The Algebraic Structure of Attributed Type Signatures

Gerald B. Penn

CMU-LTI-00-164

School of Computer Science
Language Technologies Institute
Carnegie Mellon University
Pittsburgh, PA

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

To three respected scholars and friends:

Arunas L. Liulevicius,
Lawrence A. McElwee,
and
William Morrison

Abstract

Feature structures are related to frames in artificial intelligence and to record structures in many programming languages. They are widely used as a data structure for natural language processing and their formalizations often include multiple inheritance and subtyping, which allow for terser descriptions and a logical control over non-determinism during search. While it is widely known that problems in empirical linguistics often under-determine the formal devices that must be employed in their formal expression, it has never been formally proven what, if anything, is gained by using subtypes, parametric types and/or features in a feature logic. This, in turn, has hampered our understanding of how typed feature structures relate to other algebraic structures used in natural language processing and logic programming, such as systemic networks and lattices of Prolog or first-order terms. Given a fixed signature, a declaration of types and features, what kinds of information can feature structures distinguish with types relative to what they can distinguish with features? Are types, parametric or otherwise, just a convenient shorthand for bundles of features? If there is a formal trade-off, can we use that to our advantage for better "compilation" of practical large-scale grammars, when viewed as logic programs over typed feature structures?

This dissertation is a study of the algebraic structures that underlie attributed type signatures and the universes of typed feature structures that they induce. Specifically, it proposes definitions of signature subsumption and equivalence to answer these questions with reference to the logic of typed feature structures as formalized in Carpenter [1992]. In addition to the obvious advantages of understanding what the ramifications of changing the signature of a program/grammar are, the present study also demonstrates, with the support of empirical results, that the view of signatures proposed here can substantially improve the efficiency of programs written over the logic of typed feature structures by showing how to embed a significant class of signatures into the lattice of Prolog terms. In so doing, it demonstrates that efficient computation with typed feature structures reduces to the more general problems of standard logic programming in Prolog, graph coloring and matrix multiplication.

Acknowledgements

The research program that ultimately yielded this dissertation began when I enrolled at Carnegie Mellon University to study categorial grammar and mathematical linguistics with Bob Carpenter. Just one semester, he assured me, studying feature logic and some of the computational issues surrounding its treatment in his forthcoming book, *The Logic of Typed Feature Structures*, and then we could look at the good stuff. That was nine years ago and, as the reader will quickly notice, this is not a dissertation about categorial grammar.

The fruit of that first (and second) semester of study was a logic programming language and grammar development system based on typed feature structures called The Attribute Logic Engine (ALE). Its motivation was the observation that a very simple method for coping with hitherto unwieldy feature-based grammars could be arrived at by adapting a few proposed solutions for some more general, and now well-understood, problems in the theory of programming languages and compiler design to the needs of computational linguistics, and that by being simple and well-adapted, it would be a good method as well. It worked. ALE has been very widely used as a programming and development tool for a wide range of problems both in natural language processing and elsewhere, and to have been involved with it may very probably remain the most satisfying experience of my professional career. Bob no longer actively works in the area of feature logic (having subsequently written a very nice book on categorial grammar), but for having provided the inspiration and vision for this line of research, the first acknowledgement and an enormous debt of gratitude goes to him. Perhaps more importantly, he has also shown me that research can be quite a lot of fun.

The second acknowledgement and debt of thanks goes to the ALE users whose questions, comments and bug reports have been an indirect but still very great source of encouragement for me to continue with its support and with further research in this area. The work described in this dissertation, in fact, was undertaken with very much the same motivation that ALE originally had: the search for simple, well-adapted methods, this time mostly from discrete mathematics, to extend the state of the art in computing with typed feature structures. Its results will eventually be incorporated into future versions of ALE as well.

I am also particularly grateful to Frank Pfenning for the untiring faith and guidance he contributed to this work, and for his encyclopedic ability to connect my questions to relevant problems in other areas of mathematics and computer science. Dana Scott's advice was also quite helpful in the same respect. As for the other members of my thesis committee, John Lafferty, Alon Lavie, and Chris Manning, I am also grateful for their support and willingness to answer my questions. Herbert Simon also provided me with a great deal of insight into the role played by early artificial intelligence research in the development of typed feature logic.

Very special thanks are due to Sri Sri Swami Satchidananda, Sri Swami Karunananda Ma, Sri Swami Divyananda Ma, Sri Swami Sevananda, Abhaya Thiele, Palita Piperidis and the other inmates of Satchidananda Ashram for the guidance, support and friendship they provided throughout the course of my graduate studies.

As for the good friends I made during the four years I spent at Carnegie Mellon, I am humbled by their comradeship and patient indulgence of my penchant for Indian vegetarian cuisine: Stuart Eisenstadt, Michael Mecca, Massimo Paolucci, Thomas Polzin, Shrisha Rao, and Akira Ushioda.

This research was primarily conducted during my three years of employment as a research scientist in the Sonderforschungsbereich 340 at Eberhard-Karls-Universität Tübingen. To the directors of that project, Erhard Hinrichs and Dale Gerdemann, I am enormously indebted for their patience and financial support for this work. I also benefitted greatly from the friendship and interaction I had with Frank Richter, Manfred Sailer, Mike Calcagno, Thilo Goetz, Shuly Wintner and my other colleagues in Tübingen: Bjoern Aldag, Kordula DeKuthy, Frederik Fouvry, John Griffith, Paul King, Valia Kordoni, Sandra Kübler, Detmar Meurers, Guido Minnen, and Adam Przepiórkowski.

This research also benefitted from an internship at the Multimedia Communications Research Laboratory at Bell Laboratories during the summer of 1998. I am indebted to the director, Sid Ahuja, and Bob Carpenter for providing that opportunity. I also acknowledge and thank Bell Laboratories for making available one of the grammars used in the evaluation section at the end of this dissertation during that internship.

Most of this dissertation was actually written at the Language Technologies Institute at Carnegie Mellon University in late 1999. I am very grateful to the director, Jaime Carbonell, for providing me with the opportunity, resources and freedom to complete this document as quickly as possible.

Other individuals have contributed greatly to the contents of this dissertation, through their publications and personal communications to me during the course of this work: Mats Carlsson, Gregor Erbach, Andrew Fall, Bob Kasper, Chris Mellish and Drew Moshier. My understanding of feature-structure-based abstract machines was greatly improved upon by my opportunity to lecture on the subject at the Seminar für Sprachwissenschaft at Universität Tübingen in the summer of 1996 and again with Shuly Wintner at the Graduate Summer School for Behavioral and Cognitive Neurosciences at Rijksuniversiteit Groningen in the summer of 1997.

Portions of this dissertation have already been published as Penn, 1997; Penn, 1998; Penn and Carpenter, 1999; Penn, 1999a; Penn, 1999c; Penn, 1999d and Penn, 1999e; and were presented at the Schloss Dagstuhl IBFI Seminar on "Efficient Language Processing with High-level Grammar Formalisms" in late 1999. I am very grateful for the comments and discussion that have resulted from the early dissemination of these results.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Feature Structures

Feature structures have enjoyed a very wide use in linguistics, psychology and elsewhere for the past fifty years and have been employed at every level of linguistic theory. They are related to the record structures found in many programming languages, and to the "frames" proposed in the context of knowledge representation in artificial intelligence.

While the formal definitions used in these applications, where they are formally defined at all, often vary in some important details, feature structures are fundamentally characterized by a finite mapping from *features*, sometimes called *attributes*, to values. Figure 1.1 shows one way of depicting a feature structure that might be used to express the properties of a linguistic entity in three aspects that are salient to subject-verb agreement in English. These aspects, which are the features, are conventionally shown in small capitals at the left-hand side of each pair shown, with their respective values given on the right. Out of respect for this manner of depiction, feature structures are sometimes called *attribute-value matrices*, or *AVMs.* They can also be depicted as labelled directed graphs, as shown in Figure 1.2.

$$\begin{bmatrix} \textsc{person} & \text{third} \\ \textsc{number} & \text{singular} \\ \textsc{gender} & \text{masc} \end{bmatrix}$$

Figure 1.1: A feature structure for subject-verb agreement in English.

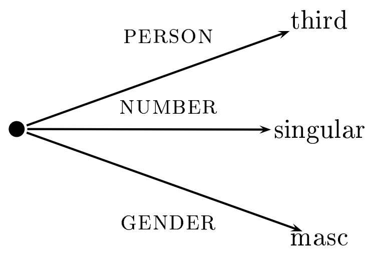

Figure 1.2: A directed-labelled-graph representation of Figure 1.1.

$$\begin{bmatrix} \text{index} & \\ \textsc{person} & \text{third} \\ \textsc{number} & \text{singular} \\ \textsc{gender} & \text{masc} \end{bmatrix}$$

Figure 1.3: A typed feature structure for subject-verb agreement.

In this case, the values might come from some assumed collection of atoms that are available as values for any feature. Often, it is assumed that values can be feature structures themselves, in which case it makes sense to speak of values that lie at the end of some path, or finite sequence, of features. The principal benefit of feature structures is that they provide named access to properties or substructures in the formal representation of an entity by means of these paths. This is in contrast to first-order terms, for example, whose subterms are referred to by means of ordinals: *first argument*, *second argument of the first argument*, etc.

1.2 Types

Semantic typing of feature structures is as old as feature structures themselves, since features were originally used to relate concepts, not specific instances of concepts. Types are typically arranged in a partial order interpreted by set inclusion, called *type hierarchies.* Typing and type hierarchies serve as an additional dimension along which to classify or organize knowledge. In the present study, the type of a feature structure will be indicated in the upper left-hand corner of its AVM representation, as in Figure 1.3, which shows the same feature structure having been assigned the type *index.*

Instead of using features and values, we could use types alone to represent subject-verb agreement properties in a large type hierarchy rooted at the

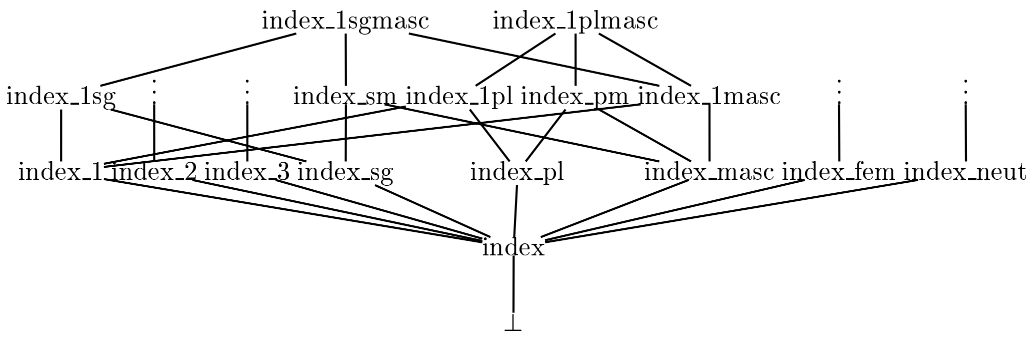

Figure 1.4: A featureless type hierarchy for subject-verb agreement.

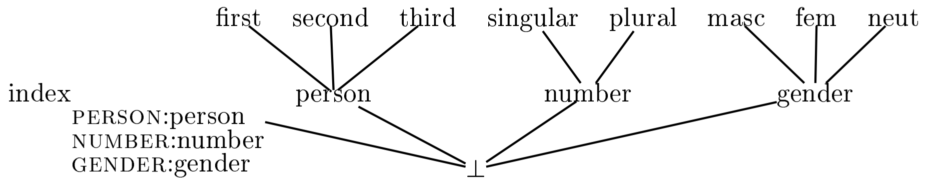

Figure 1.5: A type hierarchy for subject-verb agreement.

type, *index*, part of which is shown in Figure 1.4. Many times, these partial orders are written with the opposite orientation; but following Carpenter [1992], they will be depicted here with ⊥ (pronounced "bottom") as the most general type, with more specific subtypes written above their more general supertypes, with joins corresponding to least upper bounds, and with meets corresponding to greatest lower bounds. A central concern to computer scientists who work with typed feature structures is the efficient computation of joins, called *unification*, which corresponds to the consistent combination of information about concept membership.

We could also use feature-value pairs along with types, but in a more restricted way, by imposing a set of *appropriateness conditions* to specify which types of feature structures can bear a certain feature, which types of feature structures must bear a certain feature, what the type of a feature's value must be (also called *value restrictions*), or any combination of these. A specification of a type hierarchy, a set of features, and appropriateness conditions is known as a *type signature* or *attributed type signature*, with the specific interpretation of the appropriateness conditions remaining implicit. Figure 1.4 is a type signature with no features. Figure 1.5 is a type signature with types, features and appropriateness suitable for supporting the expression of subject-verb agreement properties tacitly assumed in Figure 1.3. The appropriateness conditions are depicted as subscripts on types, with value restrictions occurring after the feature names.

Figure 1.3 and Figure 1.4 have something in common: they are both signatures over which feature structures for describing a view of subject-verb agreement can be articulated. On the other hand, they are not the same signatures — they have different types, features and appropriateness conditions — and they may have certain practical qualities that would cause us to prefer one over the other. Other possibilities also exist. For example, one could view *index* as a parametric type that maps a triple of *person*, *number* and *gender* subtypes to a featureless type such as one of those in Figure 1.4.

1.3 Statement of Thesis and Objectives

The purpose of this thesis, broadly speaking, is to formalize what it means for two signatures to bear this kind of similarity, with an eye towards understanding the structure that exists among the feature structures they induce, particularly with respect to least upper bounds or unification. It will be shown that this minimalist view of signatures and the algebraic structures they induce can be used to improve our understanding of several practical problems in computer science as they pertain to logics of computation with typed feature structures.

In a sense, this dissertation does not argue for any new solutions to logic programming with typed feature structures. Instead, the claim is that efficient computation with typed feature structures, normally logic programming or some fragment of it such as natural language parsing or generation, is merely a collection of already well-studied problems in computer science — logic programming in Prolog, matrix multiplication and graph coloring, to name a few salient examples — disguised as new problems by more superficial differences that have occupied an all-too-central position within feature-structure-based research, particularly in computational linguistics. The disguise is removed by a proper understanding of the algebraic structure of attributed type signatures, their feature structures and the potential equivalences among them.

1.3.1 Thesis

The algebraic structures underlying attributed type signatures, their specifications, and the sets of feature structures that they induce admit a precise formalization of the equivalences that can intuitively appear to exist among the information states distinguished by different signatures. This formal equivalence can be used to substantially increase the efficiency of the practical task of programming with typed feature structures, and lends a better understanding to several more theoretical problems in mathematics and computer science.

1.3.2 Objectives

This dissertation provides a justification of this thesis by making the following specific contributions:

1. **Join-preserving Embeddings:** it provides a better and more general abstraction of what characterizes a join-preserving embedding, i.e., embeddings of one meet semi-lattice into another that preserve the results of unification, than the conventional definition.
2. **Signature Subsumption and Equivalence:** using that generalized definition, it presents a formal definition of these concepts that corresponds to the intuitive similarity mentioned in the last section.
3. **Feature-Subtype Equivalence:** it proves that every type signature is equivalent to a feature-free type signature, i.e., a type hierarchy, but that features do add more expressivity when one considers the case of finite signatures because their equivalent feature-free signatures may be infinite. Conventional wisdom on this point, while acknowledging that there is a difference in expressive power, has incorrectly pointed to another cause.
4. **Parametric Typing:** it presents the first formalization of parametric typing that is general enough to accord with its use in both the theory of programming languages and feature-structure-based linguistic theories such as Head-driven Phrase Structure Grammar (HPSG, Pollard and Sag, 1994).
5. **Parametric Typing Equivalence:** it locates an equivalence between an important class of parametric type signatures and non-parametric

type signatures that can be used to extend the abilities of existing programming languages based on attributed type signatures to handle parametric types efficiently and with minimal modification. Parametric types provide yet another dimension, together with subtypes and features, along which signatures can vary while remaining equivalent. In contrast to features, it is proven that parametric types add no extra expressive power from a formal standpoint, but provide a far more elegant means of higher-order reasoning in a type system, while allowing for significantly more compact encodings of information.

6. **Term Encoding:** It uses the generalized notion of join-preserving embeddings to provide an embedding of typed feature structures over any statically-typable attributed type signature into Prolog terms, thus reducing logic programming over typed feature structures to a Prolog preprocessing step, and admitting easy solutions to the problems of coroutining and constraint logic programming with typed feature structures.

7. **Signature Specifications:** it corrects several previous misconceptions about algebraic closure operations on partial orders, demonstrates that the compilation of any attributed type signature reduces entirely to matrix-theoretic operations on sparse matrices, and characterizes a new class of sparse matrices useful for knowledge representation for which specialized multiplication algorithms can be developed. Preliminary results suggest that these methods can improve compilation times on large signatures by a factor of 800 or more over naive transitive closure algorithms and by up to a factor of 5000 over closure by optimized matrix multiplication algorithms such as Strassen's algorithm.

8. **Optimal Term Encoding:** combining the matrix-theoretic view of type signatures in the absence of features with the Prolog term encoding of typed feature structures, it solves the hitherto open problem of finding the optimal join-preserving flat first-order-term encoding of an arbitrary finite semi-lattice, along with a complexity analysis.

The combination of the practical insights derived from this work have led to the development of an improved version of the Attribute Logic Engine (ALE, Carpenter and Penn, 1996), a logic programming language based on the logic of typed feature structures, and its reference English grammar that is faster than the current version of ALE by a factor of slightly more than 113,000.

1.4 Structure of the Dissertation

Because of the interdisciplinary nature of this work, it will be useful to consider the broad structure of this dissertation in outline from more than one perspective, each based on a discipline that will perhaps be informed by it.

1.4.1 Mathematics and Theoretical Computer Science

First and foremost, this dissertation represents an attempt to arrive at a proper understanding of the algebra of information states that is induced by combining a partially ordered set of types with appropriateness conditions, a certain kind of constraint on the presence and values of their features.

Feature structures can be thought of as representing local environments or call-stack frames, in which case feature values represent logical variables defined within the scope of those environments. The connection between earlier versions of the logic of typed feature structures and the theory of programming languages was first made by Aït-Kaći [1984], and later developed by Moshier [1988]. The version of the logic used in the present dissertation together with its connection to domain theory was presented by Carpenter [1992]. Several connections to category theory have also been drawn by Moshier [1997a,b]. Central to all of these contributions has been the attempt to isolate and explore the significance of combining internally structured objects — records, essentially — with the external structure of inclusional polymorphism and the subsumption that it induces.

From a mathematical and philosophical point of view, attributed type signatures can be thought of as very elegant and compact representations of (often infinite) partial orders of elements that characterize the possible partial states of knowledge that one can entertain for some empirical domain. While there is a substantial body of work on the theory of lattices and partial orders that has certainly informed the development of these signatures — most significantly, the importance of least upper bounds, i.e., unification — the evolution of attributed type signatures has been driven mainly by empirical demands, mostly from very early work in artificial intelligence and psychology on the representation of human memory and reasoning, and from recent work in computational linguistics on the view of parsing a sentence as the consistent combination of partial information about grammaticality and meaning that the words of which it is comprised provide. As a result, attributed type signatures exhibit some rather odd but still meaningful departures from more mainstream work in lattices, the theory of programming languages and knowledge representation.

There has been some previous formal work on logics of feature terms or typed feature structures. Nearly all of that work has elected to focus on the formal problem of finding a model-theoretic denotational semantics that appropriately relates feature structures to the collections of phenomena in the world that their partial information describes, rather than to the partial information states themselves. Certainly, many of the earlier attempts at knowledge representation from which feature structures grew are painful reminders of how essential such an external criterion is for verifying correctness. While this endeavor is certainly faithful to those reminders and, indeed, consistent with the introductory paragraphs of a very large body of generative linguistics literature, if that denotational semantics can only parrot the syntax of feature structures or their signatures, as these models inevitably have done, it is a sign either that we know far too little about the phenomena we seek to describe or that the correct level of abstraction and the essential properties of the correct objects in these models have still not been found. In the present case, it is likely a combination of both.

The assumption that underlies the present study is that, introductory paragraphs notwithstanding, the occupation in which generative linguists are actively employed is one of providing a formal description not of language itself but of the means by which the grammaticality of an utterance is elegantly ascertainable, i.e., a simple process that functionally and precisely corresponds to formal competence while still making no epistemic commitment to the biological mechanisms of linguistic comprehension. Such an assumption entails that the use of features or subtypes is one governed directly by practical considerations of the behavior of partial information states relative to the algebra that captures the essential or defining characteristics of that process, rather than by its consequences with respect to predicting the existence or nature of its empirical subjects. If in providing some mathematical relief to the more immediate concerns of the former, it forsakes what is arguably more germane to the study of language, this dissertation can only offer an apology in chorus with the adherents of the current generative enterprise.

This algebraic structure itself is thus the criterial semantics that we should be seeking. Specifically, it provides a criterion that can be used to evaluate the correctness of an attributed type signature as a specification of the organization and interaction of different sources of knowledge (for example, knowledge of language) relative to the process of consistently combining them. Individual feature structures themselves can then be taken to model individual descriptions of information relative to that signature. This is a tacit assumption underlying the work of Carpenter [1992] as well, although the focus there favors a detailed treatment of the relationship between descriptions and feature structures, much in the style of the feature structure modelers, rather than of that between signatures and algebras of feature structures. The present study favors the latter primarily because of appropriateness and the non-modular influence that it exerts on signatures as a means of classifying partial knowledge.

The next chapter presents an introduction to the logic of typed feature structures as presented in Carpenter, 1992, with the crucial difference that all assumptions of finiteness are removed except the assumption that there are finitely many features, on which appropriateness intuitively seems to depend. Mathematically, the assumption that there are finitely many types, or that feature structures have finitely many substructures is an arbitrary restriction that obstructs a very elegant insight: that the induced algebras of information states can be viewed as signatures themselves. This chapter also defends Carpenter's [1992] view of signatures as bounded complete partial orders, however, by noting that the appropriate abstraction of closure under bounded completeness in this context is the same as what is known in mathematics as the Dedekind-MacNeille completion and that, given the assumption that most pairs of types in the original partial order are join-incompatible, this completion can be performed efficiently in practice. Finally, it attempts to put typed feature structures into an historical context in order to explain some of their anomalies relative to other description logics and knowledge representation languages.

Because the algebraic structure of feature structures is one defined almost exclusively by the unification operator, a central concern in this study is how to compare the behavior of two carriers with respect to this operator. Unification-preserving or *join-preserving* maps are a well-studied class of functions in both lattice theory and knowledge representation theory as providing the essential characterization of that comparison. They are not, however. Chapter 3 presents the classical definition of a join-preserving embedding and then generalizes it to what is argued to be a better, more essential characterization. It also presents an abstraction of feature structures as the formalization of our intuitive notion of "information states," and combines the two to define the notions of signature subsumption and signature equivalence, which provide an initial formal response to the question posed at the beginning of this introduction: when can two signatures be regarded as equivalent? It is then shown that the symmetric closure of signature subsumption entails signature equivalence only in the finite case, and that the collection of all signatures forms a proper pre-order.

Chapter 4 presents a different kind of abstraction of feature structures and uses it to show how attributed type signatures can tractably encode some related conceptual taxonomies in computational linguistics and computer science, such as systemic networks, an encoding problem that was previously thought to be intractable. It then uses it to re-introduce Carpenter's assumptions of finiteness, and presents a classification of types and signatures that corresponds to the preservation of finiteness in the induced algebra. For practical purposes, the relevant interpretation of the above question, of course, is what sort of equivalence can exist among *finite* signatures only. This also establishes a result regarding the expressive power of features in an attributed type signature: finite signatures with features can express algebras of information states that only an infinite signature could without them.

The remaining chapters present some noteworthy extensions and applications of attributed type signatures. Chapter 5 considers the addition of parametric typing to attributed type signatures. Previous work on combining inclusional and parametric polymorphism has not been general enough to account for their use in computational linguistics. This chapter provides an elegant formalization of the intuition behind their use, again using the algebraic structure of information states as an external criterion to ensure the correctness of the formalization. It also considers the expressive power of parametric types and shows that, in contrast to features, parametric types do not add any additional expressive power under assumptions of finiteness. A construction is also presented for inducing finite subsignatures of infinite signatures, which provides a kind of modularity that opens up new potential applications for parametric types relative to how they are currently used in linguistics.

Chapter 6 considers the question of Prolog term encodings of typed feature structures, beginning with a review of some important lesser-known work on first-order term encodings of lattices. It has been an open question for the last eight years as to how to extend this work to attributed signatures, i.e., to the existence of features and appropriateness conditions. This chapter shows that, surprisingly, the collection of Prolog-term-encodable attributed signatures is exactly the same as the collection of statically typable attributed signatures, i.e., those signatures that require no run-time type inference. In other words, the potential for non-static typability (due to the value restrictions of appropriateness conditions) is the only factor that distinguishes algebras of typed feature structures from the lattice of Prolog terms. As a result, appropriateness can be viewed as a set of empirically mandated constraints that steer what would otherwise be a record algebra with infinite-branching terms back to a very conservative extension of the same terms used to generalize the partial information states of first-order terms, thus providing a very elegant, although unwitting, convergence of the needs of empirical linguistics and computational logic.

Chapter 7 takes a slight departure and considers the relationship between signatures and specifications of signatures, which typically assume various closure operations such as the transitive closure of subsumption. This chapter reconsiders the algebra that underlies transitive closure, which, contrary to some earlier misconceptions, is argued to be matrices over the closed boolean semi-ring. It is shown that an extension of this algebra can be constructed from any finite bounded complete partial order with features and appropriateness, i.e., any finite signature. It also identifies a new class of sparse matrix multiplication algorithms that will be particularly important to typed programming language and knowledge representation research as the number of concepts or types in their inheritance or class hierarchies becomes sufficiently large.

Chapter 8 reconsiders the problem of Prolog term encoding in the light of the results of Chapter 7. The same matrix-theoretic reduction given there can also be used to solve the open problem of finding the smallest-arity flat first-order term encoding of a finite lattice. This can also be viewed as an interesting extension of the classical keyword conflicts or minimal intersection graph problems. An empirical evaluation of some potential encoding algorithms for typed feature structures is also given at the end.

1.4.2 Linguistics

This dissertation concerns provable, logical equivalences of attributed type signatures and their consequences for the representation of partial empirical knowledge with typed feature structures. The logic of typed feature structures is also widely used as a means of stating principles in theoretical linguistics, although the boundary between its usage as a formal theoretical device and its usage as a tool for grammar engineering has often been blurred. The present study focuses on partial information about empirical objects, which is relevant not only to considerations of processing but also to the representation of theoretically interesting sets of those objects, which partial information states can be taken to denote. The first-class representation of these sets, rather than just as a disjunction of elements, is a much more elegant way to state generalizations over them in the principles of the grammar.

The goal of the linguist in the theoretical pursuit is often to provide a statement of those principles that is the simplest to conceive of, the easiest to extend to other human languages, the most conservative in its perturbation of commonly accepted principles as it encompasses new data, and/or the most natural in terms of capturing generalizations that are important to a given research community. One of the consequences of the logical equivalences that exist between attributed type signatures is that empirical data, from linguistics or any other domain, necessarily under-determine their own expression in the logic of typed feature structures when considered from a purely formal point of view. More aesthetic criteria such as those listed above can be, and have been used to argue for the superiority of one or another of various equivalent expressions as signatures.

This dissertation rejects the use of these criteria here on the grounds that:

1. they assume a prior set of formally equivalent alternatives, and thus assume a common and correct knowledge of formal equivalence among attributed type signatures that did not exist (until now);
2. they have not yet been formalized themselves to the extent that they can be used to objectively judge otherwise equivalent signatures; and
3. none of them enjoy a consensus of opinion on their importance or truth.

Some of the results of this dissertation can, in fact, be used to furnish the alternatives to such debates. An understanding of the formal equivalences presented here, however, is a necessarily independent and preliminary one to their own appreciation.

On the other hand, a proper understanding of attributed type signatures alone may be enough to resolve many representation issues. For example, a wide range of agreement phenomena in language is represented by what is often called *structure-sharing*, here called *re-entrancies*, among the person-number-gender indices referred to above. A semantic state of affairs in which someone likes himself may be represented as in Figure 1.6. The numerical tag

$$\begin{bmatrix} \text{liking} & \\ \text{LIKER} & \begin{bmatrix} \boxed{1}\ \text{index} & \\ \text{PERSON} & \text{third} \\ \text{NUMBER} & \text{singular} \\ \text{GENDER} & \text{masc} \end{bmatrix} \\ \text{LIKED} & \boxed{1} \end{bmatrix}$$

Figure 1.6: A typed feature structure in which the LIKER and LIKED are referred to by the same *index*-typed feature structure.

$$\begin{bmatrix} \text{liking} & \\ \text{LIKER} & \begin{bmatrix} \boxed{4}\ \text{index} & \\ \text{PERSON} & \boxed{1}\ \text{third} \\ \text{NUMBER} & \boxed{2}\ \text{singular} \\ \text{GENDER} & \boxed{3}\ \text{masc} \end{bmatrix} \\ \text{LIKED} & \begin{bmatrix} \boxed{5}\ \text{index} & \\ \text{PERSON} & \boxed{1} \\ \text{NUMBER} & \boxed{2} \\ \text{GENDER} & \boxed{3} \end{bmatrix} \\ \boxed{4} \not\leftrightarrow \boxed{5} & \end{bmatrix}$$

Figure 1.7: A typed feature structure in which the LIKER and LIKED are referred to by indices with the same substructures.

indicates that the value of LIKER is extensionally the same feature structure as the value of LIKED. But the signature that induces this feature structure also induces the feature structure in Figure 1.7. In this feature structure, only the PERSON, NUMBER and GENDER values are re-entrant. The potential re-entrancy of the indices themselves is prohibited by an *inequation*, a kind of negative re-entrancy. The same signature also induces the feature structure in which both indices and all of their substructures are distinct (inequated), which conventionally means that the indices refer to two different individuals.

The question for the theoretical linguist then arises as to how it might be possible to construct an experiment that would determine when Figure 1.7 is the correct representation. If it is to be interpreted as meaning that the individuals are the same, then the experiment should distinguish it from cases in which Figure 1.6 is correct, because these are not mutually consistent, i.e., they cannot refer to the same object in the world. If it is to be interpreted as meaning that the individuals are different, then the experiment should distinguish it from cases in which the third, fully-inequated alternative is correct, because these are not mutually consistent either. This is an important question — if one principle of grammar prohibits agreement by inequating indices, and another principle requires identical PERSON, NUMBER and GENDER information, the feature structure in Figure 1.7 is still licensed by the grammar.

This is a case in which a convention of representation, here the convention of using features for person, number, and gender information in indices, has persisted in the absence of a coherent picture of what information states are actually posited by its signature (Figure 1.5). The alternative signature, one in which only subtypes are used (Figure 1.4), does not suffer from the existence of a spurious extra possible feature structure, but was presumably avoided because of the nuisance of working with so many explicit types. This is a matter of convenience, however, not a matter of representation. As Chapter 5 shows, there is another alternative, one in which a parametric type, *index(third,singular,masc)*, is used to represent indices. This simultaneously is as compact as the feature-based representation, but avoids dealing with an extra possibility like Figure 1.7 because parameters cannot be re-entrant. Parametric types have not been used in theoretical linguistics apart from lists and sets of other types, again because of a lack of formal understanding of their consequences.

Theoretical linguists working in the realm of typed-feature-structure-based formalizations of grammar will hopefully be able to glean from this dissertation a better understanding of the consequences that the design decisions and changes they make with their attributed type signatures will have. Some of those will render their grammars more or less efficiently parseable. The more important consequences, however, pertain to precisely predicting all and only the grammatical utterances of some (fragment of) natural language. The promissory note that every "constraint-based" theory of grammar, including HPSG, has issued is that there will be something fundamentally more modular, declarative, transparent, and, ultimately, empirically revealing about grammars articulated as collections of constraints that delineate regions of ungrammaticality on an otherwise grammatical easel. That this note can actually be paid is not at all a certainty: every large-scale attempt at a feature-structure-based grammar, again including those based on HPSG, has inevitably been forced to retreat into a more deductive perspective, to a great extent because of the overwhelming number of common-sense constraints that must be explicated in order to exclude pathological absurdities from consideration in a truly constraint-based approach — an extra principle, for example, could be added above to "unlicense" Figure 1.7. These absurdities must be addressed because they exist on that easel; and that easel is created by the structure of the presumed attributed type signature. If there is any chance of producing a large-scale constraint-based grammar using the logic of typed feature structures, it surely demands a precise understanding of the algebra of typed feature structures induced by its signature.

Chapter 2 presents an introduction to typed feature structures as formalized in Carpenter, 1992. This can be viewed as a formalization of the approach to feature structures taken in early work on HPSG [Pollard and Sag, 1987], and, when augmented with constraints that force every feature structure to have a maximally specific type and every pair of substructures to be either inequated or re-entrant, as a formalization of the feature structures used in later work, as epitomized in Pollard and Sag, 1994. An overview of their development in psychology and artificial intelligence research is also provided. One important issue that arises in the course of this survey is whether typed feature structures are the right representation language for semantic information that typically involves the use of a wider range of functional abstractions than can elegantly be encoded using simple types and features.

Chapter 3 presents the definitions of signature subsumption and signature equivalence. These establish two formal criteria for determining the equivalence of signatures. Signature equivalence is of more direct concern to theoretical linguists, as it provides a bijective (and thus one-to-one) mapping between the feature structures that obey appropriateness in one signature and those of another. Principles of grammar over one signature, translated into the terminology of another signature, are guaranteed to have the same effect in that other signature. Crucially, the grammaticality predictions of a set of translated principles are also preserved. Signature equivalence thus provides the formal criterion for determining whether a signature plus principles of grammar, recast into a more elegant or explanatory terminology, is genuinely equivalent to an original grammar. It is also proven for the case of finite signatures, i.e., signatures with a finite number of feature structures licensed by appropriateness, that this equivalence can be inferred when two signatures mutually subsume each other, in the sense defined by signature subsumption.

Chapter 4 shows how attributed type signatures can be used to encode multi-dimensional inheritance and systemic networks without an explosion in the number of types or features. A proper understanding of the expressive power of features in an attributed signature reveals how this can be accomplished. That expressive power is then shown to be strictly greater than the expressive power of signatures with no features, due to the existence of *recursive types*, types, such as the type for non-empty lists, to which both a feature structure and one of its substructures can simultaneously belong. A great many types in HPSG are recursive, with some important linguistic consequences. Lists have routinely been used in HPSG to express the generalization that a linguistic sign can stand in a particular relation with an unbounded number of other linguistic objects. If, in principle, syntactic heads can have any number of arguments, or any number of long-distance dependencies can exist in a single sentence, or any number of quantifiers can exist, with readings corresponding to any permutation of their scopes, or even if the principles of immediate dominance allow for recursive phrase structures, such as an unbounded nesting of complement clauses or of noun phrases within relative clauses, then features must be used in these encodings. Another interesting example is the encoding of mutual subcategorization between an attributive adjective and its noun in Pollard and Sag, 1994. This requires the use of not only recursive types but cyclic feature structures, i.e., feature structures that are re-entrant with one or more of their substructures. Cyclic feature structures can only exist in the presence of recursive types. The number of features used can be reduced to a bare minimum, of course, which is a trend that has emerged within very recent research in HPSG. An understanding of recursive types and their relationship to the finiteness of the induced algebra of feature structures provides the formal criterion for guiding that minimization.

Chapter 5 is probably the most important chapter relative to current feature-structure-based linguistics. Parametric types are discussed here as an example of how to use the algebraic structure of attributed type signatures as a guide for extending the formal language of typed feature structures itself. Parametric types are very widely used in HPSG-based linguistics. They are used exclusively with lists and sets and typically as a kind of macro in which the parameter is a description of a feature structure. In this chapter, it is shown that parametric types cannot sensibly be regarded as macros, and that parameters cannot be sensibly regarded as descriptions if the description language contains variables. The good news is that signatures with parametric types can, with some restrictions, be regarded as macros for normal signatures, and that the restricted application of parametric types to lists and sets is unnecessary. As mentioned above, parametric types can be used much more prolifically to provide elegant, compact, “type-based” encodings of other linguistic objects.

Chapter 6 examines the relationship between Prolog term encodings and typed-feature-structure encodings of partial empirical knowledge. The use of typed feature structures in linguistics was originally justified on the basis of its possession of named attributes and the fact that it provided strictly more expressive power than first-order-term encodings. In spite of that original claim, the last twenty years of linguistic applications of feature structures have seen a stronger type discipline and restrictions on the use of features — culminating in appropriateness conditions — that have actually pushed feature structures back towards first-order terms in expressive power. It has been known for several years that at least one formalization of HPSG is first-order equivalent, namely SRL [King, 1989], in which feature structures denote total information and therefore have only the discrete information ordering among them. This formalization had a profound impact on the view of feature structures taken in Pollard and Sag, 1994 and later work on HPSG. The logic of Carpenter [1992] can be viewed as a generalization of SRL that accommodates the representation of partial information. The present chapter shows that a significant fragment of the logic of Carpenter [1992] is equivalent in its expressive power to Prolog terms, which in turn generalize first-order terms. The final chapter pursues the practical application of this equivalence.

1.4.3 Practical Grammar/Software Development

In addition to concerns of importance to the theoretical linguist, grammar developers, as well as developers working with knowledge representation tools in other domains, must pay particular attention to the efficiency and scalability of their code, with respect to both parsing and generation in the case of natural language processing, as well as to certain other development tasks such as incremental compilation in general. Typed feature structures are quite useful for this kind of development because the combination of semantic typing, subtyping, named attributes and appropriateness allows one to use a very terse description language to refer to a sparse amount of information over what are typically very large structures or terms. The reduction of a large amount of processing to the one fairly efficient operation of unification is also quite appealing from the standpoint of constructing simple but efficient implementations.

Modularity in this framework, however, is quite another matter. The pre-eminence of unification, when combined with appropriateness and the reliance on paths of features relative to a common superstructure to refer to the sharing of information, confers upon feature structures a disturbing degree of non-modularity, which can reflect poorly on their grammars' scalability and robust modification. A recent trend within HPSG, which was motivated to a great extent by this, has been to enumerate classes of linguistic information as explicit types whenever possible rather than as feature values. This does make the representation more modular, in practice. Features cannot be made to disappear altogether, however, although not for the reason that one might at first suspect, and as a result, the signature remains the basic modular unit of this logic.

This dissertation can be viewed as a study of that unit, along with some practical consequences. In particular, the question of determining when two signatures are equivalent invites some speculation as to when it might be practically advantageous to convert from one signature to another equivalent one, either internally within a development system or explicitly by a grammar developer, since not all logically equivalent signatures will necessarily have the same computational properties. Where a representation prefers subtypes, as in Figure 1.4, those types have typically been represented as strings, which are hashed by a programming language compiler. Where a representation prefers feature values, as in Figure 1.5, the feature structures containing those values have typically been represented as records, such that access to the feature values involves some amount of pointer chasing at run-time. Both of those are purely conventional. In Chapter 8, it will be shown that in certain circumstances it is actually preferable to encode semi-lattices of types in a record-like structure for transparency and efficiency, whereas in Chapter 4 it will be shown that, in other circumstances, feature-value-based representations can be unfolded into subtype-based ones for an improvement in efficiency.

Chapter 2 presents an introduction to the logic of typed feature structures as formulated in Carpenter, 1992 along with an historical overview of its development. This is a very widely used version of typed feature logic, and is the logical basis of the ALE system and its successors. Chapter 3 then presents the formal definitions of signature subsumption and signature equivalence that are required in order to certify the correctness of these transformations. For internal transformation by a grammar development system, signature subsumption of the transformed signature by the original signature is all that is required. Such a system can map principles, queries etc. into the terminology of the transformed signature by the map that witnesses signature subsumption, carry out the computations there, and map back so that the answer is stated in terms of the original signature. For explicit transformation by the grammar developer, signature equivalence, a stronger condition, is necessary in order to ensure that the explicitly transformed signature and grammar can be modified or augmented with the same effect as would have taken place with the original signature and grammar. It is also proven that symmetric signature subsumption is equivalent to signature equivalence only in the finite case.

It is commonly believed that there are certain feature values that cannot be equivalently expressed as subtypes in a signature. That perception is based on the unique ability of feature values to participate in re-entrancies. Chapter 4 shows that re-entrancies are only contingently related to the extra expressive power that features provide, and that the real source of extra expressive power is the possible presence of *recursive types*, types, such as the type for non-empty lists, to which both a feature structure and one of its substructures can simultaneously belong. Recursive types can only exist in the presence of features under certain appropriateness conditions, and the same information that they are capable of conveying can be conveyed only by potentially infinitely many subtypes alone. Recursive types are the unique aspect of signatures that can prevent a totally unconstrained mapping among feature-based and subtype-based encodings of information through transformations of finite signatures.

Chapter 5 extends the logic of typed feature structures by adding parametric types. While parametric types are already used in HPSG, they have only been used for lists and sets, and even then somewhat informally. Properly understood, they can be used much more widely to provide more compact encodings of information than subtypes alone can, while still retaining many of the same benefits. Parametric types effectively provide a third alternative to using features or subtypes to encode information in attributed type signatures. While signatures often serve as a finite presentation of what would otherwise be an infinite number of types, it is also possible to induce a smaller subsignature relative to a fixed grammar that is sufficient for processing with that fixed grammar. Even in the absence of parametric types, this is a very useful idea for grammar development, as the number of types and features in a signature typically does correlate with how efficient the encoding of feature structures relative to that signature can be.

The remaining chapters will probably be of the most relevance to grammar developers and other software developers working in knowledge representation who are interested in using the logic of typed feature structures given their great, direct potential to improve upon the performance of feature-structure-based logic programming languages. Chapter 7 considers the algebraic structure underlying the specification of signatures given by users of knowledge representation tools and object-oriented programming languages. This structure can be used to improve the efficiency of compiling the transitive closure of signature declarations and other closure operations that are implicitly assumed to hold by grammar development systems and typed programming language compilers that use record-like data structures. The development of a large conceptual knowledge base, such as a signature, typically involves making a very large number of small changes to the knowledge base, possibly interleaved with tests that ensure the validity of intermediate stages of development. Compilation of the declarations that define the knowledge base is essential for ensuring this validity, and yet incremental compilation is extremely difficult due to the non-modularity of the signatures themselves and the non-locality of least-upper-bound computations. As the number of types and features becomes larger, the cost of complete, non-incremental compilation can become very expensive unless close attention is paid to the algorithms used. It is also conceivable that as object-oriented programming languages become more modular, secure and portable, the number of declared classes in programs written in those languages could also become prohibitively large for their compilers. Chapter 7 shows that all of the tasks that must be performed in compiling and verifying the well-formedness of an attributed type signature can be reduced to operations on sparse matrices. Preliminary results have been so promising that incremental compilation of signatures may not be necessary at all — the closures might simply be recomputed in their entirety.

Chapters 6 and 8 consider the problem of encoding typed feature structures as Prolog terms in an implementation of a programming language or grammar development system. Logic programming languages based on typed feature structures such as ALE and its successors bear a great deal of similarity to Prolog, but it has been assumed since the first release of ALE itself that it was impossible to encode typed feature structures as Prolog terms in such a way that feature structure unification could reduce simply to Prolog term unification. Such a reduction has obvious practical advantages both for system developers and system users, since both the heavily optimized compilation for the core logic programming language itself and many of the enhanced pieces of functionality offered by commercial Prolog systems become available essentially for free. As it happens, a very large and significant class of signatures, namely those that are statically typable, do admit such a reduction. This is proven in Chapter 6, and some optimizations of the reduction and an evaluation of its performance in a practical setting is provided in Chapter 8. These results require a serious reappraisal of using commercial Prolog technology to support feature-structure-based logic programming and grammar development as an alternative to designing and building customized abstract-machine-based compilers.

Chapter 2

Attribute-Value Logic

The present study will use the logic of typed feature structures as formulated in Carpenter, 1992 as its starting point. This formulation is a relatively recent one, by comparison, and is general enough in its view of typing and feature structures that it captures the essential characteristics of most other formulations as specific instances. It is also one of the more widely used in both linguistic and formal work based on attribute-value logic. Many of the practical results of the present study, in addition, cannot accrue unless appropriateness conditions with a fairly restrictive interpretation are assumed to apply to type signatures, as they are through much of the development of Carpenter, 1992.

The next section of this chapter provides an introduction to the typed feature logic of Carpenter [1992] with some passing comparisons to several related logics. It also takes two digressions to consider the feasibility of assuming that all type systems are bounded complete partial orders, which, in the finite case, is equivalent to assuming meet-semi-latticehood (Section 2.1.2), and of assuming that the set of types to which any feature is appropriate has a unique most general type — sometimes called unique feature "introduction" at that most general type (Section 2.1.8). Both of these assumptions are fairly restrictive for a knowledge representation language and have been widely criticized since the publication of Carpenter, 1992 as being inconvenient to adhere to and computationally intractable to restore when not adhered to. As these two sections show, restoring meet-semi-latticehood in a compilation stage can, in fact, be efficiently performed in practice, although it is intractable in theory, and restoring unique feature introduction can always be achieved in time bounded by a low-degree polynomial.

The third section provides an account of the early history of feature structures to place their design in its proper context.

2.1 The Logic of Typed Feature Structures

The key to understanding Carpenter, 1992 is that, in spite of its title, it does not present a logic *of* typed feature structures, but a logic *about* typed feature structures. In particular, the logic developed in the course of that work is one *of* expressions from a description language with features and types drawn from a fixed signature. The rules of the various versions of the logic presented there choose among several classes of axiom schemes that enforce a wide range of different requirements on these descriptions, from various hygienic properties such as the associativity and distributivity of their connectives, to several degrees of discipline in typing relative to the type system of the signature. Depending on the choice of rules, a respective collection of typed feature structures relative to the same signature serves as the semantic model for the closure of derivations over descriptions with the chosen logic. In this respect, Carpenter, 1992 bears more similarity to work in domain theory, in which feature structures can be taken as models of recursive computations (following, for example, Pereira and Shieber [1984]), than to the more classical model-theoretic treatment of descriptions [King, 1989, Smolka, 1988] or of typed feature structures as syntactic terms in their own right [Johnson, 1988], which characterizes much of the other work on attribute-value logics.

The actual logics themselves will not be introduced here — the interested reader is referred to Carpenter, 1992 for their definitions and proofs of their soundness and completeness. The present study will consider only the algebraic operations on feature structures that were proven in that book to correspond exactly to the closure of descriptions under those logics, i.e., to the least fixed points of the functions corresponding to inference steps in these calculi.

2.1.1 Type Hierarchies

In Carpenter, 1992, feature structures are typed, and those types are related to each other in a particular kind of partial order.

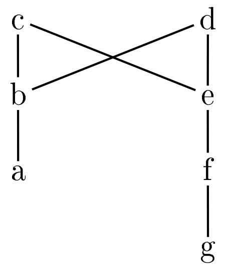

Figure 2.1: An example of a non-bounded-complete partial order.

Definition 2.1. *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 $x = y$,*
- ***(transitivity)*** *if $x \leq y$ and $y \leq z$, then $x \leq z$.*

The partial order we consider is *subsumption*, written $\sqsubseteq$. If types are interpreted as sets, then subsumption is interpreted as the inverse inclusion relation on those sets. Intuitively, $a \sqsubseteq b$ says that every feature structure of type b is also of type a. Figure 2.1 shows an example of a partially ordered set. We write them in a way that assumes reflexivity, anti-symmetry, and transitivity, so that only a base *immediate subsumption* relation, whose reflexive and transitive closure is the real subsumption relation, needs to be given. a (immediately) subsumes b, because a is lower than b, and therefore b does not subsume a. Similarly, b (immediately) subsumes c; so a also subsumes c by transitivity. a also implicitly subsumes itself.

Not every a and b may be comparable with $\sqsubseteq$; but we can identify subsets of P that are totally ordered by $\sqsubseteq$:

Definition 2.2. *A* chain *is a subset, C, of a partially ordered set, $\langle P, \leq \rangle$, such that for every $x, y \in C$, either $x \leq y$ or $y \leq x$.*

The type system is presented as a partially ordered set, rather than just as a set of incomparable types because the intention is to use types as labels on feature structures to represent *knowledge or information about objects*, rather than objects in the world themselves. This view of types is, in part, the legacy of knowledge representation languages such as KL-ONE [Brachman, 1977], which could perform certain automated classification tasks to assist the user in drawing conclusions from partial information about the world. It has also been reinforced by research in computational linguistics, notably by Head-driven Phrase Structure Grammar (HPSG, Pollard and Sag, 1987, 1994, which used feature structures to represent partial information about linguistic entities to create a sophisticated formal language for specifying constraints on the structure of language that could be used to parse sentences.

Carpenter [1992] is particularly interested in partial orders of types for which unification makes sense, i.e., partial orders in which the combination of consistent partial information about type membership results in the inference of membership in some least specific type that can be used to label a feature structure. Unification may still fail because not all of our type hierarchies will have a greatest element. Alternatively, one could also require the existence of a greatest element, $\top$, and say that it is "implemented" as failure in practice during unification. This is the approach taken in Aït-Kaći, 1984 and Fall, 1996, for example. We will also need to look at partial orders of types in this way in Chapter 7; but it is trivial to add a topmost element to any BCPO.

Definition 2.3. *Given a partially ordered set, $\langle P, \leq \rangle$, the set of* upper bounds *of a subset $S \subseteq P$ is the set $\mathrm{S}^{\mathrm{u}} = \{y \in P \mid \forall x \in S.x \leq y\}$. The set of* lower bounds*, S^{l}, is defined dually.*

Definition 2.4. *A partially ordered set, $\langle P, \leq \rangle$, is* bounded complete (BCPO) *iff, for every $S \subseteq P$ such that $S^u \neq \emptyset$, S^u has a least element, called the* least upper bound*, or* join*, of S, written $\bigvee S$.*

The greatest lower bound*, or* meet*, of S, when it exists, is defined dually and is written $\bigwedge S$.*

Definition 2.5. *Given a BCPO, $\langle P, \leq \rangle$, $p \in P$ is* join reducible *iff there exist distinct consistent $q, r \in P$ not equal to p such that $\bigvee \{q, r\} = p$.* Meet reducibility *is defined dually.*

Definition 2.6. *A* type hierarchy *is a non-empty, countable, bounded complete, partially ordered set.*

In the case of type hierarchies, where the partial order is subsumption, we then write $\bigvee S$ as $\bigsqcup S$, and in the special case where S has only two elements x and y, as $x \sqcup y$. Least upper bounds realize our intuitions about unification on partially ordered sets of types. Figure 2.1 is not a BCPO. For example, the set, $S = \{b, e\}$ has the set of upper bounds, $S^u = \{c, d\}$, which has no least element. Figure 2.2, on the other hand, is a BCPO. h is a join-reducible

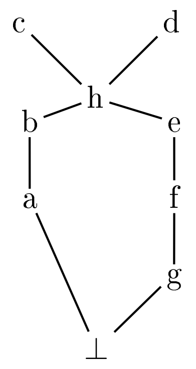

Figure 2.2: An example of a bounded complete partial order.

element. These are represent the "interesting" cases of type unification.

The biggest departure here from Carpenter, 1992 is that the latter admits only finite BCPOs as type hierarchies. Many of the remarks made in the course of this dissertation will be specific to finite BCPOs, because of their obvious computational importance; and it will be indicated explicitly where finiteness is assumed.

