Cleaning the HPSG Garage: Some Problems and Some Proposals

Carl Pollard

July 31, 2001

1 Introduction

I'm one of those people who can't get through life—or even through a day—without lists of things I have to do. In part the lists help me keep track of things that MUST get done, or else life would quickly become very unpleasant, like taking the vegetable scraps down to the compost heap. But the lists also make reference to tasks that get put off indefinitely, like remodeling the kitchen and replacing the rotten joists under the rear deck. Even though I never actually do these things, the fact that they are on one of my lists promotes the illusion that they are on my agenda and therefore I am somehow in control of the situation. Intermediate between these two categories of task are the ones that I put off for months, even years, but get to eventually, like cleaning the garage. I'm happy to report that after more than two years, I actually DID just clean the garage, with the help of some energetic teenagers.

My way of dealing with the so-called real world carries over into my professional life, and that includes my involvement with HPSG. Specifically: for years I have kept a list of things related to HPSG that I think need to be fixed. Some of these are routine things that just involve twiddling some feature, and then there are the analogs of remodeling the kitchen, the things I will never get to, like developing a psycholinguistically realistic processing model. Of more interest are the tasks of intermediate difficulty, the analogs of cleaning the garage, ones that I have finally gotten around to doing something about. Naturally I've kept a list of these, which appears in (1)

```
(1) Problems with HPSG to stop putting off fixing

    A. The formalism requires that unsaturated valence feature values,
       like all feature structures, be totally well typed, but it is
       unclear what values their features should have; the same
       applies to the features MOD and SPEC used for syntactic
       selection of heads by modifiers and specifiers.

    B. Lexical rules are problematic, inasmuch as so-called meta-level
       (Calcagno-Pollard style) lexical rules have no adequate
```

formalization, and description-level (Meurers-style) lexical rules require, counterintuitively, that the inputs be totally well-typed (Meurers to appear).

C. Hoehle's problem: There is widespread spurious ambiguity as to whether or not two structurally identical subparts of a feature structure are token-identical.

D. Set-valued features, though needed, are difficult to integrate cleanly into the feature logic.

E. The augmentation of the feature logic to enable formulation of constraints (other than ones of equality), viz. RSRL, depends on idiosyncratic technology (the 'chains' of Richter 2000, which have been shown by Kepser (2001) to lead to undecidability of model checking).

F. Recent proposals (Dalrymple and Kaplan 2000, Daniels 2001, Levy 2001, Pollard and Levy 2001) within HPSG and LFG for dealing with coordination of unlikes and the distinction between ambiguity and neutrality/syncretism (the so-called Ingria problem, which arises in coordination constructions, free relatives, parasitic gaps, etc.) seem unduly complicated.

G. There is no straightforward way to handle constructions such as gapping, right node raising, and comparative ellipsis, which can be seen as essentially involving lambda-abstraction in the syntax.

H. The semantics of many constructions (e.g. TOO/ENOUGH constructions, WH-questions) require abstraction (Richter and Sailer 1999, Ginzburg and Sag 2001), which must then be simulated within feature logic (along with the relation of beta-reduction).

If you glance over this list, you'll notice that all the items are problems with syntax, semantics, or the interface between them, that have been around for a number of years. In this paper I will sketch you my own proposals for dealing with some of these problems. One big difference between problems like those in (1) and cleaning the garage is that after I finally DID clean the garage, it was still a garage. But the kinds of solutions I've come up for the problems in (1) are such that, once they are carried out, it's no longer quite clear whether what's left still deserves to be called HPSG.

Instead of talking about the problems piecemeal, I want to start with a rough sketch of the framework I have wound up with, and then show how the solutions to some of the problems in question fit into the framework. The sketch is necessarily rough because a lot of the pieces are not yet in place. In order to distinguish it from more recognizable variants of HPSG, I'm going to refer to this framework-in-the-making as HIGHER ORDER GRAMMAR, not because it is grammar of a higher order—well, maybe it is, but that's not the reason for the name—but rather because the description languages involved in writing grammars are higher-order logics, not feature logics like Kasper-Rounds logic or RSRL. Notice I said higher order logICS (plural), not higher order logic – that's because the language used to describe meanings is different from the one used to describe syntactic entities, in ways that I will come back to a little later. But to explain how I wound up with a framework that involves two higher-order logics, I need to backtrack a little bit.

