(comp:plus,spec:plus)
specified_num_without_complements	num_word(comp:minus,spec:plus)
specified_number	num_word(spec:plus)
unspecified_num	num_word(spec:minus)
unspecified_num_with_complements	num_word(comp:plus,spec:minus)

continued on next page

continued from previous page

ERG name	ICEBERG name
unspecified_num_without_complements	num_word(tkey:plus,approx:minus, comp:minus,spec:minus)
norm_card_word	num_word(type:card)
ord_word	num_word(type:ord)
card_only_word	num_word(comp:minus,type:card)
approx_bare_unspecified_card_le	num_word(tkey:plus,le:plus,approx:plus, comp:minus,spec:minus,type:card)
bare_unspecified_card_le	num_word(tkey:plus,le:plus,approx:minus, comp:minus,spec:minus,type:card)
bare_specified_card_le	num_word(le:plus,comp:minus,spec:plus, type:card)
bare_specified_ord_le	num_word(le:plus,comp:minus,spec:plus, type:ord)
bare_unspecified_ord_le	num_word(tkey:plus,le:plus,approx:minus, comp:minus,spec:minus,type:ord)
complemented_specified_card_le	num_word(le:plus,comp:plus,spec:plus, type:card)
complemented_unspecified_card_le	num_word(le:plus,comp:plus,spec:minus, type:card)

A.7.10 Prepositions

Prepositions generally appear before a noun phrase and express the syntactic and/or semantic relation of the noun phrase to another constituent in the containing clause. The ERG contains prepositions for various kinds of complements and semantic relations.

ICEBERG implements prepositions with a parametric type, *prep_word*, which subtypes *word*, refines four inherited parameters, and introduces parameters for complement type, complement optionality, transitivity, modifier presence, subject presence, and temporal-modifying prepositions. Three simple types are added by ground-instance subtyping links.

ICEBERG description

```
comp_type  sub   [pp, vp, cp, np, s, vp_aux, nomp, nbar, none].
trans_type sub   [atrans, intrans, trans, ditrans].

word       sub   prep_word( tkey  : plus,
                            conj  : minus,
                            aff   : minus,
                            hcphr : plus,
                            comp  : comp_type,
                            copt  : bool,
                            trans : trans_type,
                            mod   : bool,
                            subj  : bool,
                            temp  : bool ).

prep_word  close trivial(grammar).

prep_word(rel:minus,que:minus,slash:minus,
          mc:na,copt:minus,temp:plus)          sub before_prep_le.
prep_word(rel:minus,que:minus,slash:minus,
          mc:na,temp:plus,trans:trans)         sub hour_prep_le.
prep_word(msg:minus,mc:na,comp:nomp,subj:minus) sub prep_nomod_of_le.
```

ICEBERG ground instances

ERG name	ICEBERG name
	prep_word(comp:comp_type)
basic_prep_word	prep_word(comp:nomp,trans:trans)
pp_idiom_le	prep_word(le:plus,trans:intrans)
prep_cp_le	prep_word(le:plus,comp:s,trans:trans)
prep_idiom_le	prep_word(le:plus,copt:minus,subj:minus,trans:trans)
prep_nbar_comp_le	prep_word(le:plus,comp:nbar,trans:trans)
prep_pp_le	prep_word(le:plus,comp:pp,trans:trans)

continued on next page

continued from previous page

ERG name	ICEBERG name
temp_prep_le	prep_word(le:plus,temp:plus)
prep_le	prep_word(le:plus,comp:nomp,copt:minus,trans:trans)
prep_no_n_mod_le	prep_word(comp:nomp,mod:minus,copt:minus, trans:trans)
prep_optcomp_le	prep_word(le:plus,comp:nomp,copt:plus,trans:trans)
reg_prep_le	prep_word(comp:nomp,copt:minus,temp:minus, trans:trans)
ditrans_prep_le	prep_word(slash:minus,que:minus,rel:minus,mc:na, le:plus,trans:ditrans)
prep_idiom_nomod_le	prep_word(mc:na,msg:minus,le:plus,comp:nomp, subj:minus)
prep_nomod_le	prep_word(mc:na,msg:minus,le:plus,comp:nomp, mod:minus)
pp_le	prep_word(slash:minus,que:minus,rel:minus,mc:na, msg:minus,trans:intrans)

A.7.11 Time words

ERG time words encode lexical items which express various temporal relations, including hours, minutes, days of the month, days of the week, months, and years.