There are actually three particular kinds of finiteness that one can impose on type hierarchies as defined here. Non-empty bounded complete partial orders always have a least type (and thus are always non-empty), because $S = \emptyset$ has the non-empty set of upper bounds, $S^u = P$, which must have a least element. This element is written as $\perp$ (pronounced "bottom"). Denotationally, $\perp$ corresponds to the set of all objects in a model, or to put it another way, the set of objects that satisfy an empty set of constraints or information that we have about those objects. Viewed in this way $\perp$ is empty in the information that it provides about its objects — anything could be of type $\perp$. We will be looking at partially ordered types as successive refinements of $\perp$ that add information about their objects. Many times, it will be convenient to assume that a type's denotation cannot be achieved by an infinite number of distinct such refinements:

Definition 2.7. *A partially ordered set, $\langle P, \leq \rangle$, is* well-founded *iff it has no infinite descending chains.*

Well-foundedness will be critical at several points, because it will be convenient to define characteristic functions that map types to cardinals by exploiting the natural isomorphism that exists between a given type and the set of "paths" through the type hierarchy from $\bot$ to that type. With well-foundedness, those paths are all finite, which will allow us to dispense with transfinite cardinals as potential values of the characteristic functions. The most important of these functions is:

Definition 2.8. *Given a well-founded type hierarchy, $\langle T, \sqsubseteq\rangle$, and a type $t \in T$, the* path length of t *is given by the function $\delta : T \longrightarrow \mathsf{Nat}$, where:*

$$\delta(t) = \begin{cases} 0 & \textit{if } t = \bot, \\ 1 + \max\limits_{t' \in \{t\}^l} \delta(t') & \textit{otherwise.} \end{cases}$$

The path length of t is the length of the longest path of immediate subsumption links, measured in types, from $\bot$ to t. In Figure 2.2, $\delta(a) = 1$, $\delta(b) = 2$, but $\delta(h) = 4$, because of the path $\bot - g - f - e - h$.

A second kind of finiteness also concerns how "deep" subtyping can extend.

Definition 2.9. *A partially ordered set, $\langle P, \leq\rangle$, is* Noetherian *iff its dual, $\langle P, \geq\rangle$, is well-founded.*

Noetherian sets have no infinite ascending chains, and well-founded sets have no infinite descending chains. An infinite ascending chain is a chain with an infinite number of elements and no greatest element. These chains still allow us to talk about path length, for example, since the values on an infinite ascending chain will all be finite, although unbounded.

The third kind of finiteness concerns how "broad" subtyping can fan out, i.e., how many subtypes a given type can immediately subsume.

Definition 2.10. *Given a type hierarchy, $\langle T, \sqsubseteq\rangle$, and a type $t \in T$, the* branching factor of t *is given by the function $b : T \longrightarrow \mathsf{Nat} \cup \{\infty\}$, where:*

$$b(t) = |\{x \in \{t\}^u \mid \forall y \in \{t\}^u.(y \sqsubseteq x \Rightarrow y = x)\}|\,.$$

$b(t)$ is the number of minimal elements of the set of upper bounds of t.

Definition 2.11. *A type hierarchy, $\langle T, \sqsubseteq\rangle$, is* finitely branching *iff there is an $n \in \mathsf{Nat}$ such that for all $t \in T$, $b(t) \leq n$.*

The n for Figure 2.2 is 2, which is attained at h and $\bot$.

These three structural conditions are sufficient to characterize finiteness in the usual sense of cardinality:

Definition 2.12. *A type hierarchy, $\langle T, \sqsubseteq \rangle$, is* finite *iff $|T|$ is finite.*

Theorem 2.1. *A type hierarchy is finite iff it is:*

- *well-founded,*
- *Noetherian, and*
- *finitely branching.*

Proof. If $\langle T, \sqsubseteq \rangle$ is finite, then it is trivially well-founded, Noetherian and finitely branching. Suppose $\langle T, \sqsubseteq \rangle$ is well-founded, Noetherian and finitely branching. Since it is well-founded, we can use our path length function, δ. Since it is finitely branching, there is an upper bound, n on $b(T)$, the image of b on T. Since T has a least element, there are at most n types with a δ-value of 1. Since every type, t, with $\delta(t) > 1$, lies on a path that passes through a type, t', with $\delta(t') = \delta(t) - 1$, there at most n^k types with δ-values of k. Thus for any path length, there are a finite number of types with that path length.

Hence, if there is a bound, d, such that for all $t \in T$, $\delta(t) \leq d$, then $|T|$ is finite. Suppose there is no such bound. T has at least one maximal type or else T contains an infinite ascending chain, and is thus not Noetherian. T has an infinite number of maximal types, or else the path length of the maximal type(s) with the largest path length provides the bound d. Similarly, for any path length p, there are infinitely many maximal types with path length p or greater. Let $t_0 = \bot$, which has path length 0 and subsumes infinitely many of the maximal types (in fact, all of them). For every t_i, since it has only finitely many subtypes of path length $i+1$, there is a subtype, $t_{i+1} \sqsupset t_i$ such that $\delta(t_{i+1}) = i+1$ and t_{i+1} subsumes infinitely many of the maximal types. The sequence, $t_0 \sqsubset t_1 \sqsubset \ldots$ forms an infinite ascending chain in T. So the bound, d, exists, and $|T|$ is finite. □

The only fact required from bounded completeness is the existence of a least element. There is also a useful dual notion of branching factor, to which we did not need recourse for characterizing finiteness:

Definition 2.13. *Given a type hierarchy, $\langle T, \sqsubseteq \rangle$, and a type $t \in T$, the* supertype branching factor of t *is given by the function $\sigma : T \longrightarrow \mathsf{Nat} \cup \{\infty\}$, where:*

$$\sigma(t) = \left| \{x \in \{t\}^l \mid \forall y \in \{t\}^l.(x \sqsubseteq y \Rightarrow y = x)\} \right| .$$

$\sigma(t)$ is the number of maximal elements of the set of lower bounds of t.

2.1.2 Meet Semi-lattice Completions

Just because it would be convenient for unification to be well-defined does not mean it would be convenient to think of any empirical domain's concepts as a bounded complete partial order, or that it would be convenient to add all of the types necessary to a would-be type hierarchy to ensure bounded completeness. In the case of finite partial orders, bounded completeness is equivalent to another more localized condition:

Definition 2.14. *A partial order, $\langle P, \sqsubseteq\rangle$, is a* meet semi-lattice *iff for any $x, y \in P$, $x \sqcap y\downarrow$.*

Proposition 2.1. *A finite partial order is bounded complete iff it is a meet semi-lattice.*

$\sqcap$ is the binary greatest lower bound, or *meet* operation, and is the dual of the join operation. Figure 2.1 is not a meet semi-lattice because c and d do not have a meet, nor do a and g, for example.

The question then naturally arises as to whether it would be possible, given any finite partial order, to add some extra elements (types, in this case) to make it a meet semi-lattice, and if so, how many extra elements it would take, which also provides a lower bound on the time complexity of the completion.

It is, in fact, possible to embed any finite partial order into a lattice that preserves existing meets and joins by adding extra elements. The resulting construction is the finite restriction of the Dedekind-MacNeille completion [Davey and Priestley, 1990, p. 41].

Definition 2.15. *Given a partially ordered set, P, the Dedekind-MacNeille completion of P, $\langle DM(P), \subseteq\rangle$, is given by:*

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

This route has been considered before in the context of taxonomical knowledge representation [Aït-Kaći et al., 1989, Fall, 1996]. While meet semi-lattice completions are a practical step towards providing a semantics for arbitrary partial orders, they are generally viewed as an impractical preliminary step to performing computations over a partial order. Work on more efficient encoding schemes began with Aït-Kaći et al., 1989, and this seminal

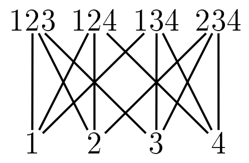

Figure 2.3: A worst case for the Dedekind-MacNeille completion at $n = 4$.

paper has in turn given rise to several interesting studies of incremental computations of the Dedekind-MacNeille completion in which LUBs are added as they are needed [Bertet et al., 1997, Habib and Nourine, 1994].

There are partial orders P of unbounded size for which $|DM(P)| = \Theta(2^{|P|})$. As one family of worst-case examples, parametrised by n, consider a set $S = \{1, \ldots, n\}$, and a partial order P defined as all of the size $n-1$ subsets of S and all of the size 1 subsets of S, ordered by inclusion. Figure 2.3 shows the case where $n = 4$. Although the maximum subtype and supertype branching factors in this family increase linearly with size, the partial orders can grow in depth instead in order to contain this.

That yields something roughly of the form shown in Figure 2.1.2, which is an example of a recent trend in using type-intensive encodings of linguistic information into typed feature logic in HPSG, beginning with Sag [1997]. These explicitly isolate several dimensions[1] of analysis as a means of classifying complex linguistic objects. In Figure 2.1.2, specific clausal types are selected from among the possible combinations of CLAUSALITY and HEADEDNESS subtypes. In this setting, the parameter n corresponds roughly to the number of dimensions used, although an exponential explosion is obviously not dependent on reading the type hierarchy according to this convention.

There is a simple algorithm for performing this completion, which assumes the prior existence of a most general element ($\perp$), given in Figure 2.5. Most instantiations of the heuristic, "where there is no meet, add one" [Fall, 1996], do not yield the Dedekind-MacNeille completion [Bertet et al., 1997], and other authors have proposed incremental methods that trade greater efficiency in computing the entire completion at once for their incrementality.

Proposition 2.2. *The MSL completion algorithm is correct on finite partially ordered sets, P, i.e., upon termination, it has produced $DM(P)$.*

[1]It should be noted that while the common parlance for these sections of the type hierarchy is *dimension*, borrowed from earlier work by Erbach [1994] on multi-dimensional inheritance, these are not dimensions in the sense of Erbach [1994] because not every n-tuple of subtypes from an n-dimensional classification is join-compatible.

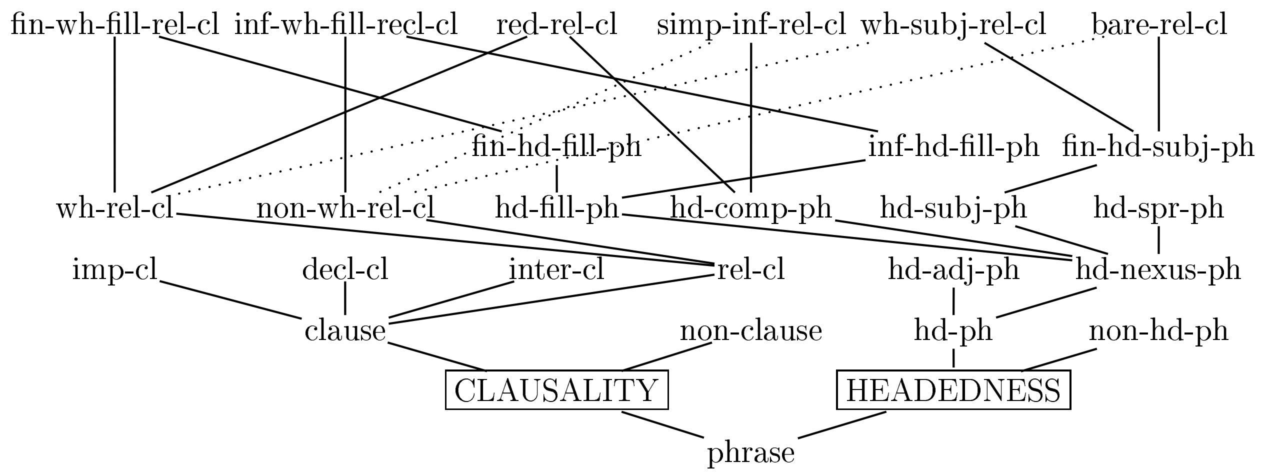

Figure 2.4: A fragment of an English grammar in which supertype branching distinguishes "dimensions" of classification.

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).

Figure 2.5: The MSL completion algorithm.

Proof: Let $V(P)$ be the partially ordered set produced by the algorithm. Clearly, $P \subseteq V(P)$. It suffices to show that (1) $V(P)$ is a complete lattice (with $\top$ added), and (2) for all $v \in V(P)$, there exist subsets $A, B \subseteq P$ such that $v = \bigvee_{V(P)} A = \bigwedge_{V(P)} B$.[2]

Suppose there are $v, w \in V(P)$ such that $v \sqcap w \uparrow$. There is a least element, so v and w have more than one maximal lower bound, l_1, l_2 and others. But then $\{l_1, l_2\}$ is upper-bounded and $l_1 \sqcup l_2 \uparrow$, so the algorithm should not have terminated. Suppose instead that $v \sqcup w \uparrow$. Again, the algorithm should not have terminated. So $V(P)$ with $\top$ added is a complete lattice.

Given $v \in V(P)$, if $v \in P$, then choose $A_v = B_v = \{v\}$. Otherwise, the algorithm added v because of a bounded set $\{t_1, t_2\}$, with minimal upper bounds, $u_1, \ldots u_k$, which did not have a least upper bound, i.e., $k > 1$. In this case, choose $A_v = A_{t_1} \cup A_{t_2}$ and $B_v = \bigcup_{1 \leq i \leq k} B_{u_i}$. In either case, clearly $v = \bigvee_{V(P)} A_v = \bigwedge_{V(P)} B_v$ for all $v \in V(P)$. $\square$

Termination is guaranteed by considering, after every iteration, the number of sets of meet-irreducible elements with no meet, since all completion types added are meet-reducible by definition.

In LinGO [LinGO, 1999], the largest publicly-available LTFS-based grammar, and one which uses such type-intensive encodings, there are 3414 types, the largest supertype branching factor is 19, and although dimensionality is not distinguished in the source code from other types, the largest subtype branching factor is 103. Using supertype branching factor for the most conservative estimate, this still implies a theoretical maximum of approximately 500,000 completion types, whereas only 893 are necessary, 648 of which are

[2]These are sometimes called the *join density* and *meet density*, respectively, of P in $V(P)$ [Davey and Priestley, 1990, p. 42].

inferred without reference to previously added completion types.

Whereas incremental compilation methods rely on the assumption that the joins of most pairs of types will never be computed in a corpus before the signature changes, this method's efficiency relies on the assumption that most pairs of types are join-incompatible no matter how the signature changes. In LinGO, this is indeed the case: of the 11,655,396 possible pairs, 11,624,866 are join-incompatible, and there are only 3,306 that are consistent (with or without joins) and do not stand in a subtyping or identity relationship. In fact, the cost of completion is often dominated by the cost of transitive closure, which is discussed in Chapter 7 in more detail.

While the continued efficiency of compile-time completion of signatures as they further increase in size can only be verified empirically, what can be said at this stage is that the only reason that signatures like LinGO can be tractably compiled at all is sparseness of consistent types. In other geometric respects, it bears a close enough resemblance to the theoretical worst case to cause concern about scalability. Compilation, if efficient, is to be preferred from the standpoint of static error detection, which incremental methods may elect to skip. In addition, running a new signature plus grammar over a test corpus is a frequent task in large-scale grammar development, and incremental methods, even ones that memoise previous computations, may pay back the savings in compile-time on a large test corpus. It should also be noted that another plausible method is compilation into logical terms or bit vectors, in which some amount of compilation (ranging from linear-time to exponential) is performed with the remaining cost amortised evenly across all run-time unifications, which often results in a savings during grammar development.

2.1.3 Feature Structures

A type hierarchy, $\langle T, \sqsubseteq \rangle$, along with a finite set of features, *Feat*, and a set of nodes, Q, induces a set of typed feature structures:

Definition 2.16. *A* typed feature structure *is a tuple,* $F = \langle Q, \bar{q}, \theta, \delta, \nleftrightarrow \rangle$ *where:*

- Q *is a countable set of* nodes*,*
- $\bar{q} \in Q$ *is the* root node*,*
- $\theta : Q \longrightarrow T$ *is a total* node typing *function,*

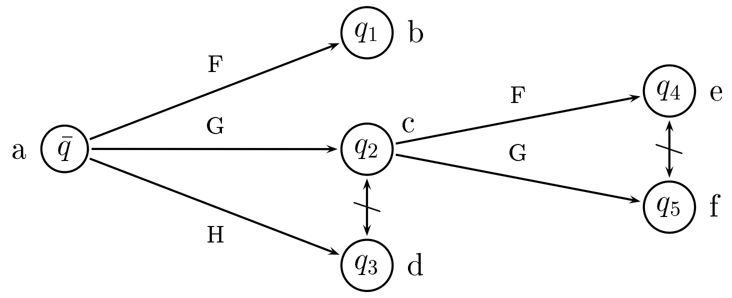

Figure 2.6: An example typed feature structure.

- $\delta : Feat \times Q \longrightarrow Q$ *is a partial* feature value *function, and*
- $\not\leftrightarrow \subseteq Q \times Q$ *is an anti-reflexive and symmetric* inequation *relation.*

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

An example feature structure induced by the type hierarchy in Figure 2.2 and the set of features $Feat = \{\text{F}, \text{G}, \text{H}\}$ can be represented as a directed graph, as shown in Figure 2.6. It has the set of nodes $Q = \{\bar{q}, q_1, q_2, q_3, q_4, q_5\}$ with the following node typing and feature value functions, and inequation relation:

$$\begin{array}{lll}
\theta(\bar{q}) = a & \delta(\text{F}, \bar{q}) = q_1 & q_2 \not\leftrightarrow q_3 \\
\theta(q_1) = b & \delta(\text{G}, \bar{q}) = q_2 & q_4 \not\leftrightarrow q_5 \\
\theta(q_2) = c & \delta(\text{H}, \bar{q}) = q_3 & \\
\theta(q_3) = d & \delta(\text{F}, q_2) = q_4 & \\
\theta(q_4) = e & \delta(\text{G}, q_2) = q_5 & \\
\theta(q_5) = f & &
\end{array}$$

Notice that in this formalization of feature structures, types are not values themselves, but only decorate the real values — nodes — through the typing function, θ.

The relation, $\not\leftrightarrow$, represents the set of *inequations* that hold between nodes in a feature structure. This is not simply the complement of equality. These are persistent, negative constraints on the identification of nodes, much like those proposed for Prolog II [Colmerauer, 1984, 1987].

Not much can be said about nodes apart from the fact that they can bear types and features with values. The set, Q, has no algebraic structure of its own, and, with the exception of extensional typing, the description language used in Carpenter, 1992 provides no means of regulating the number of feature structures in $\mathcal{F}$, or the number of them of any particular type. It also provides no means of determining whether two different feature structures have intersecting node sets unless they are both substructures of a common feature structure, so that they can be referred to by paths of features. In fact, no description language can, whose descriptions are intended to describe single feature structures (and their substructures), rather than arbitrary collections of them.

Nodes are not types; and the fact that $\delta(\text{F})$ maps nodes to nodes rather than types to types is a significant departure (with precedents) from early conceptions of feature structures, as elaborated upon in Section 2.2. They correspond roughly to instances of a particular concept or type; but we are entirely dependent upon their types, features values and the existence of a common root node (and therefore, a common super-structure) to distinguish them, i.e., to observe whether two nodes $q_1, q_2 \in Q$ are such that $q_1 = q_2$ or $q_1 \neq q_2$. In general, there are some nodes that we simply cannot distinguish from each other, particularly if they do not belong to a set Q of a common feature structure.

Following one of the extensions discussed in Carpenter, 1992, the set of nodes for a given feature structure is allowed to be countably infinite here, which admits two possible kinds of infinity for typed feature structures. Just as with type hierarchies, typed feature structures can be "deeply" infinite, by having unboundedly long paths of nodes, or "broadly" infinite, by having nodes, $q \in Q$, for which there are infinitely many features, $f \in \mathit{Feat}$ such that $\delta(\text{F}, q)\!\downarrow$. The latter is rejected here by the assumption that the set of features, *Feat*, is finite, following Carpenter [1992].[3] This is often restricted further by appropriateness, as explained below. This assumption is ultimately responsible for almost every practical benefit of the logic of typed feature structures documented in this dissertation or earlier; and its validity hinges on an apparent lack of empirical necessity in any domain for formal descriptions of objects with an infinite number of attributes or, put in the language of first-order logic, infinitely branching terms.

[3]It also depends on the assumption that δ is a function, i.e., that it can only take one value for every pair of feature and node. With the arguable exception of set-valued features [Carpenter, 1993a, Manandhar, 1994, Moshier and Pollard, 1994, Richter, in prep.] this assumption seems to be fairly universal, and even for set-valued features, an auxiliary accessibility relation is typically used instead of δ).

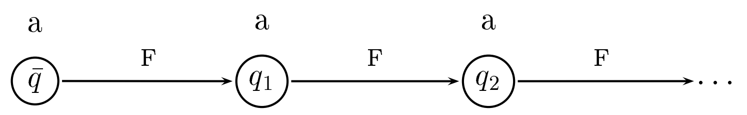

Figure 2.7: A typed feature structure with an infinite number of nodes.

To understand the former better, we need a formal notion of paths and path values:

Definition 2.17. *A* path *is a finite sequence of features, $\pi \in Feat^*$.*

Definition 2.18. *Given a typed feature structure, $F = \langle Q, \bar{q}, \theta, \delta, \leftrightarrow\!\!\!\!\!/\ \rangle$, its partial* path value *function is a function, $\delta' : Feat^* \times Q \longrightarrow Q$ such that:*

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

Following Carpenter [1992], δ will be used to refer to both the feature value function and the path value function.

Definition 2.19. *If $F = \langle Q, \bar{q}, \theta, \delta \rangle$, and $\delta(\pi, q)\downarrow$, then the* restriction of F to π *is $F@\pi = \langle Q', \bar{q}', \theta', \delta' \rangle$, where:*

- *$Q' = \{\delta(\pi\pi', \bar{q}) | \pi' \in Feat^*\}$,*
- *$\bar{q}' = \delta(\pi, \bar{q})$,*
- *$\delta' = \delta|_{F \times Q'}$, and*
- *$\theta' = \theta|_{Q'}$.*

Definition 2.20. *A typed feature structure, $F = \langle Q, \bar{q}, \theta, \delta \rangle$, is* finite *iff Q is a finite set.*

Figure 2.7 depicts part of a feature structure with an infinite number of nodes, each of which is of type a and has a single attribute, F. This is a well-defined typed feature structure, but it is not finite.

Typed feature structures, even finite ones, can also be "infinite" in the sense of having cycles, and therefore having a path value function that takes a value for infinitely many paths. For example, a feature structure can have two nodes, $\bar{q}$ and q such that $\delta(\text{F}, \bar{q}) = q$ and $\delta(\text{F}, q) = \bar{q}$. That can be depicted as a labelled directed graph with cycles, as shown in Figure 2.8, or as an AVM, as shown in Figure 2.9. The boxed numerical tags are used

$$\begin{tikzcd} \text{a}\ \bar{q} \arrow[r, bend left, "\text{F}"] & q\ \text{a} \arrow[l, bend left, "\text{F}"] \end{tikzcd}$$

Figure 2.8: The directed graph representation of a cyclic feature structure.

$$\begin{bmatrix} \boxed{1}\ \text{a} & \\ \text{F} & \begin{bmatrix} \text{a} & \\ \text{F} & \boxed{1} \end{bmatrix} \end{bmatrix}$$

Figure 2.9: The AVM representation of Figure 2.8.

to indicate the identity of nodes relative to paths, or "re-entrant nodes," i.e., nodes that have an in-degree of greater than 1 in the directed graph representation of a feature structure.[4]

A related ability of feature structures is to have acyclic *re-entrancies* between paths, i.e., sharing of substructures. Figure 2.10 is distinguished from Figure 2.11, for example, in that its F and G values are not only of the same type with the same features, but actually the same node. Boxed numerical tags are also used in AVM representations to identify nodes that participate in the inequation relation. Figure 2.12 shows the AVM representation of the feature structure shown in Figure 2.6. In AVM representations, if no $\not\leftrightarrow$ pairs are shown, it is assumed that $\not\leftrightarrow = \emptyset$.

2.1.4 Appropriateness and Attributed Type Signatures

Given a set of types, T, and a finite set of features, *Feat*, the set of feature structures, $\mathcal{F}$, includes, for every combination of features from *Feat*, a feature

[4]The exception is the root node, $\bar{q}$, which is considered re-entrant if it has an in-degree greater than 0.

$$\begin{bmatrix} \text{a} & \\ \text{F} & \boxed{1} \begin{bmatrix} \text{b} & \\ \text{H} & \text{c} \end{bmatrix} \\ \text{G} & \boxed{1} \end{bmatrix}$$

Figure 2.10: The AVM representation of a feature structure with an acyclic re-entrancy.

$$\begin{bmatrix} \text{a} & \\ \textsc{f} & \begin{bmatrix}\text{b} & \\ \textsc{h} & \text{c}\end{bmatrix} \\ \textsc{g} & \begin{bmatrix}\text{b} & \\ \textsc{h} & \text{c}\end{bmatrix}\end{bmatrix}$$

Figure 2.11: The AVM representation of a feature structure with structurally identical but non-re-entrant substructures.

$$\begin{bmatrix} \text{a} & \\ \textsc{f} & \text{b} \\ \textsc{g} & \begin{bmatrix} \boxed{1}\ \text{c} & \\ \textsc{f} & \boxed{2}\ \text{e} \\ \textsc{g} & \boxed{3}\ \text{f}\end{bmatrix} \\ \textsc{h} & \boxed{4}\ \text{d} \\ \boxed{1} \not\leftrightarrow \boxed{4} & \\ \boxed{2} \not\leftrightarrow \boxed{3} & \end{bmatrix}$$

Figure 2.12: The AVM representation of the feature structure in Figure 2.6.

structure that has a node bearing that combination, with, for every selection of types from T, values of those respective types. This may not always make sense. We may, in the case of linguistic knowledge representation, want to allow some nodes that represent what is known about the syntactic status of a verb to bear a feature MOOD and other nodes that represent what is known about the syntactic status of a noun to bear a feature CASE; but it would not make sense to bestow attributes such as MOOD and CASE on the same node. In addition, a feature such as MOOD could reasonably have a value of a type such as *indicative* or *subjunctive*, but not of the type *nominative* or *masculine*.

Pollard and Sag [1987, p. 38] first suggested that the type system could be used not only to classify different kinds of objects in the world, but to prevent the cooccurrence of certain features in formal representations of those objects. Its importance derives, again, from viewing feature structures as models of states of partial information about objects, in which it is important to distinguish between features whose values are inapplicable or irrelevant to a particular state of information and features whose values are relevant but unknown or missing. Of course, the type system can also be used to declare

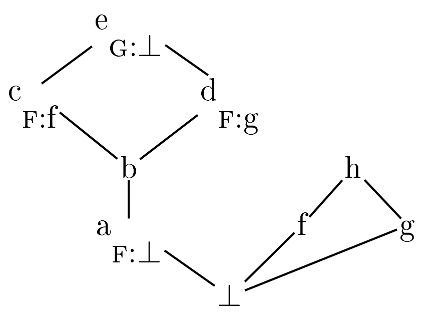

Figure 2.13: An example type signature with upward closure and right monotonicity assumed.

which values make sense for a given feature.

King [1989] first reified the intuition that only certain features are relevant or appropriate for a particular type. Knowledge of which features these are for each type augments our knowledge about the types and features used in the specification of a grammar. The formalization of that extra knowledge used here, known as *appropriateness conditions*, is taken from Carpenter [1992].

Definition 2.21. *Given a type hierarchy, $\langle T, \sqsubseteq\rangle$, and a finite set of features, Feat, an* appropriateness specification *is a partial function, Approp : Feat $\times$ $T \longrightarrow T$ such that, for every* $\textsc{f} \in$ *Feat:*

- ***(Feature Introduction)*** *there is a type Intro*$(\textsc{f}) \in T$ *such that:*
  - *Approp*$(\textsc{f}, \textit{Intro}(\textsc{f}))\!\downarrow$*, and*
  - *for every $t \in T$, if Approp*$(\textsc{f}, t)\!\downarrow$*, then Intro*$(\textsc{f}) \sqsubseteq t$*, and*
- ***(Upward Closure / Right Monotonicity)*** *if Approp*$(\textsc{f}, s)\!\downarrow$ *and $s \sqsubseteq t$, then Approp*$(\textsc{f}, t)\!\downarrow$ *and Approp*$(\textsc{f}, s) \sqsubseteq$ *Approp*$(\textsc{f}, t)$*.*

Definition 2.22. *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.*

Figure 2.13 depicts a type signature. It looks like a type hierarchy, but feature-value pairs have been added to some of the types. By convention, a feature annotates its introducing type, and the value (type) it occurs with is the value of *Approp* at that type. For example, in the example shown, *Intro*(F) = a and *Approp*(F, a) = $\bot$. Upward closure is assumed, so all subtypes of an introducing type also have its introduced features. In this way, features can be left implicit where their presence can be inferred from upward closure and their values can be inferred from right monotonicity. For example, F is appropriate to b in Figure 2.13, also with value $\bot$. The types c and d both refine the value of *Approp* on F, but e does not, and the value of *Approp*(F, e), h, can be inferred from the unification of the values at c and d. e does introduce a new feature, G, however.

The usage of *signature* is borrowed from King [1989], although the definition of a signature and of appropriateness there is slightly different. They are also closely related to class hierarchies in object-oriented programming, in that they specify where features (methods) are introduced and how they are inherited. Just as in object-oriented programming, one could interpret these specifications as defaults and entertain various conventions for overriding them [Carpenter, 1993b, Lascarides and Copestake, 1999], although these will not be pursued here.

In terms of identifying what provides the basic vocabulary or building blocks of a feature structure or a description language, it might be more fair to refer to the set of types plus the set of features as the signature; but type subsumption and appropriateness play a very important role in the logic(s) of Carpenter [1992], in that the axiom schemes that make up those logics depend on them in order to form genuine axioms. Thus, a signature, as defined here, is precisely what is needed to construct a logic to accompany the descriptions that are constructed from a set of types and a set of features. Appropriateness is also distinguished from the more general recursive constraint system presented as an application in Carpenter, 1992 in that satisfaction is decidable.

As mentioned above, the emphasis in this presentation will be on the algebraic operations on feature structures that correspond to the closure of descriptions under various rules of inference. Some of these operations are what “enforce” the requirements that an appropriateness specification states over feature structures because the (totally) well-typed feature structures are exactly the feature structures that model those fixed points — the existence of *Approp* by itself guarantees nothing. Of principal practical interest, however, is the operation of feature structure unification and its progenitor, the subsumption relation on feature structures.

2.1.5 Subsumption and Unification

The view of feature structures as representing partial or underspecified information is what inspired us to use partially ordered sets of types. That partial order fittingly induces a partial order on feature structures themselves, that corresponds to subsumption of information content:

Definition 2.23. *Given a common signature, a typed feature structure, $F = \langle Q, \bar{q}, \theta, \delta, \not\leftrightarrow \rangle$* subsumes *another typed feature structure, $F' = \langle Q', \bar{q}', \theta', \delta', \not\leftrightarrow' \rangle$, written $F \sqsubseteq F'$ (or, where unclear, $\sqsubseteq_{\mathcal{F}}$) iff there is a total function, $h : Q \longrightarrow Q'$, called a* morphism, *such that:*

1. *$h(\bar{q}) = \bar{q}'$,*
2. *for every $q \in Q$, $\theta(q) \sqsubseteq \theta'(h(q))$,*
3. *if $\delta(\text{F}, q)\downarrow$, then $h(\delta(\text{F}, q)) = \delta'(\text{F}, h(q))$, and*
4. *if $q_1 \not\leftrightarrow q_2$, then $h(q_1) \not\leftrightarrow' h(q_2)$.*

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

The second criterion forces feature structure subsumption to obey type subsumption on individual nodes. The other criteria require the morphism to establish a correspondence between nodes that preserves the starting node, feature (and thus path) values, and inequations. The more specific feature structure, F', can thus simulate F in the sense that we can map to a corresponding node using h, ask questions about path values, path equalities, type information and inequation information, and we get the same or more specific answers that we would have received from F itself. In this sense, F' has the same information that F has plus possibly more.

The feature structure in Figure 2.11, for example, subsumes the feature structure in Figure 2.10. The morphism that witnesses this is not an injection, because the nodes corresponding to the paths F and G in the former are both mapped to the node corresponding to both the path F and G in the latter in order to satisfy the third criterion. So feature structures can be more specific by virtue of having extra path equations. The feature structure in Figure 2.11 also subsumes the feature structure in Figure 2.14. In general, a feature structure with a pair of nodes with consistent types has a pair of more specific feature structures: one where their paths map to the same node, and one where the nodes are inequated. Neither of these subsumes the other. When a pair of nodes has inconsistent types, then only the

$$\begin{bmatrix} \text{a} & \\ \textsc{f} & \boxed{1}\begin{bmatrix}\text{b} & \\ \textsc{h} & \text{c}\end{bmatrix} \\ \textsc{g} & \boxed{2}\begin{bmatrix}\text{b} & \\ \textsc{h} & \text{c}\end{bmatrix} \\ \boxed{1} \not\leftrightarrow \boxed{2} & \end{bmatrix}$$

Figure 2.14: The AVM representation of a feature structure with structurally identical but inequated substructures.

$$\begin{bmatrix} \text{a} & \\ \textsc{f} & \text{b} \end{bmatrix} \sqsubseteq \begin{bmatrix} \text{a} & \\ \textsc{f} & \text{b} \\ \textsc{g} & \text{c} \end{bmatrix}$$

Figure 2.15: An example of feature structure subsumption.

inequated variant exists. This is the more intensional version of inequated feature structure presented in Carpenter, 1992. The other, called *fully inequated* feature structures [Carpenter, 1992, p. 120], has neither, because it assumes that inequations can only hold between nodes in a feature structure when there is a more specific feature structure in which their paths could be shared.

The single direction of implication in the fourth criterion says that they can also have extra path inequations.

To consider another example, the feature structures in Figure 2.15 stand in the subsumption relation shown because of the condition that $\delta(\textsc{f}, q)\!\downarrow$ in the third criterion. Thus, feature structures can also be more specific by virtue of having extra features defined on a particular node.

Just as type subsumption induces type unification by forming least upper bounds, feature structures subsumption induces feature structure unification. The problem, as mentioned above, is that we have no way to determine that two feature structures have non-intersecting node sets. We do not want the unification of the feature structures in Figure 2.16 to differ from the unification of those in Figure 2.17. As individual feature structures, both structures in both pairs have the same information, so unification should yield a feature structure with the same information as well. This is related to the fact that feature structure subsumption is not a partial order because

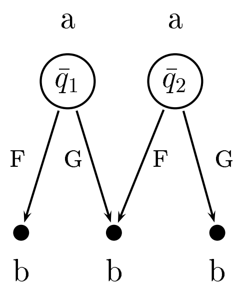

Figure 2.16: Two feature structures whose sets of nodes intersect.

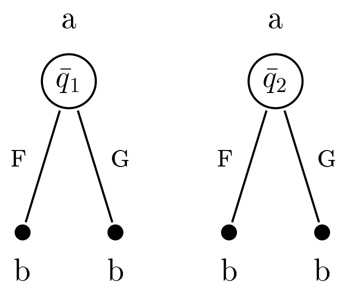

Figure 2.17: Two feature structures whose sets of nodes do not intersect.

there can exist F_1 and F_2 such that $F_1 \sqsubseteq F_2$ and $F_2 \sqsubseteq F_1$. We should not care *which* feature structure unification returns, provided that it has the right information content.

We can solve both concerns by creating an equivalence relation that relates mutually subsuming feature structures. Subsumption modulo this relation is then a partial order. Using the axiom of choice, we can also find equivalent feature structures under this relation with non-intersecting nodes as a precursor to unification. In the next chapter, it will be shown that this relation corresponds exactly to abstraction away from the set of actual nodes in a feature structure, leaving only its relevant information.

Definition 2.24. *Given a set, S, an* equivalence relation *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 2.25. *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 2.3. *$\sim$ is an equivalence relation.*

Definition 2.26. *Given a set, S, and an equivalence relation, $\approx$, and an element $s \in S$, the* equivalence class of s under $\approx$ *is:*

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

Definition 2.27. *Given a set, S, and an equivalence relation, $\approx \subseteq S \times S$, the* quotient set of S modulo $\approx$ *is:*

$$S/\!\approx\, = \{[s]_{\approx} | s \in S\}.$$

Definition 2.28. *Given a common signature, and $F \sim \langle Q, \bar{q}, \theta, \delta, \leftrightarrow\!\!\!\!\!/\,\rangle$ and $F' \sim \langle Q', \bar{q}', \theta', \delta', \leftrightarrow\!\!\!\!\!/\,'\rangle$ such that $Q \cap Q' = \emptyset$, let $\bowtie$ be the finest-grained equivalence relation on $Q \cup Q'$ such that:*

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

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

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

where:

- $\theta^{\bowtie}([q]_{\bowtie}) = \bigsqcup \{\theta(q) \sqcup \theta'(q') | q' \bowtie q\}$,
- $\delta^{\bowtie}(\text{F}, [q]_{\bowtie}) = \begin{cases} [\delta(\text{F}, q)]_{\bowtie} & \textit{if } q \in Q, \\ [\delta'(\text{F}, q)]_{\bowtie} & \textit{if } q \in Q' \end{cases}$
- $[q]_{\bowtie} \not\leftrightarrow^{\bowtie} [q']_{\bowtie}$ *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 2.4. *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 [1988] and extended to the typed case by Carpenter [1992]. □

Notice that $F \sqcup F' \in \mathcal{F}$ because our lack of interest in the structure of nodes allows $F \sqcup F' \in \mathcal{F}$ to use a set of equivalence classes as its set of nodes. Each equivalence class, in turn, contains nodes of F and F'. As a result, $F \sqcup F'$ corresponds to one of many alphabetically variant minimal upper bounds of F and F' with respect to $\sqsubseteq$, and so $\sqcup$ can be viewed as a partial function from $\mathcal{F} \times \mathcal{F}$ to $\mathcal{F}$. Notice that the result may have a different type than either of the operands because of joins in the type hierarchy.

By considering feature structures and subsumption modulo alphabetic variance, the structure assumed for types, a bounded complete partially ordered set, is mirrored in the structure of feature structures:

Proposition 2.5. *$\sqsubseteq_{\mathcal{F}}$ is reflexive and transitive.*

Proof. It is reflexive because the identity function, $id : Q \longrightarrow Q$ is a morphism. It is transitive because composition of morphisms yields a morphism. □

Definition 2.29. *Given $\sqsubseteq$ and $\sim$,* invariant subsumption, *$\sqsubseteq^{\sim} \subseteq \mathcal{F}/{\sim} \times \mathcal{F}/{\sim}$ is defined such that $[F_1]_{\sim} \sqsubseteq^{\sim} [F_2]_{\sim}$ iff there exist $F_1' \in [F_1]_{\sim}$, $F_2' \in [F_2]_{\sim}$, such that $F_1' \sqsubseteq F_2'$.*

Theorem 2.2. *$\langle \mathcal{F}/{\sim}, \sqsubseteq^{\sim}\rangle$ is a type hierarchy.*

Proof. Countability follows from the countability of node sets and of the (implicit) set of types, T.

$\sqsubseteq^{\sim}$ is reflexive and transitive because $\sqsubseteq$ is reflexive and transitive. Suppose $[F_1]_\sim \sqsubseteq^{\sim} [F_2]_\sim$ and $[F_2]_\sim \sqsubseteq^{\sim} [F_1]_\sim$. Then there exist $F_1' \in [F_1]_\sim$, $F_2' \in [F_2]_\sim$, such that $F_1' \sqsubseteq F_2'$ and $F_1'' \in [F_1]_\sim$, $F_2'' \in [F_2]_\sim$ such that $F_2'' \sqsubseteq F_1''$. But then $F_2' \sim F_2'' \sqsubseteq F_1'' \sim F_1'$, so $F_1' \sim F_2'$ and $[F_1]_\sim = [F_2]_\sim$. So invariant subsumption is a partial order.

Given $[F_1]_\sim$ and $[F_2]_\sim$, suppose there exists $[F_3]_\sim$ such that $[F_1]_\sim \sqsubseteq^{\sim} [F_3]_\sim$ and $[F_2]_\sim \sqsubseteq^{\sim} [F_3]_\sim$. Then there exist $F_1' \in [F_1]_\sim$, $F_2' \in [F_2]_\sim$, and $F_3', F_3'' \in [F_3]_\sim$, such that $F_1 \sim F_1' \sqsubseteq F_3'$ and $F_2 \sim F_2' \sqsubseteq F_3'' \sim F_3'$. By Proposition 2.4, $F_1 \sqcup F_2{\downarrow}$ and $F_1 \sqcup F_2 \sqsubseteq F_3'$; so $[F_1 \sqcup F_2]_\sim{\downarrow}$ and $[F_1 \sqcup F_2]_\sim \sqsubseteq^{\sim} [F_3]_\sim$. So $\sqsubseteq^{\sim}$ is bounded complete. □

Of course, the use of the term "type hierarchy" here is chosen mostly for shock value. We are really proving that it is a countable BCPO. For the case of feature structures without inequations, this follows directly from the proof in Carpenter, 1992 that $\langle \mathcal{F}/{\sim}, \sqsubseteq^{\sim}\rangle$ is a domain. We were not thinking of the elements in a type hierarchy themselves as sets of feature structures (although these are their most natural *denotations*); but because we never specified exactly what our types were, they could just as well be. $\langle \mathcal{F}/{\sim}, \sqsubseteq^{\sim}\rangle$ may not be the same type hierarchy as the one in the signature that induced it; but recognizing this duality is the first step towards understanding the underdetermination of signature encodings by empirical data. We will examine this issue more carefully — in particular, the conditions under which feature structures form a finite BCPO — in Chapter 4.

2.1.6 Well-Typing

The next class of algebraic operations identify subsets of $\mathcal{F}$ that respect the appropriateness specification of its implicit signature. There are a few other noteworthy subsets of $\mathcal{F}$ that will not be discussed in detail here. One is the set of feature structures that respect a stronger version of type inferencing called *extensionality*, where nodes whose feature values and types are identical are assumed to be the same node. Another is of those that are *fully inequated*, i.e., that inequate every pair of paths whose nodes have inconsistent types. A third is of those that are *sort-resolved*, i.e., that label their nodes with only maximally specific types. The first two are elaborated upon in Carpenter, 1992. The third is discussed somewhat there, but was not completely understood until Carpenter and King, 1995.

Semi-well-typedness

There are at least three ways in which one can interpret and enforce appropriateness specifications. One, perhaps the most basic, is as a specification of which features a node is permitted to bear, given its type.

Definition 2.30. *A typed feature structure, $F = \langle Q, \bar{q}, \theta, \delta, \not\leftrightarrow \rangle$ is* semi-well-typed *iff for every $q \in Q$, if $\delta(\text{F}, q)\downarrow$, then $Approp(\text{F}, \theta(q))\downarrow$.*

$\mathcal{STF}$ denotes the set of semi-well-typed feature structures.

Definition 2.31. *Let TypDom : $\mathcal{F} \longrightarrow \mathcal{F}$ be the partial function such that, given $F = \langle Q, \bar{q}, \theta, \delta, \not\leftrightarrow \rangle \in \mathcal{F}$, $TypDom(F) = \langle Q, \bar{q}, \theta^D, \delta, \not\leftrightarrow \rangle$, where $\theta^D(q) = \theta(q) \sqcup \bigsqcup\{Intro(\text{F}) | \text{F} \in Feat \& \delta(\text{F}, q)\downarrow\}$ provided the joins in the definition of θ^D exist, and is undefined otherwise.*

Proposition 2.6. *For every $F \in \mathcal{F}$, if $TypDom(F)\downarrow$, then $TypDom(F) \in \mathcal{STF}$.*

Proof. If the joins required by θ^D exist, then we know $Approp(\text{F}, \theta^D(q))$ wherever $\delta(\text{F}, q)\downarrow$ by feature introduction ($Approp(\text{F}, Intro(\text{F}))\downarrow$), the fact that $Intro(\text{F}) \sqsubseteq \theta^D(q)$, and by upward closure. □

The name, *semi-well-typed*, was bestowed by King and Goetz [1993], who used it to present an alternative to well-typing without reference to a unique introducing type for every feature, although unique introducing types are retained here (see Section 2.1.8). One can show that *TypDom* promotes any feature structure to its least semi-well-typed extension, if one exists:

Proposition 2.7. *If $F \in \mathcal{F}$, and $F' \in \mathcal{STF}$, then $F \sqsubseteq F'$ iff $TypDom(F) \sqsubseteq F'$.*

Proof. Proven by King and Goetz [1993]. □

Corollary 2.1. *$F \in \mathcal{STF}$ iff $TypDom(F) \sim F$.*

Proof. By Propositions 2.7 and 2.6, the forward direction holds. The reverse holds since $\mathcal{STF}$ is clearly upward closed. □

Another central issue is whether unification preserves the condition that this operator establishes. As it happens, once we know that we are working in $\mathcal{STF}$, unification will allow us to stay there, so we will not need *TypDom* anymore:

Theorem 2.3. *If $F, F' \in \mathcal{STF}$ and $F \sqcup F'\downarrow$, then $F \sqcup F' \in \mathcal{STF}$.*

Proof. Unification can only make types more specific, which is consistent with *Approp* by upward closure. □

Another way to look at this result is that unification in $\mathcal{F}$ automatically gives us unification in $\mathcal{STF}$ — we never need to worry about dealing with non-semi-well-typed feature structures if we wish to avoid them.

Well-typedness

A second view of appropriateness is that of a *value restriction* — using the value of *Approp* to constrain the value of an appropriate feature. Of course, this view will only make sense if we combine it with semi-well-typedness, to make sure that *Approp* has a value, where necessary.

Definition 2.32. *A typed feature structure, $F = \langle Q, \bar{q}, \theta, \delta, \not\leftrightarrow \rangle$ is* well-typed *iff for every $q \in Q$, if $\delta(\text{F}, q)\downarrow$, then $Approp(\text{F}, \theta(q))\downarrow$ and $Approp(\text{F}, \theta(q)) \sqsubseteq \theta(\delta(\text{F}, q))$.*

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

Definition 2.33. *Let TypRan : $\mathcal{STF} \longrightarrow \mathcal{STF}$ be the partial function such that, given $F = \langle Q, \bar{q}, \theta, \delta, \not\leftrightarrow \rangle \in \mathcal{STF}$, $TypRan(F) = \langle Q, \bar{q}, \theta^R, \delta, \not\leftrightarrow \rangle$, where $\theta^R(q) = \theta(q) \sqcup \bigsqcup\{Approp(\text{F}, \theta(q'))|\text{F} \in Feat \& q' \in Q \& \delta(\text{F}, q') = q\}$ provided the joins in the definition of θ^R exist, and is undefined otherwise.*

Proposition 2.8. *For every $F \in \mathcal{STF}$, if $TypRan(F)\downarrow$, then $TypRan(F) \in \mathcal{TF}$.*

Definition 2.34. *Let TypInf : $\mathcal{F} \longrightarrow \mathcal{TF}$ be the partial function such that TypInf = TypRan $\circ$ TypDom.*

Proposition 2.9. *If $F \in \mathcal{F}$, and $F' \in \mathcal{TF}$, then $F \sqsubseteq F'$ iff $TypInf(F) \sqsubseteq F'$.*

Proof. Proven by Carpenter [1992]. □

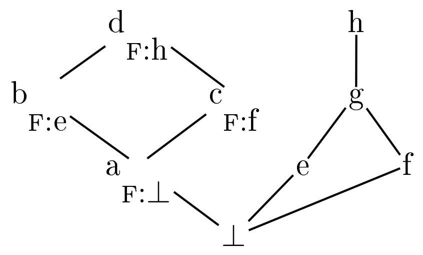

Figure 2.18: A non-statically typable signature.

Corollary 2.2. *$F \in \mathcal{TF}$ iff TypInf$(F) \sim F$.*

Proof. On analogy to Corollary 2.1. □

By analogy to *TypDom*, *TypInf* ("type inferencing") promotes any feature structure to its least well-typed extension, if one exists.

Just as with *TypDom*, we should also ask whether unification preserves the work of *TypInf*. Unfortunately, this is not always the case. Figure 2.18 shows a counter-example. If we unify a well-typed feature structure of type b with a well-typed feature structure of type c, we may wind up with a feature structure of type d, whose F value is only of type g, not h, and which is therefore not well-typed. There are some signatures in the present formulation that are inherently not statically well-typable. As a result, unification in $\mathcal{TF}$ looks like unification in $\mathcal{F}$ followed by an application of *TypInf* in order to extend the result back into $\mathcal{TF}$:

Theorem 2.4. *If $F, F', F'' \in \mathcal{TF}$, then $F \sqsubseteq F''$ and $F' \sqsubseteq F''$ iff TypInf$(F \sqcup F') \sqsubseteq F''$.*

Proof. Proven by Carpenter [1992]. □

Figure 2.19 depicts what must happen in the case of the $\mathcal{TF}$ induced by the signature in Figure 2.18.

Total Well-typedness

The third view of appropriateness is that of necessary conditions on the occurrence of certain features. This is the converse of semi-well-typing, and says that if a feature is appropriate to a node, then the node must bear that feature, thus naturally bringing semi-well-typing along with it. We will also require well-typing.

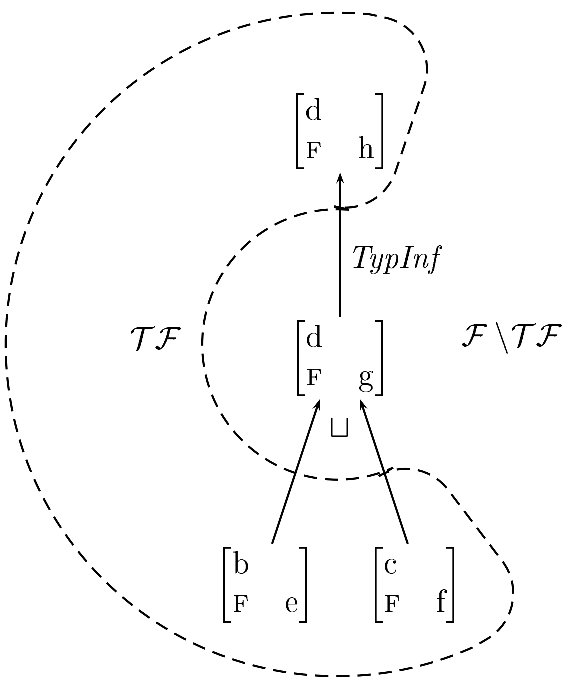
$\begin{bmatrix}\text{d}\\ \text{F}\quad \text{h}\end{bmatrix}$
TypInf
$\mathcal{TF}$
$\begin{bmatrix}\text{d}\\ \text{F}\quad \text{g}\end{bmatrix}$
$\mathcal{F}\backslash\mathcal{TF}$
$\sqcup$
$\begin{bmatrix}\text{b}\\ \text{F}\quad \text{e}\end{bmatrix}$
$\begin{bmatrix}\text{c}\\ \text{F}\quad \text{f}\end{bmatrix}$

Figure 2.19: Well-typed unification in a non-statically typable signature.