Back before things started to get complicated, say back in the mid-1980's, HPSG feature geometry was very straightforward, as shown in (2):

```
(2) HPSG Feature Geometry, c. 1985

    +-                    -+
    |PHON list[phonstring] |
    |SYN  syn              |
    |SEM  sem              |
    +-                    -+
```

This geometry was intended to formalize the notion of a Saussurean sign, an association between two mental entities, a signifiant and a signifie, with the addition of a middle component, a syntactic entity of some kind, what is usually referred to as a "syntactic category", that determined how the sign in question interacted with other signs to form bigger signs. Of course the feature structure encoding is inessential; we could just as well encode a sign in this sense as a tuple, as in (3):

```
(3) A sign
                       <pIg, PIG, pig'>

            where pIg is a phonological entity

                  PIG  is a syntactic entity (see (20)c)

                  pig' is a semantic entity
```

In fact I'll argue below (see (14)) that actually the best way to think of a feature structure is as a tuple.

Now the notion of a sign as a triple like the one in (3) is really a very mainstream idea in formal linguistics. It was borrowed into early HPSG from my 1984 dissertation, which in turn borrowed from work in the late 1970's by Gazdar and Pullum, although they didn't call such things signs and the notation was slightly different. And I suppose Gazdar and Pullum got the idea from Montague's PTQ, where an expression is associated with a syntactic category and an expression of intensional logic. The notion of a sign in essentially this same sense has been commonplace in categorial or type-logical grammar for a long time, and I think that the idea of a grammar generating tuples in this sense is also shared by LFG and even by Jackendoff's representational modularity.

For frameworks that employ such a notion of sign, a grammar consists of:

```
(4) Architecture of sign-based grammar frameworks

    a. PHONOLOGICAL COMPONENT: specification of a universe of possible
       phonological entities

    b. SYNTACTIC COMPONENT: specification of a universe of possible
       syntactic entities (``categories'')

    c. SEMANTIC COMPONENT: specification of a universe of possible
       semantic entities

    d. LEXICON: specification of a set of lexical signs

    e. RULES: a set of schemata, each of which allows signs to be
       generated from (or deduced from) other signs.

    f. The totality of signs is recursively defined by the schemata,
       with the recursion being grounded at the lexicon.
```

Of course HPSG is constraint-based, and at first blush it might look as though the architecture sketched in (4) doesn't leave any room for constraints. Actually it does though, as shown in (5):

```
(5) Fitting constraints into sign-based grammar

    a. Purely phonological constraints are dealt with in the
       phonological component.

    b. Purely syntactic constraints (e.g. feature cooccurence
       restrictions) are dealt with in the syntactic component.

    c. Purely semantic constraints (analogs of Montague's meaning
       postulates) are dealt with in the semantic component.

    d. Interface constraints are dealt with in the schemata.
```

Of course this is inconsistent with the idea of using the same logic for all constraints, but outside of the HPSG community nobody ever seemed to be very impressed with this idea anyway.

Now the reason I mention sign-based is because higher-order grammar is a kind of sign-based grammar, and the most efficient way to describe it is by describing its components, as listed in (6).

(6) Higher order grammar is a species of sign-based grammar

a. PHONOLOGICAL COMPONENT: another indefinitely postponed task (like remodeling the kitchen). For now, let's say the phonology of a sign is a string of phnological words, and the phonological component of the grammar specifies what are the possible well-formed phonological words.

b. SYNTACTIC COMPONENT: a theory in a higher-order logic called the syntactic logic. THIS IS ONE OF THE TWO KEY INNOVATIONS OF HIGHER-ORDER GRAMMAR. The syntactic logic and the role it plays will be described in detail below.

c. SEMANTIC COMPONENT: a theory in another higher-order logic called the meaning logic. The role of the meaning logic is analogous to the role of IL in PTQ, or of Ty2. (I.e. the nonlogical propositions of the theory are essentially meaning postulates.) But rather than being an intensional logic (with its notorious foundational problems) it is a hyperintensional logic (Lappin and Pollard 1999, 2000; Fox and Lappin 2001). THE HYPERINTENSIONAL MEANING LOGIC IS THE OTHER KEY INNOVATION OF HIGHER-ORDER GRAMMAR.

d. LEXICON: a designated set of signs of the form <p, s, m> where

   i. p is a string of phonological words;
   ii. s is a term of the syntactic logic; and
   iii. m is a term of the meaning logic.

e. RULES: there is enough flexibility here to allow for rules that look like they could have been written either constraint-based grammarians or type-logical grammarians.