The ERG time words appear sprinkled thoughout the type hierarchy and, in most cases, look to be afterthoughts to the hierarchy rather than full-fledged members. In order to impose some conceptual structure on these words, ICEBERG has gathered them into two parametric types: *time_word* and *day_part_word*. *time_word* subtypes *word*, refines six inherited parameters, and adds parameters for time word type, ersatz time words, and approximate times. *day_part_word* subtypes *time_word* and adds a parameter for definiteness.

ICEBERG description

```
time_type     sub   [hour, minute, ampm, dow, month,
                     month_year, dom, mealtime].

word          sub   time_word    ( aff    : minus,
                                   slash  : minus,
                                   rel    : minus,
                                   conj   : minus,
                                   mc     : na,
                                   msg    : minus,
                                   time   : time_type,
                                   ersatz : bool,
                                   approx : bool ).

time_word     sub   day_part_word( def    : bool ).

time_word     close trivial(grammar).
day_part_word close trivial(grammar).
```

ICEBERG ground instances

ERG name	ICEBERG name
	time_word(time:type_type)
abstr_hour_word	time_word(que:minus,tkey:plus,time:hour)
approx_hour_le	time_word(que:minus,tkey:plus,le:plus,approx:plus, time:hour)
hour_le	time_word(que:minus,tkey:plus,le:plus,approx:minus, time:hour)
abstr_minute_word	time_word(que:minus,hcphr:plus,ersatz:plus, time:minute)
minute_le	time_word(que:minus,hcphr:plus,le:plus,approx:minus, ersatz:plus,time:minute)
approx_minute_le	time_word(que:minus,hcphr:plus,le:plus,approx:plus, ersatz:plus,time:minute)
np_word_no_quant	time_word(que:minus,hcphr:plus,ersatz:plus)
am_pm_le	time_word(que:minus,hcphr:plus,le:plus,ersatz:plus, time:ampm)

continued on next page

continued from previous page

ERG name	ICEBERG name
day_of_week_le	time_word(que:minus,le:plus,time:dow)
month_le	time_word(que:minus,le:plus,time:month)
month_year_le	time_word(que:minus,le:plus,time:month_year)
day_of_month_le	time_word(que:minus,tkey:plus,hcphr:plus,le:plus, time:dom)
mealtime_le	time_word(que:minus,hcphr:plus,le:plus,time:mealtime)
	day_part_word(def:bool)
def_day_part_le	day_part_word(que:minus,tkey:plus,hcphr:plus,le:plus, def:plus)
day_part_le	day_part_word(hcphr:plus,le:plus,def:minus)

A.7.12 Unknown words

ERG unknown words encode word types for words which appear in corpus data but are not in the lexicon. There are only eight of such types, four of which ICEBERG encodes with a parametric subtype, *unknown_word*, of *word*. The four remaining unknown word types are added by ground-instance subtyping links.

ICEBERG description

```
word                     sub   unknown_word( msg   : minus,
                                             conj  : minus,
                                             mc    : na,
                                             hcphr : plus).

unknown_word             close trivial(grammar).

nale_word                syn   unknown_word(slash:minus,que:minus,rel:minus).

unknown_word(tkey:plus)  sub   [intr_noun_word_nale, intrans_adj_nale,
```

```
                                mv_np_trans_nale, mv_unerg_nale].
```

ICEBERG ground instances	
ERG name	ICEBERG name
basic_unknown_word	unknown_word(tkey:bool)
unknown_word	unknown_word(tkey:plus)
proper_nale	nale_word(tkey:minus)
adv_word_nale	nale_word(tkey:plus)

A.7.13 Miscellaneous words

This section sketches the ERG word types not captured by previous classifications. Included in this set of miscellaneous word types are root markers, imperatives, 'how about' words, explitive 'it' and 'there', and title words. The miscellaneous types also include a number of simple types which encode properties of raising but which appear to be implemented as afterthoughts to the original ERG word type hierarchy.