Definition 2.35. *A typed feature structure, $F = \langle Q, \bar{q}, \theta, \delta, \not\leftrightarrow\rangle$ is* totally well-typed *iff it is well-typed and, for every $q \in Q$,$\text{F} \in Feat$, if $Approp(\text{F}, \theta(q))\downarrow$, then $\delta(\text{F}, q)\downarrow$.*

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

Definition 2.36. *Let $Fill : \mathcal{TF} \longrightarrow \mathcal{TF}$ be a total function such that, given $F = \langle Q, \bar{q}, \theta, \delta, \not\leftrightarrow\rangle \in \mathcal{TF}$, $Fill(F) = \langle Q \cup Q^{Fill}, \bar{q}, \theta^{Fill}, \delta^{Fill}, \not\leftrightarrow\rangle$, where Q^{Fill} is a smallest set such that:*

- *$Q^{Fill} \cap Q = \emptyset$, and for every $q^{Fill} \in Q^{Fill}$, if $\delta^{Fill}(\text{F}, q) = q^{Fill}$ and $\delta^{Fill}(\text{F'}, q') = q^{Fill}$, then $\text{F} = \text{F'}$ and $q = q'$,*
- $$\delta^{Fill}(\text{F}, q) = \begin{cases} \delta(\text{F}, q) & \textit{if } \delta(\text{F}, q)\downarrow, \\ q^{Fill} & \textit{some } q^{Fill} \in Q^{Fill}, \textit{ if } \delta(\text{F}, q)\uparrow \textit{ and} \\ & Approp(\text{F}, \theta(q))\downarrow \\ \textit{undefined} & \textit{otherwise} \end{cases}$$
- $$\theta^{Fill}(q) = \begin{cases} \theta(q) & \textit{if } q \in Q, \\ Approp(\text{F}, \theta(q')) & \textit{if } q \in Q^{Fill}, q = \delta^{Fill}(\text{F}, q') \end{cases}$$

Proposition 2.10. *For every $F \in \mathcal{TF}$, $Fill(F) \in \mathcal{TTF}$.*

Notice that *Fill* is a total function. We can promote any well-typed feature structure to a totally well-typed one. If we combine it with *TypInf*, then we obtain a unification function for $\mathcal{TTF}$ — a partial function that promotes any $\mathcal{TTF}$-extensible $\mathcal{F}$-unification to its minimal totally well-typed extension.

Proposition 2.11. *If $F \in \mathcal{TF}$ and $F' \in \mathcal{TTF}$, $F \sqsubseteq F'$ iff $\mathit{Fill}(F) \sqsubseteq F'$.*

Proof. Proven by Carpenter [1992]. $\square$

Definition 2.37. *Let $TWT : \mathcal{F} \longrightarrow \mathcal{TTF}$ be the partial function such that $TWT = \mathit{Fill} \circ \mathit{TypInf}$.*

Corollary 2.3. *$F \in \mathcal{TTF}$ iff $TWT(F) \sim F$.*

Proof. On analogy to Corollary 2.1. $\square$

Theorem 2.5. *If $F, F', F'' \in \mathcal{TTF}$, then $F \sqsubseteq F''$ and $F' \sqsubseteq F''$ iff $TWT(F \sqcup F') \sqsubseteq F''$.*

Proof. Proven by Carpenter [1992]. $\square$

Total well-typing is the interpretation of appropriateness used throughout the rest of this dissertation. Some of its advantages with respect to simulating non-total but well-typed interpretations will be demonstrated later; but there are two important prior arguments for this choice as well. The first is that total well-typing captures one of the main intuitions of appropriateness — distinguishing unknown or absent information from irrelevant information. Feature structures of a particular type represent our knowledge about objects in the world, and those objects are known to have certain attributes on the basis of belonging to that type, whether we know what their attributes' values are or not. The view of feature structures as partial information allows us to use non-maximally specific types as the types of vague values of unknown attributes, so the existence of the attributes themselves is still consistent with that view, and should be inferred.

The other argument is a computational one. Total well-typing allows us to infer exactly how many attributes a feature structure of a particular type will have. That means we have information about the size of its representation, which can be used to simplify memory allocation and unification algorithms that act on it. Because features can be introduced at any type, that still does not mean that the arity or size of a feature structure's representation will never change. When unified with another feature structure, the type of the result could promote, and it may acquire new features. The type of a feature's value could also promote upon application of *TypInf* because of a different (but still consistent) value restriction. The fact that the type can in fact promote and the fact that its arity may change when it does so are the two significant sources of complexity in working with the unification of typed feature structures in comparison to first-order terms; but it would be even worse if the arity were not constant even for a fixed type.

For totally well-typed feature structures, we also have the same duality as with general feature structures:

Corollary 2.4. $\langle \mathcal{TTF}/{\sim}, \sqsubseteq^{\sim}_{\mathcal{TTF}} \rangle$ *is a type hierarchy.*

Proof. $\sqsubseteq^{\sim}_{\mathcal{TTF}}$ is just the restriction of $\sqsubseteq^{\sim}$ to $\mathcal{TTF}$, so it is also a partial order. By Theorem 2.5, it is bounded complete. ☐

This can also be proven [Carpenter, 1992] by showing that *TWT* is a closure operator over domains.

2.1.7 Join Preservation

In the last subsection, we saw that there are some signatures for which $\mathcal{F}$-unification does not extend automatically to $\mathcal{TF}$- or $\mathcal{TTF}$-unification. Those signatures can be explicitly characterized (this definition will be reformulated in Chapter 6):

Definition 2.38. (Tentative) *An appropriateness specification is said to* preserve joins *iff, for all features* $\textsc{f} \in \mathit{Feat}$*, for all types* s, t *such that* $s \sqcup t{\downarrow}$*:*

$$\mathit{Approp}(\textsc{f}, s \sqcup t) = \begin{cases} \mathit{Approp}(\textsc{f}, s) \sqcup \mathit{Approp}(\textsc{f}, t) & \textit{if } \mathit{Approp}(\textsc{f}, s){\downarrow} \textit{ and} \\ & \mathit{Approp}(\textsc{f}, t){\downarrow} \\ \mathit{Approp}(\textsc{f}, s) & \textit{if only } \mathit{Approp}(\textsc{f}, s){\downarrow} \\ \mathit{Approp}(\textsc{f}, t) & \textit{if only } \mathit{Approp}(\textsc{f}, t){\downarrow} \\ \textit{unrestricted} & \textit{otherwise} \end{cases}$$

Proposition 2.12. *Approp is join-preserving iff for any* $F, F' \in \mathcal{TF}$ *such that* $F \sqcup F'{\downarrow}$*,* $F \sqcup F' \in \mathcal{TF}$*.*

Proof. The forward direction was proven by Carpenter [1992]. Suppose *Approp* is not join-preserving. Then there are types s and t and a feature

$f \in$ *Feat* for which $s \sqcup t\!\downarrow$ and one of the three conditions above do not hold, depending on whether $\mathit{Approp}(\textsc{f}, s)\!\downarrow$ and/or $\mathit{Approp}(\textsc{f}, t)\!\downarrow$. Let $F_s, F_t \in \mathcal{F}$ be feature structures with no features and one node of type s, and t respectively. Clearly, $TWT(F_s)$ and $TWT(F_t)$ exist, which are the least (by subsumption) totally well-typed feature structures of types s and t respectively. They are also in $\mathcal{TF}$. Since they are least, they have no re-entrant nodes, i.e., there is a unique node q for every pair of feature and node, F' and q', for which $\delta(\text{F'}, q')\!\downarrow$, and $\theta(q) = \mathit{Approp}(\text{F'}, \theta(q'))$. By right monotonicity of *Approp* and the bounded completeness of the type hierarchy, $TWT(F_s)$ and $TWT(F_t)$ are then unifiable. But in $TWT(F_s) \sqcup TWT(F_t)$, consider $q = \delta(\textsc{f}, \bar{q})$. Its type is either $\mathit{Approp}(\textsc{f}, s)$, $\mathit{Approp}(\textsc{f}, t)$, or $\mathit{Approp}(\textsc{f}, s \sqcup t)$, depending on which case of join-preservation s and t belong to. They violate that case, however. By right monotonicity, $\theta(q) \sqsubseteq \mathit{Approp}(\textsc{f}, s \sqcup t)$, so $\mathit{Approp}(\textsc{f}, s \sqcup t) \not\sqsubseteq \theta(q)$, and thus $TWT(F_s) \sqcup TWT(F_t) \notin \mathcal{TF}$. □

Proposition 2.13. *Approp is join-preserving iff for any $F, F' \in \mathcal{TTF}$ such that $F \sqcup F'\!\downarrow$, $\mathit{Fill}(F \sqcup F') \in \mathcal{TTF}$.*

Proof. Suppose *Approp* is join-preserving. By the preceding proposition, for every $F, F' \in \mathcal{TTF}$, $(F \sqcup F') \in \mathcal{TF}$. *Fill* is a total function from $\mathcal{TF}$ to $\mathcal{TTF}$, so $\mathit{Fill}(F \sqcup F') \in \mathcal{TTF}$.

Suppose *Approp* is not join-preserving. In the proof of the preceding proposition, two feature structures in $\mathcal{TTF}$ were used, and their unification was shown not to be in $\mathcal{TF}$. So they fall out of the domain of *Fill*. □

With join-preservation, one can use only *Fill* plus $\mathcal{F}$-unification; and *Fill* is total, unlike *TWT*. Carpenter [1992] identifies two further restrictions on *Approp*, either one of which allow one to eliminate *Fill* as well, leaving only $\mathcal{F}$-unification.

2.1.8 Signature Completion

Feature introduction has been argued not to be appropriate for certain empirical domains either. Just as with the condition of bounded completeness, we may ask whether it is possible to take a would-be signature without feature introduction and restore this condition through the addition of extra unique introducing types for certain appropriate features. The following algorithm achieves this:

1. Given candidate signature, S, find a feature, F, for which there is no unique introducing type. Let K be the set of minimal types to which F is appropriate, where $|K| > 1$. If there is no such feature, then stop.

2. Add a new type, v, to S, to which F is appropriate, such that:

   - for all $k \in K$, $v \sqsubseteq k$,
   - for all types, t in S, $t \sqsubseteq v$ iff for all $k \in K$, $t \sqsubseteq k$, and
   - $Approp(\text{F}, v) = Approp(\text{F}, k_1) \sqcap Approp(\text{F}, k_2) \sqcap \ldots \sqcap Approp(\text{F}, k_{|K|})$, the generalization of the value restrictions on F of the elements of K.

3. Go to (1).

In practice, the same signature completion type can be used for different features, provided that their minimal introducers are the same set, K. This clearly produces a partially ordered set with a unique introducing type for every feature. It may disturb bounded completeness, however, which means that the result must undergo meet semi-lattice completion, as described in Section 2.1.2. If generalization has already been computed, the signature completion algorithm runs in $O(fn)$, where f is the number of features, and n is the number of types.

2.1.9 Descriptions and Most General Satisfiers

The final algebraic operation to be considered is that of the most general satisfier. This operation, unlike the others, maps a description to a feature structure, and is essentially what links the description language presented in Carpenter, 1992 to feature structures themselves.

Definition 2.39. *The set of* descriptions *over a countable set of types, T a finite set of features, Feat and a countable set of variables, Var, is the least set Desc such that:*

- *$x \in Desc$, for all $x \in Var$,*
- *$\neq x$, for all $x \in Var$,*
- *$t \in Desc$, for all $t \in T$,*
- *$\pi : \phi \in Desc$, for all $\pi \in Feat^*, \phi \in Desc$,*

- $\phi \wedge \psi, \phi \vee \psi \in$ *Desc, for all* $\phi, \psi \in$ *Desc.*

NonDisjDesc is the disjunction-free fragment of this language.

This language is a consolidated and somewhat simplified form of the language plus its various extensions used in Carpenter, 1992. In the spirit of Höhfeld and Smolka [1988], and as suggested by Carpenter [1992] and pursued in Steinicke and Penn, 1999, the path equations and path inequations established by variables are taken as constraints on variable assignment functions, and the scope of variables (which in many practical applications, is larger than a single description) is implicitly closed under existential quantification of the variable assignment. For example, we should read the description, $\text{F} : \text{G} : \text{H} : x \wedge \text{I} : \text{J} : \text{K} : x$ as denoting those feature structures F, for which there exists a feature structure, G such that the value of the path $\text{F} : \text{G} : \text{H}$ in F and the value of the path $\text{I} : \text{J} : \text{K}$ in F are both G. One may also notice the absence of a general negation operator. In a description language over types with subsumption, general negation is potentially non-monotonic. In Figure 2.13, for example, $\neg b$ is true of feature structures of type a, but not of feature structures of any subtype of a. As proven by Carpenter and Penn [1993], provided one works in the domain of sort-resolved feature structures (in which subsumption reduces to identity), general negation on a variable-free fragment of this language reduces to syntactic sugar. As proven by Steinicke and Penn [1999], that reduction does not hold in the full language. One can only have two of variables, inequations, and general negation — with all three, there is again no sound and complete calculus for satisfiability. Inequations are necessary in order to encode the negation of path equations. Variables are quite useful to have in a logic programming setting, and have been shown to reduce the size of disjunctive descriptions exponentially [Kasper, 1987b]. So general negation is excluded here.

A description from this language describes a feature structure. Typically, many feature structures are described by a particular description. Carpenter [1992] gives a semantic notion of satisfiability in which feature structures are taken to model descriptions:

Definition 2.40. *An* assignment *is a total function* $\alpha : Var \longrightarrow \mathcal{F}$. *Let Assign by the set of all assignments.*

Definition 2.41. *Where* $F \in \mathcal{F}$ *is said to* satisfy $\phi \in$ *Desc, written* $F \models^{\alpha} \phi$, $\models^{\alpha}$ *is the smallest relation such that:*

- $F \models^\alpha x$ *iff* $\alpha(x) = \bar{q}$,
- $F \models^\alpha \neq\! x$ *iff* $\alpha(x) \neq \bar{q}$,
- $F \models^\alpha t$ *iff* $t \sqsubseteq \theta(\bar{q})$,
- $F \models^\alpha \pi : \phi$ *iff* $F@\pi \models^\alpha \phi$,
- $F \models^\alpha \phi \wedge \psi$ *iff* $F \models^\alpha \phi$ *and* $F \models^\alpha \psi$,
- $F \models^\alpha \phi \vee \psi$ *iff* $F \models^\alpha \phi$ *or* $F \models^\alpha \psi$.

ϕ *is* satisfiable *iff there is a* $F \in \mathcal{F}$ *such that* $F \models^\alpha \phi$.

In the absence of disjunction, there is a least such feature structure for a satisfiable description, and the set of feature structures that satisfy a description is upward closed:

Definition 2.42. *Given a description* $\phi \in Desc$, *the* satisfiers *of* ϕ *are the feature structures that satisfy* ϕ *under some assignment function:* $Sat(\phi) = \bigcup_{\alpha \in Assign} \{F | F \models^\alpha \phi\}$.

Proposition 2.14. *If* $F \in Sat(\phi)$ *and* $F \sqsubseteq F'$, *then* $F' \in Sat(\phi)$.

Proof. Proven by Carpenter [1992]. □

Proposition 2.15. *If* $\phi \in NonDisjDesc$, *and* ϕ *is satisfiable, then there is a most general satisfier* $MGSat(\phi) \in \mathcal{F}$ *such that* $F \in Sat(\phi)$ *iff* $[MGSat(\phi)]_\sim \sqsubseteq^\sim [F]_\sim$.

Proof. Proven by Carpenter [1992]. □

MGSat can also be generalized to a function whose codomain is a set of feature structures in the case of disjunctive descriptions. In the non-disjunctive case, these two propositions establish a one-to-one correspondence between descriptions and feature structures, up to alphabetic variance of nodes — the set of feature structures that satisfy a description is rooted at a unique least equivalence class of satisfiers, whose principal filter in $\langle \mathcal{F}/\sim, \sqsubseteq^\sim \rangle$ (the set of feature structure equivalence classes that it subsumes) is none other than the equivalence classes of that set of satisfiers.

In practice, one normally uses $TWT \circ MGSat$, which provides a representative from the equivalence class of least totally well-typed feature structures that satisfy a description. Totally well-typed feature structures have the very nice property that their descriptions can be extremely terse. When a feature, F, is mentioned in a description, ϕ, the type, $Intro(\text{F})$, is inferred for $TWT(MGSat(\phi))$ because of feature introduction; and when a type is mentioned or inferred, all of the features appropriate to that type are known to be present on $TWT(MGSat(\phi))$. In this way, total well-typing allows descriptions to identify only those paths in a potentially very large feature structure to which reference to salient information, i.e., information more specific than that provided by a most general satisfier, is needed. This extra level of indirection combined with a syntactic notion of variables that remains independent of feature structures themselves are the primary practical advantages of description logics such as those of Carpenter [1992] or Smolka [1988], which also employs a description language with variables, over term or record logics such as the ψ-terms of Aït-Kaći [1984]. In the latter, *tags*, the analogue of variables, are actually part of the terms or records themselves, and thus introduce an additional level of alphabetic variance, in a manner roughly similar to that in which nodes in the formalization presented here do. ψ-terms can also have inconsistent types. Feature structures as defined in Carpenter, 1992 do not — there are simply some descriptions with inconsistent types that are not satisfiable by any feature structure.

For the purposes of this study, the charm of most general satisfiers also works the other way — we can simply continue to work with and think about (totally well-typed) feature structures rather than descriptions, because we know that satisfiable descriptions have most general satisfiers. The only problem that remains is the alphabetic variance introduced by nodes, which will be addressed in the next chapter.

2.2 A Brief History of Typed Feature Structures

> It has been said that Aristotle sometimes mistook the rules of Greek grammar for immutable verities of logic. We can raise an analogous issue about computer simulation. To what extent do we make implicit assumptions of psychological theory when we decide to write a simulation program in an information-processing language? ...it is probable that psychological postulates enter the simulation by way of the structure of the programming lan-

guage. — Allen Newell and Herbert A. Simon, *Computers in Psychology*, p. 422, 1963

The feature structure has, under various names, occupied a central position in artificial intelligence research from the very beginning. It began life in experimental psychology as a means of characterizing mental representations of concepts. From the study of volitive acts, Narziss Ach (1871 – 1946) and his colleagues in the Würzburg school had concluded that human thought was not only guided by associations, crudely put, but also by what Ach termed "determining tendencies," influences from the presentation of a goal or task that direct or determine the associations that most prevalently or strongly accrue on the perception of a stimulus [Ach, 1951, 1905]. These determining tendencies were alleged to give thought its intentional, ordered nature, as opposed to some cacophonous chorus of random associations [Humphrey, 1951, pp. 83–4].

This idea was later used by psychologists working on concept learning and category formation to formulate descriptions of objects that were used to illustrate concepts to be learned, in which these determining tendencies were encoded as attributes, and the concepts themselves were characterized as collections of attribute-value pairs [Hunt, 1962]. This began with the more formal work of Hovland [1952] on learning theory, and became popular with its adoption for experimental work on concept learning, notably in Bruner et al., 1956.

2.2.1 Description Lists

In terms of its use as a structure for representing knowledge in actual implementations, feature structures were first used as a data structure for encoding relational formulae from logic in a computer's memory in a manner inspired by and faithful to research in experimental psychology. When Allen Newell and Herbert Simon embarked upon the task of creating a programming language for simulating human reasoning, they knew that they would need very expressive data structures that could also serve as representations of human memory. They saw that it was possible to encode arbitrary quantifier-free predicates using nested lists of attribute-value pairs by regarding attributes as binary relational or functional symbols. They also saw that lists of attribute-value pairs can directly represent Ach's directed associations, where the attribute is used to select the most appropriate association given a particular context for a stimulus, and thus its determining tendencies (H. Simon, p.c.). These lists, called *description lists*, were realized as early as 1956 in Logic Language (LL), a language for supporting the automated theorem proving system, Logic Theorist (LT) [Newell and Simon, 1956]. LL later developed into the more widely circulated series of languages, IPL [Newell and Shaw, 1957], where their use was quickly generalized to empirical domains other than logic itself, e.g., relations among goals and moves in a chess-playing program [Newell et al., 1958]. Plain lists were also included in IPL for representing normal (unlabelled) associations.

Having been incorporated into IPL, description lists were also one of the topics presented at the Dartmouth Conference (H. Simon, p.c.) where the term "artificial intelligence" is said to have been coined. As a result of that meeting, they were borrowed, along with lists, into LISP [McCarthy, 1960], although not as a full-fledged construct, but rather as a useful internal device (renamed "property lists") for tracking state information in early LISP interpreters. Their subservient status, however, did not prevent their continued use as an explicit knowledge representation device. The SIR (Semantic Information Retrieval) system [Raphael, 1964, 1968] was a natural language dialog system with no built-in world knowledge that could acquire directed associations from interaction with a user and then answer questions based on those associations. It was implemented in LISP and used property lists to represent the directed associations.

SIR was a first in several respects. It appears to have been the first natural language processing system to use this kind of data structure, although it used it for representing only knowledge about the domain under discussion, not linguistic knowledge to support the vehicle of discussion. SIR also marks a methodological turning point in that it appears to have been one of the first systems to allow the choice of this data structure to guide its decisions about knowledge representation rather than *vice versa*: in Raphael, 1968 (p. 48), the choice of semantic representation is very candidly ascribed to one among what was available in the artificial intelligence programming languages of the day. Among these decisions, and perhaps the most lasting legacy of SIR, was the decision to explicitly represent hyponymy or subsumption as an attribute-value pair, along with a select, closed class of other common semantic relationships. This is the first use of the so-called *ISA link* in an implementation of knowledge-based reasoning or memory.[5] The attribute,

[5]Semantic networks [Quillian, 1968] or frames [Minsky, 1975] are perhaps more com-

SUBSET, was represented in SIR just as any other attribute, although the justification given for its inclusion and its prevalence in the examples given in Raphael, 1968 clearly indicate that it was already the *princeps principium* of attributes.

2.2.2 Semantic Networks

Quillian [1968] also used IPL, along with its description lists and plain lists, to implement a semantic network for encoding the meanings of words roughly as they appear in dictionary definitions. Each concept or word meaning could be analyzed as a bundle of properties, according to Quillian [1968, p. 242]. These properties were represented by *token nodes* connected in a subnetwork (called a *plane* by Quillian [1968]) to represent their association with the concept being defined, represented by a *type node*. The "meaning" of a concept was then defined as the subnetwork, taken as a whole, accessible from its type node. The only properties that could define concepts were other concepts, so there were also special links, *interplanar links*, to connect type nodes to their token node instances in other planes.

The associations themselves in the network were generally unlabelled. Attributes were only used with numerical values, and were drawn from a closed class including intensity, number and criteriality to indicate the "degree" to which a property was associated with other nodes. Quillian [1968, p. 229] was aware of the relevance of directed associations, of the use of attributes to represent them in mathematical psychology, and of the potential contradiction to that work that existed in assuming that associations were undirected, but somehow believed that the use of IPL as the underlying programming language conferred an air of respectability on this philosophical problem, i.e., that there was no contradiction because there existed a programming language that provided lists and description lists at the same time.

In other details, Quillian [1968] was very much driven by the practical problem of encoding dictionary definitions in a fairly literal fashion. It used the convention that any associative link between a type node and a token node in a plane, i.e., other than interplanar links, implied a hyponymous relationship, as commonly found in dictionary entries of the form, "An X is a Y that is/has A, B and C," in which X is taken to be a subclass of

monly credited with the idea; but the former never directly linked two concepts in a hyponymous or any other relationship and the latter explicitly argued against "inflexible, inclusion-oriented" knowledge representation schemes [Minsky, 1975, p. 251].

Y. Links between token nodes were used to represent adjectival or adverbial modification. There were also special links for representing conjunctive lists and disjunctive lists in definitions, and a special kind of three-way link that connected triples of token nodes with a relation-subject-object interpretation to encode events and their role assignments. Planes of token nodes were connected mainly in a tree pattern, although token nodes could also have multiple associations as a result of pronominal or other kinds of coreference in a dictionary definition, in which case the network was conventionally depicted with a special tag to indicate the identity of those paths. This is the historical antecedent of re-entrant nodes in feature structures. Attributes and their numerical values were only employed to encode the shades of meaning contributed by modal quantifiers such as "most" or "probably" in dictionary definitions.

Although Quillian [1968] believed that concepts could be decomposed into more elemental properties, concepts (as type nodes) stood in a network alongside the properties (as token nodes) of which they were composed, and the properties themselves were merely other associated concepts. This is not the same type-token distinction used in philosophical literature on language; in fact, it was introduced simply in order to partition the network into swaths of nodes, consisting of one type node along with many token nodes, in order to use their configuration as a representation of meaning for the type node's concept [Quillian, 1968, p. 234]. In no formal sense did this kind of network provide a semantics for its concepts. Type nodes remained uninterpreted abstractions of addresses in a computer's memory for the purposes of describing a particular computer program more lucidly, not concepts.

On the other hand, this graph-theoretic view of knowledge representation essentially characterizes all subsequent work in this area. Typed feature structures are no exception. Of course, type nodes have been moved to a network of their own, the type signature, and types decorate the (token) nodes of graphs that correspond to individual feature structures with an assignment function. The legacy becomes acutely apparent, however, whenever feature structures are used to encode higher-order functions or abstractions other than the first-order entity-to-truth mapping that the type assignment function can straightforwardly encode, and whenever non-trivial choices must be made between positing sub-concepts and positing individuals or instances of those concepts. In the former case, as can happen with feature-structure-based representations of natural language semantics, such as HPSG's treatment of quantifiers or representation of situation-style nuclei, the representation of functions as nodes in a graph alongside representations of objects to which they could potentially apply has the same flat-earthed feel to it as semantic networks and, in particular, makes the prospects of finding a transparent, syntactic representation of state, abstraction, application or composition rather grim. The latter case arises, for example, in the choice between subtype-based encodings and feature-value-based encodings of distinctions among information states that is one of the central subjects of this study. That choice exists to a great extent because there is no external criterion or constraint to evaluate what the semantic types should correspond to. A model-theoretic denotation could be constructed so that nodes, for example, are interpreted in a very heterogeneous universe of entities in the world, functions on those entities, abstract properties that they may have such as number and gender, and whatever else is necessary — the model theories that currently exist for typed feature structures permit that — but at that point, feature structures are not being used as a formal device to represent knowledge, but as a formal device to represent data structures that encode formal devices to represent knowledge. The problem is that, historically, this was all that they were intended to achieve in work such as that of Quillian [1968].

Curiously, one of the few sources of external criteria came at around this time from Chomsky and Halle [1968], who were inspired in their choice of feature structures directly by their continued application in psychology. It was also the first use of feature structures, although of a very simple form, to represent properly linguistic knowledge, which is now where they find their widest range of application. Values of features were always plus or minus, and the features themselves were a "substantively" universal collection of phonetic features, over which any language must articulate its phonological rules. Although there was no explicit semantic typing of these structures (as with all early uses of feature structures) and although a featureless network of types could have been used to achieve the same effect, given the very limited range of feature values, there was a clearly expressed agenda of using feature structures to decompose language-specific phonemes into bundles of these language-universal phonetic properties. This is certainly much more in keeping with the conceptual decomposition to which Quillian [1968] had alluded, and has become the standard way of using feature structures within the area of phonology. The use of claims about universal substantives in grammar has never been generalized to the application of feature structures to other areas of linguistics. In syntax, for example, the trend has been to claim that particular implicational constraints of grammar are either language-universal or language-specific, but that all of those constraints are free to avail themselves of the same types and features in a common signature.

2.2.3 KL-ONE

A very useful history and survey of KL-ONE and its successors can be found in Woods and Schmolze, 1990, upon which much of this subsection is based.

The advent of KL-ONE [Brachman, 1977, Brachman and Schmolze, 1985] marked the beginning of a formally and computationally mature approach to knowledge representation and classification networks. Although the inventors of languages such as IPL had thought a great deal about necessary and sufficient conditions for representations of concepts, memory etc. from the perspective of research in empirical psychology, subsequent applications of those languages had become increasingly parochial and informal in their coining of primitives and conventions of usage in order to achieve a particular sufficiency for the domain in which they were to be applied, primarily because no precise prescription came with these languages to indicate what the psychologically inspired structures they provided actually meant, or how they were to be properly used.

The original goal of Brachman [1977] was to provide a level of "epistemological" primitives (including ISA links) below that of actual concepts, that could be argued to be sufficient regardless of the empirical domain being represented. The desire to certify the validity of that argument formally ultimately led to the additional adoption of an external, denotational semantics that could be used to prove correctness, culminating in the KL-ONE successor language, KRYPTON [Brachman et al., 1983], and every successor since then.

KL-ONE followed the tradition of using graphs as an abstraction for relating concepts, but did not decorate the edges of the graphs with attributes. Instead, a different flavor of node was used to represent attributes (called "roles" in KL-ONE) in the graph, which could then be connected to the concepts that bore them. In this way, the connections between nodes could be restricted to a closed class of sufficient primitives, while still allowing for an open class of labelled associations at the conceptual level. KL-ONE was also the first formal system to provide value restrictions, although it and its successors have typically had a much more flexible approach to appropriateness than the one assumed by Carpenter [1992].

KL-ONE was also crucially influenced by work on frames [Fahlman, 1977, 1979, Minsky, 1975], which, along with contemporaneous work on object-oriented programming languages, was responsible for a paradigm shift in artificial intelligence from the view of relatively atomic concepts being associated to other external such concepts to a view of concepts as classes of internally structured instances that possess attributes and values that are, in turn, internally structured instances of other concepts. This concept-oriented or object-oriented view of knowledge representation structures in turn provoked an inquiry into how instances of subconcepts acquire or *inherit* attributes and values from superconcepts that they are also instances of. Until KL-ONE, however, work on frames centered on providing a precise operational characterization of how inheritance was computed as well as efficient algorithms and data structures for achieving that [Fahlman, 1979], rather than a denotational characterization of what the nature and structure of a given instance actually was. The object-oriented, recursive nature of feature structures as well as the importance attached to the inheritance of attributes stems from this work, although inheritance was initially conceived of as inherently subject to default reasoning, a trend that did not return to formalizations of feature logic until relatively recently [Lascarides and Copestake, 1999].

The use of denotational semantics as a means of ensuring correctness did have some antecedents as far back as work in semantic networks, e.g., Schubert, 1976, and frames, e.g., Hayes, 1979. The most significant contribution that the early KL-ONE languages made was in showing that a formal notion of correctness plus a sufficient, universal substrate of concept-structuring primitives could be used to automate significant portions of the classification process itself. The first so-automated portion was a pair of greatest lower bound and least upper bound constructions called "most specific subsumer" and "most general subsumee" [Woods, 1979], with the latter being roughly analogous to a most general satisfier. Such algorithms were made possible because of common agreement on the sufficiency and meaning of the provided primitives and thus of higher-order notions based on them. The correctness of the algorithms that implemented these operations could then be derived from the formal semantics of the language. Semantic networks, by contrast, required human intervention to ensure that a new concept was inserted in its proper place.

Along with automatic classification come concerns about the tractability of the tasks that are being automated. The goal of providing formally correct as well as tractable automated classification in a knowledge representation language was first articulated by Levesque [1981a,b], and eventually implemented in KRYPTON by distinguishing a "T-box," or terminological reasoning component, in which conceptual terms that can participate in constraints or rules are defined, from an "A-box," or assertional reasoning component, in which the constraints or rules themselves are stated. Although not every subsequent system has been as strict in enforcing this separation, the T-box is generally where restricted but automated, tractable reasoning and classification can occur, while the A-box is typically where functionality is supported that is still very much in demand but cannot necessarily support automatic or tractable inference. Some languages, e.g., LOOM [MacGregor, 1991, 1988], include full first-order reasoning in their assertional component. Within the realm of typed feature structures, signatures correspond to T-boxes — they define the types and features that can occur in rules or constraints. Computation of least upper bounds (unification), inheritance (subsumption) and appropriateness (value restriction) are all typical operations that a KL-ONE-style language would support in its T-box.

KRYPTON actually drew a few other subtle but important distinctions among the statements of its language [Brachman, 1983]. Those included *modality*, i.e., whether a statement is an analytic or contingent truth, *quantification*, i.e., whether a statement is universally true or simply true by default, and *matrix*, i.e., whether an ISA statement is to be interpreted set-theoretically as inclusion or predicatively as a material conditional. These distinctions have largely been eroded in successive languages simply because of the difficulty that exists in classifying some statements according to one or more of them

In the present formulation of typed feature structures, quantification is not an issue in the signature; but several programming languages based on the logic of typed feature structures employ Prolog-like reasoning with negation-by-failure, which effectively admits defaults into their assertional components. The duality between set-theoretic and material conditional interpretations of signatures can be seen by comparing Carpenter, 1992 with its contemporaries' mostly set-theoretic treatments of feature logic. Well-typedness of feature values, for example, is enforced by the implication, "if $\delta(\text{F}, q)\downarrow$, then $\mathit{Approp}(\text{F}, \theta(q)) \sqsubseteq \theta(\delta(\text{F}, q))$," without an indication of what the denotations of $\mathit{Approp}(\text{F}, \theta(q))$ and $\theta(\delta(\text{F}, q))$ actually are, or of what nodes such as q and $\delta(\text{F}, q)$ really represent. Of course, a set-theoretic interpretation can easily be provided; in fact, all of the formulations of feature logic that do provide models say next to nothing about the universe of objects which they claim that nodes represent. It is also interesting to note that, of the four distinctions drawn by Brachman [1983], matrix is the only one not implemented in KRYPTON.[6]

As for modality, another trend in typed feature logic, notably in King and Goetz, 1993, Gerdemann and King, 1994, and Gerdemann, 1995a, has been to simultaneously reject total well-typing as the interpretation of appropriateness, the unique feature introduction requirement, and the use of assertional-component functionality, e.g., general implicational constraints, to specify necessary conditions that might otherwise be relegated to appropriateness, such as feature value cooccurrence restrictions or subtype partitioning conditions. Total well-typing is a necessary and sufficient interpretation of appropriateness conditions, while well-typing with unique feature introduction is only sufficient, and without it, neither necessary nor sufficient for deterministic type inference. The result is an impaired ability to exploit necessary conditions in type inferencing, and an *ad hoc* criterion for distributing constraints between the T-box (signature) and A-box (theory) (specifically, one based on a very literal reading of Pollard and Sag, 1987, 1994). The trend in KRYPTON and other KL-ONE-like languages has been in the exactly opposite direction: to exclude purely sufficient conditions entirely, or at the very least from the T-box, or to blur the entire T-box/A-box distinction, as in CLASSIC [Borgida et al., 1989, Patel-Schneider et al., 1998], in which case sufficient conditions are still excluded from the definitional statements that are used to drive automatic classification. There, the emphasis has been on isolating a collection of necessary conditions (sometimes even contingent necessary conditions) that can tractably and infallibly apply to terms in the assertional component to force early contradictions.

2.2.4 Feature Structure Unification and Beyond

A very good introduction to feature logics in the modern sense can be found in Keller, 1993.

Feature structures began to take their present form beginning with Kay [1979], who used them as the basis of a language of descriptions in a model

[6]In a later version, KRYPTON eventually prohibited primitive concepts from having necessary conditions attached to them in the T-box, which can be read as a rejection of material conditional content from the T-box.

of human language production and comprehension. While informally presented, the exposition makes it clear that feature structures were viewed as being composed of sets of "basic" equations between paths and either atomic values (roughly, types) or other paths (re-entrancies), much like the abstract feature structures that will be presented here in Chapter 3. Kay [1979] also identified unification as the operation *par excellence* to be performed on these structures, as it was the role of grammar in this model to state the constraints on making incomplete descriptions of utterances more complete, either by adding semantic and functional information to a description of a phonological string, which constitutes parsing, or by adding the necessary phonological string to a description of semantic and functional constraints, which constitutes generation or surface string realization. Graph-theoretic approaches to knowledge representation prior to this time had only explored the possibility of unifying individuals' attribute graphs themselves (as opposed to calculating least upper bounds of concept pairs) to a limited extent as a means of inference in artificial intelligence. A salient example in vision research was that of Winston [1975], who calculated similarities and differences between pairs of graphs for scene comparison and identification in a process that closely resembles graph unification between feature structures.

This new view of grammar, presumably inspired by early work on unification [Robinson, 1965] and logic programming [Kowalski, 1974] in computer science, sparked a revolution in formal approaches to natural language syntax in the early 1980s, leading to the advent of three new and very productive schools of grammar, all with different variations on what feature structures were and how they were to be used: Lexical-Functional Grammar (LFG) [Kaplan and Bresnan, 1982], Generalized Phrase Structure Grammar (GPSG) [Gazdar et al., 1985], and Kay's own theory, which developed into what is now called Functional Unification Grammar (FUG) [Kay, 1979, 1984]. Kay [1985] elaborated on the significance of feature structures and unification in computational linguistics relative to developments in logic programming (which experienced an explosion of interest at around the same time), and claimed that the two major benefits of feature structures over logic programming with Prolog-like terms was the use of attributes for named access to substructures, and unbounded arity. While this dissertation rejects the empirical necessity of infinitely branching terms as a matter of principle, the formalization of appropriateness presented by Carpenter [1992] and followed here does still allow for a limited degree of arity incrementation relative to the type system (which Kay [1979, 1985] did not have). As discussed earlier in this chapter, all feature structures of a given type have the same arity, but as that type is refined, the arity can increase in fixed signature-specified increments.

The first formal treatment of feature description languages in this "modern" view came with Aït-Kaći [1984]. ψ-terms, as they were called there, have already been discussed in the previous section. Aït-Kaći [1984], particularly his use of semantic types with type subsumption, along with the influence of LFG and FUG, caused a substantial revision of GPSG, called Head-driven Phrase Structure Grammar (HPSG, Pollard and Sag, 1987). HPSG was the first of these linguistic theories to take typing seriously. HPSG also had a significant impact on the logic of Carpenter [1992], in so far as a straightforward encoding of HPSG's type system and principles was an important special case of its overall application. That effect can be felt in both its treatment of types — Carpenter, 1992 and (again, by way of HPSG) King, 1989 are the only two formal approaches to feature structures that use a partially ordered set of types — and appropriateness, which was actually first conceived of by King [1989] as a formalization of typing constraints that appeared to be assumed in HPSG's treatment of feature structures. It is also apparent in the applications presented by Carpenter [1992], such as recursive type constraints and logic programming.

The intuition behind feature structures in HPSG initially took the view that feature structures were partially ordered terms corresponding to successively refined information states about an utterance from the perspective of language processing, as had Kay [1979]. A later version of the theory [Pollard and Sag, 1994] partially rejected that view in favor of a mixture of a theory of language proper and a practical view of how to parse relative to that in a tractable manner. That revision came mostly as a result of the persuasion of King [1989], who had recast HPSG in light of some of the philosophical aspirations expressed in Pollard and Sag [1987], and in so doing, recast the language of typed feature structures as a conservative fragment of the language of first-order logic. This essentially shifted the meaning of "feature structure" from the descriptions of utterances to formal entities denoting the utterances themselves — a distinction that was not entirely clear to begin with in HPSG. Relative to those, Carpenter [1992] falls somewhere in between, taking feature structures not to be the same as descriptions, but still thinking of them as partially ordered.

The specific allocation of types and features in signatures also drifted between Pollard and Sag, 1987 and Pollard and Sag, 1994. Pollard and Sag,

$$\begin{bmatrix} \text{basic-circumstance} & \\ \textsc{reln} & \text{see} \\ \textsc{seer} & \text{kim} \\ \textsc{seen} & \text{sandy} \end{bmatrix}$$

Figure 2.20: A semantic representation from Pollard and Sag, 1987.

$$\begin{bmatrix} \text{see } (\sqsupseteq \text{ basic-circumstance}) & \\ \textsc{seer} & \text{kim} \\ \textsc{seen} & \text{sandy} \end{bmatrix}$$

Figure 2.21: A semantic representation from Pollard and Sag, 1994.

1987, for example, represented semantic content using feature structures of type, *circumstance*, with a feature, RELN, describing the kind of semantic relation that holds, along with various other features appropriate (in the non-technical sense) to that relation. Figure 2.20, for example, could depict the circumstance of someone named Kim seeing someone named Sandy, where *basic-circumstance* is a subtype of *circumstance* for semantic relations that are quantifier-free and coordination-free. In Pollard and Sag, 1994, types such as *see* instead appear as subtypes of *basic-circumstance* (renamed *quantifier-free-psoa*), with the features appropriate to seeing being introduced by *see* itself and RELN eliminated entirely, as in Figure 2.21. This adjustment reflects an effort on the part of the authors to define signatures as closely as possible to their intended use, specifically, to the intended cooccurrences that should exist among types and features, as well as an adherence to some intuitions concerning how types in the logic should correspond to the semantic types being represented. In simple quantifier-free, coordination-free cases such as those shown here, that correspondence can be rather transparent.

The early view of feature structures in LFG was formalized by Johnson [1988], essentially as only a path function on "attribute-value elements," that roughly correspond to nodes in Carpenter, 1992. They also include atomic values which, as in the original proposal of Kay [1979], could be cast into Carpenter, 1992 as mutually incomparable extensional types with no appropriate features. This implies, among other things, that two feature paths that terminate in an atomic value are considered to be re-entrant iff they terminate in the same atomic value. Feature structures to Johnson [1988] are also not partially ordered, but represent "total" information about actual linguistic entities. Just as in Carpenter, 1992, they are also taken to be models of an attribute-value description language, but unlike Carpenter, 1992, are taken themselves to be interpretable on Gorn trees — this particular interpretation is connected with LFG's tight association of feature structures that represent functional linguistic information, called *f-structures* with phrase-structure trees that represent certain information pertaining to constituency, called *c-structures*. Johnson [1988] was also one of the first to consider cyclic feature structures, along with Moshier [1988]. The use of regular expressions in feature paths, called *functional uncertainty* and quite common in LFG research beginning with Kaplan and Zaenen [1986], however, was not incorporated into a feature logic until Keller, 1993. One may also note, in this context, the possible extension of feature logic to negative and disjunctive feature values [Karttunen, 1984], which appear quite frequently in feature-structure-based linguistics literature, as well as set-valued features [Carpenter, 1993a, Manandhar, 1994, Moshier and Pollard, 1994, Richter, in prep.], which have been essential to HPSG among other approaches.

A closely related treatment of feature logic to those of Johnson [1988] and King [1989] was that of Smolka [1988], who presented a description language along with a model-theoretic semantics of which feature structures (as unordered, totally informative structures again) were one admissible model. Smolka [1988] was the first to combine a semantic notion of typing with the proper level of intensionality in a description language for talking about feature structures. Although *sorts* with no partial order among them were used, terms could describe objects that were both sorted and feature-bearing.

The view of feature structures and description language in Carpenter, 1992 itself was more heavily influenced by earlier work that took feature structures to be partially ordered information states, starting with Pereira and Shieber, 1984 and culminating in Moshier, 1988, which is perhaps the closest work in spirit to Carpenter, 1992, and description languages for them, most notably Rounds-Kasper Logic [Kasper, 1987a, Kasper and Rounds, 1986, 1990, Rounds and Kasper, 1986]. Both Moshier, 1988 and Rounds-Kasper Logic were untyped in the semantic sense, although they both provided access to a collection of atoms, much like Kay, 1979.

Carpenter [1992] also was the first to generalize the type system to include both intensional and extensional types, the first to provide a notion of appropriateness that can essentially be used to define the approaches to typing taken in other accounts, and the first to give inequations a first-class status both in feature structures and the description language. Smolka, 1988 and successor languages that viewed objects being described as total have used classical negation, with the usual set-theoretic interpretation; but, in fact, negation was a significant sticking point for earlier work that assumed partially ordered objects. Inequations are a particular subset of negated descriptions that can be treated very elegantly and monotonically in a partially ordered setting. The influence towards this choice again came from logic programming, specifically treatments of negation in constraint logic programming, as in Colmerauer, 1984, 1987. An influential alternative proposal was the use of intuitionistic negation in the description language [Moshier and Rounds, 1987].

2.3 Summary

This chapter presented some basic facts from Carpenter, 1992 that will be useful in the development presented later on, along with a brief history of the use of feature structures and related structures within artificial intelligence. Feature structures have proven to be useful because of their ability to combine named attributes with subsumption in a way, due to appropriateness and total well-typing, that allows terse descriptions to pinpoint a sparse amount of information within potentially very large data structures.

We have also seen our first glimpse of the duality that exists between subtype-based and feature-based encodings of information by noting that the set of totally well-typed feature structures modulo alphabetic variance is essentially a type hierarchy. That duality is visible due to certain generalizations in the treatment of typed feature logic here that depart from a more mundane, practical view of feature structures in which infinite feature structures are disallowed and the number of types must be finite rather than merely countable.

On the other hand, some of the other restrictions conventionally assumed in the course of studies of feature logics, such as unique feature introduction and bounded completeness, have been adopted here as well. These have been widely criticized as restricting the applicability of feature logic to linguistics. The sections on meet semi-lattice completions and signature completions have argued that those assumptions are warranted insofar as even very large non-compliant signatures to date can be modified so as to observe them in a very small amount of time.

Chapter 3

Abstract Feature Structures and Signature Subsumption

In the last chapter, several results were presented that allow us to regard $\mathcal{F}$ and $\mathcal{TTF}$ as type hierarchies, in a rather loose rendering of the term, provided that we first collect their feature structures into equivalence classes as defined by alphabetic variance. In the same way that a semantic view of feature structures abstracted away from some of the irregularities that syntactic descriptions must admit, we can further abstract away from the alphabetic variance that the nodes of feature structures admit. *Abstract feature structures* were first introduced in the context of acyclic feature structures with typed terminal nodes by Moshier and Rounds [1987], extended to cyclic feature structures by Moshier [1988], and extended to typed feature structures by Pollard and Moshier [1990]. They are also discussed by Carpenter [1992]. Their presentation here extends them to typed feature structures with inequations. It will also be shown that the notion of total well-typing, together with its inclusion operation, TWT, can be extended to abstract feature structures.

Totally well-typed abstract feature structures are the right level of abstraction for looking at the information content of feature structures. As such, they are also the right level of abstraction for thinking about the ability of signatures to simulate the behavior of other signatures with respect to unification, because that simulation intuitively means that one can conduct computations for one signature in another one, translate back, and emerge with the same information, not with any particular correspondence between nodes.

After introducing abstract feature structures, we will be ready to define signature equivalence and signature subsumption. The former formalizes the question posed in the introduction about what it means for two signatures to be empirically equivalent. The latter also establishes an encoding of one signature by another but only in one direction. This can have some practical advantage since not all logically equivalent signatures are computationally equivalent. It is also related to term encoding problems in knowledge representation, where the embedding is implicitly into the "signature" of first-order terms, or a similar canonical representation. These topics will be addressed in later chapters.

Central to the idea of signature equivalence and subsumption is the mathematical construct known as a join-preserving encoding or embedding. This rather well-studied class of functions provides a characterization of what it means for a correspondence between partial orders to preserve the behavior of their elements under unification. It will be argued here that the standard definition of this class is not quite general enough to capture all of the salient join-preserving correspondences that can exist, and a generalization that accommodates these exceptions is then given. This generalization will be important in Chapter 6, when it is used to establish a useful correspondence between feature structures and Prolog terms.

Given a notion of subsumption among signatures themselves, it is only natural to ask what structure the collection of all signatures possesses — possibly even enough structure to be a signature itself. The fourth section of this chapter considers this question.

3.1 Abstract Feature Structures

Moshier's key insight was that nodes are valuable only insofar as they identify paths. Instead of defining types, path equations and path inequations on nodes, then, we use types, subject to a few commonsensical restrictions:

Definition 3.1. *Given a set of types, T, and a set of features, Feat, an* abstract feature structure *is a tuple $A = \langle \Pi, \Theta, \approx_F, \not\approx_F \rangle$ where:*

- $\Pi \subseteq$ *Feat** *is the set of* paths*,*
- $\Theta : \Pi \longrightarrow T$ *is the total* path typing function*,*
- $\approx_F \subseteq \Pi \times \Pi$ *is the* path equation relation*, and*

- $\not\approx_F \subseteq \Pi \times \Pi$ *is the* path inequation relation,

such that:

- **prefix closure:** Π *is prefix-closed,*
- **path equivalence:** $\approx_F$ *is an equivalence relation on* Π*,*
- **inequation negativity:** $\not\approx_F$ *is symmetric and anti-reflexive,*
- **inequation disjointness:** $\approx_F \cap \not\approx_F = \emptyset$*,*
- **prefix consistency:** *if* $\pi\text{F} \in \Pi$ *and* $\pi \approx_F \pi'$*, then* $\pi'\text{F} \in \Pi$ *and* $\pi\text{F} \approx_F \pi'\text{F}$*,*
- **inequation consistency:** *if* $\pi_1 \not\approx_F \pi_2$*,* $\pi_1 \approx_F \pi_1'$ *and* $\pi_2 \approx_F \pi_2'$*, then* $\pi_1' \not\approx_F \pi_2'$*, and*
- **typing consistency:** *if* $\pi_1 \approx_F \pi_2$*, then* $\Theta(\pi_1) = \Theta(\pi_2)$*.*

$\mathcal{A}$ *is the set of abstract feature structures.*

These are an abstraction from typed feature structures with normal inequations. Fully inequated feature structures [Carpenter, 1992, p. 120] would have more restrictions on where inequations can occur.

One can also define an operation on feature structures that casts them into their abstract feature structures.

Definition 3.2. *Let* $Abs : \mathcal{F} \longrightarrow \mathcal{A}$ *be the total function such that given* $F = \langle Q, \bar{q}, \theta, \delta, \not\leftrightarrow \rangle \in \mathcal{F}$*,* $Abs(F) = \langle \Pi, \Theta, \approx_F, \not\approx_F \rangle$ *such that:*

- $\Pi = \{\pi | \delta(\pi, \bar{q})\downarrow\}$*,*
- $\Theta(\pi) = \theta(\delta(\pi, \bar{q}))$*,*
- $\pi_1 \approx_F \pi_2$ *iff* $\delta(\pi_1, \bar{q}) = \delta(\pi_2, \bar{q})$*, and*
- $\pi_1 \not\approx_F \pi_2$ *iff* $\delta(\pi_1, \bar{q}) \not\leftrightarrow \delta(\pi_2, \bar{q})$*.*

Proposition 3.1. *Abs is a surjection, i.e., every abstract feature structure stands in the image of Abs.*

Proof. Given $A = \langle \Pi, \Theta, \approx_F, \not\approx_F \rangle$, let $F = \langle [\Pi]_{\approx_F}, \Theta^{\approx_F}, \delta^{\approx_F}, \not\leftrightarrow^{\approx_F} \rangle$ such that:

- $\Theta^{\approx_F}([\pi]_{\approx_F}) = \Theta(\pi)$,
- $\delta^{\approx_F}(\text{F}, [\pi]_{\approx_F})\downarrow$ iff $\pi\text{F} \in \Pi$ and $\delta^{\approx_F}(\text{F}, [\pi]_{\approx_F}) = [\pi\text{F}]_{\approx_F}$,

- $[\pi]_{\approx_F} \not\leftrightarrow^{\approx_F} [\pi']_{\approx_F}$ iff $\pi \not\approx_F \pi'$.

$\Theta^{\approx_F}$ is well-defined by typing consistency. $\delta^{\approx_F}$ is well-defined by prefix closure and prefix consistency. $\not\leftrightarrow^{\approx_F}$ is well-defined by inequation negativity, consistency and disjointness. As can easily be verified, $Abs(F) = A$. $\square$

Abstract feature structures admit a direct characterization of subsumption, just as was shown for $\langle \mathcal{F}/\sim, \sqsubseteq^{\sim} \rangle$.

Definition 3.3. *Given* $A = \langle \Pi, \Theta, \approx_F, \not\approx_F \rangle, A' = \langle \Pi', \Theta', \approx_F', \not\approx_F' \rangle \in \mathcal{A}$, A *subsumes (*is extended by*)* A', $A \sqsubseteq A'$, *iff:*

- $\Pi \subseteq \Pi'$,
- $\approx_F \subseteq \approx_F'$,
- $\not\approx_F \subseteq \not\approx_F'$, *and*
- *for all* $\pi \in \Pi$, $\Theta(\pi) \sqsubseteq \Theta'(\pi)$.