Of course so far this is still all very sketchy. In the remainder of this paper I will try to give the flavor of higher-order grammar by discussing some of the details with reference to the problems listed in (1). In fact, we can dispose of the last problem, Problem H, right off the bat:

```
(7) Solution to Problem H

    Go back to Montague (or closer, Gallin) and use a logic with
    lambda-abstraction to handle meaning. (But pick one that avoids
    the foundational problems of possible-worlds semantics.)  After
    all, if you get to the point where you're coding up lambda
    calculus in feature logic, the handwriting is on the wall.
```

Now, since the phonological component is one of the things being postponed indefinitely, let's skip over that and look at the syntactic component, which is the part of higher-order grammar that is best worked out so far. As stated in (6)b, this is a theory in the syntactic logic, which is a higher-order logic.

2 The Syntactic Logic

Here we give an informal sketch of the syntactic logic, discussing the types and then the terms.

2.1 The Types of the Syntactic Logic

```
(8) The types of the syntactic logic, Part 1: Primitive types and
    Bool.

    a. By a HIGHER-ORDER LOGIC, we mean a typed logic with
       lambda-abstraction such as Church's (1940) simple theory of
       types.

    b. Example: Montague's IL, but with some proof theory (Henkin 1950)
       thrown in and the up and down operators thrown out. This is
       the same thing as Gallin's (1975) Ty2, which is Church's logic
       but with two primitive types instead of just one.

    c. But, following Lambek and Scott 1986, our syntactic logic will
       have SEVERAL primitive types. These correspond to HPSG atomic
       types, e.g.  Case, Vform, Agr, etc.  We write Bool instead of t
       for the type of truth values.


(9) The types of the syntactic logic, Part 2: the product constructor

    a. Still following Lambek and Scott, we have a wider range of type
       constructors than just the exponential constructor =>. (Note:
       A => B is the standard way of writing what Montague wrote as
       <A,B>.] In particular, we allow the product constructor X.
```

b. Crucially, We allow product types indexed by any finite set, not just binary products. Thus if $\langle A_i \rangle_{i \text{ in } I}$ is an indexed family of types, its indexed product is written

$$X_{i \text{ in } I} A_i$$

If I = {0,1}, this is written simply $A_0 \times A_1$.

c. Indexed product types play the role of complex (nonatomic) feature structure types in HPSG.

Example 1. Let the index set I = {CASE, AGR}, i.e. the indices are feature names, and let A_{CASE} = Case, A_{AGR} = Agr.

Then we can define the type Noun to be $X_{i \text{ in } I} A_i$.
A more suggestive notation for this is

Noun = (CASE Case, AGR Agr)

That is, we notate indexed products using an AVM notation with types in the value positions.

Example 2.

Verb = (VFORM Vform, AUX Bool, INV Bool)

(10) The types of the syntactic logic, Part 2: the coproduct constructor +

a. As with product types, coproduct types can be indexed by an arbitrary finite set. The notation for the binary case is

$$A_0 + A_1$$

(10) b. Coproducts play the same role as partitions in HPSG type hierachies. For example, suppose Noun and Verb were the only parts of speech. Then we could define the type Head as

Head = Noun + Verb

(11) The types of the syntactic logic, Part 3: Defined subtypes.

a. For any type A and any formula (see below) phi of the syntactic logic with at most the variable x of type A free, there is a subtype of A written

{x in A: phi}

b. Partially specified feature structure types are implemented as defined subtypes of indexed products. For example, assuming (as we will below) that the type NP is the same as the type Noun, we could defined the subtype NPnom as

NPnom = {x in NP: CASE(x) = nom}

Eventually, we'll want the syntactic logic to have a model-theoretic interpretation. For reference, the set-theoretic interpretations of the type constructors are given in (12).

(12) In a set-theoretic interpretation of the syntactic logic,

i. D(Bool) = {0,1};

ii. D(A => B) is a (possibly proper) subset of the functions from D(A) to D(B). (cf. Henkin 1950, Gallin 1975.)

iii. D(A x B) is the binary cartesian product of D(A) and D(B), i.e. the set of functions f with domain {0, 1} such that f(0) is in D(A) and f(1) is in D(B).

More generally, D($X_{i \text{ in } I}$ A_i) is the set of all I-tuples,

i.e. functions with domain I such that for each i in I, f(i) is in A_i. These play the same role as typed feature structures in HPSG. (We'll come back to this soon.)

iv. D(A + B) is the disjoint union of D(A) and D(B). [Any two sets have a disjoint union; they needn't be disjoint!]