ICEBERG description

```
word           sub   root_marker   ( rel      : minus,
                                     que      : minus,
                                     conj     : minus,
                                     aff      : minus,
                                     hcphr    : plus ).

root_marker    sub   how_about_word( chead    : bool,
                                     copt     : bool ).

word           sub   title_word    ( aff      : minus,
                                     conj     : minus,
                                     mc       : na,
                                     msg      : minus,
                                     slash    : minus,
```



```
                                     rel      : minus,
                                     que      : minus,
                                     posthead : bool ).

word           sub   expl_word     ( aff      : minus,
                                     conj     : minus,
                                     mc       : na,
                                     msg      : minus,
                                     slash    : minus,
                                     rel      : minus,
                                     que      : minus,
                                     ind      : expl_ind(full:bool) ).

expl_word      close trivial(grammar).
title_word     close trivial(grammar).
root_marker    close trivial(grammar).
how_about_word close trivial(grammar).

root_marker(rel:minus) sub [lex_imperative].
lex_imperative         sub [dont_imp_le,lets_imp_le].

word(conj:bool)        sub [add_cont,raise_cont,hc_word].

add_cont               sub [modal_pres(tkey:plus,le:plus,aux:+),
                            modal_pres(tkey:plus,aux:+,mood:modal_subj),
                            ought_aux(tkey:plus,le:plus)].

raise_cont             sub [do_aux_word(neg:minus)].

hc_word                sub [time_word(que:minus,tkey:plus,time:hour),
                            noun_word(slash:minus,que:minus,rel:minus,
                                      tkey:plus,le:plus,comp:np,
                                      ntype:count,trans:trans) ].
```

ICEBERG ground instances	
ERG name	ICEBERG name
root_marker_word	root_marker(rel:minus)
	how_about_word(chead:bool)
how_about_n_or_p_le	how_about_word(chead:plus)

continued on next page

continued from previous page	
ERG name	ICEBERG name
how_about_vp_le	how_about_word(chead:minus,copt:plus)
how_about_s_le	how_about_word(chead:minus,copt:minus)
	title_word(posthead:bool)
title_le	title_word(le:plus,posthead:minus)
post_title_le	title_word(tkey:plus,hcphr:plus,le:plus,posthead:plus)
	expl_word(ind: expl_ind(full:bool))
expl_it_le	expl_word(le:plus,ind:expl_ind(full:plus,explind:it))
expl_there_le	expl_word(le:plus,ind:expl_ind(explind:ithere))

Bibliography

[Abn97] S.P. Abney. Stochastic attribute-value grammars. *Computational Linguistics*, 23(4):597–618, 1997.

[AIS77] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language*. Oxford University Press, 1977.

[AK84] H. Aït-Kaci. *A lattice theoretic approach to computation based on a calculus of partially ordered type structures*. PhD thesis, University of Pennsylvania, 1984.

[AKBLN89] H. Aït-Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, 1989.

[Ale64] C. Alexander. *Notes on the Synthesis of Form*. Harvard University Press, 1964.

[Ale79] C. Alexander. *A Timeless Way of Building*. Oxford University Press, 1979.

[Bac81] J. Backus. The history of Fortran I, II, and III. In R.L. Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, 1981.

[BFO02] E.M. Bender, D. Flickinger, and S. Oepen. The Grammar Matrix: An open-source starter-kit for the rapid development of cross-linguistically consistent broad-coverage precision grammars. In *Proceedings of the Workshop on

Grammar Engineering and Evaluation at the 19th International Conference on Computational Linguistics, pages 8–14, Taipei, Taiwan, 2002.