Proposition 3.2. $F \sqsubseteq F'$ *iff* $Abs(F) \sqsubseteq Abs(F')$.

Proof. The characteristics of a morphism that witnesses $F \sqsubseteq F'$ (p. 42) directly correspond to the requirements for abstract feature structure subsumption. $\square$

Significantly, the elements of $\mathcal{A}$ correspond exactly to the equivalence classes established by the alphabetic variance relation, $\sim$:

Proposition 3.3. $Abs(F) = Abs(F')$ *iff* $F \sim F'$.

Proof. By the definition of $\sim$ and the previous proposition. $\square$

As a result, the abstraction operation corresponds to reading the information from a feature structure, and abstract feature structures correspond to information states. We can now define unification over abstract feature structures without recourse to alphabetic variance to solve the problem with non-intersecting node sets. Because unification is really only intended to combine information from feature structures, it is more natural to think of it as an operation on abstract feature structures.

Definition 3.4. *Given* $A, A' \in \mathcal{A}$ *such that* $A = \langle \Pi, \Theta, \approx_F, \not\approx_F \rangle$, *and* $A' = \langle \Pi', \Theta', \approx_F', \not\approx_F' \rangle$, *the* unification *of* A *and* A' *is* $A \sqcup A' = \langle \Pi'', \Theta^C, C, I \rangle$, *where:*

- Π'' *is the prefix-consistent closure of* $\Pi \cup \Pi'$*, i.e., the least set containing* $\Pi \cup \Pi'$ *that is prefix-consistent (as defined on p. 75),*
- C *is the transitive, prefix-consistent closure of* $\approx_F \cup \approx_F{}'$*,*
- I *is the closure of* $(\not\approx_F \cup \not\approx_F{}')$ *under* C*,*
- $\Theta^C(\pi) = \bigsqcup\{\Theta^{\sqcup}(\pi') | \langle \pi', \pi \rangle \in C\}$*, and*
- $\Theta^{\sqcup}(\pi) = \begin{cases} \Theta(\pi) \sqcup \Theta'(\pi) & \textit{if} \quad \pi \in \Pi \cap \Pi', \\ \Theta(\pi) & \textit{if only} \quad \pi \in \Pi, \\ \Theta'(\pi) & \textit{if only} \quad \pi \in \Pi', \\ \bot & \textit{otherwise}, \end{cases}$

provided:

- *the joins required by* Θ^C *and* $\Theta^{\sqcup}$ *exist, and*
- $C \cap I = \emptyset$*,*

and is undefined otherwise.

Proposition 3.4. *$\mathcal{A}$-unification is well-defined.*

Proof. Prefix closure, prefix consistency and path equivalence trivially hold. Inequation symmetry follows from the inequation symmetry and inequation disjointness of A and A'. Inequation disjointness is guaranteed by the second stipulation. Inequation consistency follows from the closure of $(\not\approx_F \cup \not\approx_F{}')$ under C. Typing consistency follows from the closure under C in Θ^C. □

The two conditions correspond to our intuitions about when unification fails, namely when typing information is inconsistent or when path inequations are violated. The explicit prefix-consistent closures are necessary to handle cases like Figure 3.1,[1] Neither abstract feature structure contains the path, FH, but because of the re-entrancy between F and G on the left, and the fact that GH is defined on the right, the result must have it to remain prefix-consistent and prefix-closed. These new paths are the ones that contribute $\bot$ — no information — to the type of their equivalence class in the last clause of $\Theta^{\sqcup}$. I must be explicitly closed under C to handle cases such as Figure 3.2, in which the violation is not directly inferred from the union of their inequations. The typing function must be closed under C for the same reasons.

We also get the same duality as before without using quotient sets.

[1]Since *Abs* is a surjection, we are justified in depicting the elements of $\mathcal{A}$ by concrete feature structure representatives.

$$\begin{bmatrix} \text{a} & \\ \text{F} & \boxed{1} \\ \text{G} & \boxed{1} \end{bmatrix} \sqcup \begin{bmatrix} \text{a} & \\ \text{F} & \text{b} \\ \text{G} & \begin{bmatrix} \text{b} & \\ \text{H} & \text{c} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \text{a} & \\ \text{F} & \boxed{1}\begin{bmatrix} \text{b} & \\ \text{H} & \text{c} \end{bmatrix} \\ \text{G} & \boxed{1} \end{bmatrix}$$

Figure 3.1: An example of the necessity of prefix-consistent closure in $\mathcal{A}$-unification.

$$\begin{bmatrix} \text{a} & & \\ \text{F} & \boxed{1} & \\ \text{G} & \boxed{2} & \\ \text{H} & \boxed{3} & \\ \text{I} & \boxed{4} & \\ \boxed{1} & \not\leftrightarrow & \boxed{2} \\ \boxed{3} & \not\leftrightarrow & \boxed{4} \end{bmatrix} \sqcup \begin{bmatrix} \text{a} & \\ \text{F} & \boxed{1} \\ \text{G} & \boxed{2} \\ \text{H} & \boxed{2} \\ \text{I} & \boxed{1} \end{bmatrix} \uparrow$$

Figure 3.2: An example of the necessity of explicitly closing path inequations under path equality in $\mathcal{A}$-unification.

Proposition 3.5. *$\langle \mathcal{A}, \sqsubseteq \rangle$ is a type hierarchy.*

Proof. By Proposition 3.3, $\sqsubseteq_{\mathcal{A}}$ is a partial order because $\sqsubseteq_{\mathcal{F}}^{\sim}$ is. The fact that it is bounded complete follows from the definition of $\mathcal{A}$-unification plus the fact that set union is the least extension of consistent sets. $\square$

The notion of total well-typing also naturally extends to abstract feature structures:

Definition 3.5. $A = \langle \Pi, \Theta, \approx_F, \not\approx_F \rangle$ *is* totally well-typed *iff:*

- *for every* $\pi\text{F} \in \Pi$*, then* $\mathit{Approp}(\text{F}, \Theta(\pi))\downarrow$ *and* $\mathit{Approp}(\text{F}, \Theta(\pi)) \sqsubseteq \Theta(\pi\text{F})$*, and*
- *for every* $\pi \in \Pi$*, if* $\mathit{Approp}(\text{F}, \Theta(\pi))\downarrow$ *then* $\pi\text{F} \in \Pi$.

$\mathcal{TTA}$ is the set of totally well-typed abstract feature structures.

Well-typing consists of satisfying only the first criterion, which gives us $\mathcal{TA}$, the set of well-typed abstract feature structures. The extension is natural because *Abs* respects the total well-typing of concrete feature structures.

Proposition 3.6. *If $F \in \mathcal{TTF}$, then $Abs(F) \in \mathcal{TTA}$.*

Proposition 3.7. *If there is an $F \in \mathcal{TTF}$ such that $Abs(F) = A$, then for all $F' \in \mathcal{F}$ for which $Abs(F') = A$, $F' \in \mathcal{TTF}$.*

Proof. Given $F' \in \mathcal{F}$ for which $Abs(F') = A$, then $Abs(F') = Abs(F)$. By Proposition 3.3, $F' \sim F$. So $F' \sqsubseteq F$, and therefore has a totally well-typed extension. By Propositions 2.9 and 2.11, $TWT(F')\downarrow$ and $F' \sqsubseteq TWT(F') \sqsubseteq F$. Also, $F \sqsubseteq F'$, so $TWT(F') \sim F \sim F'$. By Corollary 2.3, F' is totally well-typed. □

Often, operations like *TWT*, *Fill*, etc. will be thought of as applying to abstract feature structures rather than concrete feature structures, which makes sense because of this respect.

Theorem 3.1. *If $A, A', A'' \in \mathcal{TTA}$, then $A \sqsubseteq A''$ and $A' \sqsubseteq A''$ iff $Abs(TWT(A \sqcup A')) \sqsubseteq A''$.*

Proof. On analogy to Theorem 2.5. □

Corollary 3.1. *$\langle \mathcal{TTA}, \sqsubseteq_{\mathcal{TTA}} \rangle$ is a type hierarchy.*

Proof. On analogy to Corollary 2.4. □

3.2 Order-Embeddings and Join-Preserving Encodings

We can begin to consider signature equivalence by first asking how, in general, two partially ordered sets may be said to be equivalent.

Definition 3.6. *Given two partial orders $\langle P, \sqsubseteq_P \rangle$ and $\langle R, \sqsubseteq_R \rangle$, a function $f : P \longrightarrow R$ is an* order-embedding *iff, for every $x, y \in P$, $x \sqsubseteq_P y$ iff $f(x) \sqsubseteq_R f(y)$.*

Definition 3.7. *$f : P \longrightarrow R$ is an* order-isomorphism *iff it is an order-embedding and onto.*

Order-isomorphisms preserve the structure of partially ordered sets exactly, and, in so doing, preserve operations such as least upper bounds, greatest lower bounds etc., that are derived from that structure. Clearly, if f is

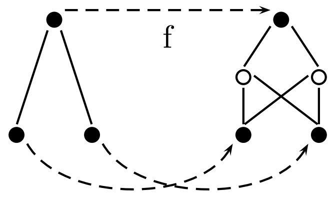

Figure 3.3: An example order-embedding that cannot translate least upper bounds.

an order-isomorphism, so is f^{-1}, so this correspondence can be used in either direction from a problem solving perspective.

Order-embeddings appear to establish that correspondence in only one direction, and they do to the extent that one can ask questions such as "does x subsume y?". As shown in Figure 3.3, however, this weaker correspondence cannot be used in general to reason about operations such as least upper bounds on bounded complete partial orders. The reason is that the image of f may not be closed under those operations in the codomain. In fact, the codomain could provide answers where none were supposed to exist, or, as in Figure 3.3, no answers where one was supposed to exist. Mellish [1991, 1992] was the first to formulate the "one-way" join-preserving encoding problem in a correct fashion, by explicitly requiring the correct behavior:

Definition 3.8. *Given two BCPOs, P and R, $f : P \longrightarrow R$ is a* classical join-preserving encoding of P into R *iff:*

- **injectivity** *f is an injection,*
- **zero preservation** *$f(p \sqcup_P q)\uparrow$ iff $f(p) \sqcup_R f(q)\uparrow$, and*
- **join homomorphism** *$f(p \sqcup_P q) = f(p) \sqcup_R f(q)$, where they exist.*

Join-preserving encodings are also order-embeddings because $p \sqcup q = q$ iff $p \sqsubseteq q$.

There is actually a more general definition:

Definition 3.9. *Given two BCPOs, P and R, $f : P \longrightarrow Pow(R)$ is a* join-preserving encoding of P into R *iff:*

- **totality** *for all $p \in P$, $f(p) \neq \emptyset$,*
- **disjointness** *$f(p) \cap f(q) \neq \emptyset$ iff $p = q$,*

- **zero preservation** *for all* $\bar{p} \in f(p)$*, and* $\bar{q} \in f(q)$*,* $p \sqcup_P q\uparrow$ *iff* $\bar{p} \sqcup_R \bar{q}\uparrow$*, and*
- **join homomorphism** *for all* $\bar{p} \in f(p)$*, for all* $\bar{q} \in f(q)$*,* $\bar{p} \sqcup_R \bar{q} \in f(p \sqcup_P q)$*, where they exist.*

When f maps elements of P to singleton sets in R, then f is a classical join-preserving encoding; but there is no reason in general to require that only one element of R can represent P, provided that it does not matter which one we choose. The utility of this generalization can seen in the following theorem, which shows that choosing total well-typing or well-typing is really a matter of expressive convenience or taste:

Theorem 3.2. *For any signature,* $S = \langle T_S, \sqsubseteq_S, Feat_S, Approp_S \rangle$*, there is:*

1. *a signature,* $R = \langle T_R, \sqsubseteq_R, Feat_R, Approp_R \rangle$ *with a function* $f : \mathcal{TF}_S \longrightarrow Pow(\mathcal{TTF}_R)$*, that induces a join-preserving encoding,* $\bar{f}$ *of* $\mathcal{TA}_S$ *into* $\mathcal{TTA}_R$*, and*
2. *a signature,* $P = \langle T_P, \sqsubseteq_P, Feat_P, Approp_P \rangle$ *with a function* $g : \mathcal{TTF}_S \longrightarrow Pow(\mathcal{TF}_P)$*, that induces a join-preserving encoding,* $\bar{g}$ *of* $\mathcal{TTA}_S$ *into* $\mathcal{TA}_P$*.*

Proof. (1) For each type in T_S, t with k appropriate features, $F = \{\text{F}_1, \ldots, \text{F}_k\}$, let T_R have 2^k types, $\{t_E | E \subseteq F\}$. Also:

- $Feat_R = Feat_S$,
- $Approp_R(\text{F}, t_E)\downarrow$ iff $Approp_S(\text{F}, t)\downarrow$ and $\text{F} \in E$, in which case $Approp_R(\text{F}, t_E) = (Approp_S(\text{F}, t))_\emptyset$, and
- $t_E \sqsubseteq_R t'_{E'}$ iff $t \sqsubseteq_S t'$ and $E \subseteq E'$.

Clearly, $\sqsubseteq_R$ is a partial order and bounded complete, where $t_E \sqcup_R t'_{E'} = (t \sqcup_S t')_{(E \cup E')}$.

Now for $F \in \mathcal{TF}_S$, let $f(F) = \{G\}$, where $G = \langle Q_F, \theta_G, \delta_F, \nleftrightarrow_F \rangle$, and $\theta_G(q) = (\theta_F(q))_{\{\text{H}|\delta_F(\text{H},q)\downarrow\}}$. Since all that is different between F and G is the node-typing function, G is obviously well-formed in $\mathcal{F}_R$. F is well-typed, so wherever $\delta_F(\text{J}, q)\downarrow$, $Approp_S(\text{J}, \theta_F(q)) \sqsubseteq_S \theta_F(\delta_F(\text{J}, q))$. To show $G \in \mathcal{TF}_R$, it must be shown that $Approp_R(\text{J}, \theta_G(q))\downarrow$ and $Approp_R(\text{J}, \theta_G(q)) \sqsubseteq_R \theta_G(\delta_F(\text{J}, q))$. The first is proven by noting that $\text{J} \in \{\text{H}|\delta_F(\text{H}, q)\downarrow\}$. To prove the second, note that $Approp_R(\text{J}, \theta_G(q)) = (Approp_S(\text{J}, \theta_F(q)))_\emptyset \sqsubseteq_R (\theta_F(\delta_F(\text{J}, q)))_\emptyset \sqsubseteq_R (\theta_F(\delta_F(\text{J}, q)))_{\{\text{H}|\delta_F(\text{H},\delta_F(\text{J},q))\downarrow\}} = \theta_G(\delta_F(\text{J}, q))$.

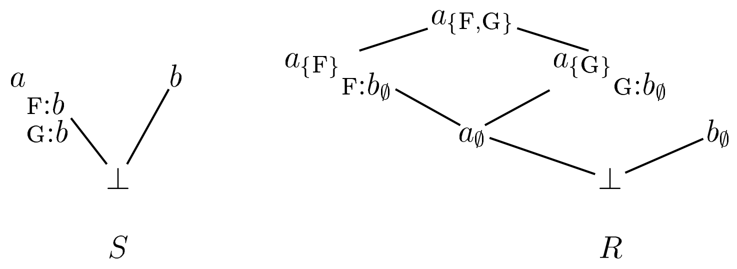

Figure 3.4: An example of S and R in the proof of Theorem 3.2.

To show $G \in \mathcal{TTF}_R$, it must also be shown that for all J and q, for which $Approp_R(\text{J}, \theta_G(q))\!\downarrow$, $\delta_F(\text{J}, q)\!\downarrow$. $\theta_G(q) = (\theta_F(q))_{\{\text{H}|\delta_F(\text{H},q)\downarrow\}}$ and $Approp_R(\text{J}, (\theta_F(q))_{\{\text{H}|\delta_F(\text{H},q)\downarrow\}})\!\downarrow$ implies, by definition, that $\text{J} \in \{\text{H}|\delta_F(\text{H}, q)\!\downarrow\}$. So $\delta_F(\text{J}, q)\!\downarrow$.

Now let $\bar{f}$ be the abstraction of f over the alphabetically variant equivalence classes of $\mathcal{TF}_S$, such that $\bar{f}(Abs_S(F)) = \{Abs_R(f(F))\}$. The fact that $\bar{f}$ is well-defined and a join-preserving encoding follows directly from the fact that $G \in f(F)$ only differs from F in its node typing function, and that $\theta_G(q)$ only differs from $\theta_F(q)$ in its type's subscript annotation with exactly the features that q bears.
(2) Let $P = S$, and for $F \in \mathcal{TTF}_S$, let $g(F) = \{F' \in \mathcal{TF}_S | \mathit{Fill}(F') = F\}$, and let $\bar{g}$ be the abstraction of g over the alphabetically variant equivalence classes of $\mathcal{TTF}_S$, such that $\bar{g}(Abs(F)) = \{Abs(F') | F' \in \mathcal{TF}_S \& \mathit{Fill}(F') = F\}$. $\bar{g}$ is disjoint because *Fill* is a function. Also, for all $Abs(\bar{F}) \in g(Abs(F))$, and $Abs(\bar{G}) \in g(Abs(G))$, $F \sqcup_{\mathcal{TTF}} G = (\mathit{Fill} \circ \mathit{TypInf})(F \sqcup G)\!\downarrow$ means $(\mathit{Fill} \circ \mathit{TypInf})(\mathit{Fill}(\bar{F}) \sqcup \mathit{Fill}(\bar{G}))\!\downarrow$. Since *Fill* is total, by Proposition 2.11 and Corollary 2.3, $(\mathit{Fill} \circ \mathit{TypInf})(\mathit{Fill}(\bar{F}) \sqcup \mathit{Fill}(\bar{G})) = (\mathit{Fill} \circ \mathit{TypInf})(\bar{F} \sqcup \bar{G})$ and thus, $\mathit{TypInf}(\bar{F} \sqcup \bar{G}) = \bar{F} \sqcup_{\mathcal{TF}} \bar{G}\!\downarrow$ and $Abs(\bar{F} \sqcup_{\mathcal{TF}} \bar{G}) \in \bar{g}(Abs(F \sqcup_{\mathcal{TTF}} G))$. The reverse of this reasoning also holds. Thus $\bar{g}$ is a join-preserving encoding. □

As an example of $\bar{f}$, one might consider the S and R shown in Figure 3.4. $\mathcal{TF}_S$ corresponds to $\mathcal{TTF}_R$ in a way that should be clear. As an example of $\bar{g}$, one can consider the encoding of $\mathcal{TTF}_S$ into $Pow(\mathcal{TF}_S)$ given in Figure 3.5, where S is again as shown in Figure 3.4. Notice that the sets are closed under unification, which is a property guaranteed by the generalized definition of join-preserving encodings.

$\bar{f}$ is a classical join-preserving encoding, but $\bar{g}$ is not. $\bar{g}$ designates *every* well-typed approximation of a totally well-typed abstract feature structure

$$\begin{array}{lcl} \bot & \longmapsto & \{\bot\} \\ \text{b} & \longmapsto & \{\text{b}\} \\ \begin{bmatrix}\text{a} & \\ \text{F} & \text{b} \\ \text{G} & \text{b}\end{bmatrix} & \longmapsto & \left\{\text{a}, \begin{bmatrix}\text{a} & \\ \text{F} & \text{b}\end{bmatrix}, \begin{bmatrix}\text{a} & \\ \text{G} & \text{b}\end{bmatrix}, \begin{bmatrix}\text{a} & \\ \text{F} & \text{b} \\ \text{G} & \text{b}\end{bmatrix}\right\} \end{array}$$

Figure 3.5: An example of $\bar{g}$ in the proof of Theorem 3.2.

as a representative of that structure in the BCPO of well-typed abstract feature structures. In fact, $\bigcup Im(\bar{g}) = \mathcal{TA}_S$, i.e., every well-typed abstract feature structure represents something; but $\bar{g}$ is not a bijection, because its codomain is $Pow(\mathcal{TA}_S)$. One could, for example, think of $\mathcal{TF}_S$ as modeling the extent to which a particular totally well-typed abstract feature structure has been filled in an application that uses a lazy filling algorithm to reduce the size of the data structures that it must copy or unify. Lazy filling could potentially change which well-typed abstract feature structure is being used as a representative in between unifications in such a system. It does not matter which representative we use because *Fill* is a total function, and can be "postponed" until all of the unifications are finished. Generalized join-preserving encoding formally delineates a set of alternatives among which lazy filling or other algebraic operations can navigate without disturbing join-preservation. Generalized join-preserving encoding will be used in the same capacity in Chapter 6.

In the case of the proof above, it is possible to extract a classical encoding from $\bar{g}$, namely the trivial inclusion encoding that chooses the single totally well-typed abstract feature structure, which maps to itself under *Fill*, from the set of well-typed abstract feature structures delineated by a set in the image of $\bar{g}$. In general, that will not be possible. Figure 3.6 shows a generalized join-preserving encoding from which no classical encoding can be extracted. In fact, there is no classical encoding at all of S into the R shown, because no three elements can be found in R that pairwise unify to yield a unique join. A generalized encoding exists because we can choose three potential representatives for d, one (h) for unifying the representatives of a and b, one (i) for unifying the representatives of b and c, and one (j) for unifying the representatives of a and c. Notice again that the representatives of d must also be closed under unification. So generalized join-preserving encoding really is a generalization, even in the absence of other non-join algebraic operations

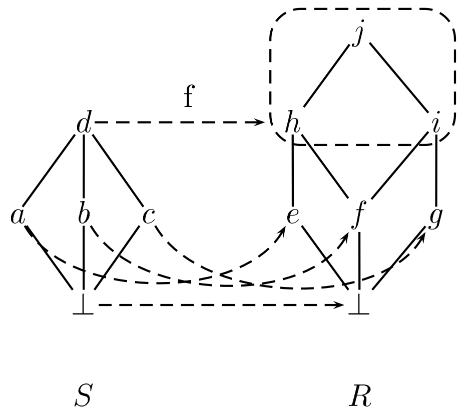

Figure 3.6: A non-classical join-preserving encoding between BCPOs for which no classical join-preserving encoding exists.

to support.

A stronger version of this idea would also require homomorphisms and zero preservation to hold for least upper bounds of infinite sets, although that will not be pursued here.

3.2.1 Symmetric Join Preserving Encodings

An important question to ask is whether the existence of a join-preserving encoding in both directions is enough to establish an order-isomorphism, i.e., equivalence in the strongest sense. Clearly if the dual join-preserving encodings are both classical and inverses of one another, then they constitute an order-isomorphism. There may, however, be cases where we only know of two join-preserving encodings, even classical ones, which only map one BCPO into a proper subset of the other. In the category of sets, the Schröder-Bernstein Theorem guarantees the existence of a bijection in this case; but that says nothing about whether that bijection preserves the structure of subsumption in the desired way.

Proposition 3.8. *If f is a classical join-preserving encoding of P into R, then f is an order-isomorphism between P and $Im(f) \subseteq R$.*

Proof. $p \sqsubseteq_P q$ iff $p \sqcup_P q = q$ iff $f(p \sqcup_P q) = f(p) \sqcup_R f(q) = f(q)$ iff $f(p) \sqsubseteq_R f(q)$. □

Theorem 3.3. *If P (or R) is finite, and there exist a classical join-preserving encoding, f, of P into R and a classical join-preserving encoding, g, of R into P, then then there exists an order-isomorphism between P and R.*

Proof. By Proposition 3.8, f is an order-isomorphism between P and $Im(f) \subseteq R$. This means that $Im(f)$ is a BCPO, and since f is a join-preserving encoding, $p \sqcup_{Im(f)} q\downarrow$ iff $p \sqcup_R q\downarrow$. So the restriction of g to $Im(f)$, $g|Im(f)$, is a join-preserving encoding. By Proposition 3.8, $g|Im(f)$ is an order-isomorphism between $Im(f)$ and $Im(g|Im(f)) \subseteq P$. Since composition of order-isomorphisms is an order-isomorphism, we have the order-isomorphism $(g|Im(f)) \circ f$ between P and $Im(g|Im(f)) \subseteq P$. Since P is finite and order-isomorphisms are bijective, $Im(g|Im(f)) = P$. But $Im(g|Im(f)) \subseteq Im(g) \subseteq P$, so $Im(g) = P$ also. By Proposition 3.8, g is an order-isomorphism between R and $Im(g) = P$. □

With the following lemma, we have the same result for generalized term encodings as a corollary:

Lemma 3.1. *If P (or R) is finite, and there exist a join-preserving encoding, f, of P into R and a join-preserving encoding, g, of R into P, then f and g map elements of P and R, respectively, to singleton sets, i.e., they induce respective classical join-preserving encodings.*

Proof. Because P is finite, $|P| \geq |\bigcup Im(g)|$, and because g is disjoint and total, $|\bigcup Im(g)| \geq |R|$, so R is finite. By the same reasoning with R and f, $|R| \geq |P|$. Thus $|P| = |R|$, and $|\bigcup Im(g)| = |P|$, so g must map elements of R to singleton sets in P, and likewise for f. □

Corollary 3.2. *If P (or R) is finite, and there exist a join-preserving encoding, f, of P into R and a join-preserving encoding, g, of R into P, then there exists an order-isomorphism between P and R.*

In the infinite case, we are not always so lucky. Figure 3.7 shows part of a classical join-preserving encoding from an infinite ascending binary tree to an infinite ascending ternary tree, and Figure 3.8 shows part of a classical join-preserving encoding in the reverse direction. These are trivially join-preserving because they are subsumption preserving and there are no join-reducible elements. There can be no order-isomorphism, however, because $b_\perp$ in one must map to $t_\perp$ in the other, since they are the only elements that subsume everything, and thus b_1 and b_2 must map to two of $\{t_1, t_2, t_3\}$ in

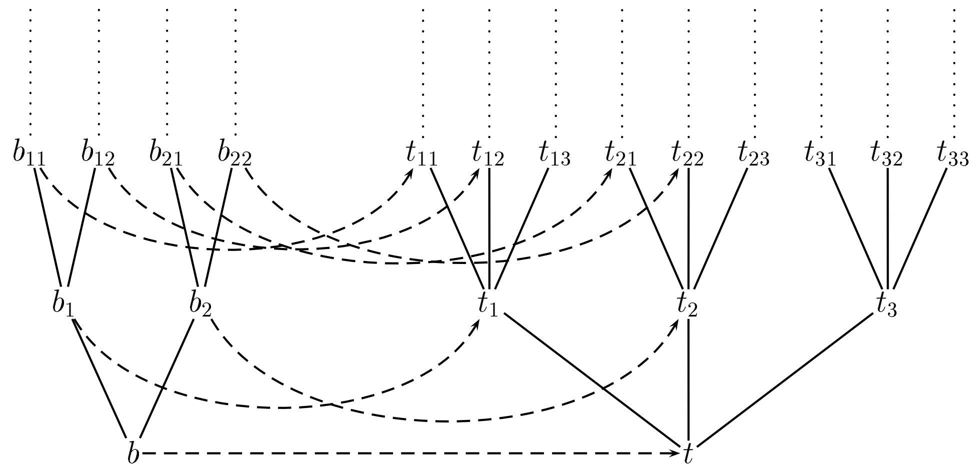

Figure 3.7: A classical join-preserving encoding from an infinite ascending binary tree to an infinite ascending ternary tree.

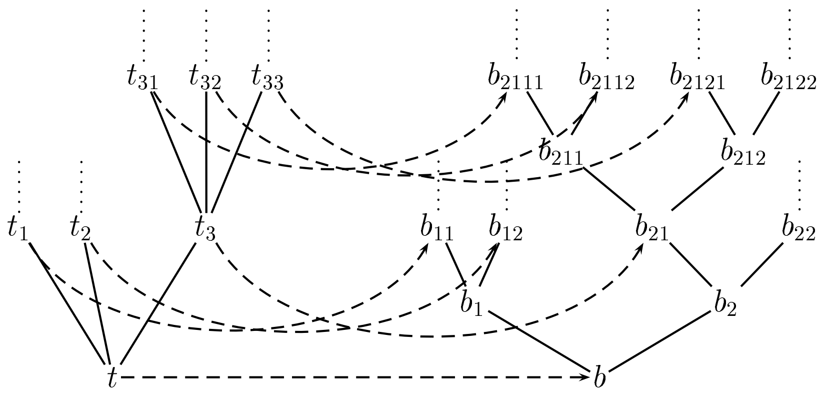

Figure 3.8: A classical join-preserving encoding from an infinite ascending ternary tree to an infinite ascending binary tree.

the infinite ternary tree, because they are the only elements subsumed only by $t_\perp$ and themselves. But then the third of $\{t_1, t_2, t_3\}$ cannot correspond to anything in the infinite binary tree without being subsumed by the correlates of the other two.

3.3 Signature Equivalence and Subsumption

In the last chapter, it was noted that totally well-typed feature structures give us the "best" interpretation of appropriateness in a sense; and in the last section, we saw that total well-typing can, in another respect, be used as a substitute for the weaker well-typing interpretation. We also saw that abstract feature structures naturally represent the "information states" provided by feature structures, and that they respect total well-typing. Putting all of this together, we finally can obtain a formal definition of signature equivalence and signature subsumption:

Definition 3.10. *Signature S is* equivalent *to signature R, written $S \approx_S R$, iff there exists an order-isomorphism between $\mathcal{TTA}_S$ and $\mathcal{TTA}_R$, the totally well-typed abstract feature structures of S and R, respectively.*

Definition 3.11. *Signature S* subsumes *signature R, written $S \sqsubseteq_S R$, iff there exists a (generalized) join-preserving encoding of $\mathcal{TTA}_S$ into $\mathcal{TTA}_R$.*

Proposition 3.9. *$\approx_S$ is an equivalence relation and $\sqsubseteq_S$ is a pre-order.*

Proof. Reflexivity of both relations follows from the fact that the identity function is an order-isomorphism. Symmetry and transitivity of $\approx_S$ follows from the fact that inversion and composition preserve order-isomorphisms. Define the composition of two join-preserving encodings, $f : P \longrightarrow Pow(R)$ and $g : R \longrightarrow Pow(S)$, $f \circ_\sqcup g : P \longrightarrow Pow(S)$, such that:

$$(f \circ_\sqcup g)(p) = \bigcup_{\bar{p} \in g(p)} f(\bar{p})$$

It can easily be verified that $f \circ_\sqcup g$ is also a join-preserving encoding. □

We are justified in calling the weaker notion "subsumption" because in that case, $\mathcal{TTA}_R$ must be at least as refined in the distinctions between information states that it makes as $\mathcal{TTA}_S$. We still need the stronger notion

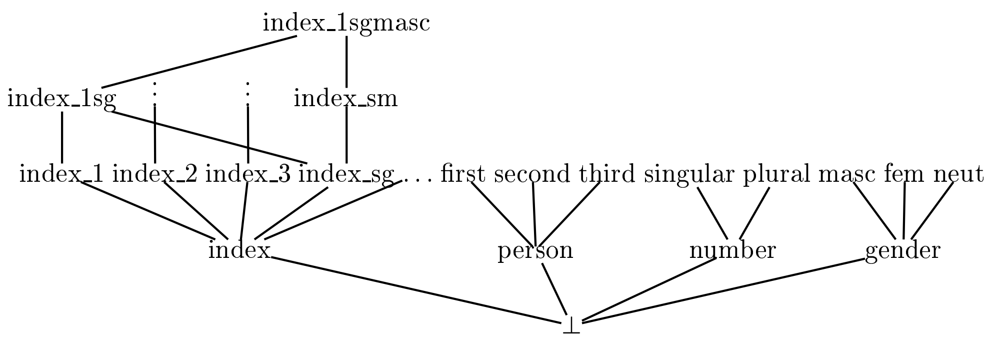

Figure 3.9: Figure 1.4 augmented to be equivalent to Figure 1.5.

because, as shown in the last section, the weaker notion in both directions does not always guarantee the stronger. The stronger, signature equivalence, formally defines the equivalence that motivated this study. The weaker notion is one that is almost always easier to establish, and that retains most of the practical benefit as well. If $S \sqsubseteq_S R$, then programs written relative to S can be translated into programs written in R, and executed on a machine built for R, with the answers translated back into the vocabulary of S. Because of the extra properties guaranteed by join-preservation, this translation works even in the face of computation of joins and other operations that "respect" joins in the manner described above for generalized join-preserving encodings.

To consider the example in the introduction again, the signatures in Figures 1.4 and 1.5 are actually not quite equivalent — Figure 1.4 requires identical *person*, *number* and *gender* branches to those in Figure 1.5 in order for feature structures of those types to have something to correspond to, as shown in Figure 3.9. Given that modification, some of the correspondence between totally well-typed abstract feature structures of type *index* is shown in Figure 3.10. As they stand, the signature in Figure 1.4 subsumes the signature in Figure 1.5, with the latter distinguishing additional information states for self-standing abstract feature structures of types *person*, *number*, and *gender*.

The next chapter considers this correspondence between signatures in more detail. In particular, one can create equivalent signatures directly by adding more types and modifying appropriateness conditions.

$$\begin{array}{lcl}
\text{index} & \longmapsto & \begin{bmatrix} \text{index} & \\ \textsc{person} & \text{person} \\ \textsc{number} & \text{number} \\ \textsc{gender} & \text{gender} \end{bmatrix} \\
\text{index\_sg} & \longmapsto & \begin{bmatrix} \text{index} & \\ \textsc{person} & \text{person} \\ \textsc{number} & \text{singular} \\ \textsc{gender} & \text{gender} \end{bmatrix} \\
\text{index\_1sg} & \longmapsto & \begin{bmatrix} \text{index} & \\ \textsc{person} & \text{first} \\ \textsc{number} & \text{singular} \\ \textsc{gender} & \text{gender} \end{bmatrix} \\
\text{index\_1sgmasc} & \longmapsto & \begin{bmatrix} \text{index} & \\ \textsc{person} & \text{first} \\ \textsc{number} & \text{singular} \\ \textsc{gender} & \text{masc} \end{bmatrix}
\end{array}$$

Figure 3.10: Part of the correspondence between *index* values in Figures 1.4 and 1.5.

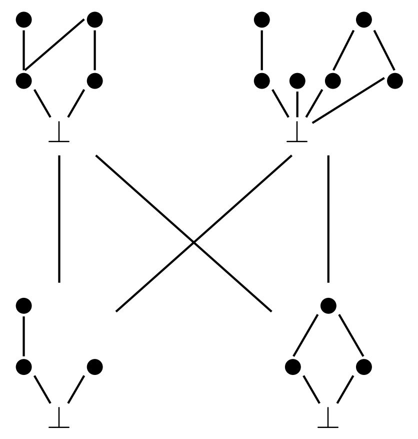

Figure 3.11: Two signatures with no least upper bounds along with two of their minimal upper bounds.

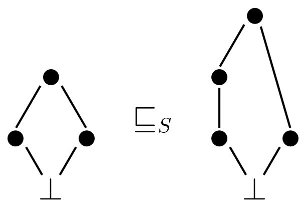

Figure 3.12: Subsumption between the two lower signatures of Figure 3.11 due to top-smashing.

3.4 A Signature of Signatures?

Since we have a pre-order on signatures, we may stop to ask whether there is a signature of signatures, and if so, at what point in the signature of signatures does it contain itself. We can construct a true partial order over (classes of) signatures by considering the one induced on the quotient of signatures modulo either $\approx_S$ or the symmetric closure of $\sqsubseteq_S$. We even have a least element, namely the signature consisting only of $\bot$.

The set of signatures is not a bounded complete partial order, however, and therefore not a signature. Figure 3.11 shows two signatures with two minimal upper bounds for which there is no least upper bound between them. This may seem rather unfair, since if we top-smashed the signature on the left, giving a representation to inconsistency within the signature, it would be subsumed by the one on the right, as shown in Figure 3.12. Even if we adopt the convention that we should top-smash signatures with

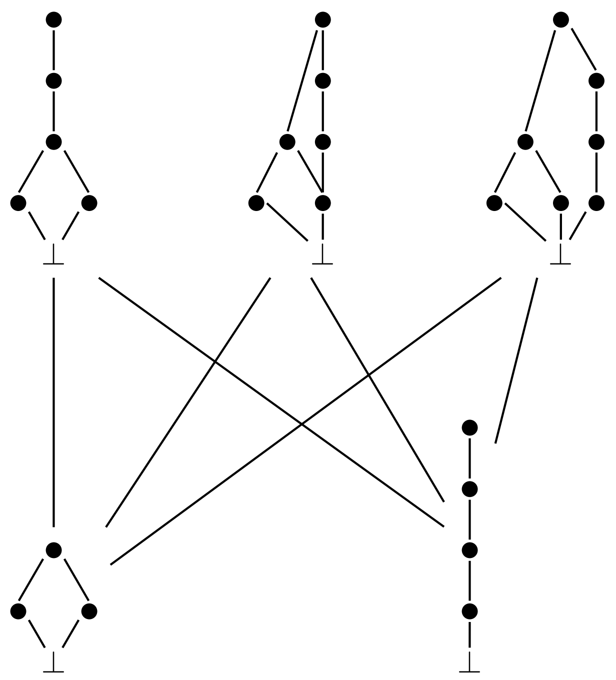

Figure 3.13: Two signatures with greatest elements that have no least upper bounds and three of their minimal upper bounds.

no greatest element, there are still counter-examples in which no least upper bound exists, such as in Figure 3.13.

As a result, we cannot straightforwardly think of a domain of signatures, following the development of feature structure domains by Carpenter [1992].

3.5 Summary

This chapter has presented some of the basic concepts that establish the view of typed feature structures and signatures upon which this dissertation is based. The most fundamental among those is the identification of totally well-typed abstract feature structures as the "information states" that are worth preserving algebraically.

Join-preserving embeddings were also generalized in a way that more essentially characterizes them. These two concepts, information and generalized join preservation, combine to give us a useful way of comparing signatures through signature subsumption and signature equivalence. The relationship between these two relations as well as the underlying structure of the set of all signatures has also been examined.

Chapter 4

Recursion, Finiteness, and Appropriate Values

We saw in the last chapter (Corollary 3.1) that $\mathcal{TTA}$ has a very similar structure to the signature that induces it, namely, that of a countable, bounded complete, partially ordered set, no matter what that signature is. We can even designate each abstract feature structure as a type and call it a type hierarchy, or a type signature with no features. We also saw how it is possible to put signatures themselves into a subsumption relationship relative to the totally well-typed abstract feature structures they induce.

These two facts about signatures are really opposite sides of the same coin. Given a total well-typedness interpretation of appropriateness, which will be assumed throughout the rest of this work, signatures with features can be regarded as a compact way of describing the structure of the totally well-typed abstract feature structures that they induce. Some are more compact than others; and the fact that there is some variation in how compact they can be is what leads to instances of signature equivalence between what, on the surface, appear to be different signatures. The signature in Figure 3.9, for example, is not very compact at all. It has no features, and is therefore isomorphic to its totally well-typed abstract feature structures, with each type having one. Figure 1.5 is more compact, but also has a $\mathcal{TTA}$ that is isomorphic to the type hierarchy in Figure 3.9.

In this view, signatures are merely a specification of a particular partially ordered set, $\mathcal{TTA}$, and appropriate features are what make such a specification more compact. They provide a kind of mapping of structure in one part of a signature's $\mathcal{TTA}$ back onto itself, thus eliminating the need for its repetition in the signature. The first section of this chapter investigates this view further, and shows how it can be used to provide tractable encodings of multi-dimensional inheritance and systemic networks, two other means of knowledge representation that are popular in linguistics which were previously thought not to be reconcilable to attributed type signatures or simple partial orders, more generally. The second section then reconsiders the discussion of finiteness from Chapter 2 in the light of signatures, i.e., how to reason about the finiteness of most general satisfiers and the finiteness of $\mathcal{TTA}$ directly in terms of signatures. In the course of this study, it will become clear that finite signatures with features do indeed possess more expressive power than finite signatures without them — the "compacting" power of features can even render an infinite $\mathcal{TTA}$ finitely presentable.

The third section examines finite signatures more closely according to the analysis of finite type hierarchies given in Chapter 2, namely, how appropriateness and total well-typing allow us indirectly to restrict $\mathcal{TTA}$ to be well-founded, finite branching and/or Noetherian. The fourth section then discusses the potential practical application of these results, by considering the transformation of signatures as a device to improve the performance of processing relative to them.

4.1 Product isomorphisms

In general, the features over which *Approp* is defined for a given type establish an isomorphism between the totally well-typed abstract feature structures of that type and the product of several *filters* of totally well-typed abstract feature structures, determined by the value of *Approp* on those features with that type.

Definition 4.1. *A subset $L \subseteq P$ of a partially ordered set, $\langle P, \sqsubseteq \rangle$ is a* filter *iff it is:*

- **directed:** *if $x, y \in L$ then there exists a $z \in L$ such that $z \sqsubseteq x$ and $z \sqsubseteq y$, and*
- **upward closed:** *if $x \in L$, $y \in P$, and $x \sqsubseteq y$, then $y \in L$.*

Filters are very closely related to the concept of most general satisfiers or "least extension" operators, such as TWT etc., in general, because the set of extensions of which they are least forms a filter.

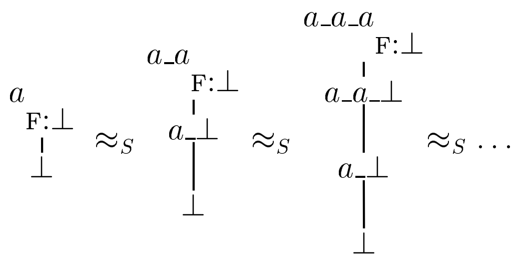

Figure 4.1: An infinite series of equivalent signatures.

Definition 4.2. *Given a signature, S, with type hierarchy $\langle T, \sqsubseteq\rangle$, let $T : T \longrightarrow Pow(\mathcal{TTA}_S)$ be such that $T(t) = \{A \in \mathcal{TTA}_S | \Theta_A(\epsilon) = t\}$, the set of totally well-typed abstract feature structures of type t.*

Definition 4.3. *Given a signature, S with type hierarchy $\langle T, \sqsubseteq\rangle$, let $A : T \longrightarrow Pow(\mathcal{TTA}_S)$ be such that $A(t) = \{A \in \mathcal{TTA}_S | t \sqsubseteq \Theta_A(\epsilon)\}$.*

Proposition 4.1. *For any $t \in T$, $A(t)$ is the filter of totally well-typed abstract feature structures rooted at $Abs(TWT(MGSat(t)))$.*

Proof. $Abs(MGSat(t))$ is the least abstract feature structure of type t, and $Abs(TWT(MGSat(t)))$ is its least totally well-typed extension. $A(t)$ is then obviously directed, because one can always choose $Abs(TWT(MGSat(t)))$ as z, and upward closed because it contains all totally well-typed abstract feature structures more specific than $Abs(TWT(MGSat(t)))$. □

If we again compare Figures 3.9 and 1.5, we can see that every type in the signature of Figure 1.5 has a corresponding filter in Figure 3.9. In the case of types like *person* or *first*, there is just a type again. No features are appropriate to those. In the case of *index*, there is a larger set of types, determined by every possible combination of *person* subtype, *number* subtype and *gender* subtype, because these are the three appropriate features to *index* in Figure 1.5. In this way, multiple appropriate features in the more compact signature correspond to a product in the less compact signature.

Sometimes, the appropriate value restriction that a feature takes can actually subsume the type to which the feature is appropriate. In this case, we can end up with an infinite series of equivalent signatures, as shown for a very simple case in Figure 4.1. In this case, we say a signature is *recursive*, or

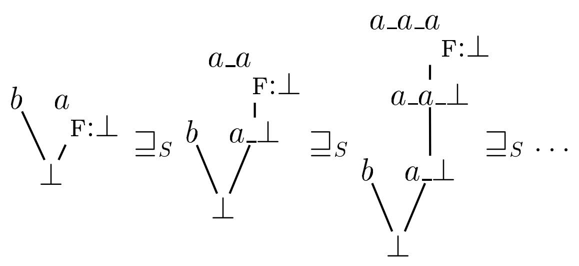

Figure 4.2: An infinite descending chain of signature approximations.

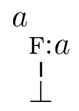

Figure 4.3: A signature with a cyclic type.

that the type a in Figure 4.1 is a *recursive type*, and F is a *recursive feature*. These will be defined formally later. All of the signatures in Figure 4.1 are equivalent, because they all have isomorphic BCPOs of totally well-typed abstract feature structures. If the unfolding of the product into types is not complete, then we have an infinite descending chain of approximations in the case of recursive types, as shown in Figure 4.2. There are no types a_b, a_a_b, etc. to correspond to an a-typed feature structure with a substructure of type b, so the unfolding is not complete. There is also the kind of recursion shown in Figure 4.3. This kind of a and F are said to be a *cyclic type*, and a *cyclic feature*, respectively. These are somewhat of a special case. They also have infinitely many equivalent signatures, but none of those signatures are finite. They still have infinite chains of finite approximations, however, as shown in Figure 4.1. Types with cyclic appropriate features also do not have finite totally well-typed most general satisfiers, and Carpenter [1992] excludes them by prohibiting *appropriateness loops*. All of these will be discussed further in the next section.

Unfortunately, in the case of multiple features, we do not always obtain a well-behaved product. We do in the case of Figure 1.5 because the value restrictions of these features, *person*, *number* and *gender*, together with *index* itself are pairwise join-incompatible. That means that feature structures of

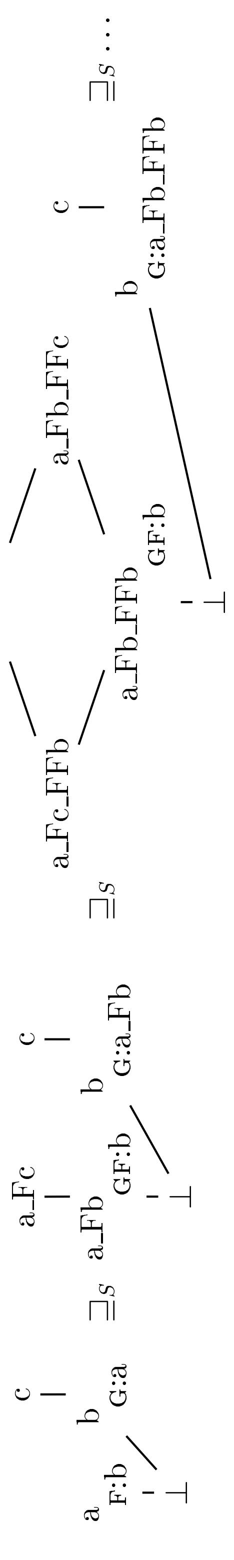

Figure 4.4: A signature with cyclic types, and its infinite descending chain of approximations.

type *index* can never have re-entrancies or cycles inside them. To handle the general case, we need a special kind of product:

Definition 4.4. *Given a finite family of filters of abstract feature structures, $\mathcal{L} = \{L_1, \ldots, L_n\}$, the* shared product of $\mathcal{L}$ with respect to t *is the set of all tuples $\langle t, A_1, \ldots, A_n, \approx, \not\approx \rangle$, such that $t \in T$ and, for all $1 \leq i, j, i_1, i'_1, i_2, i'_2 \leq n$:*

- **membership:** $A_i = \langle P_i, \Theta_i, \approx_i, \not\approx_i \rangle \in L_i$,
- **path equivalence:** $\approx$ *is an equivalence relation over* $\{\epsilon\} \uplus P_1 \uplus \cdots \uplus P_n$,
- **inequation negativity:** $\not\approx$ *is a symmetric, anti-reflexive relation over* $\{\epsilon\} \uplus P_1 \uplus \cdots \uplus P_n$,
- **inequation disjointness:** $\approx \cap \not\approx = \emptyset$,
- **prefix consistency:** *if* $\pi\text{F} \in P_i$ *and* $\langle i, \pi \rangle \approx \langle j, \pi' \rangle$, *then* $\pi'\text{F} \in P_j$ *and* $\langle i, \pi\text{F} \rangle \approx \langle j, \pi'\text{F} \rangle$,
- **inequation consistency:** *if* $\langle i_1, \pi_1 \rangle \not\approx \langle i_2, \pi_2 \rangle$, $\langle i_1, \pi_1 \rangle \approx \langle i'_1, \pi'_1 \rangle$, *and* $\langle i_2, \pi_2 \rangle \approx \langle i'_2, \pi'_2 \rangle$, *then* $\langle i'_1, \pi'_1 \rangle \not\approx \langle i'_2, \pi'_2 \rangle$,
- **typing consistency:** *if* $\langle i, \pi \rangle \approx \langle j, \pi' \rangle$, *then* $\Theta_i(\pi) = \Theta_j(\pi')$, *where* $\Theta_0(\epsilon) = t$,
- **path projection:** $\pi \approx_i \pi'$ *iff* $\langle i, \pi \rangle \approx \langle i, \pi' \rangle$,
- **inequation projection:** $\pi \not\approx_i \pi'$ *iff* $\langle i, \pi \rangle \not\approx \langle i, \pi' \rangle$,

Definition 4.5. Subsumption *on shared products,* $\sqsubseteq_L$, *is defined such that* $\langle t, A_1, \ldots, A_n, \approx, \not\approx \rangle \sqsubseteq_L \langle t', A'_1, \ldots, A'_n, \approx', \not\approx' \rangle$ *iff* $t \sqsubseteq t'$, $\approx \subseteq \approx'$, $\not\approx \subseteq \not\approx'$, *and, for all* $1 \leq i \leq n$, $A_i \sqsubseteq A'_i$.

Because of typing consistency, pairwise join-incompatible value restrictions, such as those of *index* will only allow $\approx$ to be the union of the $\approx_i$ from individual dimensions, and likewise for $\not\approx$, which means we can throw them away — subsumption will never depend on them. This is why we get a true product in the case of *index*.

Theorem 4.1. *Given a signature,* $\langle T, \sqsubseteq, \mathit{Feat}, \mathit{Approp} \rangle$, *and a type* $t \in T$, *with* n *appropriate features,* $\{\text{F}_1, \ldots, \text{F}_n\} = \{\text{F} | \mathit{Approp}(\text{F}, t)\downarrow\}$, *there exists an order-isomorphism between* $T(t)$ *and the smallest shared product of* $\{A(\mathit{Approp}(\text{F}_1, t)), \ldots, A(\mathit{Approp}(\text{F}_n, t))\}$ *with respect to* t.

Proof. Let $A = \langle P_A, \Theta_A, \approx_A, \not\approx_A \rangle \in T(t)$ correspond to the member of the shared product, $\langle t, A_1, \ldots, A_n, \approx, \not\approx \rangle$ for which, for all $1 \leq i \leq n$:

- P_A is the smallest set such that $P_A = \{\epsilon\} \cup \mathrm{F}_1 P_1 \cup \cdots \mathrm{F}_n P_n$, where $\mathrm{F}_i P_i = \{\mathrm{F}_i \pi | \pi \in P_i\}$,
- $\Theta_A(\epsilon) = t$, and $\Theta_A(\mathrm{F}_i \pi) = \Theta_i(\pi)$,
- $\approx_A$ is the smallest relation on P_A such that $\mathrm{F}_i \pi \approx_A \mathrm{F}_j \pi'$ iff $\langle i, \pi \rangle \approx \langle j, \pi' \rangle$, $\epsilon \approx_A \mathrm{F}_i \pi$ iff $\langle 0, \epsilon \rangle \approx \langle i, \pi \rangle$ (and symmetrically) and $\epsilon \approx_A \epsilon$, and
- $\not\approx_A$ is the smallest relation on P_A such that $\mathrm{F}_i \pi \not\approx_A \mathrm{F}_j \pi'$ iff $\langle i, \pi \rangle \not\approx \langle j, \pi' \rangle$, and $\epsilon \not\approx_A \mathrm{F}_i \pi$ iff $\langle 0, \epsilon \rangle \not\approx \langle i, \pi \rangle$ (and symmetrically).