(12) v. D({x in A: phi[x]} = {a in D(A): $|\mathrm{phi}(x)|^{g}$ = 1 under the assignment g(x) = a.}

Now we have types for feature structures, but so far we don't have any feature structures. For that, we need not just the types of the logic but also the terms.

2.2 The Terms of the Syntactic Logic

(13) The terms of the syntactic logic, Part 1: the usual

a. A denumerably infinite set of variables of each type.

b. Constants, e.g. nom:Case; fin:Vform; 3sf: Agr; +,-:Bool. (In general, t:A means t is a term of type A.)

c. Terms formed from application and abstraction as usual.

d. formulas (terms of type Bool) formed from formulas using the usual logical connectives.

e. Formulas of the form s = t, where s and t are terms of the same type.

f. Term equivalence is defined as usual via lambda conversion and substitution of equivalents for equivalents (changing bound variables if necessary to avoid illicit capture). (But there are additional term quivalences involving tuples and cotuples, as described below.)

(14) The terms of the syntactic logic, part 2: tuple terms

a. Pair terms: if s:C => A and t:C => B, (s,t): C => A x B. (This means you don't have to curry if you don't want to.)

More generally, we have indexed tuple terms: if $t_i : A_i$ for all i in I, then $(t_i)_{i \text{ in } I} : X_{i \text{ in } I}\, A_i$.

(14) b. We also use an AVM-like notation for indexed tuple terms, but with terms rather than types in the value positions, e.g.

(CASE nom, AGR 3sf): Noun

(VFORM fin, AUX +, INV +): Verb

Thus indexed tuples correspond to typed feature structures in HPSG. RED ALERT: the syntactic components of signs are terms, not types! This is one of many differences between higher-order grammar and type-logical grammar.

c. This solves Hoehle's Problem ((1)c), because in the model (see (17)a) an indexed tuple term simply denotes a function, so that any two structurally isomorphic feature structures are equal.

(15) The terms of the syntactic logic, part 3: projection terms

a. For any pair of types A, B, we have the terms

$p_0 : (A \times B) \Rightarrow A$ and $p_1 : (A \times B) \Rightarrow B$

More generally, for any indexed family $\langle A_i \rangle_{i \text{ in } I}$ of types, we have, for each i in I, the term

$$p_i : (X_{i \text{ in } I}\ A_i) \Rightarrow A_i$$

b. Since complex feature structure types are implemented as indexed products, we can implement HPSG features as their projections.

Example: Since Noun = (CASE Case, AGR Agr), we have

CASE: Noun => Case

(Strictly speaking, we should have called this p_{CASE} instead of CASE, but we will consistently abuse the notation in this way.

(15) c. In addition to the usual rules for term equivalence in (13)e, there are also equivalences whose effect is to let a projection select the corresponding component of a tuple. In the binary case these are:

$$p_0((s,t)) = s \qquad\qquad p_1((s,t)) = t$$

In the general case, this amounts to extracting a feature value, e.g.

AUX((VFORM fin, AUX +, INV -)) = +

(16) The terms of the syntactic logic, part 4: cotuple terms and injection terms

a. Copair terms: if s:A => C and t:B => C, [s,t]: A + B => C. (This generalizes to indexed cotuple terms.)

b. Injection terms: for any pair of types A, B, we have terms

$$i_0 : A \Rightarrow (A + B) \quad\text{and}\quad i_1 : B \Rightarrow (A + B).$$

c. In addition to the rules for term equivalence in (13)e and (15)c, there are also equivalences whose effect is to decompose a function defined by cases:

$$[s,t](i_0) = s \qquad\qquad [s,t](i_1) = t$$

(17) Denotations of new kinds of terms

a. |(s,t)| is the vector function with domain D(C) that takes c to the function f with domain {0,1} such that f(0) = |s(c)| and f(1) = |t(c)|.