[BKNS99] M. Butt, T.H. King, M. Nino, and F. Segond. *A Grammar Writer's Cookbook*. CSLI Publications, Stanford, 1999.

[BMN97] K. Bertet, M. Morvan, and L. Nourine. Laxy completion of a partial order to the smallest lattice. In *Proceedings of the International KRUSE Symposium: Knowledge Retrieval, Use and Storage for Efficiency*, pages 72–81, 1997.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, A System of Patterns*. John Wiley & Sons Ltd, Chichester, England, 1996.

[Bra77] R. Brachman. What's in a concept: structural foundations for semantic networks. *International Journal of Man-Machine Studies*, 9(1):127–152, 1977.

[Car92] B. Carpenter. *The Logic of Typed Feature Structures*. Cambridge, 1992.

[CF00] A. Copestake and D. Flickinger. An open source grammar development environment and broad-coverage English grammar using HPSG. In *Proceedings of the Second conference on Language Resources and Evaluation*, Athens, Greece, 2000.

[CFSP99] A. Copestake, D. Flickinger, I. A. Sag, and C. Pollard. Minimal recursion semantics: An introduction. Manuscript, Stanford University, 1999.

[Cop02] A. Copestake. *Implementing Typed Feature Structure Grammars*. CSLI Publications, 2002.

[CP96] B. Carpenter and G. Penn. Efficient parsing of compiled typed attribute value logic grammars. In H. Bunt and M. Tomita, editors, *Recent Advances in Parsing Technology*, pages 145–168. Kluwer, 1996.

[CP01] B. Carpenter and G. Penn. ALE user's guide version 3.2.1, 2001. http://www.cs.toronto.edu/∼gpenn/ale.html.

[Dan02] M.W. Daniels. On a type-bases analysis of feature neutrality and the coordination of unlikes. In L. Hellan F. van Eynde and D. Beermann, editors, *Proceedings of the 8th International HPSG Conference*, Stanford, 2002. CSLI Publications.

[DEH+94] C. Doran, D. Egedi, B.A. Hockey, B. Srinivas, and M. Zaidel. XTAG system – a wide coverage grammar for English. In *Proceedings of the 15th. International Conference on Computational Linguistics (COLING 94)*, pages 922–928, Kyoto, Japan, 1994.

[Des] Design Patterns for Avionics Control Systems. http://g.oswego.edu/dl/acs/acs/acs.html.

[Dev] DevGuy's Programmers' Canvas. http://www.devguy.com/fp/programmerscanvas.

[DP02] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order, 2nd Edition*. Cambridge University Press, 2002.

[EU90] G. Erbach and H. Uszkoreit. Grammar Engineering. Problems and Prospects. Claus report #1, Computational Linguistics, Saarland University, Saarbrüken, Germany, 1990.

[Fal96] A. Fall. *Reasoning with Taxonomies*. PhD thesis, Simon Fraser University, 1996.

[FB03] D. Flickinger and E.M. Bender. Compositional semantics in a multilingual grammar resource. In *Proceedings of the ESSLLI 2003 workshop "Ideas and Strategies for Multilingual Grammar Development"*, Vienna, Austria, 2003.

[FB04] D. Flickinger and E.M. Bender. Semantic selection in a cross-linguistic framework. In S. Müller, editor, *Proceedings of the 11th International HPSG Conference, Workshop on Semantics in Grammar Engineering*, Stanford, 2004. CSLI Publications.

[FCS00] D. Flickinger, A. Copestake, and I.A. Sag. HPSG analysis of English. In W. Wahlster, editor, *Verbmobil: Foundations of speech-to-speech translation*, Artificial Intelligence, pages 321–330. Springer, Berlin, Germany, 2000.

[Fel98] C. Fellbaum. *Wordnet: An electronic lexical database*. MIT Press, 1998.

[Fli00] D. Flickinger. On building a more efficient grammar by exploiting types. *Natural Language Engineering*, 6(1):15–28, 2000.