That this is an isomorphism follows from total well-typing and the trivial isomorphism between finite mappings and finite products. $\square$

Shared products explicitly factor the information in a feature structure into (1) a finite product of information from its feature values, and (2) a set of equations and inequations between them. In the case of signatures with recursive types, the order isomorphisms given by shared products establish a set of recursive equations that embed a shared product containing $T(t)$ into $T(t)$ itself.

While the trivial isomorphism between finite mappings and finite products may indeed seem trivial, it is probably worth stopping to look at a few of its ramifications, in light of how often it has been forgotten or neglected in applications of typed feature logic to the encoding of related structures.

4.1.1 Multi-dimensional inheritance

ProFIT [Erbach, 1994, 1995, 1996], a logic programming language for typed feature structures, introduced a restricted form of multiple inheritance in order to guarantee the existence of Prolog term encodings of its feature structures. Specifically, it allowed for two kinds of type subsumption declarations in its signatures:

$$Super > [Sub_1, Sub_2, \ldots, Sub_n].$$

$$Super > [Sub_{1,1}, \ldots, Sub_{1,n_1}] * \ldots * [Sub_{m,1}, \ldots, Sub_{m,n_m}].$$

The first implicitly declares $Sub_1, \ldots, Sub_n$ to be mutually exclusive, i.e., no multiple inheritance. The second declares *multi-dimensional inheritance*, essentially restricting multiple inheritance to the full products of types from different dimensions. It is a restriction in general because all possible combinations of dimensions must be attainable. For example, we could declare the subhierarchy rooted at *index* in Figure 1.4 with:

$index > [pers] * [num] * [gend].$

$pers > [1, 2, 3].$

$num > [s, p].$

$gend > [m, f, n].$

The maximally specific types, *index_1sgmasc, index_1plmasc* etc., are represented by the product of their *pers*, *num* and *gend* values.

Multi-dimensional inheritance uses finite products, but as seen earlier, multiple appropriate features implicitly use finite products as well. Multi-dimensional inheritance is not only a restriction of general multiple inheritance, but it provides nothing that the use of extra features could not already provide in the absence of all multiple inheritance. In particular, it is trivially equivalent to the use of one feature for each dimension with no path equations or inequations between them.

As will be seen in Chapter 5, parametric types effectively provide the same sort of product, but with the additional ability to introduce or remove extra dimensions at subtypes, and to link appropriateness conditions to dimensions through the use of type variables.

4.1.2 Systemic networks

Systemic networks [Kress, 1976] are a means of stating constraints over the allowable combinations of a finite set of properties (usually called *features*, but different from the use of that term here). They are used rather often in computational linguistics, whose connection to inheritance-based reasoning with feature structures has been explored in a series of papers by Kasper [1986, 1988, 1989]. The formulation presented here is adapted from Carpenter and Pollard [1991], Carpenter [1992] and Henschel [1995].

Definition 4.6. *Given a set of properties,* PROP*, the set of* entry expressions of PROP*, $E(\text{PROP})$, is the smallest set such that:*

- $\text{PROP} \subseteq E(\text{PROP})$,
- $e_1 \wedge e_2 \in E(\text{PROP})$, *for all* $e_1, e_2 \in E(\text{PROP})$, *and*

- $e_1 \vee e_2 \in E(\textsc{prop})$, *for all* $e_1, e_2 \in E(\textsc{prop})$.

Definition 4.7. *Given a set of properties,* PROP*, a* system *is a pair* $s = \langle e(s), out(s) \rangle$*, where* $e(s) \in E(\textsc{prop})$ *is the* entry condition of s *and is in disjunctive normal form, and* $out(s) \subseteq \textsc{prop}$ *is the set of* output properties of s.

Definition 4.8. *Given a set of systems,* S *and a set of properties,* PROP*, the* immediate dependency relation of S, $Dep \subseteq \textsc{prop} \times \textsc{prop}$*, is defined such that* $Dep(p, q)$ *iff there is an* $s \in S$ *such that* p *appears in* $e(s)$ *and* $q \in out(s)$.
The dependency relation of S, Dep^**, is the reflexive and transitive closure of* Dep.

Definition 4.9. *A* systemic network *is a triple* $\langle \bar{p}, \textsc{prop}, S \rangle$*, where* PROP *is a finite set of properties,* $\bar{p} \in \textsc{prop}$ *is the distinguished* start property*, and* S *is a finite set of systems such that:*

- *the output sets of* S *are a partition of* $\textsc{prop} \backslash \{\bar{p}\}$,
- *the dependency relation of* S, Dep^**, is anti-symmetric, and*
- *for all* $p \in \textsc{prop}$, $Dep^*(\bar{p}, p)$.

Systemic networks are interpreted as sets of constraints on subsets of PROP. An allowable subset is called a *selection expression*:

Definition 4.10. *Given a systemic network* $SN = \langle \bar{p}, \textsc{prop}, S \rangle$*, a* selection expression of SN *is a subset* $\sigma \subseteq \textsc{prop}$ *such that:*

- $\bar{p} \in \sigma$,
- *if* $\{p_1, \ldots, p_n\} \subseteq \sigma$*, with* $n \geq 1$*, then for every system* $s \in S$ *such that* $p_1 \wedge \ldots \wedge p_n$ *is a disjunct of* $e(s)$*, there is exactly one property,* $p' \in out(s)$ *such that* $p' \in \sigma$*, and*
- *if* $p' \in \sigma$ *and there exists an* s *for which* $p' \in out(s)$*, then there exists a disjunct of* $e(s)$, $p_1 \wedge \ldots \wedge p_n$*, such that* $\{p_1, \ldots, p_n\} \subseteq \sigma$.

Systemic networks are conventionally drawn as shown in Figure 4.5, a systemic network for describing unmarked SP(O) sentences with active voice from the NIGEL grammar [Mann and Matthiessen, 1983] (and cited by

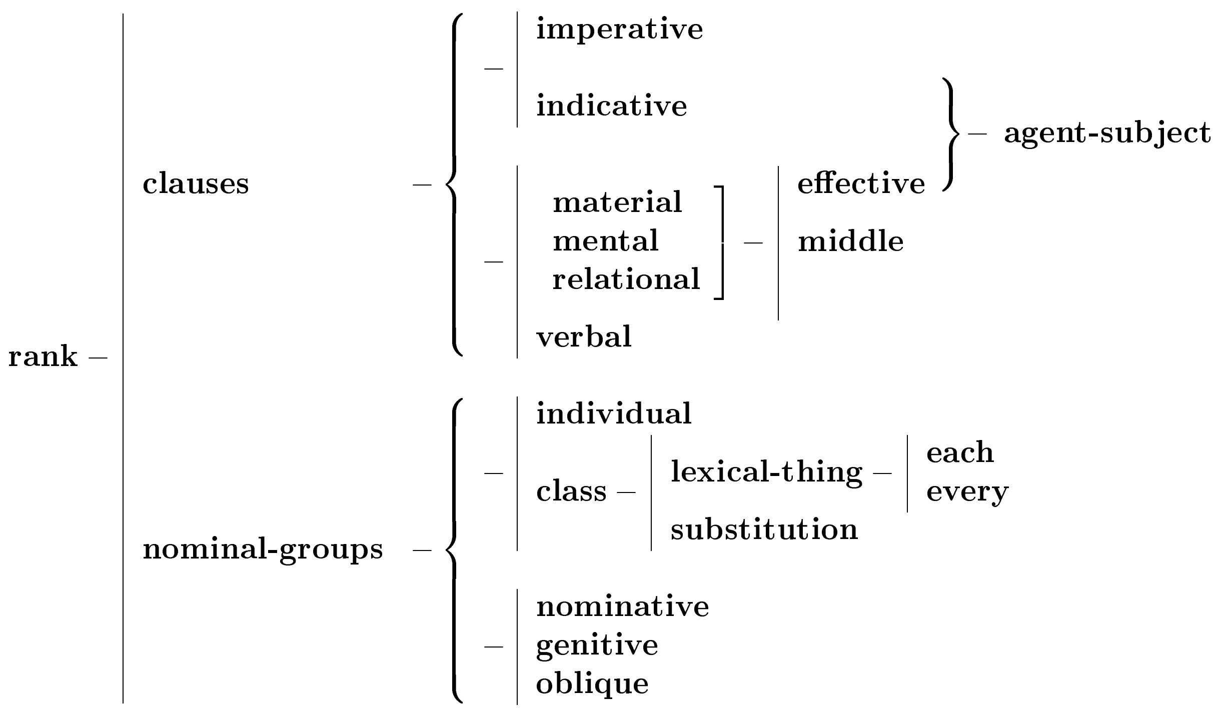

Figure 4.5: An example of a systemic network.

Henschel [1995]). The drawing convention relies on the use of four connectives: the choice connective (`|`), the conjunctive choice connective (`{`), the conjunctive precondition connective, (`}`), and the disjunctive precondition connective, (`]`). The choice connective indicates a system with more than one output property. Systems with a single output property, such as `agent-subject`, are indicated by the property itself. The network shown in Figure 4.5 has a total of nine systems, with `root` being the distinguished start property and the entry condition to the system with output properties `clauses` and `nominal-groups`. The conjunctive choice connective indicates that a property is used in the entry condition to more than one system, such as `clauses` in the example, which is the entry condition to both the `imperative/indicative` system and the `material` etc. system. Conjunctive and disjunctive precondition connectives are used to express the conjunctions and disjunctions, respectively, of complex entry conditions. The entry condition to the `agent-subject` system is `imperative` $\wedge$ `indicative`, and, to the `effective/middle` system is `material` $\vee$ `mental` $\vee$ `relational`.

Carpenter [1992, pp. 30–32] presents a BCPO construction for representing valid partial information states in systemic networks, i.e., subsets of properties that subsume at least one selection expression, without the disjunctive precondition connective, i.e., disjunctive entry conditions. Henschel [1995] extends the construction to all systemic networks. She observes that the construction requires 2^n types for an n-property systemic network in the worst case, and that the worst case is caused by the conjunctive choice connective. Neither considers the use of features in their constructions, for some reason — only BCPOs of types.

In fact, systemic networks do have polynomially bounded encodings in the logic of typed feature structures, although, to the author's knowledge, not in the fragment presented in this work. The meet semi-lattice completion of Figure 4.6 is the encoding for Figure 4.5, for example.

Definition 4.11. *Given a feature structure F with node set Q on a signature with types T, the* types of F *are* $\Theta(F) = \{\tau \in T | \exists q \in Q. \tau \sqsubseteq_T \theta(q)\}$.

The encoding reduces the question, "Is $\sigma \subseteq$ PROP a subset of a selection expression?" to the question, "Is there a totally well-typed feature structure F of type $\bar{p}$ such that $\sigma \subseteq \Theta(F)$?" and, if yes, the question, "Is that σ a selection expression?" to "Is the F corresponding to σ maximal in $\langle \mathcal{TTF}, \sqsubseteq_{\mathcal{TTF}} \rangle$?" It relies crucially on the use of non-maximally-specific *extensional* types, indicated in Figure 4.6 with boxes around them. While a

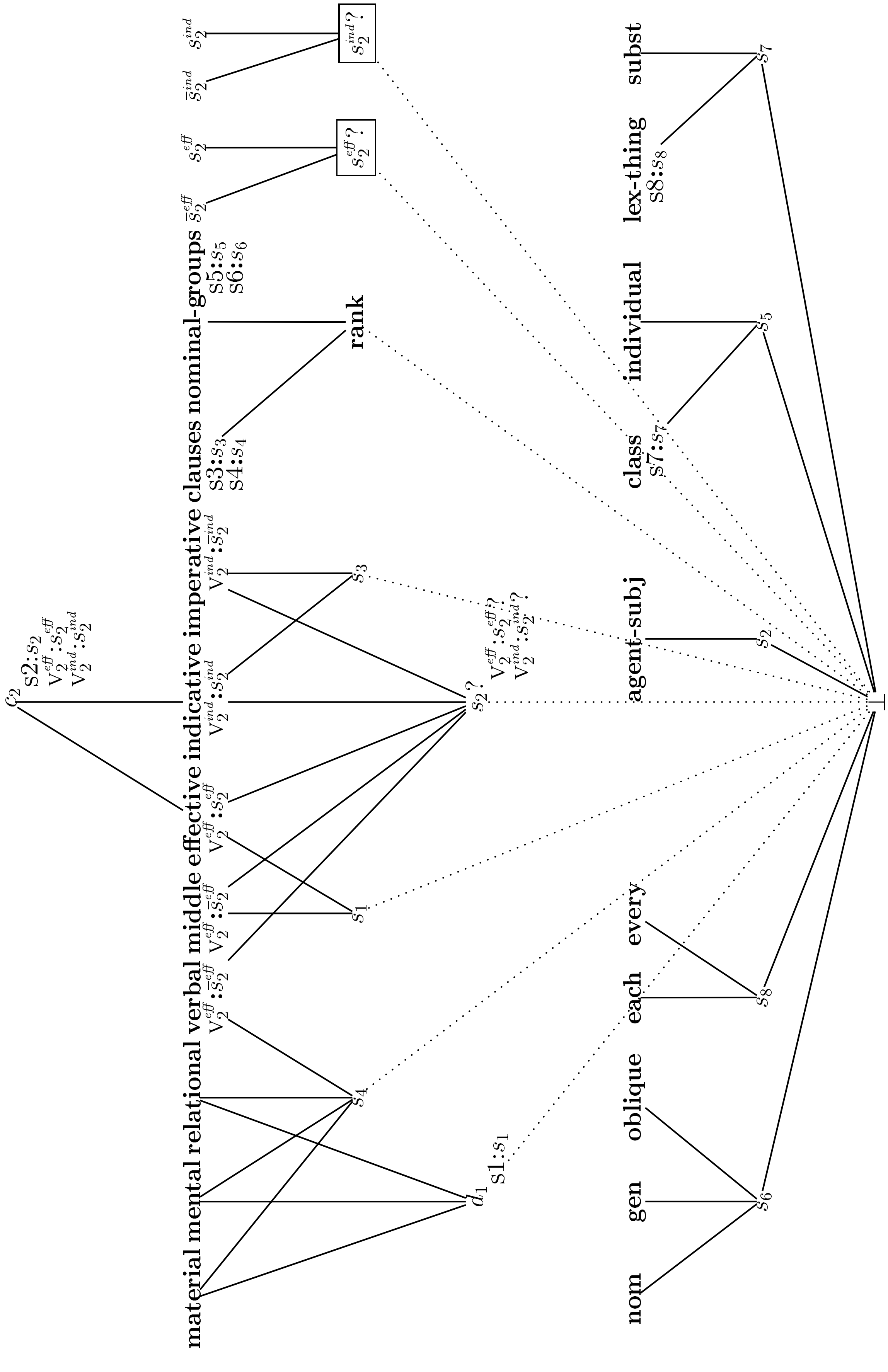

Figure 4.6: An attributed type signature (after MSL completion) that encodes the systemic network in Figure 4.5.

formalization of extensional type inference and, thus, of this encoding is out of the scope of the present discussion, it should at least be noted that conjunctive choice connectives, i.e., the use of properties in more than one system's entry condition, are encoded by introducing multiple features (corresponding to systems) at the types that correspond to those multiply used properties. The product implicitly encoded in multiple appropriate features combined with the non-maximally-specific value restrictions that permit those feature values to vary independently over system output sets allows for a polynomially bounded presentation of any systemic network as an attributed type signature.[1] `clauses` and `nominal-groups`, for example, introduce two features each because they are used in the entry conditions to two systems each.

4.2 Finiteness

So far, we have seen two kinds of finiteness: finiteness of type hierarchies themselves, i.e., a finite number of types, and finiteness of feature structures, i.e., a finite number of nodes. In this section, we shall also consider the finiteness of $\mathcal{TTA}$ and the finiteness of filters, $A(t) \subseteq \mathcal{TTA}$, i.e., a finite number of totally well-typed abstract feature structures of an individual type. As it happens, none of these notions of finiteness are the same. In particular, a finite signature (a signature with a finite type hierarchy) can still admit finite descriptions with infinite most general satisfiers and/or an infinite number of totally well-typed abstract feature structures in its $\mathcal{TTA}$.

The purpose of this section is to characterize the other two kinds of

[1]The extensional types are necessary in order to handle another potential source of combinatorial explosion that emerges once conjunctive choice connectives are dispensed with, namely the use of complex disjuncts, i.e., conjunctions of three or more properties, in entry conditions. Using a properties-as-types encoding, representing the conjunction of three properties as the join of their types must be unfolded into a distributive sublattice, or else any pair of properties is sufficient to entail the join. That unfolding increases the number of types exponentially as a function of the number of conjuncts. Extensional types can be used essentially to synchronize the values at different paths in the same feature structure. Each one represents a vote cast by another property as to whether a complex conjunction is satisfied. If and only if all of the votes are "yes," e.g., s_2^{eff} and s_2^{ind} in Figure 4.6, then a maximal extension of $s_2?$ must be c_2, which introduces the output system for that conjunction. Extensional types thus perform an "end run" around appropriateness, which cannot otherwise enforce path equations, path inequations (among join-compatible types) or type constraints on paths more than one feature long — restrictions that in this domain are admittedly rather arbitrary.

finiteness within finite signatures, i.e., signatures with finite type hierarchies. Finiteness is of obvious computational interest. It also is related to our ability to transform finite signatures into other equivalent finite signatures. The properties that allow us to "unfold" one or more features into a signature with more but still finitely many types are the same properties that ensure the finiteness of $\mathcal{TTA}$ or at least of some filter-shaped piece of it. In fact, we can think of $\mathcal{TTA}$ itself as a signature with no appropriate features, as a result of Corollary 3.1. The question is how to spot whether or not $\mathcal{TTA}$ is finite simply by looking at the signature.

4.2.1 Cyclic Types and Finite Most General Satisfiers

To begin where Carpenter [1992] left off, infinite most general satisfiers can be characterized as follows:

Definition 4.12. *Given a signature, $S = \langle T, \sqsubseteq, Feat, Approp \rangle$, the* appropriateness graph of *S, $A(S)$, is a labelled directed graph $\langle T, \{\langle t_1, t_2, \mathtt{F}\rangle |$ $Approp(\mathtt{F}, t_1)\downarrow$ and $Approp(\mathtt{F}, t_1) = t_2\}\rangle$, whose vertices are the types of S and whose edges map from types to value restrictions of their appropriate features.*

Definition 4.13. *Given a signature, $S = \langle T, \sqsubseteq, Feat, Approp \rangle$, and a type, t, t is a* cyclic type *iff there exists a non-empty path from t to t in $A(S)$. The features on this path are* cyclic features.

Carpenter [1992, pp. 97–99] referred to such paths as "appropriateness loops," and to its edges minus the labels as the "substructure requirement" relation.

Definition 4.14. *Given a signature, $S = \langle T, \sqsubseteq, Feat, Approp \rangle$, and a type, t, t is* finitely satisfiable *iff there is no $t' \in T$ such that there is a path from t to t' in $A(S)$ and t' is cyclic.*

Proposition 4.2. *If t is not finitely satisfiable and $t \sqsubseteq t'$, then t' is not finitely satisfiable.*

Proof. This follows from the fact that *Approp* is upward closed. □

Proposition 4.3. *For any finite description, $\phi \in NonDisjDesc$, over a finite signature, S, such that $M = TWT(MGSat(\phi))\downarrow$, M is finite iff for all $t \in \Theta(M)$, t is finitely satisfiable.*

Note that $\Theta(M)$ is always finite because the signature is finite. Computation with infinite most general satisfiers is actually not an impossibility, because they always have finite presentations — at the very least, the descriptions themselves — although this direction will not be pursued further in this study.

A direct consequence of this is that cyclic types themselves do not have finite most general satisfiers. What is much more interesting is that, although their most general satisfiers are not finite, they *do* have finite satisfiers, namely cyclic feature structures with finitely many nodes. For example, the cyclic type a in Figure 4.3 has the following finite satisfier:

$$\begin{bmatrix} \boxed{1}\ \mathrm{a} & \\ \mathrm{F} & \boxed{1} \end{bmatrix}$$

If we were studying infinite signatures, this would not necessarily be the case.

4.2.2 Recursive Types and Finite Filters

We can easily generalize our definitions of appropriateness graph and cyclic types in order to characterize finite filters of feature structures of a given type. The intuition is that, if we assume a finite number of types, the only way in which a finite feature structure, F, can subsume infinitely many feature structures is if at least one type is assigned to infinitely many nodes in those structures. Specifically, it must be the case that for some substructure of F, G, of type t, the substructure corresponding to G in one of the feature structures F subsumes has a proper substructure which is a subtype of t again. Because feature structure subsumption is ultimately induced by type subsumption, we must look at type subsumption and appropriateness together to understand how this could happen.

Definition 4.15. *Given a signature, $S = \langle T, \sqsubseteq, Feat, Approp \rangle$, assuming that there is no feature called* S*, the* subtype-appropriateness graph *of S, $SA(S)$, is a labelled directed graph $\langle T, \{\langle t_1, t_2, \mathrm{F}\rangle \mid Approp(\mathrm{F}, t_1)\downarrow$ and $Approp(\mathrm{F}, t_1) = t_2\} \cup \{\langle t_1, t_2, \mathrm{S}\rangle | t_1 \sqsubseteq t_2\}\rangle$, whose vertices are the types of S and whose edges consist of the edges of $A(S)$ plus edges with label* S *that map from types to their subtypes.*
Given $t \in T$, let $SA(t)$ be the subgraph of $SA(S)$ consisting of all and only those nodes, t', for which there is a path from t to t' in $SA(S)$. Also, let $\rightsquigarrow$ be the labelled path accessibility relation in $SA(S)$.

Definition 4.16. *Given a signature, $S = \langle T, \sqsubseteq, \mathit{Feat}, \mathit{Approp}\rangle$, and a type, t, t is a* recursive type *iff there is a path from t to t in $SA(S)$.*
Such paths contain at least one feature label, F, *because $\sqsubseteq$ is, by definition, anti-symmetric. Such features are called* recursive features.

Proposition 4.4. *If t is cyclic then t is recursive.*

We can now characterize the conditions on a finite filter of feature structures of a given type in terms of properties of that type derived from solely from its signature:

Definition 4.17. *Type, t, is* finite *iff $|A(t)|$ is finite.*

Definition 4.18. *Type, t, is* provably finite *iff:*

- *every subtype of t is provably finite,*
- *for every feature,* F, *such that $\mathit{Approp}(\text{F}, t)\downarrow$, $\mathit{Approp}(\text{F}, t)$ is provably finite, and*
- *t is not recursive.*

Theorem 4.2. *If S is finite and t is not provably finite, then t is not finite.*

Proof. By induction on the length, λ, of the shortest path in $SA(S)$ from t to a recursive type. t has such a path, because, if not, then there are no recursive types in $SA(t)$, thus $SA(t)$ is acyclic, and by Lemma 4.2, t is provably finite after all.

If $\lambda = 0$, then t itself is recursive, and by Lemma 4.1, t is not finite. For $\lambda > 0$, t must not be recursive, so because it is not provably finite, there is some subtype or value restriction, i.e., a successor of t in $SA(S)$, that is not provably finite. One of these is the successor, t', on the shortest path from t to a recursive type — otherwise, by Lemma 4.2, $SA(t')$ is acyclic, and so there is no recursive type accessible from t', which is a contradiction. Furthermore, the shortest length from t' to a recursive type must be $\lambda - 1$, so by induction, t' is not finite.

Either t' is a subtype of t, or t' is a value restriction of t. If t' is a subtype, then by definition, $A(t') \subseteq A(t)$, and therefore t is not finite. If t' is a value restriction, then by Theorem 4.1, $T(t) \subseteq A(t)$ is order-isomorphic to a smallest shared product with respect to t, one of whose dimensions is $A(t')$, and thus t is not finite. $\square$

Theorem 4.3. *If S is finite and t is provably finite, then t is finite.*

Proof. By induction on the length, λ, of the *longest* path in $SA(S)$ from t to a maximally specific type with no appropriate features. There is such a path because, by Lemma 4.2, $SA(t)$ is acyclic and there are finitely many types in S.

If $\lambda = 0$, then t itself is maximally specific and has no appropriate features, and is thus trivially finite. For $\lambda > 0$, all of t's successors in $SA(S)$ are provably finite and each has a longest path strictly less than that of t, so by induction, all of them are finite. $A(t) = T(t) \cup \bigcup_{t' \sqsupset t} A(t')$. Each of the $A(t')$ are finite, and there are finitely many t' since S is finite. $T(t)$ is finite because by Theorem 4.1, it is order-isomorphic to a shared product of $\{A(\mathit{Approp}(\mathrm{F}_1, t)), \ldots, A(\mathit{Approp}(\mathrm{F}_n, t))\}$ with respect to t, each of which is finite. So t is finite. $\square$

Lemma 4.1. *If t is recursive, then t is not finite.*

Proof. Choose a cycle of length n from t to itself in $SA(S)$: $t \overset{\mathrm{X}_1}{\to} t_1 \overset{\mathrm{X}_2}{\to} \cdots t_{n-1} \overset{\mathrm{X}_n}{\to} t$, where the X_i are either the distinguished label, S, or features, $\mathrm{F}_i \in \mathit{Feat}$. Let $j_1 \ldots j_k$ be the indices for which X_{j_i} is a feature. $k \geq 1$ because $\sqsubseteq$ is anti-symmetric. Let $\psi : \mathit{Desc} \longrightarrow \mathit{Desc}$ be defined such that $\psi(\phi) = \mathrm{F}_{j_1} : \mathrm{F}_{j_2} : \cdots : \mathrm{F}_{j_k} : \phi$. Now define the infinite sequence of totally well-typed abstract feature structures, $C_i = \mathit{Abs}(\mathit{TWT}(\mathit{MGSat}(t \wedge x \wedge \psi^i(x))))$, where $x \in \mathit{Var}$, $i \geq 1$. For all $i \geq 1$, $C_i \in A(t)$, and all of them are distinct. $\square$

Lemma 4.2. *If S is finite, then t is provably finite iff $SA(t)$ is acyclic.*

Proof. To prove the forward direction, it suffices to show that if t is provably finite, then $SA(t)$ has no recursive types. This can be proven by induction on the length, λ, of the shortest path from t, which, by definition of $SA(t)$, all of its nodes have. The base case follows from the third condition of provable finiteness. The reverse direction is proven by induction on the length of the longest path to a maximally specific type with no appropriate features, which each node has because $SA(t)$ is acyclic and finite. $\square$

Notice that recursive types prevent us from using a simple inductive argument and require us to introduce the third condition into the definition (Definition 4.18) of provable finiteness. Otherwise, even in a simple signature like Figure 4.7, in which a is a recursive type, it would be consistent to say

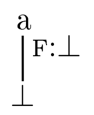

Figure 4.7: A simple signature with a recursive type.

that both a and $\perp$ were provably finite or that neither a nor $\perp$ were provably finite.

Corollary 4.1. *If S is finite, t is finite and $t \sqsubseteq t'$, then t' is finite.*

Corollary 4.2. *If S is finite, t is finite and there exists an* F *such that $Approp(\text{F}, t)\downarrow$ and $Approp(\text{F}, t) = t'$, then t' is finite.*

Corollary 4.3. *If S is finite, then $\mathcal{TTA}_S$ is finite iff S has no recursive types.*

Proof. $\mathcal{TTA}_S = A(\perp)$ and the previous two corollaries. □

When $\mathcal{TTA}_S$ is finite, we can "unfold" every feature in the signature, leaving only types — one for each totally well-typed abstract feature structure. The reader may note that potential re-entrancies among feature structures do not play a direct role in this characterization. Conventional wisdom, among both logicians and grammar developers, has been that the ability of features to share their values extensionally by means of re-entrancies is what fundamentally defines their expressive power relative to typing with inclusional polymorphism. This is not true. Features are only a more expressive device when they are used to create recursive types, in the sense defined here. Of course, recursive types always allow for cyclic re-entrancies, but it is the quality of being recursive that characterizes the difference. Even in a logic that prohibited cyclic feature structures, features that create recursive types still could not be unfolded.

4.3 Properties of Finite Signatures

One can also say more precisely which different kinds of finiteness hold of $\mathcal{TTA}_S$. Recall from Chapter 2 that a finite type hierarchy corresponds to one that is well-founded, Noetherian and finitely branching. Looking at infinite signatures, we can lose these properties independently, because the type hierarchy itself can be infinite and may discard any one or more of them independently. Finite signatures, however, are more predictable, again based on the interaction of their appropriateness and subtyping relations. A few more definitions will be convenient here.

Definition 4.19. *Given a signature, $S = \langle T, \sqsubseteq, Feat, Approp \rangle$, and a type, t, t is a* properly recursive type *iff there is a path from t to t in $SA(S)$ whose first edge is labelled with a feature, i.e., corresponds to appropriateness, not subtyping.*

Proposition 4.5. *If t is recursive, then there is a t' such that $t \sqsubseteq t'$ and t' is properly recursive.*

Proof. The cycle that witnesses a recursive type, t, can be rotated along its initial S-edges to the first feature-labelled edge emanating from some subtype, t'. Every cycle must have one feature-labelled edge because $\sqsubseteq$ is antisymmetric. If there are no initial S-edges, then $t = t'$ and t is properly recursive. □

Definition 4.20. *A path in $SA(S)$ is an* S-path *if it consists only of edges labelled with* S.

Definition 4.21. *Given a path in $SA(S)$, $a \overset{\pi}{\leadsto} b \overset{\mathrm{S}^+}{\leadsto} c \xrightarrow{\mathrm{F}} d \overset{\pi'}{\leadsto} e$, $b \overset{\mathrm{S}^+}{\leadsto} c$ is a* deletable S-path *iff there is also a path, $a \overset{\pi}{\leadsto} b \xrightarrow{\mathrm{F}} c' \overset{\mathrm{S}^*}{\leadsto} d \overset{\pi'}{\leadsto} e$ (where $c' = d$ if $c' \overset{\mathrm{S}^*}{\leadsto} d$ is of length zero).*

The intuition behind deletable S-paths is that they are the ones that are unnecessary as a result of the upward closure and right monotonicity of *Approp*: if $Approp(\mathrm{F}, b) = c'$ and $b \sqsubseteq c$, then $d = Approp(\mathrm{F}, c)\downarrow$ and $c' \sqsubseteq d$.

Definition 4.22. *Given a signature, $S = \langle T, \sqsubseteq, Feat, Approp \rangle$, and a type, t, t is* acyclically recursive *iff there is a cycle in $SA(S)$ containing t in which there is at least one* S*-path that is not deletable.*

Deletable S-paths are a way of isolating feature-labelled edges that introduce properly more restrictive value restrictions on a type relative to its supertypes. This is necessary because appropriateness is upward closed.

Note that acyclically recursive is not the same as recursive and not cyclic. A type can be simultaneously cyclic and acyclically recursive, although because of different cycles. Figure 4.8 outlines the classification of types given in this section.

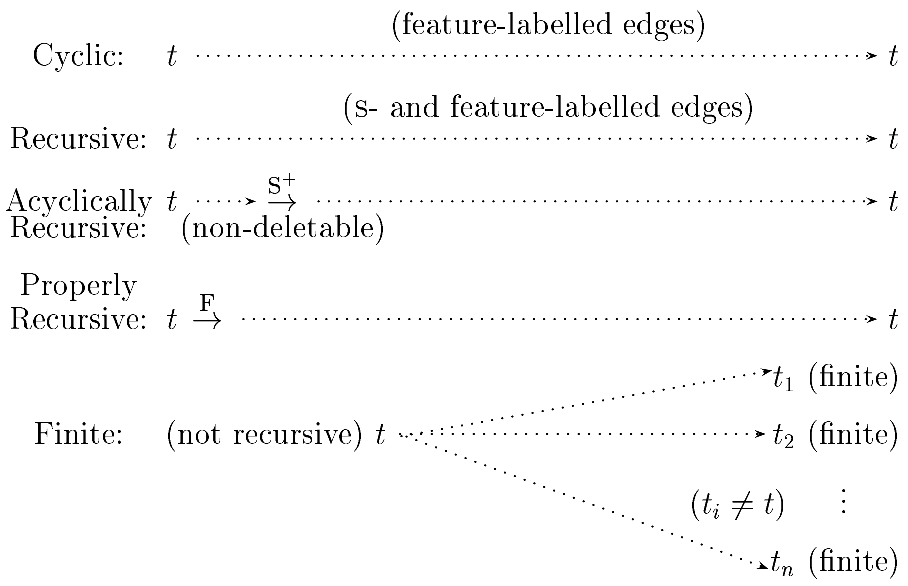
(feature-labelled edges)
Cyclic: t t
(S- and feature-labelled edges)
Recursive: t t
Acyclically Recursive: t $\overset{S^+}{\rightarrow}$ (non-deletable) t
Properly Recursive: t $\overset{F}{\rightarrow}$ t
Finite: (not recursive) t
t_1 (finite)
t_2 (finite)
$(t_i \neq t)$
t_n (finite)

Figure 4.8: An outline of the classification of types.

Theorem 4.4. *$\mathcal{TTA}_S$ is well-founded iff S has no recursive types.*

Proof. Cyclic feature structures stand at the top of infinite descending chains of approximate cyclic feature structures. Acyclic feature structures are well-founded, as proven by Wintner and Francez [1999]. □

Theorem 4.5. *$\mathcal{TTA}_S$ has only finite feature structures iff S has no recursive types.*

Proof. Below the infinite descending chain of cyclic approximations stands a limit point which has infinitely many nodes. □

Theorem 4.6. *$\mathcal{TTA}_S$ is finitely super-branching iff S has no recursive types.*

Proof. Every cyclic feature structure has an infinite supersumption branching factor. As an example, consider the simple signature in Figure 4.7. Let $C_i = Abs(TWT(MGSat(a \wedge x \wedge (\textsc{f}{:})^i x)))$, for all $i \geq 1$. By the definition of subsumption, $C_i \sqsubseteq_{\mathcal{TTA}_S} C_j$ iff $i \mid j$. Now consider the set $P = \{C_p \mid p \text{ prime}\}$. None of the abstract feature structures in P are comparable, and all of them immediately subsume C_1. Because there are infinitely many primes, C_1 has an infinite supersumption branching factor. □

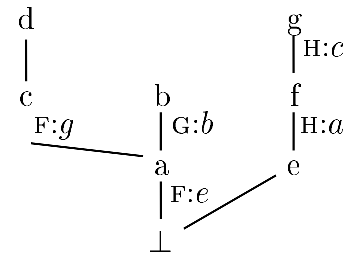

Figure 4.9: An example finite signature for demonstrating the failure of properties given in Section 4.3.

Theorem 4.7. *$\mathcal{TTA}_S$ is Noetherian iff S has no acyclically recursive types.*

Proof. If a recursive type is acyclically recursive, then beneath its cyclic feature structures are infinite ascending chains of acyclic approximations to their limits. □

Theorem 4.8. *$\mathcal{TTA}_S$ is finitely branching iff for every cycle, π, in $SA(S)$, there is no properly recursive type, t, traversed by π such that t has an out-degree greater than 1.*

Proof. Infinite feature structures have infinitely many nodes labelled with a properly recursive type. To have an infinite (subsumption) branching factor, there must be an infinite number of ways of adding only one piece of information to a feature structure. This is attainable on infinite feature structures iff it is possible to refine the types of infinitely many nodes (if a prohibited extra emanating edge is labelled with S) or it is possible to add structural information, i.e., path equations or inequations, between infinitely many pairs of nodes (if a prohibited extra emanating edge is labelled with a feature). □

Note that the definition of inequations used in this work, borrowed from Carpenter [1992, p. 112], allows an inequation to be either added or not added between two nodes that have no extension in which they can be identical — for instance, if their types are incompatible. If instead the notion of *fully inequated* feature structure [Carpenter, 1992, p. 120] were used, we would also need to check for the existence of such an extension in case of a feature-labelled edge.

An example of these properties is shown in Figure 4.11, relative to the signature in Figure 4.9, whose subtype-appropriateness graph is shown in Figure 4.10. In this signature, all of the types except $\perp$ are recursive. Of

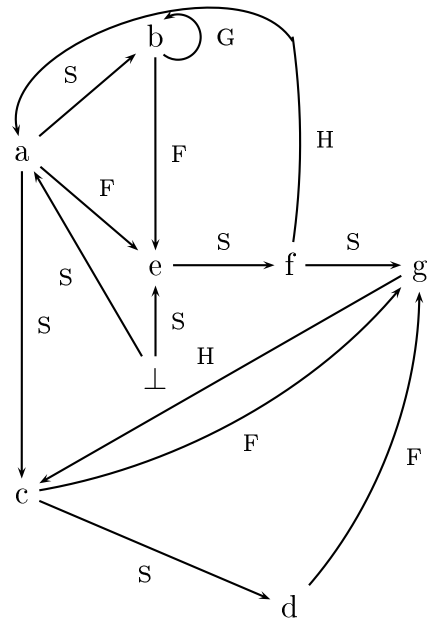

Figure 4.10: The subtype-appropriateness graph of Figure 4.9.

those, all but e are properly recursive. The types b, c and g are cyclic. b is also acyclically recursive, because of the path $b \overset{\text{F}}{\rightarrow} e \overset{\text{S}}{\rightarrow} f \overset{\text{H}}{\rightarrow} a \overset{\text{S}}{\rightarrow} b$. There is a path $c \overset{\text{S}}{\rightarrow} d \overset{\text{F}}{\rightarrow} g \overset{\text{H}}{\rightarrow} c$, but its S-path is deletable (because of the edge, $c \overset{\text{F}}{\rightarrow} g$), so c is not acyclically recursive. a is acyclically recursive, because the path $a \overset{\text{S}}{\rightarrow} b \overset{\text{F}}{\rightarrow} e \overset{\text{S}}{\rightarrow} f \overset{\text{H}}{\rightarrow} a$ has one deletable S-path ($a \overset{\text{S}}{\rightarrow} b$) but also one non-deletable S-path ($e \overset{\text{S}}{\rightarrow} f$). d, e, and f are also acyclically recursive.

The fact that b is cyclic means that every totally well-typed acyclic feature structure of type b has infinitely many nodes. In Figure 4.11, the values of G are not shown for this reason — they consist of infinitely many copies of the b-typed feature structures that contain them. The fact that b is recursive means that $A(b)$ contains an infinite descending chain of cyclic feature structures shown at the top of the figure. They also have infinite supersumption branching factors, as proven above. The fact that b is properly recursive and has an out-degree of 2 in Figure 4.10 means that the limit of this chain, shown in the middle of Figure 4.11, has an infinite subsumption branching factor, in which substructures terminating in two of infinitely many G-terminated paths are re-entrant (or inequated). Because b is also acyclically recursive, it has an infinite ascending chain rooted at its most general satisfier, shown

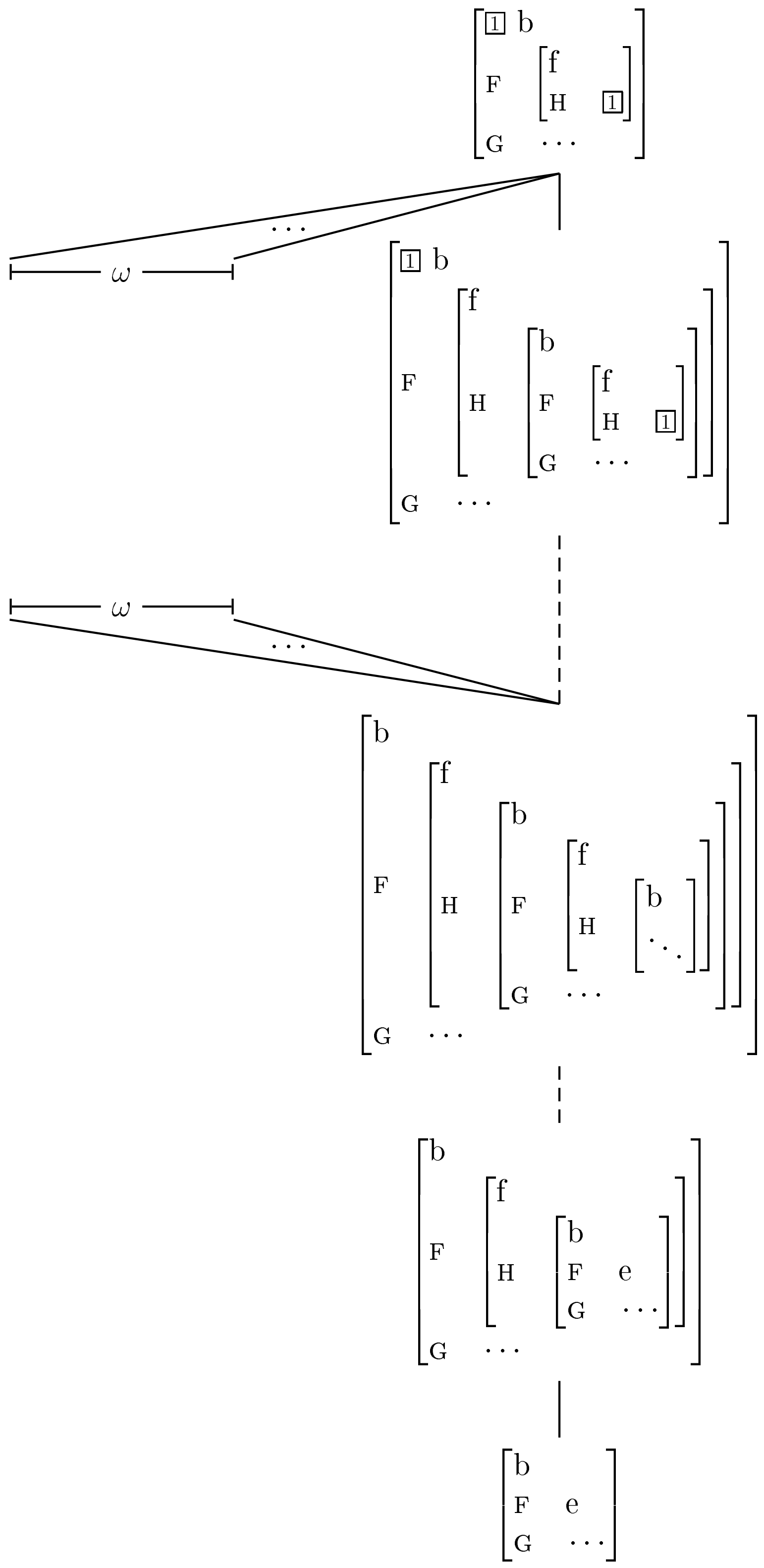

Figure 4.11: Part of $\mathcal{TTA}$ for the signature in Figure 4.9.

at the bottom of Figure 4.11, approaching the same limit.

4.4 Signature Unfolding

Armed with a knowledge of how features genuinely provide extra expressive power over simple type hierarchies and, when they do not, of the filters of types that the totally well-typed abstract feature structures they induce are equivalent to, one might suspect that it is possible to convert finite signatures into finite signatures that are logically equivalent but that have better computational properties.

The most important of these properties, of course, is the speed with which one can unify two totally well-typed feature structures induced by the signature in question. Whereas the unification of two types, or of two feature structures with no features, can usually be performed by indexing those types in a hash table, the unification of feature structures that have features typically involves an additional set of dereferencing operations which obtain matching feature values for recursive unification calls. In the case of Figure 1.5, for example, the unification of two feature structures of type *index* would probably involve three recursive unification calls to unify their PERSON, NUMBER and GENDER values, respectively. This arises from the choice of a data structure that closely mirrors the structure of the feature structure itself, of course. In Prolog, for example, those dereferencing operations are realized as pointer-chasing on the heap in order to locate the subterms that represent feature values. It is conceivable that, by converting the signature in Figure 1.5 to the one in Figure 3.9, those dereferencing operations could be avoided at a very small cost to the efficiency of the hash function that must cope with the increased number of types.

In practice, when that transformation is possible, it does result in more efficient unification. Converting just the *index* values to a purely type-based representation in an HPSG grammar, for example, typically results in an improvement in parsing times of between 25% and 33%, increasing with the size of the input, due to the large number of *index*-unification calls as a percentage of total unification calls. The problem is that, in practice, such a transformation is almost never possible.

The first factor that can block such a transformation is recursive types. As shown in Lemma 4.1, recursive types are never finite and therefore could only be unfolded into a feature-free signature that had infinitely many types.

One can, of course, choose an arbitrary bound and only partially unfold, using a feature everywhere else. In the case of lists, for example, one could unfold so that zero-length, one-length and two-length lists have their own feature-free types but lists of greater lengths still use features. This corresponds to traveling a bounded number of times around the cycle in $SA(S)$ that witnesses that a type is recursive. This may still improve efficiency if, for example, most lists are of length two or less. In any case, the feature cannot be completely eliminated without altering the logical properties of the signature. In fact, provable finiteness (Definition 4.18) tells us that the presence of a recursive type can render a great many other types infinite, i.e., with infinite $A(t)$ filters, and therefore non-unfoldable.

The second factor that can block unfolding is the sharing of variables. It was proven in Section 4.2.2 that re-entrancies by themselves do not result in extra expressive power. This is one realization of shared variables, in which the sharing actually exists within the feature logic, and indeed, re-entrancies pose no problem to unfolding. Most practical applications of feature description languages, however, use variables to share information in an extra-logical fashion as well. Logic programming languages based on typed feature structures, for example, have relations with feature-structure arguments that could be shared. The relation itself, and thus sharing between arguments (as opposed to within an argument, which corresponds to a re-entrancy) exists outside the logic. Parsing rules also typically share structure between mother categories and their daughter categories. These are also extra-logical (as opposed to sharing within the description of a single category, which also corresponds to a re-entrancy). It is, of course, possible to add extra types and features to a signature in order to bring relations, parsing rules, etc. within the scope of the logic again, e.g., to regard instances of relational goals as feature structures of the corresponding relational type in which arguments are feature values. The cost in efficiency of casting them into the feature logic, however, is typically greater than the gain in efficiency of unfolding features. In the case of logic programming again, casting relations into the feature logic would force one to meta-interpret them in a Prolog implementation, rather than compile them directly into Prolog predicates.

A third factor is the interaction between re-entrancies and recursive types. The type, *index*, for example, has a finite filter, $A(index)$, taken in isolation and all of its features can be unfolded. If the same signature also has a recursive type for lists of any element, however, as in Figure 4.4, then there will be lists of multiple *index*-valued structures, any pair of which can share

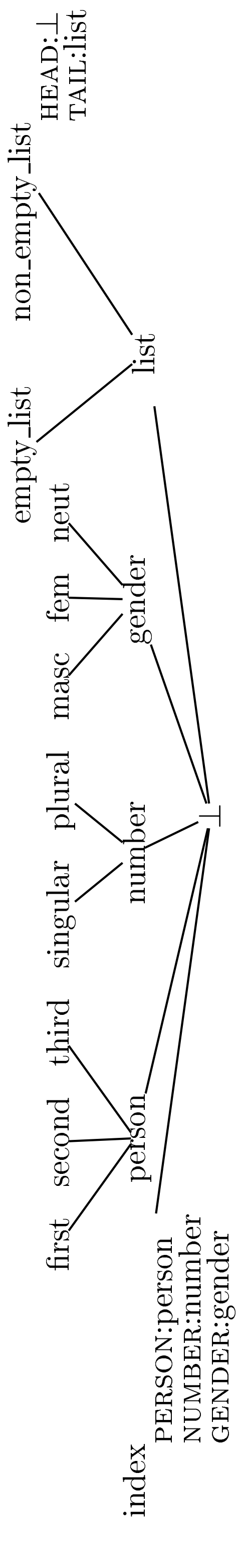

Figure 4.12: The signature in Figure 1.5 plus a recursive type for lists.

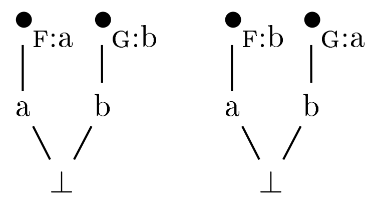

Figure 4.13: Two equivalent minimal signatures for which no apparent normalization criterion is forthcoming.

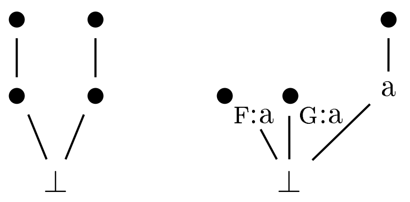

Figure 4.14: Two equivalent minimal signatures with finite $\mathcal{TTA}$.

only its PERSON or NUMBER or GENDER values. This means that *index* can only be unfolded if *list* is unfolded, but since *list* is recursive, it cannot be, at least not entirely. Note that *index* itself is still finite in this signature. The presence of a recursive type that refers to *index* in a remote corner of the same signature is enough to spoil the unfolding of *index*.

There are reasons other than efficiency to unfold a signature. Perhaps the most compelling is to use unfolding as a means of converting signatures to a normal form — to simply a proof theory over signatures, for example. Type-free signatures such as $\mathcal{TTA}$ itself are one possible normal form, although they are not always finite. They are also a strong normal form in the sense that if two signatures are equivalent, then they will have the same unique equivalent type-free signature. Strong normal forms in general can be used to prove that two signatures are equivalent. Finding a strong normal form for signatures that is guaranteed to be finite is more difficult. The two signatures in Figure 4.13 are equivalent and minimal (in the sense that no signature with fewer types is equivalent), for example, but it is difficult to see a criterion for preferring one over the other. Even when $\mathcal{TTA}$ is finite, as with the two equivalent and minimal signatures in Figure 4.14, no obvious alternative to preferring signatures with no features presents itself.

In the absence of a clear efficiency advantage for the unfolding transformation or a clear goal to which to transform, unfolding will not be pursued further here. The discovery of a sensible finite normal form for signatures is a topic for future research.

4.5 Summary

This chapter used the notions developed in the last chapter to examine finite signatures more closely. In particular, the ability of features with appropriateness to create recursive types is what makes finite signatures with features potentially more expressive than finite signatures without them. Feature-based encodings are also potentially more compact. Along the way, two other approaches to knowledge representation were considered, multi-dimensional inheritance and systemic networks, and they fall rather neatly within the range of polynomially encodable sets of information states by this logic, which is possible due to the compact products that feature-based encodings provide.

The presence of recursive features as well as the use of variables with extra-logical scope, however, typically precludes the unfolding of the features in a signature into a signature with more types as a practical device to improve the efficiency of deductive or parsing strategies over typed feature logic. Another problem that arises is the discovery of a suitably strong normal form that is guaranteed to remain finite yet restrictive enough to be of assistance in proving the equivalence of a pair of signatures. This remains a topic of future research.