This generalizes in the obvious way to arbitrary finite indexing sets I.

b. Projection terms denote projection functions of (indexed) cartesian products onto their factors.

```
(17) c. |[s,t]| is the function with domain the disjoint union of
        D(A) and D(B) whose restriction to (the embedded copy of)
        D(A) is |s| and whose restriction to (the embedded copy of
        D(B) is |t|.

     d. Injection terms denote the injection functions (embeddings) of
        cofactors into their disjoint unions.
```

We now have the entire syntactic logic in place. As I mentioned above in (11)b, one important use of the syntactic logic is for defining subtypes.

```
(18)  Some uses of the syntactic logic

     a. Subtypes can be defined by restricting a type by an open formula,
        as in (11)b.

     b. Subtypes can be used in turn to impose purely syntactic
        constraints. Example: in English the only verbs that invert
        are finite auxiliaries. This means we should have defined
        Verb not as

                   (VFORM Vform, AUX Bool, INV Bool)

        but rather as

         {x in (VFORM  Vform, AUX Bool, INV Bool):
                   INV(x) = + -> (AUX(x) = + & VFORM(x) = fin)}

     c. The syntactic component of an instance of a grammar schema
        is a lambda-term that operates on the the syntactic components
        of the inputs to yield the syntactic component of the oputput
        as described below in (21).
```

3 Exponentials and Powertypes

Now so far it may look as though all I'm doing is coding up HPSG constructs in lambda calculus, but that's because I haven't yet used the one type constructor Church and Montague did have, namely the exponential. The exponential plays the same role relative to the product constructor that the categorial slashes play relative to the dot (tensor) product in type logical grammar. That statement can be made mathematically precise as in (19). This and similar categorical technicalities are discussed at length in my FGMOL01 paper.

(19) The exponential constructor =>

For any three types A, B, and C there is a natural isomorphism from the set of arrows from C x A to B to the set of arrows from C to (A => B. Thus we can curry and uncurry:

```
                   C x A          g = curry(f)
                     |  \
            g x id   |   \ f     f = uncurry(g)
                  A  |    \
                    \|/    _|    f(c,a) = (g(c))(a)
            (A => B) x A --> B
                        eval
```

Here we focus instead on the practical importance of the exponential. First:

(20) We use exponentials to define the analogs of HPSG's unsaturated types or categorial grammar's slash categories. For example:

a. Define S and NP to be _verb_ and _noun_ respectively.

b. Define VP to be NP => S

TVP to be NP => VP

N' to be DetP => NP, etc.

(This will be revised below to allow for coordinate arguments.)

c. Thus, e.g. the syntactic component of an unsaturated sign is a lambda-term, e.g. PIG in (3) is the term of type N'

```
lambda      (AGR 3s, CASE nom_acc)
      x:DetP
```

d. In the models exponential types denote sets of functions. This has nothing to do with strings or residuation. This is another major difference between higher-order grammar and type-logical grammar. In Curry's (1961) terminology, the syntactic logic deals strictly with tectogrammatics, not phenogrammatics.

(20) e. This solves the problem in (1)A about total-well-typing of unsaturated valence features, since we are treating valence by lambda-abstraction, not by a feature: in order to completely characterize a function, you don't have to completely characterize its argument, but only tell what type of argument the function takes!

The solution to problem (1)B is similar.

f. A possible elaboration is to model grammatical functions as different "modes" of exponential (analogous to different modes of slash in multimodal categorial grammar, e.g.

```
define VP to be NP =>    S
                  SUBJ

       N' to be DetP =>   NP
                     SPR
```

(21) We use exponential terms in the syntactic components of the grammar schemata. Simple example: a unary schema to convert a present participial phrase to a verbal gerund:

```
    <p, s:VPprp, m>
 -----------------------
    <p, F(s), G(m)>
```

where F is the syntactic term

```
lambda       lambda      (CASE nom-acc, AGR 3s)
      s:VPprp      x:DetP
```

and G is a term of the meaning logic that encodes the semantic effect of nominalization.

(assuming Montague-style rules rather than an underspecificational or glue-logical syntax-semantics interface)

(22) Set values can also be treated in terms of exponentials, since for any type A, we can define the powertype of A as

```
Pow(A) = (A => Bool)
```

In the model, these are interpreted as power sets. This solves problem (1)D.