[FY97] B. Foote and J.W. Yoder. Big ball of mud. In *Proceedings of the Fourth Conference on Pattern Languages of Programs*, Monticello, Illinois, September 1997.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[GS01] J. Ginzburg and I.A. Sag. *Interrogative Investigations: the form, meaning, and use of English Interrogatives*. CSLI Publications, Stanford, California, 2001.

[HH03] L. Hellan and P. Haugereid. NorSource - an exercise in the Matrix grammar-building design. In *Proceedings of the ESSLLI 2003 workshop "Ideas and Strategies for Multilingual Grammar Development"*, Vienna, Austria, 2003.

[Hil] Hillside.net - Home of the Patterns Library. http://www.hillside.net.

[Jes24] O. Jespersen. *The Philosophy of Grammar*. George Allen & Unwin LTD, London, 1924.

[Kay85] M. Kay. Unification in grammar. In V. Dahl and P. Saint-Dizier, editors, *Natural Language Understanding and Logic Programming*, pages 233–240. Elsevier Science Publishers, 1985.

[KB82] R.M. Kaplan and J. Bresnan. Lexical-functional grammar: A formal system for grammatical representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–281. MIT Press, 1982.

[Kel93] B. Keller. *Feature Logics, Infinitary Descriptions, and Grammar*. Number 44 in CSLI Lecture Notes. CSLI Publications, 1993.

[Kin89] P. King. *A Logical Formalism for Head-Driven Phrase Structure Grammar*. PhD thesis, University of Manchester, 1989.

[Kle91] E. Klein. Phonological data types. In S. Bird, editor, *Declarative Perspectives on Phonology*, number 7 in Edinburgh Working Papers in Cognitive Science, pages 127–138. University of Edinburgh, 1991.

[KM01] D. Klein and C.D. Manning. Distributional phrase structure induction. In *The Fifth Conference on Natural Language Learning*, 2001.

[KN03] V. Kordoni and J. Neu. Deep grammar development for Modern Greek. In *Proceedings of the ESSLLI 2003 workshop "Ideas and Strategies for Multilingual Grammar Development"*, Vienna, Austria, 2003.

[KR90] R.T. Kasper and W.C. Rounds. The logic of unification in grammar. *Linguistics and Philosophy*, 13(1):35–58, 1990.

[KS94a] H.U. Krieger and U. Schäfer. $\mathcal{TDL}$—a type description language for constraint-based grammars. In *Proceedings of the 15th International Conference on Computational Linguistics*, pages 893–899, Kyoto, Japan, 1994.

[KS94b] H.U. Krieger and U. Schäfer. $\mathcal{TDL}$—a type description language for HPSG. Part 1: Overview. Research Report RR-94-37, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1994.

[KS94c] H.U. Krieger and U. Schäfer. $\mathcal{TDL}$—a type description language for HPSG. Part 2: User manual. DFKI Document D-94-14, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1994.

[LHC01] R. Levine, T. Hukari, and M. Calcagno. Parasitic gaps in English: some overlooked cases and their theoretical implications. In P. Culicover and P. Postal, editors, *Parasitic Gaps*, pages 181–222. MIT Press, Cambridge, MA, 2001.

[LP02] R. Levy and C. Pollard. Coordination and neutralization in HPSG. In L. Hellan F. van Eynde and D. Beermann, editors, *Proceedings of the 8th International HPSG Conference*, Stanford, 2002. CSLI Publications.

[Mal00] R. Malouf. Verbal gerunds as mixed categories in HPSG. In R Borsley, editor, *The Nature and Function of Syntactic Categories*, number 32 in Syntax and Semantics, pages 133–166. Academic Press, New York, 2000.

[MK00] S. Müller and W. Kasper. HPSG analysis of German. In W. Wahlster, editor, *Verbmobil: Foundations of speech-to-speech translation*, Artificial Intelligence, pages 238–253. Springer, Berlin, Germany, 2000.

[Mos88] M.A. Moshier. *Extensions to Unification Grammar for the Description of Programming Languages*. PhD thesis, University of Michigan, 1988.