Chapter 5

Parametric Types

As mentioned in Chapter 2, there have been a number of changes over time in the way in which feature structures and their types have been used in computational linguistics. Perhaps one of the more radical, but still very intuitive changes was the introduction of parametric types to classify lists and sets of linguistic objects. The reader may consider, for example, Figure 5.1, a fragment of the type signature proposed in Head-driven Phrase Structure Grammar (HPSG, Pollard and Sag, 1994), a feature-structure-based linguistic theory. The idea of treating *list* as a parametric type in HPSG was first

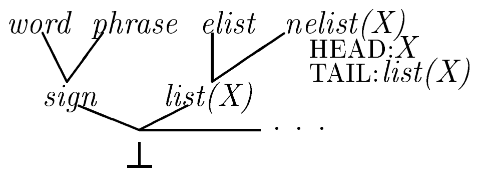

Figure 5.1: A fragment of the HPSG type signature.

broached in a footnote by Pollard [1990], and later adopted by Pollard and Sag [1994] with the restriction (again in a footnote) that parametricity could only extend "one level deep," i.e., that a parameter itself must be a type other than a list or set. That restriction has, of course, been violated many times over and in a few different ways by linguists working with parametrically typed lists since that time.

There has been some use of parametric types in computational linguistics independently of this, perhaps most prolifically by Klein [1991] to represent hierarchical structure in phonology. Certainly, the treatment of lists as parametric types is not new or unreasonable, even in the absence of the proposed depth restriction. What makes this particular change radical is that no feature logic has ever been proposed that incorporates parametric types in a general enough way to support their manner of use in HPSG, and, in contrast to the historical oscillation between feature-based and subtype-based information encodings, parametric types in HPSG have been used without exception only for lists and sets, as if either something might go terribly wrong if their use were extended to other types, or else linguists should feel ashamed for resorting to them where not absolutely necessary. At the same time, there has been an almost universal consensus among HPSG linguists that parametric types as they are currently employed are just "macro" descriptions for lists and sets — a consensus that is reflected in their liberal application within the confines of lists and sets. What is it about lists and sets that makes them absolutely necessary on the one hand, and still no better than a macro?

This chapter uses the algebraic perspective developed for signatures so far to present an incorporation of parametric types into the typed attribute-value logic of Carpenter [1992], thus providing a natural extension to the type system for programming languages based on that logic, such as the Attribute Logic Engine (ALE, Carpenter and Penn, 1996). This enquiry has yielded a more flexible interpretation of parametric types with several specific properties necessary to conform to their current usage by linguists and implementors who work with feature-based formalisms. Again, it will be assumed that total well-typing is the interpretation of choice for the appropriateness conditions found in parametrically typed signatures.

Parametric polymorphism has been combined with inclusional polymorphism before to provide natural type systems for Prolog [Dietrich and Hagl, 1988], HiLog [Yardeni et al., 1992], and constraint resolution languages [Smolka, 1989]. Previous approaches, however, have required that every parameter of a subtype should be a parameter of all of its supertypes, and *vice versa*; thus, it would not be possible to encode Figure 5.1 because $\bot \sqsubseteq \mathit{list}(X)$, and if $\bot$ were parametric, then all other types would be. The present one eliminates this restriction (Section 5.1) by requiring the existence of a non-parametric most general type (which Carpenter's [1992] logic requires anyway), which is then used during type-checking and inferencing to interpret new parameters.

The only previous attempt at an account of parametric types as they are employed in HPSG has been in King and Goetz, 1993, which consisted merely of the informal suggestion that parametric types stand for unordered sets of non-parametric types, such that, in Figure 5.1 for example, *list(word)* is not subsumed by *list(sign)*, which clearly runs against intuition. All other previous approaches to parametric polymorphism deal only with fixed-arity terms; and none but one uses a feature logic, with the one, CUF [Dorna, 1992], being an implementation of a logic without parametric types that permits parametric lists with HPSG's depth restriction as a special "hard-wired" case. The present approach (Section 5.3) provides a generalization of appropriateness that allows for a proper interpretation of subsumption, unrestricted parametricity and incremental feature introduction.

The belief that parametric types are macros is erroneous, as is the belief that their use naturally extends to parameters drawn from a general description language, e.g., *list*(LOCAL : CAT : HEAD : *verb*). This possibly arose from a confusion between type descriptions, which are part of the description language, and types, which are part of the type system. Even so, parametric types have a very wide range of potential application to computational linguistics and knowledge representation in general, just as normal types and features do; and there is no reason why they cannot be used as prolifically once they are understood. To use an earlier example, *person*, *number*, and *gender* could all be parameters of a parametric type, *index*, rather than values of features appropriate to *index*. In fact, parametrically typed encodings yield more compact specifications than simply typed encodings because they can encode products of information in their parameters, like features. Unlike features, however, they can lend their parameters to appropriateness restrictions, thus refining the feature structures induced by the signature to a closer approximation of what is actually required in the grammar itself.

It is possible, however, to regard parametric type *signatures* as a shorthand for non-parametric *signatures*. The interpretation of parametric type hierarchies is introduced in Section 5.2 by way of establishing equivalent, infinite non-parametric counterparts. Section 5.4 considers whether there are any finite counterparts, i.e., whether in actual practice finite parametric signatures are only as expressive as finite non-parametric ones, and gives a qualified "yes." These questions are formalized and answered relative to the definitions of signature equivalence and signature subsumption defined in Chapter 3.

In spite of this qualification, there is an easy way to compute with parametric types directly in an implementation, as described in Section 5.5. The two most common previous approaches have been to use the most general instance of a parametric type, e.g., *nelist*(⊥), without its appropriateness, or manually to "unfold" a parametric type into a non-parametric sub-hierarchy that suffices for a fixed grammar, e.g. Figure 5.2. The former does not suffice

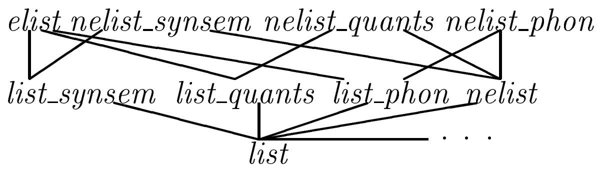

Figure 5.2: A manually unfolded sub-hierarchy.

even for fixed grammars because it simply disables type checking on feature values. The latter is error-prone, a nuisance, and subject to change with the grammar. As it happens, there is an automatic way to perform this unfolding, which turns out to be a very useful tool for the extraction of a minimal subsignature for a small fixed grammar even when parametric types are not used.

5.1 Parametric Type Hierarchies

Parametric types are not types. They are functions that provide access or a means of reference to a set of types (their image) by means of argument types, or "parameters" (their domain). Figure 5.1 has only unary functions; but in general, parametric types can be n-ary functions that map n-tuples of types to a type. *Parametric type* will be used in this chapter to refer to such a function, written as the name of the function, followed by the appropriate number of *type variables*, variables that range over some set of types, in parentheses, e.g. *list*(X). *Type* will refer to both *simple types*, such as ⊥ or *elist*; and *ground instances* of parametric types, i.e., types in the image of a parametric type function, written as the name of the function followed by the appropriate number of actual type parameters in parentheses, such as *list*(⊥), *set*(*word*) or *list*(*set*(⊥)). The letters t, u, and v will be used to indicate types; capital letters, to indicate type variables; capitalized words, to indicate feature names; p, q, and r, as names of parametric types; and g, to indicate ground instances of parametric types, where the arguments do not need to be expressed.

This means that hierarchies that use parametric types are not "type" hierarchies, since they express a relationship between functions that map types to types (we can regard simple types as nullary parametric types). For simplicity, it will be assumed here that parametric type hierarchies are finite.

Definition 5.1. *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, i.e., no ground instances of parametric types, and includes the most general type, $\bot$, which is simple, i.e., $arity(\bot) = 0$,*
- *For $p, q \in P$, $a_P(p, q, i)$, written $a_p^q(i)$, is defined iff $p \sqsubseteq_P q$ and $1 \leq i \leq arity(p)$,*
- *$0 \leq a_p^q(i) \leq arity(q)$, when it exists, and*
- *if $a_p^q(i) \neq 0$ and $a_p^q(i) = a_p^q(j)$, then $i = j$.*

As with (simple) type hierarchies, bounded completeness allows us to talk about unification, because we have a unique most-general unifier for every unifiable pair of types. The argument assignment function encodes the identification of parameters between a parametric type and its parametric subtype. The number, n, refers to the nth parameter of a parametric type, with 0 referring to a parameter that has been dropped. In practice, this is normally expressed by the names given to type variables. In the parametric type hierarchy of Figure 5.1, *list* and *nelist* share the same variable, X, because $a_{list}^{nelist}(1) = 1$. If $a_{list}^{nelist}(1) = 0$, then *nelist* would use a different variable name. As a more complicated example, in Figure 5.3, $a_b^d(1) = 1$, $a_b^d(2) = 3$,

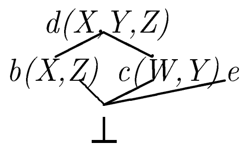

Figure 5.3: A subtype that inherits type variables from more than one supertype.

$a_c^d(2) = 2$, $a_c^d(1) = 0$, and $a_\bot$ and a_e are undefined ($\uparrow$) for any pair in $P \times \mathsf{Nat}$.

5.2 Induced Type Hierarchies

The relationship expressed between two functions by $\sqsubseteq_P$, informally, is one between their image sets under their domains, while each image set internally preserves the subsumption ordering of its domain. One could explicitly restrict these domains with *parametric restrictions*, with a function parallel to *Approp*, which specifies value restrictions on feature values. Here, it is assumed that these domains are always the set of all types in the signature. This is the most expressive case of parametric types, and the worst case to deal with computationally.

It is, thus, possible to think of a parametric type hierarchy as "inducing" a non-parametric type hierarchy, populated with the ground instances of its parametric types, that obeys both of these relationships.

Definition 5.2. *Given parametric type hierarchy, $\langle P, \sqsubseteq_P, arity, a\rangle$, the induced (type) hierarchy, $\langle I(P), \sqsubseteq_I\rangle$, is defined such that:*

- $I(P) = \bigcup_{n<\omega} I_n$*, where the sequence* $\{I_n\}_{n<\omega}$ *is defined such that:*
  - $I_0 = \{p \mid p \in P, arity(p) = 0\}$,
  - $I_{n+1} = I_n \cup \{p(t_1, \ldots, t_{arity(p)}) \mid p \in P, t_i \in I_n, 1 \leq i \leq arity(p)\}$, *and*
- $p(t_1, \ldots, t_{arity(p)}) \sqsubseteq_I q(u_1, \ldots, u_{arity(q)})$ *iff* $p \sqsubseteq_P q$*, and, for all* $1 \leq i \leq arity(p)$*, either* $a_p^q(i) = 0$ *or* $t_i \sqsubseteq_I u_{a_p^q(i)}$.

Note that $I(P)$ contains all of the simple types of P, including $\bot$, which is also the least type in $I(P)$. In the case where g_1 and g_2 are simple, $g_1 \sqsubseteq_I g_2$ iff $g_1 \sqsubseteq_P g_2$.

Figure 5.4 shows a fragment of the type hierarchy induced by Figure 5.1. If *list* and *nelist* had not shared the same type variable ($a_{list}^{nelist}(1) = 0$), then it

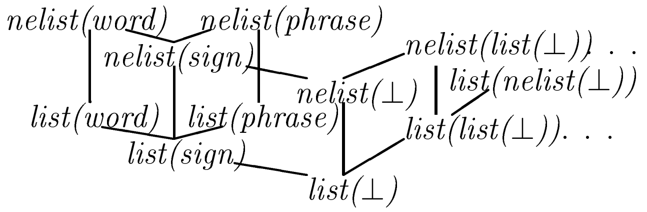

Figure 5.4: Fragment induced by Figure 5.1.

would have induced the type hierarchy in Figure 5.5. In the hierarchy induced by Figure 5.3, for example, $b(e, e)$ subsumes types $d(e, Y, e)$, for any type Y, for example $d(e, c(e, e), e)$, or $d(e, b(\bot, e), e)$, but not $d(c(\bot, e), e, e)$, since $e \not\sqsubseteq_I c(\bot, e)$. Also, for any types, W, X, and Z, $c(W, e)$ subsumes $d(X, e, Z)$.

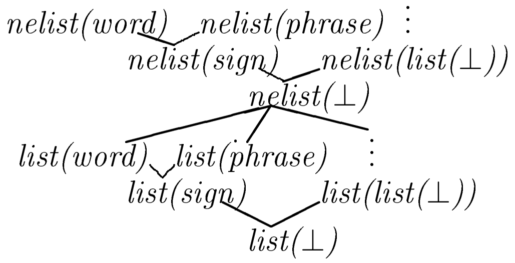

Figure 5.5: The would-be induced hierarchy of Figure 5.1 if $a^{nelist}_{list}(1)$ were 0.

The present approach permits parametric types in the type signature, but only ground instances in a grammar relative to that signature. If one must refer to "some list" or "every list" within a grammar, for instance, one may use $list(\bot)$, while still retaining groundedness. An alternative to this approach would be to attempt to cope with type variable parameters directly within descriptions. From a processing perspective, this is problematic when closing such descriptions under total well-typing, as observed by Carpenter [1992]. The most general satisfier of the description, $list(X) \wedge (\textsc{head} : \textsc{head} : Y \wedge \textsc{tail} : \textsc{head} : Y)$, for example, is an infinite feature structure of the ground instance, $nelist(nelist(\ldots))$ because X must be bound to $nelist(X)$.

We can distinguish such types with the following useful classification of types in $I(P)$:

Definition 5.3. *Given a parametric hierarchy, $\langle P, \sqsubseteq_P, arity, a\rangle$, the* parametric depth *of a type, $g = p(t_1, \ldots, t_n) \in I(P)$, $\pi(g)$, is defined such that:*

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

So, for example, $\pi(list(list(list(\bot)))) = 3$.

The construction of $I(P)$ thus excludes ground instances with infinite parametric depths. That exclusion, from the perspective of algebraic hygiene, is a rather arbitrary one; but its motivation is the present author's inability to make denotational sense of such types. The effect of this prejudice, in any case, is that $I(P)$ is not necessarily bounded complete, even when intuition tells us that it provides a proper algebraic interpretation of P. The hierarchy in Figure 5.6, for example, has the infinite set $\{\bot, a(\bot), a(a(\bot)), \ldots\}$ with upper bounds $\{b, a(b), a(a(b)), \ldots\}$, whose limit, and thus putative least upper

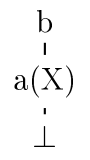

Figure 5.6: A parametric type hierarchy for which $I(P)$ is not a BCPO.

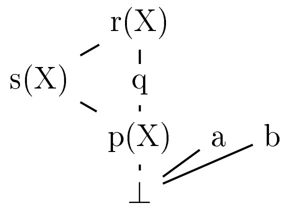

Figure 5.7: A parametric type hierarchy for which $I(P)$ is not a partial order.

bound must be the excluded limit type, $a(a(\ldots))$. This arises in Figure 5.1 where $a = \mathit{list}$ and $b = \mathit{elist}$.[1]

There are, in fact, some parametric type hierarchies, P, as defined above, for which $\langle I(P), \sqsubseteq_I \rangle$ is not even a partial order. Figure 5.7 is one such example. We should take the use of the variable X in this case to mean that $a_p^s(1) = 1$, $a_s^r(1) = 1$, and $a_p^r(1) = 1$. $a_p^q(1) = 0$, however, and thus $p(a) \sqsubseteq_I q$ and $q \sqsubseteq_{I(P)} r(b)$, but $p(a) \not\sqsubseteq_{I(P)} r(b)$. The problem is that different paths from p to r disagree on what to do with the parameter.

We can generalize the usual notion of *coherence* from programming languages, so that a subtype can add, and in certain cases drop, parameters with respect to a supertype without this disagreement:

Definition 5.4. *$\langle P, \sqsubseteq_P, arity, a_P \rangle$ is* semi-coherent *iff, for all $p, q \in P$ such that $p \sqsubseteq_P q$, all $1 \leq i \leq arity(p)$, $1 \leq j \leq arity(q)$:*

- *$a_p^p(i) = i$,*
- *either $a_p^q(i) = 0$ or for every chain, $p = p_1 \sqsubseteq_P p_2 \sqsubseteq_P \ldots \sqsubseteq_P p_n = q$, $a_p^q(i) = a_{p_{n-1}}^{p_n}(a_{p_{n-2}}^{p_{n-1}}(\ldots a_{p_1}^{p_2}(i) \ldots))$, and*
- *If $p \sqcup_P q\downarrow$, then for all i and j for which there is a $k \geq 1$ such that $a_p^{p \sqcup_P q}(i) = a_q^{p \sqcup_P q}(j) = k$, the set, $\{r | p \sqcup_P q \sqsubseteq_P r$ and $(a_p^r(i) = 0$ or*

[1]The reader may also note that same problem would have arisen with abstract feature structures if, as in most of the presentation of Carpenter [1992], infinite feature structures had been excluded. Without those limit points, $\mathcal{TTA}_S$ would often not be bounded complete either.

$a^r_q(j) = 0)\}$ *is empty or has a least element (with respect to* $\sqsubseteq_P$*).*

Proposition 5.1. *If* $\langle P, \sqsubseteq_P, arity, a_P\rangle$ *is semi-coherent, then* $\langle I(P), \sqsubseteq_I\rangle$ *is a partial order.*

Proof. 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 $\sqsubseteq_P$. If $k > 0$, then we also know that, for all $1 \leq i \leq n$, either $a^q_p(i) = 0$ or $t_i \sqsubseteq_I u_{a^q_p(i)}$. In the former case, if $a^r_p(i) \neq 0$, then the chain $p \sqsubseteq_P q \sqsubseteq_P r$ would violate semi-coherence, so $a^r_p(i) = 0$. In the latter case, if $a^r_q(a^q_p(i)) = 0$, then by the same reasoning, $a^r_p(i) = 0$. Otherwise, we have that $t_i \sqsubseteq_I u_{a^q_p(i)} \sqsubseteq_I v_{a^r_q(a^q_p(i))} = v_{a^r_p(i)}$, and thus, by induction, $t_i \sqsubseteq_I v_{a^r_p(i)}$. This applies to all $1 \leq i \leq n$, so $g_1 \sqsubseteq g_3$.

Reflexivity and anti-symmetry follow from a similar inductive proof. □

For the sake of generality, we can relax the requirement of bounded completeness to meet-semi-latticehood for now. Section 5.4 will present a further refinement of parametric type hierarchies for independent reasons that restores bounded completeness. Section 5.5 will consider the use of finite subsets of $I(P)$ for practical purposes, and bounded completeness is equivalent to meet-semi-latticehood on all finite partially ordered sets. As it happens, semi-coherence is also enough to ensure that $I(P)$ is a meet semi-lattice.

Proposition 5.2. *If* $\langle P, \sqsubseteq_P, arity, a_P\rangle$ *is semi-coherent, then* $\langle I(P), \sqsubseteq_I\rangle$ *is a meet semi-lattice. 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* $a^{p\sqcup q}_p(i) = a^{p\sqcup q}_q(j) = k$, *then* $t_i \sqcup_I u_j\downarrow$ *or* $a^s_p(i) = 0$ *or* $a^s_q(j) = 0$.

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* $1 \leq h \leq l$*:*

$$v_h = \begin{cases} t_i \sqcup_I u_j & \textit{if there exist } i \textit{ and } j \textit{ such that} \\ & \quad a^r_p(i) = h \textit{ and } a^r_q(j) = h \\ t_i & \textit{if such an } i\textit{, but no such } j \\ u_j & \textit{if such a } j\textit{, but no such } i \\ \perp & \textit{if no such } i \textit{ or } j. \end{cases}$$

Proof. ($\Rightarrow$): By contraposition using induction on the greater parametric depth, k, of types $g_1 = p(t_1, \ldots, t_n)$ and $g_2 = q(u_1, \ldots, u_m)$ in $I(P)$.

If $k = 0$ and $p \sqcup_P q\uparrow$, then by the bounded completeness of P, $\{g_1, g_2\}^u = \{p, q\}^u = \emptyset$. Otherwise, if $k = 0$, then $g_1 \sqcup_I g_2 = (p \sqcup_P q)(\bot_1, \ldots, \bot_{arity(p \sqcup q)})$, which is the least instance of $p \sqcup_P q$.

Suppose $k > 0$. If $p \sqcup_P q\uparrow$, then again, $\{g_1, g_2\}^u = \emptyset$. Otherwise, suppose that for all $s \sqsupseteq_P p \sqcup_P q$, there exist i, j and a $k > 0$ such that $a_p^{p \sqcup q}(i) = a_q^{p \sqcup q}(j) = k$ and $t_i \sqcup_I u_j\uparrow$ and $a_p^s(i) \neq 0$ and $a_q^s(j) \neq 0$. Now consider some $g_3 = s(v_1, \ldots, v_l)$ such that $g_1 \sqsubseteq_I g_3$. So $p \sqsubseteq_P s$. Either $q \sqsubseteq_P s$ or not. If not, then $g_2 \not\sqsubseteq_I g_3$ and $g_3 \notin \{g_1, g_2\}^u$. If so, then $p \sqcup_P q \sqsubseteq_P s$. Consider the i, j and k of s as specified above. $t_i \sqcup_I u_j\uparrow$, so by induction, $\{t_i, u_j\}^u = \emptyset$. Since $g_1 \sqsubseteq_I g_3$ and $a_p^s(i) \neq 0$, $t_i \sqsubseteq_I v_{a_p^s(i)}$, so $u_j \not\sqsubseteq_I v_{a_p^s(i)}$. $p \sqsubseteq_P p \sqcup_P q \sqsubseteq_P s$ and $q \sqsubseteq_P p \sqcup_P q \sqsubseteq_P s$ are chains, so by semi-coherence, $a_p^s(i) = a_{p \sqcup q}^s(a_p^{p \sqcup q}(i)) = a_{p \sqcup q}^s(a_q^{p \sqcup q}(j)) = a_q^s(j)$. So $u_j \not\sqsubseteq_I v_{a_q^s(j)}$ and since $a_q^s(j) \neq 0$, $g_2 \not\sqsubseteq_I g_3$. Thus, in either case, $\{g_1, g_2\}^u = \emptyset$.

($\Leftarrow$): It is sufficient to show that when such an s exists, there is a least such 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 $a_p^{p \sqcup q}(i) = a_q^{p \sqcup q}(j) = k > 0$, $t_i \sqcup_I u_j\uparrow$, and either $a_p^s(i) = 0$ or $a_q^s(j) = 0$. If there are no such $\langle i, j, k \rangle$, then $t_i \sqcup u_j\downarrow$ 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. $\square$

In the induced hierarchy of Figure 5.3, for example, $b(e, \bot) \sqcup_I b(\bot, e) = b(e, e)$; $b(e, e) \sqcup_I c(\bot) = d(e, \bot, e)$; and $b(e, e)$ and $b(c(\bot), e)$ are not unifiable, as e and $c(\bot)$ are not unifiable. The first two conditions of semi-coherence ensure that a_P, taken as a relation between pairs of pairs of types and natural numbers, is an order induced by the order, $\sqsubseteq_P$, where it is not, taken as a function, zero. The third ensures that joins are preserved even when a parameter is dropped ($a_P = 0$). Note that joins in an induced hierarchy do not always correspond to joins in a parametric hierarchy. In those places where $a_P = 0$, types can unify without a corresponding unification in their parameters. Such is the case in Figure 5.5, where every instance of $list(X)$ ultimately subsumes $nelist(\bot)$. One may also note that semi-coherent induced hierarchies can have not only deep infinity, i.e., non-Noetherianity, where there exist infinitely long subsumption chains, but broad infinity, where certain types can have infinite supertype (but never subtype) branching factors, as in the case of $nelist(\bot)$ or, in Figure 5.1, *elist*.

5.3 Appropriateness

So far, we have formally considered only parametric type hierarchies, with no appropriateness. Appropriateness constitutes an integral part of a parametric type signature's expressive power, because the scope of its type variables can extend to include it.

Definition 5.5. *The* restriction *of $I(P)$ to $p \in P$, $I_p(P)$, is defined such that $I_p(P) = \{p(t_1, \ldots, t_{arity(p)}) \in I(P) \mid t_i \in I(P), 1 \leq i \leq arity(p)\}$.*

Definition 5.6. *Given a parametric type, p, for all $i > 0$, the i^{th}* parametric projection *is a partial function, $\pi_i : I(P) \longrightarrow I(P)$ such that for any $g = p(t_1, \ldots, t_{arity(p)})$, with $arity(p) \geq i$, $\pi_i(g) = t_i$.*

Definition 5.7. *A function $f : I(P) \longrightarrow I(P)$ 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 5.8. *A parametric (type) signature is a semi-coherent parametric type hierarchy, $\langle P, \sqsubseteq_P, arity, a_P\rangle$, along with a finite set of features, $Feat_P$, and a partial (parametric) appropriateness specification, $Approp_P : Feat_P \times P \longrightarrow (I(P) \longrightarrow I(P))$, such that:*

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

$Approp_P$ maps a feature and the parametric type for which it is appropriate to a function that defines value restrictions on the image of that parametric type. The last two conditions are extensions of Carpenter's [1992] conditions on appropriateness (Definition 2.21, this dissertation). The first says that appropriateness conditions on one parametric type are binding on all of the types in its image, and on none of the types in the image of any other parametric type. All three kinds of parametrically determined functions are realized in practice for $Approp_P(\text{F}, p)$. In HPSG, for example, one finds:

- a constant function, at the feature, SUBCAT, which is introduced by a simple type, *cat*, whose value restriction is *list(synsem)* (not shown in Figure 5.1).
- a projection function, in Figure 5.1, at the type, *nelist(X)*, for which the feature HD's value restriction is simply the parameter, *X*.
- a parametrically decomposable function, again at the type *nelist(X)*, for which the feature, TL, has the value restriction, *list(X)*.

The ability to reflect parameters in value restrictions is what conveys the impression that ground instances of lists or other parametric types are more deeply related to their parameter types than just in name.

The use of parameters in appropriateness restrictions is also what prevents us from treating instances of parametric types in descriptions as instantiations of macro descriptions. These putative "macros" would be, in many cases, equivalent only to infinite descriptions without such macros, and thus would extend the power of the description language beyond the limits of HPSG's own logic and model theory. Lists in HPSG would be one such case, moreover, as they place typing requirements on every element of lists of unbounded length. Ground instances of parametric types are also routinely used in appropriate value restrictions, whose extension to arbitrary descriptions would substantially extend the power of appropriateness as well. This alternative will not be pursued further here.

A parametric signature induces a type hierarchy as defined above, along with the appropriateness conditions on its ground instances.

Definition 5.9. *The* induced appropriateness function, $Approp_{I(P)} : Feat_P \times I(P) \longrightarrow I(P)$, *is a partial function defined such that, for every feature,* $\mathrm{F} \in Feat_P$, *every ground instance,* $g = p(t_1, \ldots, t_{arity(p)}) \in I(P)$, $Approp_{I(P)}(\mathrm{F}, g)\downarrow$ *iff* $Approp_P(\mathrm{F}, p)\downarrow$, *and when defined,* $Approp_{I(P)}(\mathrm{F}, g) = Approp_P(\mathrm{F}, p)(g)$.

Proposition 5.3. *If* $\langle P, \sqsubseteq_P, arity, a_P\rangle$ *is a parametric type signature, then* $Approp_{I(P)}$ *is an appropriateness specification.*

5.4 Subsumption with Parametric Signatures

Now that parametric type signatures have been formalized, one can ask whether parametric types really add something to the expressive power of typed attribute-value logic. As seen in Chapter 3, there are at least two ways in which to present that question:

Question 5.1. *For every (semi-coherent) parametric signature,* P, *is there a non-parametric signature,* N, *such that* $P \approx_S N$*?*

If, for every parametric signature P, there is an order-isomorphism between the totally well-typed abstract feature structures induced by P (by way of $I(P)$) and those of some non-parametric signature N, then parametric signatures add no expressive power at all — their feature structures are just those of some non-parametric signatures painted a different color. This is still an open question. There is, however, a weaker but still relevant question:

Question 5.2. *For every parametric type signature,* P, *is there a non-parametric type signature,* N, *such that* $P \sqsubseteq_S N$*?*

If for every parametric P, there is a join-preserving encoding that embeds the totally well-typed abstract feature structures of P into those of N, then it is possible to embed problems (specifically, unifications) that we wish to solve from P into N, solve them, and then map the answers back to P. In this reading, programmers or linguists who want to think about their programs with P must accept no non-parametric imitations because N may not have exactly the same structure of information states; but an implementor of a interpreter for a language based on feature logic, for example, could secretly perform all of the work for those programs in N, and no one would ever notice.

Under this reading, many parametrically typed encodings add no extra expressive power. This class of signatures also has the very fortunate property of ensuring the bounded completeness of their induced signatures.

Definition 5.10. *A parametric type hierarchy,* $\langle P, \sqsubseteq_P, \mathit{arity}, a_P\rangle$ *is* persistent *iff* a_P *never attains zero.*

Proposition 5.4. *If* $\langle P, \sqsubseteq_P, \mathit{arity}, a_P\rangle$ *is persistent, then* $\langle I(P), \sqsubseteq_I\rangle$ *is a BCPO.*

Proof. In light of Proposition 5.2, it suffices to show that if any set $S \in I(P)$ is bounded, then S is finite. Given S, with bound b, since P is persistent, the parametric depth of every type in S is bounded by $\pi(b)$. Since P is finite, there are finitely many types of any bounded parametric depth in $I(P)$, so S is finite. □

Along with Proposition 5.3, this means that $\langle I(P), \sqsubseteq_I, \mathit{Approp}_{I(P)}\rangle$ is a signature.

Theorem 5.1. *For any persistent parametric signature, P, there is a finite non-parametric signature, N, such that $P \sqsubseteq_S N$.*

The proof is given in the appendix to this chapter. As a first approximation, one might guess that such an N could be found by using extra features to encode parameters. That guess is essentially correct.

If *elist* in Figure 5.1 retained the parameter of $\mathit{list}(X)$, then HPSG's type hierarchy (without sets) would be persistent. This is not an unreasonable change to make. The encoding, however, requires the use of *junk slots* [Aït-Kaći, 1984, Carpenter, 1992], attributes with no empirical significance whose values serve as workspace to store intermediate results.

There are at least some non-persistent P, including the portion of HPSG's type hierarchy explicitly introduced by Pollard and Sag [1994] (without sets), that subsume a finite non-parametric N; but the embeddings are far more complicated. It can be proven, for example, that for any such P, some of its acyclic feature structures must be encoded by cyclic feature structures in N; and the encoding cannot be injective on the equivalence classes induced by the types of P, i.e., the feature structures of some type in N must encode the feature structures of more than one type from P. While parametric types may not be formally necessary for the grammar presented by Pollard and Sag [1994] in the absolute sense, their use in that grammar does roughly correspond to cases for which the alternative would be quite unappealing. Of course, parametric types are not the only extension that would ameliorate the formulation of an adequate signature. The addition of relational expressions, functional uncertainty, or more powerful appropriateness restrictions can completely change the picture.

5.5 Finiteness

It would be ideal if, for the purposes of feature-based natural language processing, one could simply forget the encodings, unfold any parametric type signature into its induced signature at compile-time and then proceed as usual. This is not possible for systems that precompute all of their type operations, as the induced signature of any parametric signature with at least one non-simple parametric type contains infinitely many types.[2] On the other hand, at least some precompilation of type information has proven to be an empirical necessity for efficient processing. Even with respect to earlier untyped versions of feature logic, sensible implementations will use *de facto* feature cooccurrence constraints to achieve much of the same effect. Given that one will only see finitely many ground instances of parametric types in any fixed theory, however, it is sufficient to perform some precompilation specific to those instances, which will involve some amount of unfolding. What is needed is a way of determining, given a signature and a grammar, what part of the induced hierarchy could be needed at run-time, so that type operations can be compiled only on that part.

One way to identify this part is to consider only those types whose parametric depth is bounded by some constant. The problem with this method

[2]With parametric restrictions (p. 126), this is not necessarily the case.

is that a depth-bounded set of types may not be closed under unification in $I(P)$, $\sqcup_{I(P)}$.

Another way to identify this part is to identify some set of ground instances (a *generator set*) that are important for computation, and explicitly close that set under $\sqcup_{I(P)}$:

Definition 5.11. *The* sub-algebra generated by G, $I(G) \subseteq I(P)$, *is the smallest subset of* $I(P)$ *such that:*

- $G \subseteq I(G)$, *and*
- *if* $g_1 \in I(G)$, $g_2 \in I(G)$, *and* $g_1 \sqcup_{I(P)} g_2 \downarrow$, *then* $g_1 \sqcup_{I(P)} g_2 \in I(G)$.

To prove that $I(G)$ is finite, we need the following variation on parametric depth:

Definition 5.12. *Given a parametric hierarchy,* $\langle P, \sqsubseteq_P, arity, a\rangle$, *the* fringed parametric depth *of* $g = p(t_1, \ldots, t_n) \in I(P)$, $\phi(g)$, *is defined such that:*

$$\phi(g) = \begin{cases} -1 & \textit{if} \quad g = \bot, \\ 0 & \textit{if} \quad g \neq \bot, n = 0, \\ 1 + \max_{1 \leq i \leq n} \phi(t_i) & \textit{if} \quad n > 0 \end{cases}$$

For any $k < \omega$, *a set,* $G \subseteq I(P)$, *is* k-fringed *iff for all* $g \in G$, $\phi(g) \leq k$.

Proposition 5.5. *If* G *is finite, then there is a finite* $k \geq 0$ *such that* G *is* k*-fringed.*

Proposition 5.6. *If* G *is* k*-fringed, then* $I(G)$ *is* k*-fringed.*

Proof. By induction on k. The crucial case is the base case, in which the join of two simple types is either simple or an instance of a non-simple parametric type with every parameter equal to $\bot$, which therefore does not change the fringed depth. □

Proposition 5.7. *If* G *is* k*-fringed, then* G *is finite.*

Theorem 5.2. *If* G, *is finite, then* $I(G)$ *is finite.*

Proof. By Proposition 5.5, there is a $k \geq 0$ such that G is k-fringed. By Proposition 5.6, $I(G)$ is k-fringed. By Proposition 5.7, $I(G)$ is finite. □

$|I(G)|$ is exponential in $|G|$ in the worst case; but if the maximum *parametric depth* of G can be bounded (thus bounding $|G|$), then it is polynomial in $|P|$, although still exponential in the maximum arity of P: In practice, the maximum parametric depth should be quite low,[3] as should the maximum arity. A standard closure algorithm, such as the meet semi-lattice completion algorithm given in Section 2.1.2, can be used. One could also perform the closure lazily during processing to avoid a potentially exponential delay at compile-time. All of the work, however, can be performed at compile-time. One can easily construct a generator set: simply collect all ground instances of types attested in the grammar, or collect them and add all of the simple types, or add the simple types along with some extra set of types distinguished by the user at compile-time. The partial unfoldings like Figure 5.2 are essentially manual computations of $I(G)$.

The benefit of this approach is that, by definition, $I(G)$ is always closed under unification of consistent types in $I(P)$. In fact, $I(G)$ is the least set of types that is adequate for unification-based processing with a grammar based on G. So far, features have not been considered, however. In practice, one needs to close not only under unification but also under $Approp_{I(P)}$ so that a type will always appear in a sub-algebra along with the types that the feature values of its most general satisfier must take. The easiest way to ensure this is to require, for all $p \in P$, $\text{F} \in Feat$, and $g \in I_p(P)$, that $\phi(Approp_P(\text{F}, p)(g)) \leq \phi(g)$ whenever $Approp_P(\text{F}, p)\!\downarrow$. Then closure under appropriateness is also guaranteed not to increase the fringed depth of the original set, and thus remain finite. Other restrictions could be found. Clearly, this method of sub-signature extraction can be used even in the absence of parametric types, and is a useful, general tool for large-scale grammar design and signature re-use.

5.6 Appendix: Proof of Theorem 5.1

A very straightforward proof exists for a class of parametric signatures that are very well-behaved in their parameters:

Definition 5.13. *A parametric signature, P, is* parametrically join-preserving*, iff for all $p, q \in P$ such that $r = p \sqcup q$ is defined, for all* $\text{F} \in$ *Feat such that*

[3]With lists, so far as the present author is aware, the potential demand has only reached $\pi = 3$ [Manning and Sag, 1998] in the HPSG literature to date, and $\pi = 6$ overall [Manning, 1996].

$$\begin{tikzcd}
pq(p_1q_1, p_2q_2) \arrow[d, dash] \arrow[dr, dash] & pr(p_1r_2, p_2r_1) \arrow[dl, dash] \arrow[dr, dash] & qr(q_1r_1, q_2r_2) \arrow[dl, dash] \arrow[d, dash] \\
p(p_1, p_2) \arrow[dr, dash] & q(q_1, q_2) \arrow[d, dash] & r(r_1, r_2) \arrow[dl, dash] \\
 & \bot &
\end{tikzcd}$$

Figure 5.8: A parametric type hierarchy for which a straightforward mapping of parameters to features fails.

Approp(F, r)$\downarrow$, *for all* $g_1 \in I_p(P)$ *and* $g_2 \in I_q(P)$ *such that* $g_3 = g_1 \sqcup_I g_2\downarrow$, *and for all* $1 \leq k \leq arity(r)$*:*

$$\pi_k(Approp(\text{F}, r)(g_3)) = \begin{cases} \pi_i(Approp(\text{F}, p)(g_1)) & \textit{if there exist } i, j \textit{ such that } a^r_p(i) = a^r_q(j) = k, \\ \sqcup\ \pi_j(Approp(\text{F}, q)(g_2)) & \\ \pi_i(Approp(\text{F}, p)(g_1)) & \textit{if there exists such an } i \textit{ but no such } j, \\ \pi_j(Approp(\text{F}, q)(g_2)) & \textit{if there exists such a } j \textit{ but no such } i, \\ \bot & \textit{if there exist no such } i \textit{ or } j \end{cases}$$

Note that in the first case, an upper bound must exist by the definition of appropriateness specifications, so the least upper bound must exist by Proposition 5.2.

In the case of signatures induced by parametrically join-preserving parametric signatures, parameters can be replaced in an equivalent non-parametric signature by extra features. Even then, how these features are allocated is not entirely trivial. Figure 5.8 shows an example of a parametric type hierarchy for which we cannot simply assign one feature to every parameter. In this hierarchy, $a^{pq}_p(1) = 1$, and $a^{pq}_q(1) = 1$, so p_1 and q_1 would need to map to the same feature, and likewise for q_1 and r_1 because of qr; thus, p_1 and r_1 must map to the same feature. On the other hand, $a^{pr}_p(1) = 1$, but $a^{pr}_r(1) = 2$ and instead $a^{pr}_r(2) = 1$, so p_1 and r_1 should not map to the same feature. This can be solved for *persistent* parametric signatures by allocating one feature for every parameter position of every maximally specific type. By persistence, all parameters of a type must eventually be reflected in one parameter of each of that type's maximal extensions. That means, however, that parameters must potentially be mapped to multiple features in the encoding.

Lemma 5.1. *If P is persistent and parametrically join-preserving, then there is a finite non-parametric signature, N, such that $P \sqsubseteq_S N$.*

Proof. Define $\mathit{Feat}_N = \mathit{Feat}_P \cup \{X^m_k \mid m \in P$, maximally specific, $1 \leq k \leq \mathit{arity}(m)\}$, $\bar{N} = P$, and $\sqsubseteq_{\bar{N}} = \sqsubseteq_P$. For all $\textsc{f} \in \mathit{Feat}_P$ and $p \in \bar{N}$, let $\mathit{Approp}_{\bar{N}}(\textsc{f}, p)\downarrow$ iff $\mathit{Approp}_P(\textsc{f}, p)\downarrow$, and when it exists:

$$\begin{aligned}&\mathit{Approp}_{\bar{N}}(\textsc{f}, p) = \\ &\quad \sqcap_P\{q \mid \text{ there exists } g \in I_p(P) \text{ such that } \mathit{Approp}_P(\textsc{f}, p)(g) \in I_q(P)\}\end{aligned}$$

For all $X^m_k \in \mathit{Feat}_N \backslash \mathit{Feat}_P$ and $p \in \bar{N}$, let $\mathit{Approp}_{\bar{N}}(X^m_k, p) = \bot$ if $p \sqsubseteq_P m$, and there exists an $1 \leq i \leq \mathit{arity}(p)$ such that $a^m_p(i) = k$, and be undefined elsewhere. Let N be the meet-semi-lattice completion of the signature completion of $\bar{N}$.

We can define an order-embedding, $f : \mathcal{TTA}_{I(P)} \longrightarrow \mathcal{TTA}_N$ such that $F = \langle \Pi_F, \Theta_F, \approx_F, \not\approx_F \rangle$ is mapped to $f(F) = \langle \Pi_{f(F)}, \Theta_{f(F)}, \approx_{f(F)}, \not\approx_{f(F)} \rangle$ where $\Theta_{f(F)}$ is defined inductively on parametric depth such that:

$$\Theta_{f(F)}(\pi) = \begin{cases} p & \text{if } \pi \in \Pi_F, \Theta_F(\pi) \in I_p(P) \\ \Theta_G(\pi_G) & \text{if } \pi = \pi' X^m_k \pi_G, \pi' \in \Pi_F, \Theta_F(\pi') = p(t_1, \ldots, t_{\mathit{arity}(p)}), \\ & p \sqsubseteq_N m, a^m_p(i) = k, f(\mathit{Abs}(\mathit{TWT}(\mathit{MGSat}(t_i)))) = \\ & \langle \Pi_G, \Theta_G, \approx_G, \not\approx_G \rangle, \text{and } \pi_G \in \Pi_G \\ \text{undefined} & \text{otherwise} \end{cases}$$

$\Pi_{f(F)} = \{\pi \mid \Theta_{f(F)}\downarrow\}$ and $\approx_{f(F)}$ and $\not\approx_{f(F)}$ are the smallest relations such that $\approx_F \subseteq \approx_{f(F)}$, $\not\approx_F \subseteq \not\approx_{f(F)}$, and f(F) is an abstract feature structure.

The only difference between P and N is the placement of parameters. We must unify $f(F_1)@\pi_1 X^{m_1}_{k_1}$ with $f(F_2)@\pi_2 X^{m_2}_{k_2}$ iff $m_1 = m_2$, $k_1 = k_2$, and we must unify $F_1@\pi_1$ with $F_2@\pi_2$ and for p and q such that $\Theta_{F_1}(\pi_1) \in I_p(P)$ and $\Theta_{F_2}(\pi_2) \in I_q(P)$, $p \sqsubseteq_P m$, and $q \sqsubseteq_P m$. By persistence, for all $1 \leq k \leq \mathit{arity}(m)$, there are i and j such that $a^m_p(i) = a^m_q(j) = k$. Also, $m \in \{p, q\}^u$, so $p \sqcup_P q\downarrow$, and $p \sqcup_P q \sqsubseteq_P m$, and thus by semi-coherence, there is an $h > 0$ such that $a^{p \sqcup_P q}_p(i) = a^{p \sqcup_P q}_q(j) = h$ and $a^m_{p \sqcup_P q}(h) = k$. So for all $1 \leq k \leq \mathit{arity}(m)$, $\Theta_{f(F_1)}(\pi_1 X^m_k) \sqcup_N \Theta_{f(F_2)}(\pi_2 X^m_k)\downarrow$ iff $\Theta_{F_1}(\pi_1) \sqcup_{I(P)} \Theta_{F_2}(\pi_2)\downarrow$ and since the substructures of any πX^m_k are most general satisfiers, $\mathit{TWT}(f(F_1) \sqcup_{\mathcal{TTA}_{I(N)}} f(F_2))\downarrow$ iff $\mathit{TWT}(F_1 \sqcup_{\mathcal{TTA}_{I(P)}} F_2)\downarrow$. Since P is parametrically join-preserving, $\mathit{TWT}(f(F_1) \sqcup_{\mathcal{TTA}_{I(N)}} f(F_2)) = \mathit{TWT}(f(F_1 \sqcup_{\mathcal{TTA}_{I(P)}} F_2))$, when it exists. $\square$

Not all parametric signatures are parametrically join-preserving. Figure 5.9 shows a simple counter-example. If P is not parametrically join-preserving, then $I(P)$ is not join preserving, but not *vice versa*, since it is

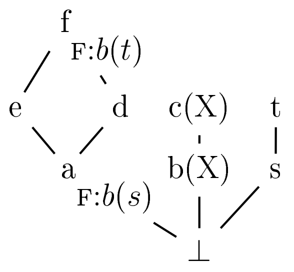

Figure 5.9: An example of a parametric signature that is not parametrically join-preserving.

possible to violate join preservation with only simply-typed value restrictions, for example, or even with non-simple value restrictions provided that the parameters themselves are assigned in a parametrically join-preserving way. If the value restriction at f had been $c(s)$, for example, the parametric signature would be parametrically join-preserving but its induced signature would not be join preserving. Parametric join preservation only constrains the parameters themselves.

It remains to be shown that all persistent parametric signatures are signature-equivalent to a persistent, parametrically join-preserving signature. The notions of equivalence and subsumption among signatures defined here are extensional in that they pay no attention to syntactic properties of a signature definition itself (such as parametric join preservation), but only to the partial order of totally well-typed abstract feature structures that it induces. Even though a parametric signature is not parametrically join-preserving, it is thus still possible to prove such an equivalence by showing that a parametrically join-preserving one could have induced the same thing.

The proof of Theorem 5.1 relies on a method for transforming parametric signatures into equivalent ones that have potentially different syntactic properties.

Definition 5.14. *Given a persistent parametric signature, P, and $i \geq 0$, the i^{th}* extension *of P, E_i, is defined such that:*

- $E_0 = P$,
- $E_{i+1} = \{e^{\langle p_1,\ldots,p_{arity(e)}\rangle} \mid e \in E_i, p_j \in E_i,\ \text{all } 1 \leq j \leq arity(e)\}$,
- $arity(e^{\langle p_1,\ldots,p_{arity(e)}\rangle}) = \sum_{j=1}^{arity(e)} arity(p_j)$,

- $e^{\langle p_1,\ldots,p_{arity(e)}\rangle} \sqsubseteq_{E_{i+1}} \bar{e}^{\langle \bar{p}_1,\ldots,\bar{p}_{arity(\bar{e})}\rangle}$ *iff* $e \sqsubseteq_{E_i} \bar{e}$ *and for all* $1 \leq i \leq arity(e)$, $p_i \sqsubseteq_{E_i} \bar{p}_{a_e^{\bar{e}}(i)}$, *and*

- *for all* $1 \leq i \leq arity(e^{\langle p_1,\ldots,p_{arity(e)}\rangle})$, $a_{e^{\langle p_1,\ldots,p_{arity(e)}\rangle}}^{\bar{e}^{\langle \bar{p}_1,\ldots,\bar{p}_{arity(\bar{e})}\rangle}}(i) = a_{p_{b(i)}}^{\bar{p}_{a_e^{\bar{e}}(b(i))}}(c(i)) + \sum_{j=1}^{a_e^{\bar{e}}(b(i))-1} arity(\bar{p}_j)$, *where*

  - $b(i)$ *is that* k *for which* $\sum_{j=1}^{k-1} arity(p_j) < i \leq \sum_{j=1}^{k} arity(p_j)$, *and*
  - $c(i) = i - \sum_{j=1}^{b(i)-1} arity(p_j)$.

Notice that all of the E_i contain the simple types of P.

Definition 5.15. *Given a persistent parametric signature, P, and its first extension, E_1, the* canonical type embedding of P into E_1 *is the function,* $\hat{\cdot} : P \longrightarrow E_1$, *defined such that:*

$$\widehat{e(t_1,\ldots,t_{arity(e)})} = \begin{cases} e & \textit{if } arity(e) = 0, \\ e^{\langle p_1,\ldots,p_{arity(e)}\rangle}(\hat{g}_{\langle 1,1\rangle},\ldots,\hat{g}_{\langle 1,arity(p_1)\rangle}, & \textit{where for all } 1 \leq i \leq arity(e), \\ \quad \hat{g}_{\langle 2,1\rangle},\ldots,\hat{g}_{\langle arity(e),arity(p_{arity(e)})\rangle}) & \quad t_i = p_i(g_{\langle i,1\rangle},\ldots,g_{\langle i,arity(p_i)\rangle}) \end{cases}$$

Proposition 5.8. $\hat{\cdot}$ *is an order-isomorphism.*

Definition 5.16. *The* i^{th} extended signature of P *is defined over the* i^{th} *extension of P, with appropriateness defined such that:*

- $Approp_{E_0} = Approp_P$,
- $Approp_{E_{i+1}}(\texttt{F}, e^{\langle p_1,\ldots,p_{arity(e)}\rangle})\downarrow$ *iff* $Approp_{E_i}(\texttt{F}, e)\downarrow$, *and*
- *when it exists,* $Approp_{E_{i+1}}(\texttt{F}, e^{\langle p_1,\ldots,p_{arity(e)}\rangle}) = \hat{\cdot} \circ Approp_{E_i}(\texttt{F}, e) \circ \hat{\cdot}^{-1}$.

Henceforth, E_i will be used to refer to the extended signature over E_i. Extended parametric signatures are technically not parametric signatures, because their appropriateness specifications may not be parametrically determined, but they clearly induce valid appropriateness specifications in $I(P)$. Since the proof of Lemma 5.1 constructs the appropriateness specification for N directly from the appropriateness specification of $I(P)$, and the construction of further extensions from this definition does the same, this is an acceptable departure. The property of parametric determination is only used below in Lemma 5.3.

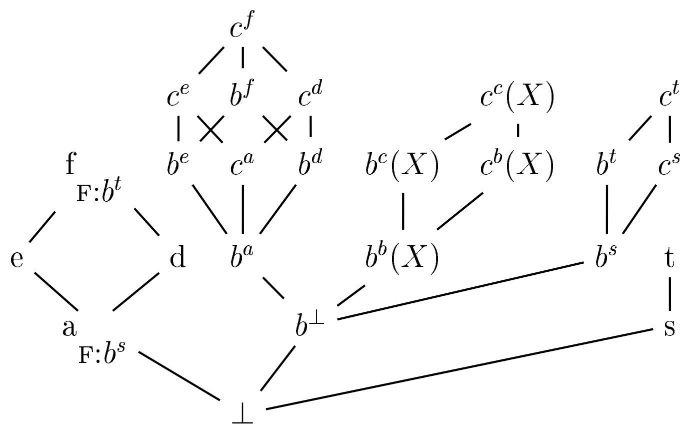

Figure 5.10: The first extended signature of Figure 5.9.

Proposition 5.9. *For all $i < \omega$, $E_i \approx_S P$.*

In order to see an extension at work, Figure 5.10 shows the first extended signature of Figure 5.9. It is parametrically join-preserving, but not join-preserving. This extended signature is also a valid parametric signature, but in general that may not be so, as explained above. Note that this same method cannot be used to convert non-statically-typable signatures into statically typable ones.

This technique will now be applied to parametric signatures in a way that depends on whether or not they satisfy another syntactic property:

Definition 5.17. *A persistent parametric signature, P, is* parametrically separated *iff:*

- *for every simple type, $s \in P$, and every non-simple type, $p \in P$, $s \sqsubseteq_P p$, and*
- *its non-simple parametric types are totally ordered.*

Parametrically separated signatures all look like the schematic hierarchy shown in Figure 5.11, where S is the set of all simple types of P, and $k_1 \leq \ldots \leq k_n$. It is easy to see that if there are any join-reducible types, they must reduce to simple types, and thus, if parametric join preservation is not satisfied, it is not satisfied by values of $Approp_P(\text{F}, s)$ that are constant functions.