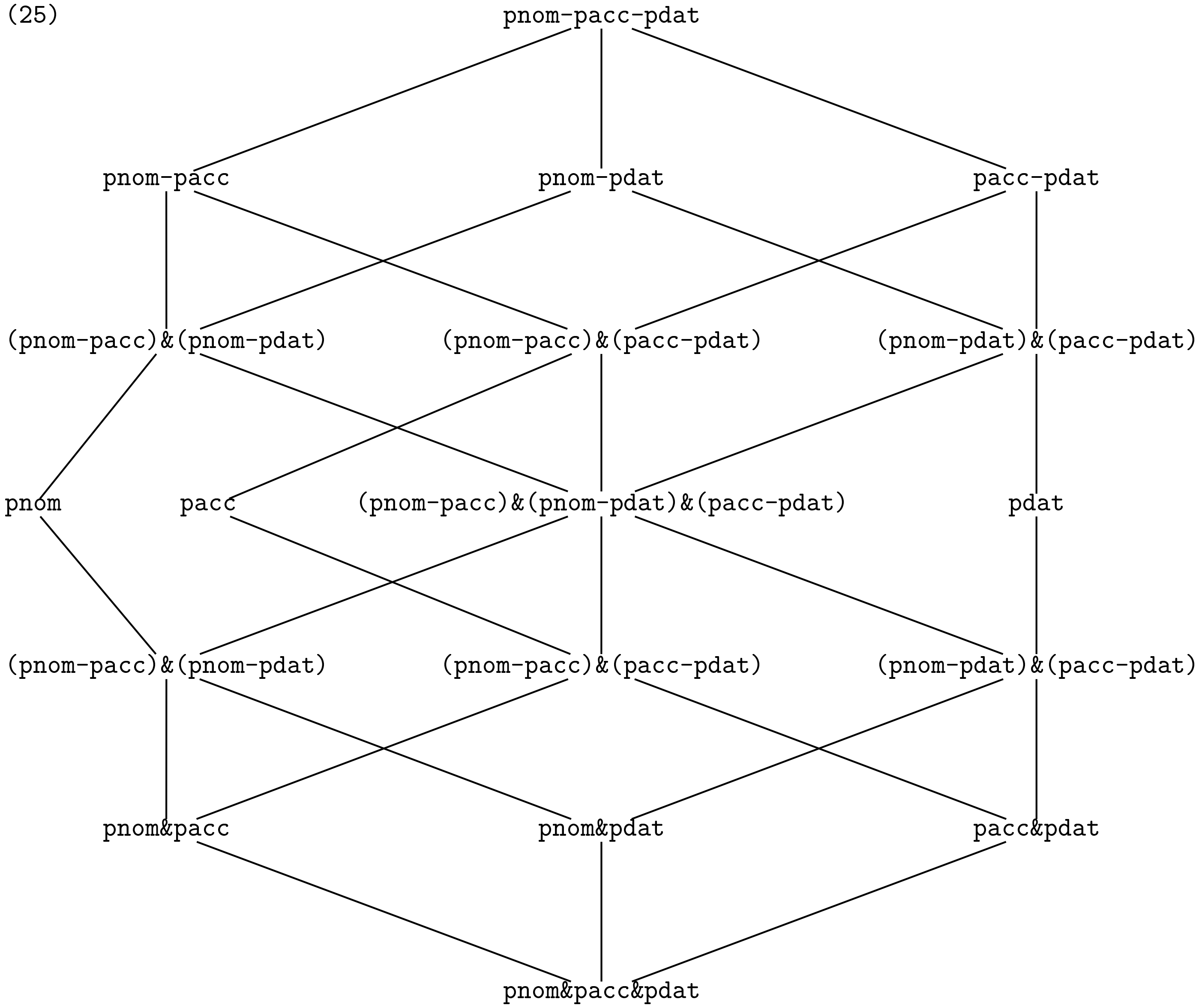
4 Neutralization and Coordination of Unlikes

Powertypes turn out to provide elegant solutions to the array of problems referred to in (1)F under the rubrics of neutralization and coordination of unlikes, exemplified in (23) and (24):

```
(23) Argument neutralization (specifically, case syncretism)

     a. Er findet und hilft FRAUEN/*MAENNER/*KINDERN. (coordination)

     b. WAS/*WER/*WEN du mir gegeben hat, ist praechtig. (free
        relative)

     c. This is someone WHO/*WHOM even close friends of __ believe __
        must be watched like a hawk. (parasitic gap)

(24) Coordination of unlikes (functor neutralization) vs. mere
     functor ambiguity (non-neutralizable polyvalence)

     a. Kim is a Republican and proud of it.

     b. *You can tuna and say that again.
```

HPSG has lagged behind type-logical grammar and LFG in confronting these problems, though Levine et al. 2001 advanced an influential proposal for case syncretism, and a couple of new accounts of neutralization and coordination using more-or-less mainstream HPSG were presented at HPSG01 (Daniels 2001 and Levy and Pollard 2001). One thing the two accounts have in common is that the analyses of case syncretization and coordination in a language with three pure cases both involve the 18-element distributive lattice in (25):

(25)



Notwithstanding the elegance of this lattice and the accounts based on it, I believe tht the resources provided by higher-order logic make a much simpler account available. Details are provided in my FGMOL01 paper; here I sketch the main ideas. To facilitate comparison with other HPSG accounts, I assume we are analyzing a language with three pure cases.