[OBC+02] S. Oepen, E.M. Bender, U. Callmeier, D. Flickinger, and M. Siegel. Parallel distributed grammar engineering for practical applications. In *Proceedings of the Workshop on Grammar Engineering and Evaluation at the 19th International Conference on Computational Linguistics*, pages 15–21, Taipei, Taiwan, 2002.

[OF98] S. Oepen and D. Flickinger. Towards systematic grammar profiling: Test suite technology 10 years after. *Journal of Computer Speech and Language (Special Issue on Evaluation)*, 12(4):411–436, 1998.

[ONK97] S. Oepen, K. Netter, and J. Klein. TSNLP - test suites for natural language processing. In J. Nerbonne, editor, *Linguistic Databases*. CSLI Publications, Stanford, CA, 1997.

[Ped] Pedagogical Patterns Project. http://www.pedagogicalpatterns.org.

[Pen98] G. Penn. Parametric types for typed attribute-value logic. In *Proceedings of 36th Annual Meeting of the Association for Computational Linguistics and the 17th International conference on Computational Linguistics*, 1998.

[Pen00] G. Penn. *The Algebraic Structure of Attributed Type Signatures*. PhD thesis, Carnegie Mellon University, May 2000.

[PH03a] G. Penn and K. Hoetmer. In search of epistemic primitives in the English Resource Grammar (or why HPSG can't live without higher-order datatypes). In Stefan Müller, editor, *Proceedings of the 10th International HPSG Conference*, pages 318–337, Stanford, 2003. CSLI Publications.

[PH03b] C. Pollard and J. Hana. Ambiguity, neutrality, and coordination in higher-order grammar. In G. Jager, P. Monachesi, G. Penn, and S. Wintner, editors, *Proceedings of Formal Grammar 2003*, pages 125–136, 2003.

[Pie02] B.C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PM90] C. Pollard and M.A. Moshier. Unifying partial descriptions of sets. In P. Hanson, editor, *Information, Language, and Cognition*, volume 1 of *Vancouver Studies in Cognitive Science*. University of British Columbia Press, 1990.

[Pol90] C. Pollard. Sorts in unification-based grammar and what they mean. In M. Pinkal and B. Gregor, editors, *Unification in Natural Language Analysis*. MIT Press, 1990.

[Pol04] C. Pollard. Type-logical HPSG. In G. Jager, P. Monachesi, G. Penn, and S. Wintner, editors, *Proceedings of Formal Grammar 2004*, pages 107–124, 2004.

[PS84] F.C.N. Pereira and S.M. Shieber. The semantics of grammar formalisms seen as computer languages. In *10th International Conference on Computational Linguistics and 22nd Annual Meeting of the Association for Computational Linguistics*, Stanford, USA, 1984.

[PS87] C. Pollard and I.A. Sag. *Information-based Syntax and Semantics*. Number 13 in CSLI Lecture Notes. CSLI Publications, 1987.

[PS94] C. Pollard and I.A. Sag. *Head Driven Phrase Structure Grammar*. University of Chicago Press, 1994.

[Qui68] M.R. Quillian. Semantic memory. In M. Minsky, editor, *Semantic Information Processing*, pages 227–270. MIT Press, 1968.

[Ric04] F. Richter. *A Mathematical Formalism for Linguistic Theories with an Application in Head-Driven Phrase Structure Grammar*. Phd thesis (2000), Eberhard-Karls-Universität Tübingen, 2004.

[Sag97] I.A. Sag. English relative clause constructions. *Journal of Linguistics*, 33(2):431–484, 1997.

[Sag03] I.A. Sag. Coordination and underspecification. In Jongbok Kim and Stephen Wechsler, editors, *Proceedings of the 9th International HPSG Conference*, Stanford, 2003. CSLI Publications.

[Sie00] M. Siegel. HPSG analysis of Japanese. In W. Wahlster, editor, *Verbmobil: Foundations of Speech-to-Speech Translation*, Artificial Intelligence, pages 264–279. Springer, Berlin, Germany, 2000.

[Smo89] G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types*. PhD thesis, Universität Kaiserslautern, 1989.