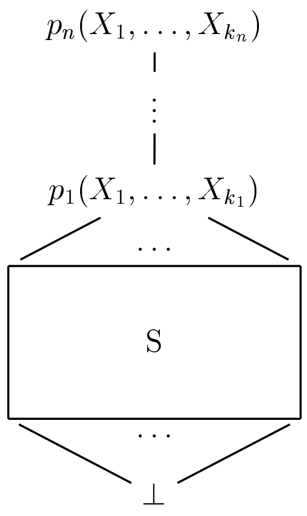

Figure 5.11: A schematic illustration of a parametrically separated parametric type hierarchy.

Lemma 5.2. *If P is parametrically separated, then there exists an $i < \omega$ such that E_i is parametrically join-preserving.*

Proof. Consider:

$$i = \max_{\substack{s \in P, \text{ simple}, \\ \textsc{f} \in \mathit{Feat}}} \pi(\mathit{Approp}_P(\textsc{f}, s)).$$

The i^{th} extension[4] of P converts the value restrictions of potentially non-compliant simple types to simple types, and thus E_i is trivially parametrically join-preserving. $\square$

Another syntactic class can be distinguished as being trivially parametrically join-preserving:

Definition 5.18. *A persistent parametric (or extended) signature, P, is* parametrically transparent *iff for all $p, q \in P$ such that $p \sqsubseteq_P q$, all $g_1 \in I_p(P)$, $g_2 \in I_q(P)$ and all $\textsc{f} \in \mathit{Feat}$ such that $\mathit{Approp}_P(\textsc{f}, p)\downarrow$, if for all $1 \leq i \leq \mathit{arity}(p)$, $\pi_i(g_1) = \pi_{a_p^q(i)}(g_2)$, then for all $1 \leq i \leq \mathit{arity}(\bar{p})$, $t_i = u_{a_{\bar{p}}^{\bar{q}}(i)}$, where $\mathit{Approp}_I(\textsc{f}, g_1) = \bar{p}(t_1, \ldots, t_{\mathit{arity}(\bar{p})})$, and $\mathit{Approp}_I(\textsc{f}, g_2) = \bar{q}(u_1, \ldots, u_{\mathit{arity}(\bar{q})})$.*

[4]Actually, $\lceil \log(i) \rceil$ is a sufficient number of extensions since every extension reduces the parametric depth of an instance in $I(P)$ by half.

Parametrically transparent signatures do not change the parameters of value restrictions at all over subsumption chains and therefore:

Proposition 5.10. *Every parametrically transparent signature is parametrically join-preserving.*

The following lemma then proves the theorem, since the k it provides can also be used to extend P to a parametrically transparent signature.

Lemma 5.3. *If P is persistent but not parametrically separated, then for all $p, q \in P$ such that $p \sqsubseteq_P q$, and for all $\mathtt{F} \in Feat$ such that $Approp_P(\mathtt{F}, p)\downarrow$, if there exist $i_0, \ldots, i_k$, $k \geq 0$, such that $\pi_{i_k} \circ \cdots \circ \pi_{i_1} \circ Approp_P(\mathtt{F}, p) = \pi_{i_0}$, then $\pi_{i_k} \circ \cdots \circ \pi_{i_1} \circ Approp_P(\mathtt{F}, q) = \pi_{i_0}$.*

Proof. If P is not parametrically separated, then there are non-simple parametric types, but either there is no greatest type that is non-simple (and, by persistence, no greatest simple type), or there is, called r, but there is also a non-simple type $p_1 \sqcup p_2$ that is join-reducible to non-simple types, p_1 and p_2, i.e., the non-simple parametric types of P are not totally ordered.

In the former case, suppose there are maximal non-simple types r_1 and r_2 and that the consequent of the lemma is false. Now consider $\pi_{i_k} \circ \cdots \circ \pi_{i_1} \circ Approp_P(\mathtt{F}, q)$ with that choice. It is not π_{i_0}, by assumption, but it must be parametrically determined, by definition, so its entire range belongs to $I_{\bar{q}}(P)$ for some $\bar{q}$. If $\bar{q}$ is not r_1 or r_2 then since $\pi_{i_k} \circ \cdots \circ \pi_{i_1} \circ Approp_P(\mathtt{F}, p) = \pi_{i_0}$, a choice of an instance in either $I_{r_1}(P)$ or $I_{r_2}(P)$ as the i_0^{th} parameter presents a contradiction, since right monotonicity would be violated. Similarly, if $\bar{q} = r_1$, then a choice of an instance in $I_{r_2}(P)$, or *vice versa*, presents the same contradiction.

In the latter case, the only way the same contradiction can be avoided is if $\bar{q} = r$, as every type subsumes r. By persistence, however, $\pi_{i_k} \circ \cdots \circ \pi_{i_1} \circ Approp_P(\mathtt{F}, q)$ must be a function that depends on the i_0^{th} parameter. The choice of an instance in $I_{p_1}(P)$ whose subparameter that subsumption-wise corresponds to the occurrence of the i_0^{th} parameter in $\pi_{i_k} \circ \cdots \circ \pi_{i_1} \circ Approp_P(\mathtt{F}, q)$ is an instance of $I_{p_2}(P)$ thus provides the same contradiction, since neither p_1 nor p_2 subsumes the other. □

Figures 5.12 and 5.13 show simple examples of each of these cases. How a parameter is reflected locally through value restrictions in subsumption chains is tightly constrained by the global shape of the type hierarchy as a result of the fact that every ground instance of every type can occur as any parameter.

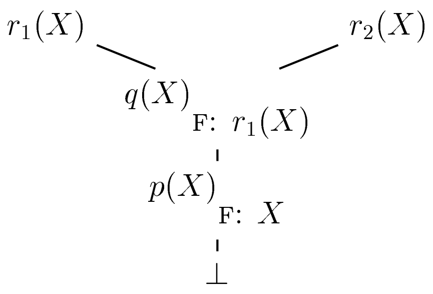

Figure 5.12: A would-be parametric signature with no greatest type that does not satisfy right monotonicity: $p(r_2(\perp)) \sqsubseteq q(r_2(\perp))$, but $r_2(\perp) \not\sqsubseteq r_1(r_2(\perp))$.

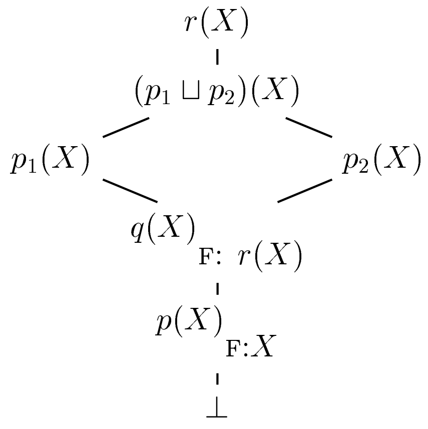

Figure 5.13: A would-be parametric signature whose parametric types are not totally ordered that does not satisfy right monotonicity: $p(p_1(p_2(\perp))) \sqsubseteq q(p_1(p_2(\perp)))$, but $p_1(p_2(\perp)) \not\sqsubseteq r(p_1(p_2(\perp)))$ because $p_2(\perp) \not\sqsubseteq p_1(p_2(\perp))$.

5.7 Summary

This chapter presented an account of parametric types that is general enough to capture their use in current linguistic theory. That account is made possible by essentially algebraic means, through formalizing the intuitive correspondence that must exist between parametric type signatures and the non-parametric signature of ground instances of parametric types. The structural conditions under which parametric types do in fact induce a well-formed BCPO were also given, which is another novel contribution. Linguists and implementors previously had no formal guidance as to how the sharing of parameters between a type and its subtypes or value restrictions should be regulated.

It was also proven that, in contrast to features, parametric type signatures do not provide any extra expressive power from a formal standpoint than non-parametric signatures. That question could be posed formally using signature subsumption as defined in Chapter 3. Parametric signatures are nevertheless a very natural and convenient means of expression. In spite of the fact that their induced equivalents are usually infinite, it was also shown that finite induced subsignatures can be induced in order to compute with them directly.

Chapter 6

Arity and Prolog Terms

We have already seen how join-preserving encodings, realized as signature subsumption and equivalence, can be used to relate different attributed type signatures to each other, to encode other kinds of signatures such as systemic networks, and to understand the relative expressive potential of attributed type signatures extended with parametric types. In this chapter, they will be used to show that the finite, inequation-free, totally well-typed feature structures of any finite signature with no cyclic types and a join-preserving appropriateness specification can be embedded into the semi-lattice of Prolog terms. When the target domain is first-order terms or Prolog terms, this embedding problem is called *term encoding.* As usual, we need to be concerned with join-preserving term encodings — those that preserve unification and unification failure. Prolog term encoding a particularly useful embedding, given the interest in logic programming among those who work with typed feature structures in the context of natural language processing. The practical application of this will be discussed at greater length in Chapter 8.

Ignoring for the moment the difference between named and positional reference to subterms, typed feature structures can be regarded as a refinement of Prolog terms in two ways. The first is that type signatures possess subsumption chains of any length. In Figure 6.1, for example, the chain from $\perp$ to *noun* is three types long (not counting $\perp$ itself). Once a Prolog variable, which corresponds to a feature structure of type $\perp$, is bound to a particular term, the principal functor of that term cannot be changed, and two terms of different principal functors cannot be unified.

The second is that, when the type of a feature structure promotes to a subtype, it may acquire more features. In Figure 6.1, this is the case

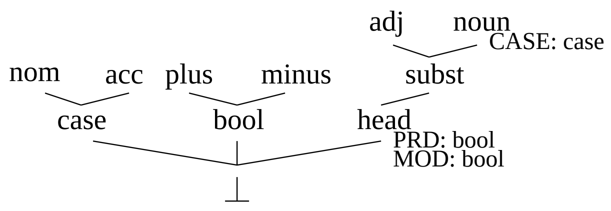

Figure 6.1: A sample tree-encodable type signature.

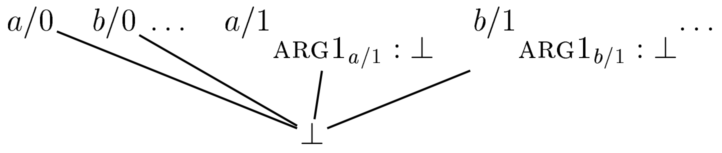

Figure 6.2: A "type signature" for Prolog terms.

when a feature structure of type *subst* promotes to type *noun*, since *noun* introduces the new feature CASE. Of course, with total well-typing, the arity of every feature structure of a particular type is fixed to a constant, namely the number of features appropriate to that type, but the arity can still change when the type promotes. In contrast, a Prolog term of a particular arity cannot later acquire a different arity, or unify with a Prolog term of a different arity. In this view, Prolog terms can be thought of as being defined over a very flat signature with infinitely many maximally specific types, as shown in Figure 6.2.

Subtyping and arity incrementation are the two main differences between totally well-typed feature structures and Prolog terms. In addition, Prolog is only a weakly typed language, so there is no way to check "well-typedness" after a unification is performed. As a result, only those signatures that do not require run-time coercion to a well-typed structure can have join-preserving Prolog-term encodings. These are the statically typable signatures, which were shown in Proposition 2.13 to correspond to signatures with join-preserving appropriateness specifications, as defined in Definition 2.38. It will be assumed here that appropriateness is join-preserving, feature structures are finite, and that there are no cyclic types, which guarantees that most general satisfiers will not fall outside the finiteness restriction. It may be possible to find join-preserving term encodings of certain cyclic types or of certain infinite feature structures over arbitrary signatures, e.g., those that depart from the most general satisfier of their type on finitely many nodes, by using some "lazy" encoding that explicitly expresses only a finite part of an infinite feature structure. It is also quite easy to extend the results presented here to feature structures with inequations in enhanced implementations of Prolog that support inequations, such as SICStus Prolog, so inequations will be ignored here as well.

The problem of encoding a typed feature structure is most easily approached by splitting it into two problems: a join-preserving encoding of type information that allows for subtypes, and a join-preserving encoding of feature values that allows for arity incrementation. The next two sections consider those problems, respectively. The key insight is proven in Lemma 6.1, in which it is shown that the definition of statically typable signatures given by Carpenter [1992] actually entails the existence of an essential extra property that makes the encoding possible.

The third section then adapts this encoding to be robust in the face of extra-logical variable bindings — these are what obstructed our potential use of signature transformations in Chapter 4 as well. No such robustness is possible given the classical view of join-preserving embeddings, but the generalized definition presented in Chapter 3 is the key to realizing that they actually exist. The fourth section then considers subsumption preservation as a special case of join preservation.

6.1 Subtyping

In this section, we consider finite type signatures without features — just finite type hierarchies. "Prolog encoding" is taken here to mean one in which the only operation necessary for feature structure unification after creating the encoding is Prolog unification of the corresponding terms. That excludes other "Prolog representations," such as the representation of feature structures in ALE, a logic programming language based on the logic of typed feature structures, which requires a dereferencing operation and table look-up at run-time [Carpenter and Penn, 1996], as well as the representation given by Gerdemann [1995b], which because of its slightly different interpretation of appropriateness conditions, requires the maintenance of extra constraints on the side in the worst case. The actual construction of any Prolog encoding can be performed at compile-time.

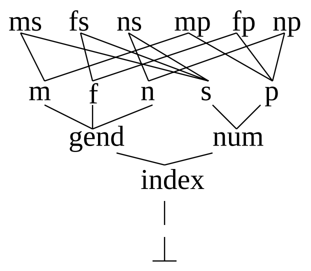

Figure 6.3: A type hierarchy with full-product multiple inheritance.

6.1.1 Tree Encodings

The first work to consider Prolog encodings of arbitrary meet semi-lattices was presented by Mellish [1991], although no general encoding algorithm was presented. Mellish [1991] was also the first to characterize the general encoding problem formally, as was presented in Chapter 3 as the "classical" definition (Definition 3.8 of join-preserving encoding. Previous work, dating back to that of Dahl [1977, 1982], concerned a restricted subset of semi-lattices that admit a Prolog *tree encoding*, i.e., an encoding by terms in which no variable is used more than once. These and other logical-term-encoding approaches are systematically presented in Fall, 1996.

Tree encodings represent a type with a term that, using subterms, represents the path(s) taken to reach that type from ⊥. In Figure 6.1, the representation of *head* would be `head(_)`; that of *subst*, `head(subst(_))`; and that of *noun*, `head(subst(noun))`. The `noun` subterm does not require a variable argument because *noun* is maximally specific.

Because the logic of typed feature structures is intensional, an extra variable argument is necessary to distinguish feature values that are variants from feature values that are extensionally identical when representing feature structures rather than just types. So, for feature structures with no features, the representation of one of type *head* would be `head(_,_)`; that of one of type *subst*, `head(subst(_),_)`; and that of one of type *noun*, `head(subst(noun),_)`. We can, for the rest of this section, ignore these extra arguments and focus on representing types.

To represent multiple inheritance, the tree-based encoding uses multiple argument positions to represent the different paths that lead to a single type. In Figure 6.3, any pair of *gend* and *num* subtypes can intersect. So *index*'s term contains two argument positions, one for *gend* and one for *num*. The

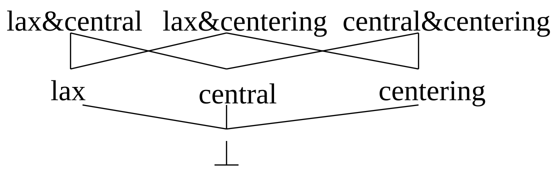

Figure 6.4: A type hierarchy with no tree encoding.

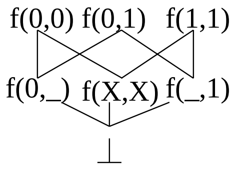

Figure 6.5: A flat-term encoding of Figure 6.4.

representations of *gend*, *s* and *ms* are `index(gend(_),_)`, `index(_,num(s))` and `index(gend(m),num(s))`, respectively.

Tree-based encoding does not work for arbitrary finite meet semi-lattices, as proven by Mellish [1991, 1992]. Figure 6.4, for example, represents a simple classification of vowels, taken from Hudson [1981] (and cited by Mellish [1991]). There are vowels that have any pair of the three properties, *lax*, *central* and *centering*; but there are no vowels that have all three at once. This has no tree encoding because separate argument positions for *lax*, *central* and *centering* would entail the consistency of *lax* with *central&centering*, for example.

6.1.2 Flat-Term Encodings

Mellish [1991, 1992] proved (non-constructively) that while not all finite meet semi-lattices admit tree encodings, they do all admit *flat-term encodings*, encodings that use terms whose substructures are all reachable from the root by a path of length at most 1. While this encoding can instantiate arguments only to constants, it can also use individual variables in more than one argument position. A flat-term encoding of Figure 6.4 is given in Figure 6.5.

Mellish [1988] showed that what is now known as *Colmerauer's method* can be used to encode systemic networks using flat terms of arity $n + 1$, where n is the number of possible property assignments allowed for by the

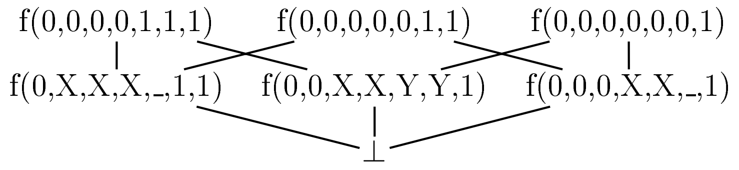

Figure 6.6: A Colmerauer-method encoding of Figure 6.4.

network. The first argument is always 0, the last argument is always 1, and every assignment is represented by a pair of arguments in between. Every description that excludes an assignment numbered i is represented by a term whose ith and $i+1$st arguments are bound to a common term. If there are 5 possible assignments, then we use a term of arity 6 as follows [Mellish, 1988]:

```
         1       2       3       4       5
         |       |       |       |       |
f(0,     _,      _,      X,      X,      1)   (excludes assignment 4)
f(0,     0,      _,      X,      X,      1)   (excludes 1 and 4)
f(0,     X,      X,      X,      1,      1)   (excludes 2, 3, and 5)
```

Fall [1996, p. 94] observed that Colmerauer's method applied to subsumption rather than assignments provides a unification-preserving encoding of arbitrary finite ordered sets, thus establishing a constructive method for flat-term encodings. A pair of arguments in an encoding is bound iff the type corresponding to the pair is not a subtype of the type being encoded. Figure 6.6 shows a Colmerauer-method encoding of the signature in Figure 6.4 with the following assignment of types to pairs of positions:

```
          lax      ctrl      ctrg      lax&      lax&      ctrl&
                                       ctrl      ctrg      ctrg
           |        |         |         |         |         |
f(0,       _,       _,        _,        _,        _,           1)
```

6.2 Arity Incrementation

As a first approximation to the encoding of typed feature structures over a finite signature, we can assume that we are given Colmerauer-style encodings of all of the types in the signature. We can then use this in an extra argument position of a term that also has one argument position for every feature in the signature, to be filled with the term encoding of its value. Cyclic feature structures correspond to circular Prolog terms.

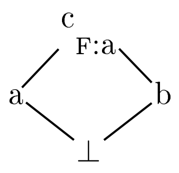

Figure 6.7: A signature that introduces a feature at a join-reducible type.

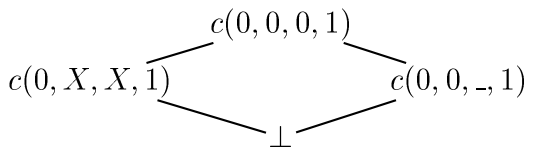

Figure 6.8: A Colmerauer encoding of the signature in Figure 6.7.

The problem with this approximation is that it does not always lead to a join-preserving encoding because features can be introduced at join-reducible types. Figure 6.7 depicts a signature for which this is the case. A Colmerauer encoding of it is shown in Figure 6.8. We can add one argument position to the term encoding of c for the value of the feature F; but then we need to make the encodings of a and b the same arity so that they unify. The detail that the approximation does not provide is what value to use in that extra position with types for which F is not appropriate. If we use a singleton variable, i.e., a variable that occurs exactly once in its term, as shown in Figure 6.9, then the encoding of a feature structure of type a unified with the encoding of a feature structure of type b yields an encoding whose F argument position does not contain a term encoding of the value restriction of F at c. On the other hand, we could use a term encoding of the most general satisfier of $\mathit{Approp}(\text{F}, \mathit{Intro}(\text{F}))$, with the understanding that the type information in the encoding will tell us when to interpret it as a real value or just to ignore it. But in Figure 6.7, that value restriction is a, one of the supertypes of c, and thus the term encoding of the most general satisfier of a

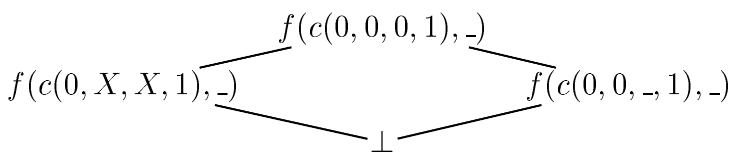

Figure 6.9: An approximate encoding of Figure 6.7 using a singleton variable for inappropriate feature positions.

would be an infinite non-circular term, since its third argument would be the term encoding of a distinct alphabetic variant of the same feature structure.

To handle recursive signatures such as these, we must adopt a convention that allows an additional argument position for a feature to contain a variable, but only in certain cases. The type information in the encoding will tell us whether that variable should be interpreted as a non-existent or introduced value. One admissible convention is given below:

Definition 6.1. *Given a (statically typable etc.) signature, $S = \langle T, \sqsubseteq, Feat, Approp\rangle$, and a Colmerauer encoding of T of arity $|T| + 1$, the* classical term encoding of *the totally well-typed feature structures of S is an injective function, $\bar{\cdot} : S \longrightarrow \bar{S}$, where $\bar{S}$ is a partially ordered set of Prolog terms of arity $|Feat| + 2$. Given $F = \langle Q, \bar{q}, \theta, \delta, \emptyset\rangle \in \mathcal{TTF}_S$ with type $\theta(\bar{q}) = t \in T$, and Q finite, its encoding is $\bar{F} = f(c(C_1, \ldots, C_{|T|+1}), \bar{F}_1, \ldots, \bar{F}_{|Feat|}, \_) \in \bar{S}$, where $c(C_1, \ldots, C_{|T|+1})$ is the Colmerauer encoding of t, and:*

$$\bar{F}_i = \begin{cases} \textit{a singleton variable} & \textit{if} \quad Approp(\mathrm{F}_i, t)\uparrow \\ \textit{a singleton variable} & \textit{if } F@\mathrm{F}_i = \langle Q^i, \bar{q}^i, \theta^i, \delta^i, \not\leftrightarrow^i\rangle \sim \\ & MGSat(Approp(\mathrm{F}_i, Intro(\mathrm{F}_i))) \\ & \textit{and } \forall q \in Q. \forall \mathrm{G} \in Feat.(\delta(\mathrm{G}, q) = \\ & \bar{q}^i) \Rightarrow (\mathrm{G} = \mathrm{F}_i) \& (q = \bar{q}), \\ \overline{F@\mathrm{F}_i} & \textit{otherwise} \end{cases}$$

This says that we can use a variable as a place-holder for some feature value, provided that it is the most general feature structure that can be a value of that feature, and it does not participate in a re-entrancy. This exploits the fact that appropriateness cannot require re-entrancies to exist[1] — an introduced feature value is never re-entrant.

In practice, of course, one can remove the extra c wrapper on Colmerauer's encoding to leave the structure of the term as flat as possible, and simply use a variable by itself to represent feature structures of type $\perp$.

Proposition 6.1. *$\bar{\cdot}$ is a classical join-preserving encoding of $\mathcal{TTF}_S$.*

Proof. By definition, $\bar{\cdot}$ is injective, which is possible because the last argument is always a singleton variable, posited to ensure the intensionality of the terms in the encoding, as described above. Because Colmerauer's encoding is zero-preserving and join-preserving, substructures are encoded in

[1]in the absence of extensional types.

subterm positions, re-entrancies are encoded as shared subterms in Prolog terms (thinking of them, too, as graphs), and the unification of singleton variables in unused term positions always succeeds, the only special cases that need to be considered for zero-preservation and join-preservation are arity incrementation, i.e., when a feature is introduced at a join, and the unification of the distinguished singleton variables with other values.

To consider the latter case first, the most general satisfier of a feature's value restriction at its introducer must subsume any other value that that feature can take in a well-typed feature structure, by upward closure and right monotonicity. So we should always get a non-variable term encoding back when we unify it with a singleton variable encoding in that position, which is exactly what happens.

Feature introduction also never fails — this is a result of the fact that *Fill* (Definition 2.36) is a total function. It also never fails in the encoding — singleton variables unify with anything. The only question is whether the right value is introduced when the type of a term changes so as to change the interpretation of the introduced feature's position. From the original definition of join preservation (Definition 2.38), it is not quite clear that this would be the case, because of the "unrestricted" clause. Lemma 6.1 establishes that this actually is true. □

Lemma 6.1. *If Approp is join-preserving,* $s \sqcup t\downarrow$*, and for some* $\textsc{f} \in \mathit{Feat}$*,* $\mathit{Approp}(\textsc{f}, s)\uparrow$ *and* $\mathit{Approp}(\textsc{f}, t)\uparrow$*, then either* $\mathit{Approp}(\textsc{f}, s \sqcup t)\uparrow$ *or* $\mathit{Approp}(\textsc{f}, s \sqcup t) = \mathit{Approp}(\textsc{f}, \mathit{Intro}(\textsc{f}))$*.*

Proof. Suppose $\mathit{Approp}(\textsc{f}, s \sqcup t)\downarrow$. Then $\mathit{Intro}(\textsc{f}) \sqsubseteq s \sqcup t$. $\mathit{Approp}(\textsc{f}, s)\uparrow$ and $\mathit{Approp}(\textsc{f}, t)\uparrow$, so $\mathit{Intro}(\textsc{f}) \not\sqsubseteq s$ and $\mathit{Intro}(\textsc{f}) \not\sqsubseteq t$. So there are three cases to consider:

$\boldsymbol{\mathit{Intro}}(\textsc{f}) = \mathbf{s} \sqcup \mathbf{t}$: then the result trivially holds.

$\mathbf{s} \sqsubseteq \boldsymbol{\mathit{Intro}}(\textsc{f})$ **but** $\mathbf{t} \not\sqsubseteq \boldsymbol{\mathit{Intro}}(\textsc{f})$ (or by symmetry, the opposite): then we have the situation in Figure 6.10. It must be that $\mathit{Intro}(\textsc{f}) \sqcup t = s \sqcup t$, so by join preservation, the lemma holds.

$\mathbf{s} \not\sqsubseteq \boldsymbol{\mathit{Intro}}(\textsc{f})$ **and** $\mathbf{t} \not\sqsubseteq \boldsymbol{\mathit{Intro}}(\textsc{f})$: $s \sqsubseteq s \sqcup t$ and $\mathit{Intro}(\textsc{f}) \sqsubseteq s \sqcup t$, so s and $\mathit{Intro}(\textsc{f})$ are consistent. By bounded completeness, $s \sqcup \mathit{Intro}(\textsc{f})\downarrow$ and $s \sqcup \mathit{Intro}(\textsc{f}) \sqsubseteq s \sqcup t$. By upward closure, $\mathit{Approp}(\textsc{f}, \mathit{Intro}(\textsc{f}) \sqcup s)\downarrow$ and by join preservation, $\mathit{Approp}(\textsc{f}, \mathit{Intro}(\textsc{f}) \sqcup s) = \mathit{Approp}(\textsc{f}, \mathit{Intro}(\textsc{f}))$. Furthermore, $(\mathit{Intro}(\textsc{f}) \sqcup s) \sqcup t = s \sqcup t$; thus by join preservation, the lemma holds. □

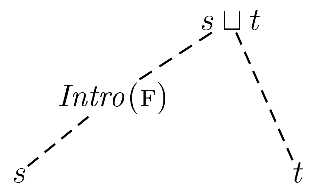

Figure 6.10: The second case in the proof of Lemma 6.1.

This lemma is a very significant result — it says that we can always predict what an introduced feature's value restriction will be in a join-preserving signature. This means that join preservation not only characterizes static typability, but also fixed-arity-term encodability in the logic of typed feature structures. We can, thus, restate join preservation as follows:

Definition 6.2. *An appropriateness specification is said to* preserve joins *iff, for all features $f \in Feat$, for all types s, t such that $s \sqcup t\downarrow$:*

$$Approp(\text{F}, s \sqcup t) = \begin{cases} Approp(\text{F}, s) \sqcup Approp(\text{F}, t) & \textit{if } Approp(\text{F}, s)\downarrow \textit{ and} \\ & Approp(\text{F}, t)\downarrow \\ Approp(\text{F}, s) & \textit{if only } Approp(\text{F}, s)\downarrow \\ Approp(\text{F}, t) & \textit{if only } Approp(\text{F}, t)\downarrow \\ \begin{cases} \textit{undefined, or} \\ Approp(\text{F}, Intro(\text{F})) \end{cases} & \textit{otherwise} \end{cases}$$

No matter which case pertains, appropriateness is never completely unrestricted if it is join-preserving.

The encoding and lemma make critical use of bounded completeness and unique feature introduction. Actually, the encoding also works if we generalize our definition of signatures to allow for multiple introducing types, provided that all of them agree on what the value restriction for a multiply introduced feature should be. Would-be signatures that multiply introduce a feature at join-reducible elements (thus requiring some kind of variable encoding), disagree on the value restriction, and still remain statically typable are rather difficult to come by, but they do exist, and for them, this encoding does not work. Figure 6.11 shows one such example. In this signature, the unification:

$$\begin{bmatrix} \text{s} & \\ \text{F} & \text{d} \end{bmatrix} \sqcup \begin{bmatrix} \text{t} & \\ \text{F} & \text{b} \end{bmatrix} \uparrow$$

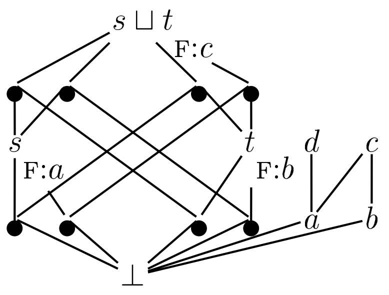

Figure 6.11: A statically typable would-be signature that multiply introduces F at join-reducible elements with different value restrictions.

does not exist, but the unification of their term encodings must succeed because the t-typed structure's F value must be encoded as a variable. To the best of the author's knowledge, there is no term encoding that can handle this generalization.

6.3 Generalized Term Encoding

A classical term encoding of typed feature structures exists, subject to the restrictions outlined at the beginning of this chapter, but in practice, it is not good for much. Programming languages that make reference to a feature structure, F, typically need to bind variables to various substructures of F, and then pass those variables outside the scope of F where they can be used to instantiate the value of another feature structure's feature, or as arguments to some function call or procedural goal. This extra-logical view of relational extensions of constraint languages has been rather commonplace in logic programming ever since it was proposed by Höhfeld and Smolka [1988]. If a subterm in an encoding is a singleton variable, we can properly understand what that variable encodes by looking at its context, i.e., the term's type etc., but outside the scope of that term, we have no way of knowing which type's most general satisfier it is supposed to encode.

A generalized term encoding provides an elegant solution to this problem without a loss of encoding ability in the form of additional, more verbose term encodings for certain feature structures. When a variable is bound to a substructure that is potentially a "lazy" singleton variable, it can be instantiated to the most general satisfier that it represents and passed out of context. The encoding of the original feature structure still remains legitimate, because the encoding sets are closed under the binding of singleton variables.

Definition 6.3. *Given a signature, $S = \langle T, \sqsubseteq, Feat, Approp\rangle$, and a Colmerauer encoding of T of arity $|T|+1$, the* term encoding of *the totally well-typed feature structures of S is a function, $\hat{\cdot} : S \longrightarrow Pow(\hat{S})$, where $\hat{S}$ is a partially ordered set of Prolog terms of arity $|Feat|+2$. Given $F = \langle Q, \bar{q}, \theta, \delta, \emptyset\rangle \in \mathcal{TTF}_S$ with type $\theta(\bar{q}) = t \in T$, and Q finite, its encoding is a set of terms, each of the form $\hat{F} = f(c(C_1, \ldots, C_{|T|+1}), \hat{F}_1, \ldots, \hat{F}_{|Feat|}, \_) \in \hat{S}$, where $c(C_1, \ldots, C_{|T|+1})$ is the Colmerauer encoding of t, and:*

$$\hat{F}_i = \begin{cases} \textit{a singleton variable} & \textit{if}\quad Approp(\mathrm{F}_i, t)\uparrow \\ \textit{a singleton variable,} & \textit{if } F@\mathrm{F}_i = \langle Q^i, \bar{q}^i, \theta^i, \delta^i, \not\leftrightarrow^i\rangle \sim \\ \textit{or } G^i & MGSat(Approp(\mathrm{F}_i, Intro(\mathrm{F}_i))) \\ & \textit{and } \forall q \in Q. \forall \mathrm{G} \in Feat.(\delta(\mathrm{G}, q) = \\ & \bar{q}^i) \Rightarrow (\mathrm{G} = \mathrm{F}_i)\&(q = \bar{q}), \\ G^i & \textit{otherwise} \end{cases}$$

where, for each $\bar{q}^i$, G^i is a unique term selected from $\widehat{F@\mathrm{F}_i}$.

Proposition 6.2. *$\hat{\cdot}$ is a join-preserving encoding of $\mathcal{TTF}_S$.*

Proof. Totality and disjointness are obvious. For those positions where a singleton variable is chosen, zero preservation and join preservation follow by the same reasoning as in the classical case. The classical encoding, in fact, is one member of the generalized encoding set for every feature structure. For those positions where an instantiated encoding of the most general satisfier is chosen, zero preservation and join preservation again follow from the straightforward correspondence between Prolog term structure and feature structure graph structure and the fact that Colmerauer's encoding (which, by itself, is still classical) is zero-preserving and join-preserving. □

6.4 Subsumption Preservation

Whenever an encoding is join preserving, it is subsumption, or order preserving because $F \sqcup G = G$ iff $F \sqsubseteq G$. But we might also expect to be able to test for subsumption in an encoding domain using a primitive subsumption test for that domain without having to perform a more expensive test unification. This does not hold in general, and, in particular, it does not hold for either the classical or generalized Prolog encodings of typed feature structures presented in the last two sections. The reason is that singleton variable placeholders in G might actually represent substructures that are more specific than an instantiated term that encodes a corresponding substructure in F, which is consistent with $F \sqsubseteq G$, but inconsistent with $\bar{F} \sqsubseteq \bar{G}$.

In the case of the classical encoding, we have no recourse to repair this problem — only one term corresponds to a given feature structure, and primitive subsumption testing using that term does not work. In the case of the generalized encoding, we can coerce a naughty term to another one that encodes the same feature structure but has filled in its value-encoding singleton variables.[2]

Definition 6.4. *Given a signature, S, let Exp : $\hat{S} \longrightarrow \hat{S}$ be the function that, for every $F \in \mathcal{TTF}_S$, maps every encoding in $\hat{F}$ to the unique term in $\hat{F}$ that has no value-encoding singleton variables.*

Proposition 6.3. *Suppose the primitive for subsumption testing among Prolog terms is called `subsumes`. Then for every $F, G \in \mathcal{TTF}_S$, $\hat{f} \in \hat{F}$, and $\hat{g} \in \hat{G}$, $F \sqsubseteq G$ iff $\hat{f}$ `subsumes` $Exp(\hat{g})$.*

A schematic overview of the generalized term encoding can be seen in Figure 6.12. Every set of terms that encode a particular feature structure has a least element, in which singleton variables are always opted for as introduced feature values. This is the same element as the classical encoding. It also has a greatest element, namely the result of *Exp*, which eliminates the variable encodings of introduced feature values. Whenever we bind a variable to a substructure, we push its encoding up within the same set to some other encoding. As a result, at any given point in time during a computation, we do not exactly know which encoding we are using to represent a given feature structure. Furthermore, when two feature structures are unified successfully, we do not know exactly what the result will be either, but we do know that it falls inside the set corresponding to the correct answer because there is always a term there with variable encodings for the values of any newly introduced features.

[2] The extra singleton variable for preserving intensionality, of course, remains a singleton variable.

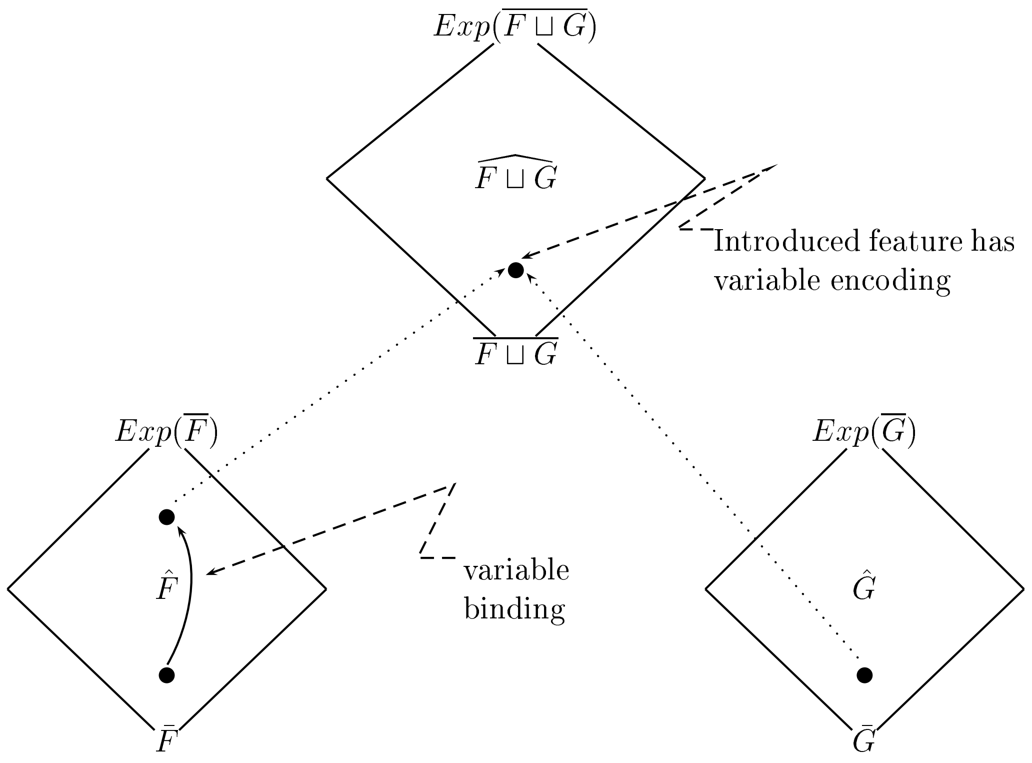

Figure 6.12: A pictorial overview of the generalized encoding.

6.5 Summary

This chapter showed that two kinds of join-preserving embeddings, one classical and one generalized, exist from any statically typable signature into the lattice of Prolog terms. The generalized embedding has the ability to withstand the necessary extra inferencing in order to use variables with an extra-logical binding scope for the purposes of logic programming, for example. The crucial step in proving their existence is Lemma 6.1, which shows that the assumption of unique feature introduction allows us to strengthen the characterization of static typability given by Carpenter [1992]. As shown in Chapter 2, this assumption can easily be restored when it is not assumed by the designer. It was also shown that subsumption preservation can be reduced to subsumption at the Prolog level using an extra closure operator, *Exp*, that can apply directly to encoding terms.

Chapter 7

The Semi-Ring Structure of Signature Specifications

Before proceeding further, it will be useful to take a different look at attributed type signatures, this time focussing on how the relations and functions that constitute signatures can be viewed as the closure of specifications of signatures. This knowledge has already been implicitly used in the way that signatures have been depicted — as graphs whose links correspond to instances of immediate subsumption, annotated with feature introduction and value restriction information only where it cannot be inferred from upward closure and/or right monotonicity. Fundamentally, what separates these specifications from the signatures themselves is a collection of *transitive closure* operations, plus various safeguards to ensure that the specifications are well-formed. Subsumption, appropriateness, transitive closure, and these safeguards can all be thought of in terms of matrices and matrix arithmetic.

The reduction of such closures to efficient operations over matrices has a wider application as well. For any programming language that aspires to support efficient object inheritance or an inclusionally polymorphic type system, two very common and important operations are type inference and the computation of least upper bounds. These can occur both during compilation to ensure the static typability of a program, or at run-time in the form of unification. Very broadly speaking, given a partially ordered set of elements, three ways have been proposed to compute encodings of those elements in order to conduct unification efficiently: table lookup, term encodings, and bit-vector encodings. Term encodings attempt to reduce unification in the object domain into unification in some other domain, such as unification of

Prolog terms [Mellish, 1988, 1992], or of sparse encodings of first-order terms [Fall, 1996]. Bit-vector encodings typically attempt to reduce unification to one or more bit-wise operations, such as AND or OR.

Inheritance among feature structures/terms is derived both from their structural properties, particularly the consistency of their shared substructures with type information, and from explicit declarations of subsumption relationships that exist among types in the type system. In practice, both of these are only indirectly defined. In the case of structural properties, this is achieved through *appropriateness conditions*, which locally enforce structural well-formedness conditions between subterms of different types. In particular, they specify which types of subterms can bear certain attributes, and what types the values of those attributes can be. The closure of those local constraints over the type system constitutes overall structural well-formedness.

In the case of types, the explicit declarations are made by way of declaring only an immediate subsumption relation over the set of types, whose transitive closure constitutes the type subsumption relation. All three methods of efficient unification thus involve computing closures at compile-time over both named and structural relations to find the implied algebraic structure underlying typed feature structures/terms. This chapter provides a uniform way of looking at these closures, particularly as they apply to programming over the logic of typed feature structures. In particular, all of the compilation required for signatures relative to structural and named well-formedness constraints in the logic of Carpenter [1992] can be reduced to matrix arithmetic.

The next section reviews the reduction of compiling unification to the transitive closure of Boolean matrices. Section 7.2 discusses the mathematical structure that must exist among the elements of these matrices for the proposed method to work, argues that the closed Boolean semi-ring is the proper one, and discusses some of the practical consequences of this choice. Section 7.3 shows how to construct a closed semi-ring that supports matrix multiplication from any finite meet semi-lattice (Definition 7.6). Section 7.4 then reduces all of the compilation steps necessary for efficient processing relative to an attributed type signature to matrix operations based on this construction.

In this chapter, only finite signatures will be considered.

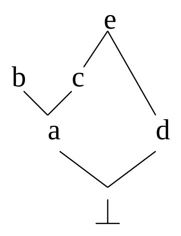

Figure 7.1: An example type hierarchy.

7.1 Subsumption Matrices and Transitive Closure

The use of Boolean matrix multiplication to compute transitive closures of graphs extends as far back as Prosser, 1959; and was improved on by Warshall's famous transitive closure algorithm [Warshall, 1962].

The application of reflexive-transitive closure to lattice representation theory extends back to the seminal paper by Aït-Kaći et al. [1989], who proposed the baseline bit-vector encoding of partially ordered types by which all others are now measured, namely one that uses n bits per code, where n is the number of types in the partial order.

Definition 7.1. *Given a finite partially ordered set, $\langle P, \sqsubseteq\rangle$, and a total ordering of P's elements, $p_1, p_2, \ldots, p_{|P|}$, the* subsumption matrix*, S, of P is a $|P| \times |P|$ Boolean matrix, where $S_{i,j} = 1$ iff $p_i \sqsubseteq p_j$.*

The crucial observation of Aït-Kaći et al. [1989] was that we can use the ith row of S to encode the type p_i, with unification corresponding to bit-wise AND. An example partial order of types is given in Figure 7.1. Here, $\perp$ is the most general type, and more specific subtypes are situated above their more general supertypes. Its subsumption matrix is given in Figure 7.2. The AND of the rows for a and d yields the row for e, for example.

We can build a base subsumption matrix, H, in the same way, by using the base, or immediate, subsumption relation rather than true subsumption. In practice, this is what is specified in type hierarchy declarations, with reflexive and transitive closure being implicit. The question then becomes how to obtain S from H. The base subsumption matrix for Figure 7.1 is given in Figure 7.3.

Two refinements are given in the same paper to produce more compact encodings. One allocates bits only for maximal types and unary branch-

$$\begin{array}{ccccccc}
 & \bot & \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{d} & \mathrm{e} \\
\bot & 1 & 1 & 1 & 1 & 1 & 1 \\
\mathrm{a} & 0 & 1 & 1 & 1 & 0 & 1 \\
\mathrm{b} & 0 & 0 & 1 & 0 & 0 & 0 \\
\mathrm{c} & 0 & 0 & 0 & 1 & 0 & 1 \\
\mathrm{d} & 0 & 0 & 0 & 0 & 1 & 1 \\
\mathrm{e} & 0 & 0 & 0 & 0 & 0 & 1
\end{array}$$

Figure 7.2: The subsumption matrix of Figure 7.1.

$$\begin{array}{ccccccc}
 & \bot & \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{d} & \mathrm{e} \\
\bot & 0 & 1 & 0 & 0 & 1 & 0 \\
\mathrm{a} & 0 & 0 & 1 & 1 & 0 & 0 \\
\mathrm{b} & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathrm{c} & 0 & 0 & 0 & 0 & 0 & 1 \\
\mathrm{d} & 0 & 0 & 0 & 0 & 0 & 1 \\
\mathrm{e} & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Figure 7.3: The base subsumption matrix of Figure 7.1.

ing types. The other allocates separate group codes for subsets that are "modular" in the way that base subsumption connects them to the rest of the network, which reduces the overall size of codes, although it makes the actual unification operation more complicated than bit-wise AND. Ganguly et al. [1994] provide another good encoding that places the burden of extra bits on types that inherit from multiple supertypes, to achieve a comparable improvement without the modularity restriction. These and others are summarized by Fall [1996]. In every case, however, these refinements are simply allocating bits more sparingly along the way to deriving a reflexively-transitively closed matrix like S from H.

For the naive encoding, one way to achieve that derivation is a reflexive-transitive closure, by directly filling in the diagonal of H with 1's (reflexive) and multiplying the result by itself until it reaches the fixed point, S (transitive). This fixed point is obviously reached after no more than $|P|$ iterations. By re-using the results of previous multiplications, one can attain it in $\lceil \log |P| \rceil$ iterations. The first refinement presented by Aït-Kaći et al. [1989] also uses matrix multiplication at one step to compute the reflexive-transitive closure of the symmetric closure of H. Aït-Kaći et al. [1989] claim that those matrix multiplication steps should be conducted in the Boolean ring of $|P| \times |P|$ bit-matrices, and that they can be performed using an efficient sub-cubic algorithm such as Strassen's algorithm [Strassen, 1969]. The correctness of this claim is considered below.

7.2 Rings, Quasi-Rings and Semi-Rings

There are actually two closely related Boolean algebras with two operations each (roughly speaking, candidates for ring-hood). They are $B_{XOR} = \langle\{0,1\}, XOR, AND, 0, 1\rangle$, where $\oplus$ corresponds to XOR, and $B_{OR} = \langle\{0,1\}, OR, AND, 0, 1\rangle$, where $\oplus$ corresponds to OR. In order to see how these differ in practice, we need to define some basic structures:

Definition 7.2. *A* monoid *is a structure $\langle P, \cdot, e\rangle$ such that:*

- *P is a set closed under $\cdot$,*
- *$\cdot$ is an associative binary operator on P, and*
- *$e \in P$ is an identity for $\cdot$.*

Definition 7.3. *A* quasi-ring *is a structure $\langle P, \oplus, \otimes, \bar{0}, \bar{1}\rangle$, such that:*

- *$\langle P, \oplus, \bar{0}\rangle$ is a monoid,*
- *$\langle P, \otimes, \bar{1}\rangle$ is a monoid,*
- *$\bar{0}$ is an annihilator of $\otimes$: $a \otimes \bar{0} = \bar{0}$ for all $a \in P$,*
- *$\oplus$ is commutative, and*
- *$\otimes$ distributes over $\oplus$: $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(b \oplus c) \otimes a = (b \otimes a) \oplus (c \otimes a)$, for all $a, b, c \in P$.*

Definition 7.4. *A* ring *is a quasi-ring with an additive inverse, i.e., for all $a \in P$, there exists $b \in P$ such that $a \oplus b = b \oplus a = \bar{0}$.*

If P is a quasi-ring, then multiplication of matrices is well-defined and has certain nice properties, such as associativity and the existence of an identity (in fact, it is also a quasi-ring).

Definition 7.5. *Given a quasi-ring,* $Q = \langle P, \oplus, \otimes, \bar{0}, \bar{1} \rangle$*, an* $m \times n$ *matrix,* A*, over* Q*, and an* $n \times p$ *matrix,* B*, over* Q*, then* $A \cdot B$ (matrix multiplication) *is the* $m \times p$ *matrix,* C*, over* Q *such that:*

$$c_{i,j} = \bigoplus_{k=1}^{n} a_{i,k} \otimes b_{k,j}.$$

B_{XOR} and B_{OR} are both Boolean quasi-rings.

B_{XOR} is also a Boolean ring; but B_{OR} is not. B_{OR} is a closed Boolean *semi-ring*, however, which, among other properties, means that OR is idempotent, i.e., that $1 \oplus 1 = 1$. This is vital for ensuring that matrix multiplication can compute a transitive closure, since transitively closed subsumption should not be "turned off" by immediate subsumption chains on more than one subtyping branch. As a result, we need idempotence in the underlying Boolean quasi-ring. XOR is not idempotent; so the Boolean ring is not the correct structure to use.

In addition, Strassen's algorithm can only compute matrix multiplication over true rings — not over all quasi-rings or closed semi-rings — because it requires the existence of an additive inverse; so Strassen's algorithm will not work with B_{OR}.

We could embed matrices over the Boolean quasi-ring, B_{OR}, into a proper ring, and find some way of restoring the result in the original structure. Such an embedding can be found in the integers, for example, as shown in Figure 7.4. This would allow us to use Strassen's algorithm. Presumably,

$$\begin{tikzcd} Z \arrow[r] & Z \arrow[d, "\begin{cases} 0 \mapsto 0, \\ x \mapsto 1\ o.w. \end{cases}"] \\ B_{OR} \arrow[u, "\iota"] & B_{OR} \end{tikzcd}$$

Figure 7.4: An embedding of B_{OR} into Z for ring multiplication.

this is what systems that purport to use either the naive encoding or its first refinement in Aït-Kaći et al., 1989 have actually been doing.

Even then, Strassen's algorithm belongs to a class of sub-cubic matrix multiplication algorithms that are only well-suited to multiplying very large, very dense matrices. While subsumption matrices can be very large, they are never very dense. Because they encode a partial order (or some approximation of its transitive closure), for every non-diagonal 1, corresponding to

$p \sqsubseteq q$, for $p \neq q$, there is a non-diagonal 0, corresponding to $q \not\sqsubseteq p$. As a result, no more than $(|P|^2 + |P|)/2$ positions are non-zero. So even if we use the above embedding to rescue Boolean ring-hood, Strassen's algorithm is still the wrong algorithm for the job. Either Warshall's classical algorithm should be used,[1] or specialized sparse matrix multiplication algorithms could be applied, most of which only require an underlying quasi-ring.

Warshall's algorithm and Floyd's extension of it to the all-pairs-shortest-path problem are both instances of a general dynamic programming algorithm on closed semi-rings [Aho et al., 1974]. Aït-Kaći et al. [1989] also present a non-standard algorithm for transitive closure by way of introducing their first refinement that (apparently unwittingly) uses B_{OR} rather than the Boolean ring. To the present author's knowledge, the fastest known general multiplication algorithm for matrices over quasi-rings is still $O(n^3)$.