(26) We simulate Levine et al.'s analysis of syncretism as follows:

a. Instead of treating Case as as a primitive type with three members

```
Case = {nom, acc, dat}
```

we treat it as the powertype

```
Case = Pow(Bcase)
```

of a primitive type of "basic cases"

```
Bcase = {bnom, bacc, bdat}
```

b. The singleton elements of Case correspond to Levine et al.'s (and Daniels') "pure cases", e.g.

```
pnom = lambda         (x = bnom)
             x: Bcase
```

c. The nonempty nonsingleton members of Case are syncretic values, e.g.

```
acc-dat = lambda        ((x = bacc) \/ (x = bdat))
                x: Bcase
```

d. This assumes all case combinations syncretize. If not, use for Case a subtype of Pow(Bcase) that omits the non-syncretizable combinations.

(27) a. A verb that governs (say) accusative will accept not just a pure accusative but also any syncretic NP whose case value Thus we must define NPacc not as

```
{x in NP: CASE(x) = nom}
```

but rather as

```
{x in NP: CASE(x)(bacc)}
```

b. By comparison, the Levy and Pollard analysis requires selecting the logical disjunction of all the case values in the lattice in (25) contained in the principal filter generated by the element pacc.

(27) c. Our account has as consequences type equalities such as:

NPacc = NPpacc union NPnom-acc

NPnom-acc = NPnom intersect NPacc

(28) Coordination

a. The fundamental pretheoretical generalization about coordination is Wasow's Generalization (Pullum and Zwicky 1986) that, leaving aside cases of "principled resolution" (Corbett 1983), the conjuncts of a coordinate structure are subject to whatever constraints are imposed on the coordinate structure as a whole.

b. HPSG accounts (prior to Levy 2001 and Daniels 2001) are unable to account for the difference (see 24) between neutralizable polyvalency (a single sign satisfying mutually inconsistent valence requirements by taking a coordinate argument) and functor ambiguity.

c. Our account preserves the fundamental properties of Dalrymple and Kaplan's (2000) account, but:

i. presents no problem about how to formulate or interpret the property distribution schema (there is none); and

ii. does not require any stipulation about what counts as a legitmate "property" (beyond being definable in the syntactic logic).

(28) d. Our account also simulates the account of Bayer and Johnson 1995, using the (truly boolean) union operation instead of the (putatively but not really boolean) join of type-logical grammar.

e. Our account shares with those of Bayer and Johnson 1995, Levy 2001, and Daniels 2001, that neutralization and coordination are algebraic duals.

(29) Proposed Analysis of Coordination

a. If the conjuncts of a (binary) coordinate structure have syntactic components a: A and b: B and the injections into the coproduct are

$$A \xrightarrow{i} A + B \xleftarrow{j} B$$

then the syntactic component of the coordinate structure is the doubleton element of Pow(A+B)

$$\text{lambda}_{x:A+B} \ ((x = i \circ a) \vee (x = j \circ b))$$

b. In the model this is just the set {a,b}, so that the syntactic component of A REPUBLICAN AND PROUD OF IT is a doubleton set. (In LFG the f-structure would be.)

c. Thus the binary coordination schema has as its syntactic component the (type-schematized) term

$$\text{lambda}_{(y,z):A \times B} \ \text{lambda}_{x:A+B} \ ((x = i \circ y) \vee (x = j \circ z))$$

d. If a and b are identified with their respective singletons in A + B, then coordination amounts to union in A + B.

(30) Ambiguity

a. We treat ambiguous signs via pairs, e.g.

PIGS: N'3p x NP3p.

Thus ambiguity is product, while neutralization (of which syncretism is a special case) is intersection.

b. The same applies to ambiguous unsaturated signs (as in (24)b above)

CAN: (NP => VP) x (VP => VP)

(30) c. The Law of Exponents holds in our type theory just as in set theory:

For any three types A, B, C, there is a natural isomorphism

(A => C) x (B => C) ~ (A + B) => C

So we could just as well have said:

CAN: (NP + VP) => VP

that is, functor ambiguity is equivalent to disjunctive subcategorization (being of an exponential type whose antecedent type is a coproduct).

(31) Coordinate structures as syntactic arguments

a. The above treatment of functor types (e.g. VP = NP => S) does not permit coordinate structures to be arguments.

b. We therefore revise the treatment of functor types as follows:

VP = Pow(NP) => S

TVP = (Pow(NP) => VP

etc. That is, predicates syntactically select ONLY coordinate arguments; noncoordinate arguments are treated as singletons.

(31) c. The same thing applies to ambiguous functors (non-neutralizable polyvalency) such as CAN (24(b)):

((Pow(NP) + Pow(VP)) => VP)

Note that the antecedent type here is a coproduct of powertypes.

(32) Neutralizable Polyvalency

a. By contrast with functor ambiguity, where the antecedent type is a coproduct of powertypes, in cases of neutralizable polyvalency the antecedent type is the powertype of a coproduct.

(32) b. Example: making the simplifying assumption that English BE selects only NP and AP (but not PP, VPpas, or VPprp), its syntactic component is of type

```
Pow(NP + AP) => VP
```

Therefore BE takes as complement a (possibly singleton) set of syntactic entities each of which is either an NP or a VP. Thus the effect of Wasow's Generalization (in this case, the distribution of the disjunctive requirement onto each of the conjuncts) is achieved type-theoretically, with no need for a Dalrymple-Kaplan style distribution schema.

(33) Neutralizably polyvalent functors are also ambiguous.

a. In our type theory, as in set theory, for any two types A and B there is a natural embedding

```
Pow(A) + Pow(B) -> Pow(A + B)
```

b. By contravariance of the exponential relative to the antecedent, for any three types A, B, and C there is a natural embedding

```
(Pow(A + B) => C) -> ((Pow(A) + Pow(B)) => C)
```

c. Thus neutralizable polyvalency implies functor ambiguity (but not the converse).

(34) Conjoined functors with differing (but overlapping) valences

a. Example (Dyla 1984): A coordination of two Polish verbs that govern accusative and genitive respectively can take an accusative-genitive syncretic NP (e.g. KOGO) as its object:

```
Kogo Janek lubi a   Jerzy nienawidzi?
who        like and       hate
```

b. We model the syntactic components of the verbs as:

```
    l: (NPacc => VP)     n : (NPgen => VP)
```

c. As discussed in (27)c, we have the following relationship among certain subtypes of NP:

```
    NPacc-gen = NPacc intersect  NPgen;
```

let a and b be the embeddings of NPacc-gen into NP-acc and NP-gen respectively:

```
                   a               b
           NPacc <---< NPacc-gen >---> NPgen
```

d. The syntactic component of the coordinate functor is

```
lambda             lambda
      x: NPacc-gen       y: VP
                        ((y = l(a)(x) \/ (y = n(a)(x)))
```

This takes a syncretic object and returns a doubleton set of VP's.

e. More generally, the schema for binary coordination of functors has as its syntactic component the type-schematized deduction

```
    lambda
          <u,v>: (A => C) x (B => C)
          lambda
                x: A intersect B
                lambda   ((y = u(a)(x)) \/ (y = v(b)(x)))
                      y:C
```

where a and b are the emedddings of A intersect B into A and B respectively.

5 Conclusion

(35) Various notions of the syntactic logic correspond directly to linguistic notions, e.g.:

terms	CAT values of signs syntactic components of phrasal schemata
primitive types	types for values of atomic features
indexed products	complex feature structure types
projections	features
coproducts	partitions in type hierachies
exponential modes	grammatical functions
exponential terms	CAT values of unsaturated signs
powertype terms	CAT values of coordinate structures
binary products	ambiguity
intersection	neutrality/syncretism
union	coordination
antecedent type is powertype of coproduct	neutralizable polyvalence
antecedent type is coproduct of powertypes	non-neutralizable polyvalence (= functor ambiguity)

(36) To-do list

a. determining what further features and type constructors are needed to handle a more robust range of constructions (e.g. there-insertion, raising, unbounded dependency constructions, gapping, comparative ellipsis, etc.) and constraints (e.g. binding principles A and B, island constraints, crossover constraints)

b. building a robust set of phrasal schemata (i.e. developing a theory of constructions);

c. establishing a syntax-phonology interface (i.e. determining the range of phenogrammatic operations and how to specify them);

d. deciding among numerous options for the syntax-semantics interface (any of various underspecification schemes, glue semantics, etc.)

e. remodel kitchen