By contrast, nearly every algorithm in the class of sub-cubic algorithms to which Strassen's algorithm belongs requires an underlying ring. The only exception of which the present author is aware is Shamir's randomized Boolean matrix multiplication algorithm [Cormen et al., 1990], which can conduct multiplications for matrices over B_{OR} using matrices over B_{XOR} with a probability of at least $1 - 1/n^k$ for any constant $k > 0$ in $O(n^{lg7} lgn)$ time. The price, of course, is the small chance of error. The fastest known "sure-fire" multiplication algorithm for matrices over proper rings is $O(n^{2.376})$ [Coppersmith and Winograd, 1990].

On the other hand, the rows and columns of H and S can be sorted so that they are upper-triangular sparse matrices. In particular, it can be shown [Aho et al., 1974] that within any closed semi-ring, the transitive closure of an upper-triangular matrix is:

$$\begin{pmatrix} A & B \\ 0 & C \end{pmatrix}^* = \begin{pmatrix} A^* & A^*BC^* \\ 0 & C^* \end{pmatrix}$$

for any square sub-matrix, A. Using the right-hand side to compute the transitive closure cuts the number of required arithmetic operations as a function of the dimension of A, although it has the same asymptotic complexity. First-order calculus shows that this cut attains a maximum of 75%, i.e., the computation takes 25% of the time required by the left-hand side,

[1] Warshall [1962] claims his algorithm increases "slightly faster" than quadratically; but it is known to be tightly bounded at cubic. Although there is no discussion of the choice of algebra, it also uses B_{OR}.

when the dimension of A is half of that of the overall matrix. A compatible sparse matrix representation may further reduce that number.

A preliminary evaluation, shown in Table 7.1, suggests that these observations can improve compilation times on large signatures by a factor of 800 or more over naive transitive closure algorithms and by up to a factor of 5000 over closure by optimized matrix multiplication algorithms such as Strassen's algorithm. The table shows results on two type hierarchies, one with 162 types from the "naive HPSG" grammar distributed with the ALE system, and one with 2763 from the LinGO project at Stanford University [LinGO, 1999]. All measurements are in seconds, and were made on

	HPSG(162)		LinGO(2763)
Strassen	41.45	sec.	huge
Strassen-32	1.21		10370.50
Strassen-64	1.07		7856.65
Naive-Z	0.35		5099.38
Naive-BOR	0.15		4368.01
Warshall	0.12		1667.37
Naive-Z*	0.08		1126.75
Naive-BOR*	0.03		707.68
Sparse-Z	<0.01		9.70
Sparse-BOR	<0.01		8.31
Sparse-BOR*	???		???

Table 7.1: Preliminary comparison of transitive closure algorithms on two type hierarchies.

a dual-450-MHz Pentium II with 1 GB of RAM running Redhat Linux 2.2. Strassen-32 (Strassen-64) is the version of Strassen's algorithm that switches to the naive multiplication algorithm on matrices of dimension 32 (64) or less, which is how Strassen's algorithm is used in practice. All variations of Strassen's algorithm are from GEMMW, the level 3 BLAS library routine that implements the Winograd variant of Strassen's algorithm in Fortran [Douglas et al., 1994]. Warshall is a standard Prolog library routine for Warshall's algorithm. All other programs were written in C by the present author. The algorithms marked with asterisks use the upper-triangular decomposition for closed semi-rings mentioned above, and as can be seen, the only non-sparse algorithms that surpass the Prolog routine are those that use the decomposition. The algorithm in the last entry has yet to be devised. While its details are an open research problem, judging from the other improvements using the upper-triangular decomposition, it should be possible to stay fairly close to the 75% reduction to achieve a performance of around two seconds on the LinGO type hierarchy.

The first refinement for computing "compact" bit-vector codes in Aït-Kaći et al., 1989 is effectively a sparse, or at least sparser, matrix encoding of S suitable for the component-wise multiplication (AND) of its rows. In particular, The observation of Aït-Kaći et al. [1989] in their first refinement can be restated as: every column corresponding to a meet-reducible type, $t = u \sqcap v$, where $u \neq t$ and $v \neq t$, can be reconstructed by component-wise multiplying the columns corresponding to u and v. Given rows i and j, $s_{i,t} \otimes s_{j,t}$ is thus $(s_{i,u} \otimes s_{i,v}) \otimes (s_{j,u} \otimes s_{j,v})$. If the underlying quasi-ring is commutative, as is the case for B_{OR}, this equals $(s_{i,u} \otimes s_{j,u}) \otimes (s_{i,v} \otimes s_{j,v})$, the product of the columns for u and v in the result. So an encoding that must preserve only component-wise multiplication of rows of S can dispense with the columns corresponding to meet-reducible types altogether. What remain are the meet-irreducible types, which are exactly the maximal types and unary-branching types in the partial orders that Aït-Kaći et al. [1989] consider.

7.3 An Extensible Quasi-Ring Construction

Now that we know that we need a quasi-ring with idempotence, we can consider which kinds of quasi-rings would be most convenient. The Boolean quasi-ring, B_{OR}, suffices for processing with simple type hierarchies, but since we are interested in totally well-typed feature structures, appropriateness conditions should also be taken into account. Only the correlates of the naive encoding of Aït-Kaći et al. [1989] will be explicitly considered here — similar improvements for compactness and speed can be made on these as well.

Our type hierarchies are (for now, finite) meet semi-lattices. Requiring type hierarchies to be finite meet semi-lattices effectively eliminates a potential source of disjunction inherent to unification in general partial orders. Bit-vector encodings capture disjunctions of types for free (as the OR of the disjuncts), but in more restricted feature logics such as ours, those disjunctions may make it difficult to articulate appropriateness conditions, and practically speaking, delay their enforcement, which exists to prune ill-formed structures. The reason for this is that individual disjuncts may have different types and therefore different appropriateness conditions — different appropriate features, for example. Instead of simply disjoining the types, the appropriateness conditions common to all of the disjuncts should be factored out, effectively creating a new meet in the type hierarchy, in a manner similar to that discussed in Chapter 2, section 2.1.8.

We can think of 0 and 1 in B_{OR} as constituting a very small type hierarchy, as shown in Figure 7.5. If $\top$ corresponds to 1, and $\bot$, to 0, then unification

$$\begin{tikzcd}
\top \arrow[d, dash] \\
\bot
\end{tikzcd}$$

Figure 7.5: The Boolean type hierarchy.

in this hierarchy corresponds to Boolean OR. We can also write this as in Figure 7.6, in which the trivial type hierarchy, consisting of just $\bot$, has been

$$\begin{tikzcd}
\bot \arrow[d, dash] \\
\underline{\bot}
\end{tikzcd}$$

Figure 7.6: The trivial type hierarchy lifted to produce the Boolean hierarchy.

bottom-lifted to add a new bottom, $\underline{\bot}$. In fact, we can do this to *any* type hierarchy. Because bottom-lifting preserves meet-semi-latticehood, we can trivially extend $\sqcup$ to any $P \cup \{\underline{\bot}\}$ where P is a finite meet semi-lattice. Now, we need something to correspond to AND:

$$a \mathbin{\underline{\sqcup}} b = \begin{cases} \underline{\bot} & \text{if } a = \underline{\bot} \text{ or } b = \underline{\bot} \\ a \sqcup b & \text{otherwise} \end{cases}$$

Definition 7.6. *Let $\langle P, \sqsubseteq \rangle$ be a finite type hierarchy. Then $Q(P) = \langle P \cup \{\underline{\bot}\}, \sqcup, \underline{\sqcup}, \underline{\bot}, \bot \rangle$, is the* quasi-ring induced by *P.*

Notice that we can define this for all P, not just the trivial type hierarchy, because in all type hierarchies in Carpenter, 1992, $\sqcup$, and therefore its extension to $P \cup \{\underline{\bot}\}$ and to $\underline{\sqcup}$, are total functions, and there is a least element. As can easily be verified:

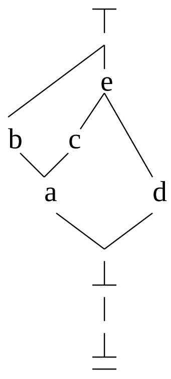

Figure 7.7: The quasi-ring constructed from Figure 7.1.

Proposition 7.1. *For all finite type hierarchies P with a greatest element, $Q(P)$ is a quasi-ring.*

The existence of a greatest element ensures that $\sqcup$ and $\underline{\sqcup}$ are closed in $P \cup \{\underline{\bot}\}$. Without loss of generality, we can assume that the greatest element, $\top$, does not explicitly appear in P, and that it does not occur anywhere else in the signature, e.g., in appropriateness conditions. $\top$ can be smashed onto any such P, and is typically implemented as type unification failure in the original signature. Figure 7.7 shows the type hierarchy in Figure 7.1 $\underline{\bot}$-lifted and $\top$-smashed to form its quasi-ring.

The benefit of using $Q(P)$ is that it allows us to generalize to other computations on signatures that require matrices with types in them rather than just 0s and 1s. The subsumption matrix of P can still be constructed using $\underline{\bot}$ and $\bot$ in place of 0 and 1, respectively. The next section presents a way of looking at all of the other closure operations and sanity checks that must hold of valid subsumption and appropriateness specifications in terms of matrices and vectors over this quasi-ring construction.

7.4 Compiling Type and Appropriateness Restrictions

7.4.1 Subtyping Cycles

The first two checks we need to make still use only $\underline{\bot}$ and $\bot$, i.e., the former 0 and 1. Unification ($\sqcup$) over all of $Q(P)$ is not necessarily well-defined until we can guarantee that P is a meet semi-lattice.

We first need a way of checking that a subsumption specification we are given is legitimate. We can assume that we are given a candidate type hierarchy in the form of its base subsumption matrix, H. We then reflexively and transitively close H as described above, to obtain its subsumption matrix, S. The first check we need to make is that P is a partial order, by checking for anti-symmetry.

Proposition 7.2. *P is not a partial order iff there exist $1 \leq i, j \leq |P|$, $i \neq j$ such that $S_{i,j} = S_{j,i} = \perp$ (the 1 element).*

An easier way to check this is to carry out the construction of S by embedding H as a matrix $\bar{H}$ in Z, as in Figure 7.4, and building its transitive closure, $\bar{S}$. The embedding method described above can still be used because H and S still contain only $\underline{\perp}$ and $\perp$.

Proposition 7.3. *P is not a partial order iff for any $1 \leq i \leq |P|$, $\bar{S}_{i,i} > 1$.*

Proof. A diagonal entry of greater than 1 indicates that that entry could have been set to 1 without an explicit reflexive closure by transitively closing over a pre-order with a symmetry. □

Once we have S, it is easy to express which types are consistent:

Definition 7.7. *The* join matrix *of P is $J = S \cdot S^T$, where S^T is the transpose of S.*

Proposition 7.4. *t_i and t_j are consistent iff $J_{i,j} = \perp$.*

Note that J is always symmetric, i.e., $J_{i,j} = J_{j,i}$ for all i,j.

7.4.2 Meet Semi-latticehood

We can check for meet-semi-latticehood by exploiting the alternative definition of meet semi-lattices in the finite case and using the rows of S as codes of types in P, in the manner of Aït-Kaći et al. [1989]:

Proposition 7.5. *Let $\vec{\underline{\sqcup}}$ be the component-wise application of $\underline{\sqcup}$ to two vectors of elements from $Q(P)$. A partial order P is a finite meet semi-lattice iff for all $1 \leq i, j \leq |P|$, if $J_{i,j} = \perp$ then there exists a (unique) $1 \leq join(i, j) \leq |P|$ such that $S_i \vec{\underline{\sqcup}} S_j = S_{join(i,j)}$.*

If two types are consistent, then their codes intersect to produce the code of another type. If P is not a finite meet semi-lattice, then the unification of some codes will be the disjunction of two or more other codes, which will not be found as a single row in S.

Along the way to verifying this, we can build the unification table for P:

Definition 7.8. *The* unification table *of P, U, is the join matrix of P, with each $\underline{\perp}$ replaced by $\top$, and each $\perp$ replaced by $t_{join(i,j)}$.*

Proposition 7.6. *For all $1 \leq i, j \leq |P|$, $U_{i,j} = t_i \sqcup t_j$.*

That S_k in Proposition 7.5 corresponds to the unification of S_i and S_j follows from the correctness of the method of Aït-Kaći et al. [1989], proven in their paper.

7.4.3 Feature Introduction

The definition of appropriateness is repeated here for convenience:

Definition 7.9. *Given a type hierarchy, $\langle T, \sqsubseteq \rangle$, and a finite set of features, Feat, an* appropriateness specification *is a partial function, $Approp : Feat \times T \longrightarrow T$ such that, for every $\text{F} \in Feat$:*

- ***(Feature Introduction)*** *there is a type $Intro(\text{F}) \in T$ such that:*
  - *$Approp(\text{F}, Intro(\text{F}))\downarrow$, and*
  - *for every $t \in T$, if $Approp(\text{F}, t)\downarrow$, then $Intro(\text{F}) \sqsubseteq t$, and*
- ***(Upward Closure / Right Monotonicity)*** *if $Approp(\text{F}, s)\downarrow$ and $s \sqsubseteq t$, then $Approp(\text{F}, t)\downarrow$ and $Approp(\text{F}, s) \sqsubseteq Approp(\text{F}, t)$.*

As mentioned above, we only expect signature specifications to specify appropriateness only by declaring (1) where a feature is introduced, along with its value restriction and (2) where a feature's value restriction cannot be inferred to be the least type that satisfies Upward Closure and/or Right Monotonicity given its value restriction on supertypes. Figure 7.8, for example, is Figure 7.1 with appropriateness declarations added. F is appropriate to a, for example, with value restriction, $\perp$. Because F is appropriate to a, it is also appropriate to b, c and e, although b refines the value restriction to c. b has two appropriate features because it also introduces G. e has two appropriate features by Upward Closure because H was introduced at d.

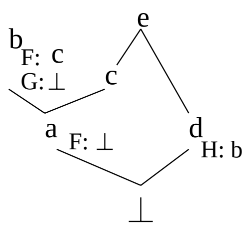

Figure 7.8: An example type signature.

	F	G	H
$\perp$	$\underline{\perp}$	$\underline{\perp}$	$\underline{\perp}$
a	$\perp$	$\underline{\perp}$	$\underline{\perp}$
b	c	$\perp$	$\underline{\perp}$
c	$\underline{\perp}$	$\underline{\perp}$	$\underline{\perp}$
d	$\underline{\perp}$	$\underline{\perp}$	b
e	$\underline{\perp}$	$\underline{\perp}$	$\underline{\perp}$

Figure 7.9: The value declaration matrix of Figure 7.8.

In order to enforce this view of appropriateness conditions on P, we can build a matrix over $Q(P)$ for these declarations:

Definition 7.10. *Given a finite set of types, P, a set of features, F, and a set of appropriateness declarations $D \subseteq F \times P \times P$, where D is a partial function, the* value declaration matrix *for D over F and P is a $|P| \times |F|$ matrix, V, over $Q(P)$, in which $V_{i,j} = u$, if there exists a $u \in P$ such that $(f_j, t_i, u) \in D$, and $V_{i,j} = \underline{\perp}$ if there is no such u.*

The uniqueness of u, when it exists, is guaranteed by the fact that D is a partial function. The value declaration matrix for Figure 7.8 is shown in Figure 7.9. The entry for type d, feature H is b because H is declared as appropriate to d with its value restricted to b.

Notice that $\underline{\perp}$ is being used here as a place-holder for pairs of type, t and feature, F, for which F is not appropriate to t. We use $\underline{\perp}$ rather than $\top$ so that feature introduction (with a value restriction of $\perp$ or greater) still respects Right Monotonicity. This trick has applications outside type signature compilation to any task that requires a uniform view of type-feature pairs or accessible feature paths while still respecting appropriateness conditions with respect to unification, since all features are effectively appropriate to all types, albeit sometimes with $\underline{\perp}$ as the value.

$$\begin{array}{cccc}
 & \mathrm{F} & \mathrm{G} & \mathrm{H} \\
\perp & \underline{\perp} & \underline{\perp} & \underline{\perp} \\
\mathrm{a} & \perp & \underline{\perp} & \underline{\perp} \\
\mathrm{b} & \mathrm{c} & \perp & \underline{\perp} \\
\mathrm{c} & \perp & \underline{\perp} & \underline{\perp} \\
\mathrm{d} & \underline{\perp} & \underline{\perp} & \mathrm{b} \\
\mathrm{e} & \perp & \underline{\perp} & \mathrm{b}
\end{array}$$

Figure 7.10: The value restriction matrix of Figure 7.8.

Definition 7.11. *The* value restriction matrix *of P is $R = S^T \cdot V$.*

Pre-multiplying V by the transpose of S closes the appropriateness declarations under subsumption. $V_{i,j}$ is thus something other than $\underline{\perp}$ iff feature j is appropriate to type i. The value restriction matrix of Figure 7.8 is shown in Figure 7.10. Notice that the entry for type e, feature H is also b because e inherits H from d. Using the value restriction matrix, we can then express the condition on unique feature introduction:

Definition 7.12. *$\delta : P \cup \{\underline{\perp}\} \to \{\underline{\perp}, \perp\}$ is the* characteristic function *for P, such that:*

$$\delta(t) = \begin{cases} \perp & \text{if } t \in P, \\ \underline{\perp} & \text{if } t = \underline{\perp} \end{cases}$$

Proposition 7.7. *R satisfies the feature introduction restriction iff for all i, there exists an $intro(i)$, such that $\vec{\delta}(R_i^T) = S_{intro(i)}$.*

δ projects the elements of $Q(P)$ back onto the trivial quasi-ring, according to whether they belong to P. Feature introduction is satisfied iff, after component-wise projection, every column of R is the same as some row of S. Rows of S encode types as the upward closed sets that they subsume. The columns of R have non-$\underline{\perp}$ values for the types to which a feature, F, is appropriate; and we know that that set is upward-closed, having left-multiplied by S^T. If that set is one of the rows of S, then that row corresponds to a minimal type, which is $Intro$(F).

Along the way, we can also find those introducing types:

Definition 7.13. *The* introduction matrix, *I, of P is a $|P| \times |F|$ matrix in which:*

$$I_{j,i} = \begin{cases} V_{j,i} & \text{if } j = intro(i), \\ \underline{\perp} & \text{elsewhere} \end{cases}$$

$$\begin{array}{cccc}
 & \text{F} & \text{G} & \text{H} \\
\perp & \underline{\perp} & \underline{\perp} & \underline{\perp} \\
\text{a} & \perp & \underline{\perp} & \underline{\perp} \\
\text{b} & \underline{\perp} & \perp & \underline{\perp} \\
\text{c} & \underline{\perp} & \underline{\perp} & \underline{\perp} \\
\text{d} & \underline{\perp} & \underline{\perp} & \text{b} \\
\text{e} & \underline{\perp} & \underline{\perp} & \underline{\perp}
\end{array}$$

Figure 7.11: The introduction matrix of Figure 7.8.

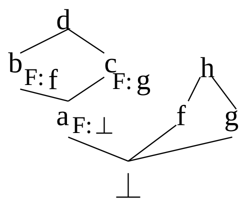

Figure 7.12: A type signature with consistent value restrictions.

The introduction matrix for Figure 7.8 is given in Figure 7.11. The entry for type *b*, feature F is $\underline{\perp}$ because, although *b* places a non-inferrable value restriction on F, it does not introduce F.

7.4.4 Value Restriction Consistency

Because of Right Monotonicity, join-reducible types can not only multiply inherit features, but also inherit value restrictions on the same feature from two or more different branches; and these must be consistent. Figure 7.12 shows an example of this. Right Monotonicity from *b* and *c* requires F to be appropriate to *d* with a value of both *f* and *g*. In Figure 7.12, this is consistent — the value of F at *d* must be of type *h*. Without *h*, it would not be consistent. We can use value restriction matrices to express this consistency check as well.

Proposition 7.8. *The value restrictions of P are consistent iff there is no i,j for which $R_{i,j} = \top$.*

$\top$ corresponds to inconsistency in the original signature.

The value declaration matrix for Figure 7.12 is given in Figure 7.13. The value restriction matrix of Figure 7.12 is given in Figure 7.14. Without *h*, the entry for *d* would have been $\top$.

$$\begin{array}{cc} & \mathrm{F} \\ \bot & \underline{\bot} \\ \mathrm{a} & \bot \\ \mathrm{b} & \mathrm{f} \\ \mathrm{c} & \mathrm{g} \\ \mathrm{d} & \underline{\bot} \\ \mathrm{f} & \underline{\bot} \\ \mathrm{g} & \underline{\bot} \\ \mathrm{h} & \underline{\bot} \end{array}$$

Figure 7.13: The value declaration matrix of Figure 7.12.

$$\begin{array}{cc} & \mathrm{F} \\ \bot & \underline{\bot} \\ \mathrm{a} & \bot \\ \mathrm{b} & \mathrm{f} \\ \mathrm{c} & \mathrm{g} \\ \mathrm{d} & \mathrm{h} \\ \mathrm{f} & \underline{\bot} \\ \mathrm{g} & \underline{\bot} \\ \mathrm{h} & \underline{\bot} \end{array}$$

Figure 7.14: The value restriction matrix of Figure 7.12.

$$
\begin{array}{c|cccccccc}
 & \perp & \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{d} & \mathrm{f} & \mathrm{g} & \mathrm{h} \\
\perp & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathrm{a} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathrm{b} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\mathrm{c} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\mathrm{d} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\mathrm{f} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathrm{g} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathrm{h} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

Figure 7.15: The convolution of Figure 7.14.

7.4.5 Appropriateness Cycles

As seen in Chapter 4, it is also often useful to require that all types have a finite most general satisfier, a finite least informative feature structure of that type that respects appropriateness conditions. This means that appropriateness conditions may not conspire so as to require a feature structure of type t to have a proper substructure of type either t or a subtype of t.This kind of appropriateness cycle can very naturally be articulated using R.

Definition 7.14. *The* convolution matrix, *C, of R is a $|P| \times |P|$ matrix over $Q(P)$ such that $C_{i,j} = \perp$ if there exists a k such that $R_{i,k} = t_j$, and $C_{i,j} = \underline{\perp}$ otherwise.*

Proposition 7.9. *P has an appropriateness cycle iff there exists an i such that $C^*_{i,i} = \perp$, where C^* is the (non-reflexive) transitive closure of C.*

$C_{i,j} = \perp$ means that type t_j is accessible as a substructure by some feature from structures of type t_i. By transitively closing C, we extend that accessibility to finite paths of features, and so can detect whether t_i is accessible from t_i. Because we convoluted C from R, which was upward closed by left-multiplication with S^T, we detect accessibility to subtypes of t_i as well. The convolution of Figure 7.14 is given in Figure 7.15 (0 and 1 are used for readability). The entry for row b, column f is $\perp$, because f is accessible from b along the feature F. In this case, the transitive closure of C is the same as C itself, because all types that occur as value restrictions of features are atomic, i.e., they have no features of their own.

In practice, this transitive closure can also be computed directly from R, without explicitly constructing C.

7.4.6 Join Preservation Condition

It can also be useful to check whether the join preservation condition is observed. This guarantees that a signature is statically typable, and, as seen in Chapter 6, can guarantee the existence of certain term encodings.

The (revised) definition of join preservation is repeated here.

Definition 7.15. *An appropriateness specification is said to* preserve joins *iff, for all features $f \in \mathit{Feat}$, for all types s, t such that $s \sqcup t\!\downarrow$:*

$$\mathit{Approp}(\textsc{f}, s \sqcup t) = \begin{cases} \mathit{Approp}(\textsc{f}, s) \sqcup \mathit{Approp}(\textsc{f}, t) & \text{if } \mathit{Approp}(\textsc{f}, s)\!\downarrow \text{ and} \\ & \mathit{Approp}(\textsc{f}, t)\!\downarrow \\ \mathit{Approp}(\textsc{f}, s) & \text{if only } \mathit{Approp}(\textsc{f}, s)\!\downarrow \\ \mathit{Approp}(\textsc{f}, t) & \text{if only } \mathit{Approp}(\textsc{f}, t)\!\downarrow \\ \begin{cases} \text{undefined, or} \\ \mathit{Approp}(\textsc{f}, \mathit{Intro}(\textsc{f})) \end{cases} & \text{otherwise} \end{cases}$$

Proposition 7.10. *$\langle P, \sqsubseteq, F, A\rangle$ satisfies the join preservation condition iff for all i, j for which $J_{i,j} = \bot$, and $U_{i,j} = t_k$, $R_i \vec{\sqcup} R_j \vec{\sqcup} (S^T I)_k = R_k$.*

Viewed in terms of R, join preservation is a linear dependence condition among consistent types — recall that in $Q(P)$, $\sqcup$ corresponds to the additive operator. In a join-preserving signature, joins cannot add new information to the system, apart from introducing new features with the value restrictions of their introducer.

7.5 Summary

This chapter presented a new closed-semi-ring construction over which all type and feature restrictions inherent to signature specifications can be computed and enforced. Along the way, the mathematical foundations of closure computations based on matrix multiplication have been clarified, as have the consequences of those foundations on the choice of algorithms for efficiently computing those closures. The induced closed-semi-ring construction also has applicability for handling computations in feature logics for which it is not the case that all features necessarily occur on all types, as it provides a uniform set of feature paths up to any finitely bounded length, that can be used for transparent classification or for indexing feature structures with appropriateness, for example.

The new construction can also serve as the basis for efficiently precompiling type signature information. What is needed now is the development of efficient sparse matrix multiplication methods that are particularly well-suited to specifications of partial orders or upward closed relations on partial orders, particularly one that allows for the efficient upper-triangular decomposition of matrices given in Section 7.2 for the case of computing subsumption matrices.

Chapter 8

Practical Prolog Term Encoding of Typed Feature Structures

In Chapter 6, it was observed that total well-typing plus unique feature introduction allows us to encode typed feature structures as Prolog terms. The encoding given there was just a proof of existence, and certainly could not be construed as a practical way of computing with typed feature structures because of the potentially large term sizes, i.e., arities, involved. Both the classical and generalized term encodings called for arities on the order of the number of types plus the number of features.

Appropriateness also allows us to encode typed feature structures as Prolog terms more efficiently. As described in Chapter 6, arity incrementation and subtyping are the two major differences between typed feature structures and Prolog terms. They are also the two major sources of complexity when encoding typed feature structures as Prolog terms. This chapter presents a collection of methods to address both sources by reducing the former to a graph coloring problem, and by presenting for the latter what is, to the author's knowledge, the first method for finding an optimal flat first-order-term encoding of any finite meet semi-lattice of types, and an approximate solution that can be derived in cubic time in the number of types from a transitively closed adjacency representation of the semi-lattice. These are presented in Sections 8.2 and 8.1, respectively. An empirical comparison of these to previous work as well as other encodings that are available using the extra functionality provided by the `library(atts)` library of SICStus

Prolog is also presented.

On a practical level, the value of Prolog term encoding stems from the value of using Prolog itself. Most systems based on typed feature structures rely on a few standard means of search in order to solve problems in natural language processing — some subset of Prolog-like SLD resolution, parsing and content-driven generation. All of these have been extensively investigated within Prolog; and, in the case of SLD resolution, current commercial implementations of Prolog have benefited from a sixteen year history of optimizations to Warren Abstract Machine (WAM) compilers [Aït-Kaći, 1991], the standard for Prolog compilation. There have been a few very WAM-like abstract machines proposed directly for typed feature structures, e.g., by Wintner [1997], Wintner and Francez [1994] and Carpenter and Qu [1995], implemented as described by Makino et al. [1998]. These inevitably rely on a recapitulation of that history for their own optimization. With the arguable exception of the term unification operation itself, any additional innovations made in the course of their development are probably better applied to (Prolog) WAM optimization, given the nearly identical requirements of the two communities and research programs. Adhering to an implementation based on Prolog term encodings also provides immediate access to the extended functionality that commercial Prologs provide, including constraint solving and constraint logic programming. These are in great demand in all areas of knowledge representation, including computational linguistics. If it could be achieved and achieved efficiently enough, clearly a Prolog-term-encoding-based implementation would be preferable, simply from the perspective of rapid development and efficient reuse of previous research.

The empirical results presented in this chapter suggest that the widely presumed futility of this endeavor is not at all beyond question. Relative to the small number of realistic, large-scale grammars currently available to the author, a Prolog-term-encoding-based implementation of a logic programming language over typed feature structures has performed remarkably well in comparison not only to the previous generation of non-term-encoding-based Prolog meta-interpreters but even to the fastest of the current implementations based on customized abstract machines for feature structures. It stands to reason, of course, that customized abstract machines must be capable of better performance in the limit, given a sufficiently large supply of political, financial and human resources. With such application-oriented research, however, the question really is not how much better it is, but whether it was already good enough, and whether what remains to be achieved constitutes a relevant research problem. The admittedly contentious tenet of the discussion in Section 8.3 is that the theoretically interesting problem of finding efficient heuristics for near-optimal flat term encodings and better non-flat term encodings overall is more worthwhile at this stage.

The approach to arity incrementation presented here, however, would be equally useful to custom-abstract-machine-based approaches; and the present approach to subtyping essentially brings Prolog-based implementations in line with the bit-vector encodings for type unification used in many abstract machines stemming from the influential paper [Aït-Kaći et al., 1989] on that subject.

8.1 Subtyping

To date, only two other systems have attempted to use Prolog encodings of typed feature structures. The algorithm used in both produces tree-encodings, as discussed in Section 6.1.1, and was actually designed [Mellish, 1988] as a general encoding algorithm for systemic networks. All systemic networks are tree-encodable because the "multiple inheritance" that they effectively provide is always a full product of two or more sub-networks. ProFIT [Erbach, 1994, 1995, 1996] first adapted this encoding to a very restricted subset of finite type hierarchies expressed in a more traditional ISA-link form. The ALEP system [Simpkins and Groenendijk, 1994] essentially carried over the same encoding for its typed-feature-structure-like records from ProFIT.

Mellish's [1988] encoding exploits the restricted inheritance provided by systemic networks to tree-encode them, and therefore cannot handle all finite BCPOs. ProFIT dealt with this limitation simply by restricting its multiple inheritance to exclude all of the counter-examples. The result is multi-dimensional inheritance, which was introduced in Section 4.1.1. Mellish's encoding works for all type hierarchies defined by multi-dimensional inheritance.

The development of flat-term encodings pursued here attempts to improve on tree encodings in terms of both coverage and speed. With regard to coverage, it has already been observed that flat-term encodings can handle arbitrary finite BCPOs (Section 6.1.2). With regard to speed, there is also some reason to suspect that flat-term encodings might be superior to tree-term encodings. Flat-term encodings, of course, are flat, i.e., their substructures are all reachable from the root by a path of length at most 1. Because first-order terms are of fixed arities, unification of a variable with a flat term can be compiled into a primitive recursive loop that compilers can statically unwind. Unification of nested compound terms, however, typically involves some amount of pointer chasing on the heap before the actual addresses can be passed to the unifier.

Thus, with all other things being equal, "broader beats deeper," i.e., two flat terms with n arguments can be unified more quickly than two terms of arity one, with subterms nested n levels deep. All other things are not equal, of course. In particular, tree encodings allow for smaller encodings of more general types, i.e., types that are closer to the root of the "tree." A tree encoding of a finite type hierarchy has terms whose depths are bounded by the maximum length of any subtype chain in the hierarchy, although with (limited) multiple or multi-dimensional inheritance, more than one path of that depth may be created; and in the case of true multiple inheritance (in the cases where the encoding works), some duplication of structure may be necessary at the ends of the paths. Flat-term encodings do not require redundant structure, but, in the *best* case, have arities equal to the length of the maximum subtype chain in the hierarchy — multiple inheritance can force it to be wider. As a result, in cases where both are applicable, it is a strictly empirical question as to whether the speed-up from flatness outweighs the fact that very general types in a tree encoding have smaller term sizes.

Colmerauer's method, introduced in Section 6.1.2, is a flat-term encoding. In the context of its original usage, namely systemic networks, the number of possible assignments of properties is, in the worst case, exponential in the number of properties in the systemic network, as is thus the arity. As a result, this method is not practical for encoding systemic networks — it can yield terms with exponentially large arities.

Fall [1996, p. 94] observed that Colmerauer's method could be used for unification-preserving encodings of arbitrary finite ordered sets, but mistakenly assumed that the method yielded exponential-sized terms for that task as well. It is exponential for systemic networks because systemic networks can, in some cases, provide exponentially compact encodings of possible assignments. For the case of an enumerated set of assignments, the method yields linear-sized terms. Colmerauer, in fact, had originally used his method for computing arbitrary set intersections. ProFIT used this encoding in the same spirit for finite domains, essentially distinguished flat finite type hierarchies with no appropriate features, but not for its type hierarchies. We

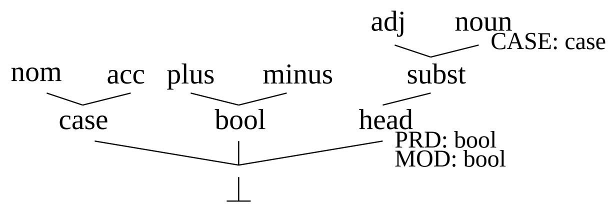

Figure 8.1: A sample tree-encodable type signature.

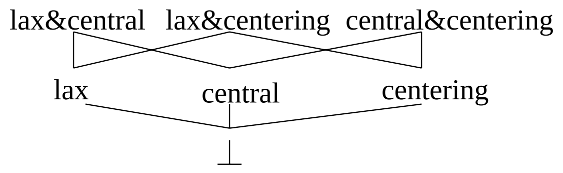

Figure 8.2: A type hierarchy with no tree encoding.

now consider a method that makes optimal use of Colmerauer's method with meet semi-lattices, a sub-case of ordered sets.

8.1.1 Modules

The first observation we can make is that meet semi-lattices can be decomposed into *modules*, pieces whose types can never unify:

Definition 8.1. *Given a finite type hierarchy, $\langle P, \sqsubseteq \rangle$, the set of* modules *of $\langle P, \sqsubseteq \rangle$ is the finest partition of $P \backslash \{\bot\}$, $M_1, \ldots M_m$, such that:*

1. *each M_i is upward-closed (w.r.t subsumption), and*

2. *if t_i and t_j are unifiable, then they belong to the same module*

Figure 8.1, for example, has three modules, one each rooted at *case*, *bool*, and *head*. In general, a module might not have a unique least type. Figure 8.2 has one module, for example. Modularizing in this fashion can be performed as the first step to any encoding strategy, because the modules can be encoded separately. The definition of module above can be generalized in order to develop hybrid tree-based / flat-term-based encodings as well, although this will not be considered further here.

8.1.2 Method 1: Colmerauer's method for meet semi-lattices

Without loss of generality, we can now consider only finite type hierarchies that have one module with no least type. As in a tree encoding, $\perp$ can be represented by a Prolog variable; and if there is a least type in a module, then it can be represented by a term with a unique variable in every argument, i.e., the most general term of that functor and arity.

Proposition 8.1. *The meet irreducible types of a finite type hierarchy are precisely the maximally specific types and the types with one immediate subtype.*

Proposition 8.2. *Every finite type hierarchy has a flat-term encoding with an arity equal to the number of meet irreducible types plus 1.*

Proof. Use Colmerauer's method on the set of meet irreducible types.[1] Every type in the meet semi-lattice can be represented by the set of meet irreducible types that it subsumes. Maximally specific types only unify with themselves. All other types can then be characterized by the maximally specific types that they subsume except the types with one immediate subtype, which would be characterized by the same maximally specific subtypes as that immediate subtype. Those types are meet irreducible as well, however, and so are distinguished by their own occurrence in the representation. ☐

One of the algorithms in Aït-Kaći et al., 1989 achieves essentially the same encoding, but for bit vectors. Finding the set of meet irreducible types takes at most cubic time — the time it takes to test all u and v for meets given a transitively closed adjacency representation of subsumption.

This method yields an optimal encoding in the sense that no encoding that uses Colmerauer's method can result in terms with a smaller arity than the number of meet irreducible types plus 1 [Fall, 1996]. Arity is the relevant measure since extra argument constants in the encoding come comparatively cheaply, and multiple occurrences of individual variables do not make unification slower. In general, however, this method does not yield an optimal flat-term encoding overall. Figure 8.3 shows a binary tree module, which has no join reducible types. No binary trees have types with only one immediate

[1] The reader may also recall the discussion at the end of Section 7.2 in this context. An appeal is made there to the same reasoning, but in the context of bit-vector encodings.

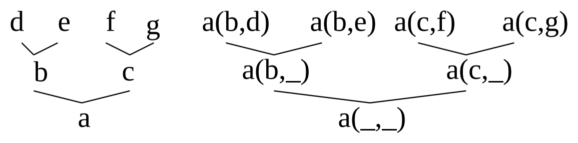

Figure 8.3: A binary tree and its optimal flat-term encoding.

subtype, so the number of meet irreducible elements is the number of maximally specific types, or half the total number of types. It can be proven, however, that the flat-term encoding of smallest arity for a binary tree, in fact for any module without join reducible types, is equal to the length of its longest subtype chain, which for binary trees, is equal to the logarithm of the total number of types. That encoding is shown in Figure 8.3.

What is the smallest arity required for arbitrary meet semi-lattices?

8.1.3 Method 2: Parametrized Search for an Optimal Encoding

A flat-term encoding is constructed from a choice of functor for the term (which is irrelevant, provided every module has one unique functor), a choice of arity of the term, and a choice of constants to instantiate some of the arguments of the term.

The set of first-order terms can be characterized as a meet semi-lattice in its own right, called the lattice of Generalized Atomic Formulae (GAF, Reynolds, 1970). We only need the sublattice of flat terms, GAF_1; and for a given module, only those flat terms of the same principal functor (which can remain implicit) and arity, a, with argument constants, 0 through some k, $GAF^a_{1,k}$. This is finite. Subsumption in this sublattice should be familiar to anyone who has used Prolog — in $GAF^2_{1,1}$, for example $f(\_,\_)$ is the most general term, $f(\_,1) \sqsubseteq_{GAF} f(0,1)$ and $f(1,1)$, $f(1,\_) \sqsubseteq_{GAF} f(1,0)$ and $f(1,1)$, and the term with the same variable in both positions, $f(X,X)$, subsumes terms with the same constant in those positions, i.e., $f(0,0)$ and $f(1,1)$.[2]

The fact that *every* finite meet semi-lattice has a Colmerauer-style encoding means that our choice of arity in a flat-term encoding never needs to be greater than the number of meet irreducible types of a module plus 1. There are other constraints on arity that can be proven as well:

[2]We will not consider terms $f(X,Y)$ with $X \neq Y$, although Prologs with inequations would allow us to use these in encodings as well.

Definition 8.2. *Given finite type hierarchy, $\langle P, \sqsubseteq \rangle$ and type $t \in P$, the* information level *of t, $\delta_P(t)$, is the length of the longest subtype chain from $\perp$ to t.*

This is exactly the same function defined as *path length* earlier (Definition 2.8), where it was used to guide inductive proofs on well-founded type hierarchies. Here, more attention will be paid to its actual value relative to other types in the same hierarchy.

We can implicitly order our types into a sequence t_i, such that if $i < j$ then $\delta_P(t_i) \leq \delta_P(t_j)$, with $\perp$ as t_1. This is equivalent to topologically sorting the type hierarchy as a directed acyclic graph. We can also extend δ to flat-term encodings, thinking of them as meet semi-lattices, $GAF^a_{1,k}$.

Proposition 8.3. *Let $\bar{P}$ be a flat-term encoding of P and $\bar{t}$ be the term corresponding to $t \in P$ in $\bar{P}$. Then:*

1. *For any t, $\delta_{\bar{P}}(\bar{t}) \geq \delta_P(t)$.*
2. *If $\bar{t}_1 \sqcup \bar{t}_2 = \bar{t}_3$, then $\delta_{\bar{P}}(\bar{t}_3) = \delta_{\bar{P}}(\bar{t}_1) + \delta_{\bar{P}}(\bar{t}_2) - \delta_{\bar{P}}(\bar{t}_1 \sqcap \bar{t}_2) \leq \delta_{\bar{P}}(\bar{t}_1) + \delta_{\bar{P}}(\bar{t}_2)$.*
3. *For any t, its supertype branching factor $\sigma(t) \leq 2^{\delta_{\bar{P}}(\bar{t})} - 1$.*

The arity of the terms in a module's encoding must be constant, so (1) implies that that the arity of the term encoding must be at least as large as the length of the longest subtype chain in P, since the type at the end of that chain must be encoded by a term of at least that arity. This lower bound can always be attained if there are no join reducible types simply by using a tree encoding. (2) says that the δ-value of the result of unification in a unification-preserving encoding cannot exceed the sum of its operands' δ-values — since we are not requiring our encodings to be meet-preserving, we might not know the value of $\delta_{\bar{P}}(\bar{t}_1 \sqcap \bar{t}_2)$. This means that a join reducible type can, in general, force an encoding to have greater arity to allow for higher δ-values of its two supertypes so that there will be enough "room" for their join. (3) essentially documents the same effect as a result of the bound on the number of terms that can possibly subsume a Prolog term of a given arity. Not every Prolog term can attain that bound, however — only those that have the same constant in every position, e.g., $f(1, 1, \ldots, 1)$.

The practical consequence of (1) is that in Figure 8.4, as the parameter d increases, the size of the encoding must increase linearly. The practical consequence of (3) is that in Figure 8.5, as the parameter x increases, the

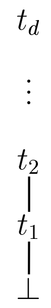

Figure 8.4: A type hierarchy whose flat term encoding grows linearly with d.

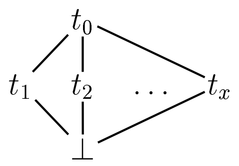

Figure 8.5: A type hierarchy whose flat term encoding grows logarithmically with x.

size of the encoding must increase logarithmically.

The choice of constants for instantiation of arguments is also bounded as a function of arity:

Proposition 8.4. *In a finite type hierarchy, P, if there is a flat-term encoding of arity a, there is a flat-term encoding of arity a that uses no more than $a \cdot max(P)$ constants, where $max(P)$ is the number of maximally specific types in P.*

Proof. Because flat-term encodings preserve unification, they also preserve subsumption, so a constant used in any term is reflected in the same argument position of the encoding of some maximally specific type. There are $a \cdot max(P)$ such positions. Of course, a Colmerauer-style encoding only uses two constants, 0 and 1. $\square$

The net result of these constraints is that there is a finite space of parameters — arity and number of constants — through which we can search for an optimal encoding, provided that we have a uniform representation of encodings through which to search. That representation can be achieved by looking at the subsumption matrices of the algebras we are trying to encode, as seen in Chapter 7, and relating it to the Prolog terms eligible to

	$\bot$	lax	cl	cng	lax& cl	lax& cng	cl& cng
$\bot$	1	1	1	1	1	1	1
lax	0	1	0	0	1	1	0
cl	0	0	1	0	1	0	1
cng	0	0	0	1	0	1	1
lax&cl	0	0	0	0	1	0	0
lax&cng	0	0	0	0	0	1	0
cl&cng	0	0	0	0	0	0	1

Figure 8.6: The subsumption matrix for Figure 8.2.

participate in our encodings. A subsumption matrix uniquely characterizes a finite meet semi-lattice's behavior with respect to unification, since joins are completely determined by subsumption. In fact, as observed by Aït-Kaći et al. [1989], each ith row can be used as an encoding of the type t_i, with unification corresponding to component-wise AND. The subsumption matrix of Figure 8.2 is shown in Figure 8.6.

For a fixed a and k, $GAF^a_{1,k}$ has a fixed subsumption matrix. With this view of finite meet semi-lattices, finding a flat-term encoding of one with a subsumption matrix S amounts to finding the right rows and columns of a GAF lattice that will behave like S. In the following, it is assumed that both are top-smashed, to give a logical point of reference to failure, as in Chapter 7, and that the rows and columns of subsumption matrices are topologically sorted (as is standard, so that the matrices will appear in upper-triangular form):

Proposition 8.5. *Every flat-term encoding of a finite type hierarchy with subsumption matrix, S, uniquely corresponds to a selection matrix, X, such that:*

1. $X \cdot GAF^a_{1,k} \cdot X^\mathsf{T} = S$*, for some arity, a, some maximum constant, k, and*
2. $X_{|S|,|GAF^a_{1,k}|} = 1$.

An optimal flat-term encoding is one with the least arity a for which such an X exists.

The fact that we are using selection matrices — matrices with exactly one 1 in any row and no more than one 1 in any column, means that we satisfy the injectivity condition of Mellish [1991] (see Section 6.1.1, this dissertation). The first condition given here means that it satisfies Mellish's homomorphism condition — the terms corresponding to the rows and columns in $GAF^{a}_{1,k}$ preserve unification because they have the same subsumption matrix. The second condition is necessary to ensure Mellish's zero-preservation condition — since $\top$ is last in the topological ordering, it means that $\top$, or failure, in one domain corresponds to failure in the other. We want Prolog unification to fail when unification in P fails.

We can thus reduce the search for an encoding to a search for a selection matrix over a finite space of parameters: arity, ranging from 0 to the number of meet irreducible types, and number of constants, ranging from 0 to $a \cdot max(P)$. S is no ordinary matrix, furthermore. If we break it into submatrices $A_{i,j}$ for rows of types with $\delta_P = i$ and columns of types with $\delta_P = j$, then for all i, j, $A_{i,i}$ is an identity matrix, and $A_{i,j}$ is a zero matrix when $i < j$. Due to the constraints mentioned above, we can consider the problem, to a great extent, independently by information level when solving for X.

The complexity of the general problem is still open, but it is quite likely to be NP-complete. Fall [1996, pp. 78–79] proved that finding an optimal join-incompatible partition[3] of an ordered set of elements is NP-complete. He observes that this can be used to construct a logical term encoding, in which there are no shared variables and every term has a constant in exactly one position. Such an encoding is not guaranteed to be zero-preserving, however, and because it is the kind of term encoding that is being restricted (rather than, for example, the kind of ordered set), it cannot straightforwardly be extended to a complexity result for the general problem, even in the absence of zero-preservation. The encoding problem without shared variables also bears a resemblance to the problem of finding a minimal intersection graph basis (Garey and Johnson, 1979, p. 204; Kou et al., 1978), in which bit vectors of a certain arity are allocated to the nodes of a graph rather than vectors (terms) of constants and variables.

[3]In the terminology of Fall [1996], it is the optimal meet-incompatible partition problem.

8.2 Features

Once we add features, we need to accommodate their values in the encoding, including possibly circular structures, which reduce straightforwardly to circular Prolog terms. As mentioned in Chapter 6, Prolog term unification, by definition, cannot handle non-statically typable signatures, so it only makes sense to focus on signatures that satisfy the join preservation condition. It is, however, possible to implement the non-statically typable ones using the functionality of an enhanced Prolog such as SICStus Prolog's attributed variables library [Holzbaur, 1990, 1992], `library(atts)`, which allows for hooks to unification. Attributed variables, in fact, look a great deal like untyped feature structures with no appropriateness. A few encodings based on that correspondence in the statically typable case are evaluated below.

Tree-based encodings can add extra arguments at subterms where features are introduced. An example typed feature structure of type *noun*, from Figure 8.1, might be encoded as `head(subst(noun(case(nom)),plus,plus))`, where the two `plus` values are for the PRD and MOD features introduced by *head*, and `case(nom)`, the encoding of the type *nom*, is for the value of CASE, which is introduced by *noun*. The logic of Carpenter [1992] allows subtypes to refine the value restrictions on features introduced by their supertypes, and for feature introduction at joins. ProFIT's declaration language, multidimensional inheritance, allows for neither of these; but a tree encoding is compatible with them, in principle.

An alternative is to encode all of the feature values of a module as extra arguments at the top level of the subtype encoding. This again appeals to the wisdom, "broader beats deeper," particularly since feature values are themselves encoded typed feature structures. It has the additional advantage that binding a variable to a feature value, another very common operation, can in many cases be compiled out to a very efficient `arg/3` call in Prolog run-time code, where the tree-based encoding would require a more expensive term traversal. It also has the same empirical caveat as with subtype encoding: that empirical domains that make reference to a large number of typed feature structures with types more general than types that introduce features may still perform better with the tree encoding, because they avoid the extra unused feature positions. Tree encodings of feature structures of types *subst* or *head* in Figure 8.1 do not need to carry an argument position for the value of CASE, for example.

How many extra argument positions do we need for features in a module's encoding? The naive answer is the number of features introduced in that module. It is possible to do better:

Definition 8.3. *The* feature graph*, $G(M)$ of module M, is an undirected graph whose vertices correspond to the features introduced in M, and in which there is an edge, (F, G), iff F and G are appropriate to a common type in M.*

Proposition 8.6. *The least number of argument positions required for the features of M in a flat encoding is the least N for which $G(M)$ is N-colorable.*

The positions correspond to the colors. This is related to using graph coloring for register allocation in compiler design. In Figure 8.1, the features PRD, MOD, and CASE form a graph that is at best 3-colorable, because they are all appropriate to the common type, *noun*.

8.3 Evaluation

The alternatives presented here have been evaluated on the task of tabulation-based parsing with two English grammars over corpora that were automatically generated with skeletal context-free grammars over the same lexicon. Measurements were made on a dual-400-MHz SPARC Ultra 450 with 512 MB of RAM running the Solaris 2.6 operating system.

Except where noted, all of the encodings were implemented as modifications of the Attribute Logic Engine (ALE, Carpenter and Penn, 1996). ALE is a logic programming language based on the logic of typed feature structures. With the exception of using typed feature structure arguments instead of first-order terms, its relational language is nearly identical to Prolog. It also has a built-in bottom-up chart parser, driven by extended feature-structure-based phrase-structure rules, which was used in the benchmarks as well. Both the ALE compiler and run-time system themselves are written in Prolog. The ALE compiler generates Prolog code which is then compiled further by a Prolog compiler. The ALE compiler, including the encoding algorithms, can thus be viewed as a preprocessing step much like the one found in the standard Prolog term-expansion mechanism. The version of Prolog used in these experiments was the SICStus Prolog 3.8.3 native code compiler. ALE's relations are extra-logical (with negation treated as negation-by-failure as in Prolog) so a generalized term encoding must be used. The most thorough, although rather biased survey of other programming languages and parsing systems based on typed feature structures can be found in Bolc et al., 1996.

The first comparison, on which Figure 8.7 is based, is a Head-driven Phrase Structure grammar (HPSG) distributed with the ALE system, sometimes called "naive HPSG." It is a very straightforward, unoptimized encoding of the first five chapters of Pollard and Sag, 1994, and a common benchmark for logic programming with typed feature structures. It uses almost every piece of functionality that ALE offers, and massively overgenerates semantic representations because of Pollard's and Sag's [1994] treatment of quantifier scope, making it very easy to find computationally intensive parses in a test corpus.

The naive HPSG grammar has 162 types and 37 features, which decompose into a large number of small modules, each having at worst 5-colorable feature graphs. All modules but two are free of join reducible types, which means that they are optimally tree-encodable; but the two, lists and sets, are heavily used within the grammar. The corpus on which it was tested consists of 64,331 sentences, of which 63,914 are grammatical, i.e., parsing succeeds. The size of the substring tables for each sentence (a rough measure of complexity) ranges from 18 to 9,619 edges. Sentence lengths range from 2 to 25 words, and the sentences are presented in in ascending order by parse time using the last (and thus fastest) encoding alternative shown. For ease of presentation, all of the results have been smoothed by a moving average with a window of 100 sentences. The top alternative depicted is the performance of a naive encoding of typed feature structures based on the SICStus Prolog attributed variables library, where the type is represented as the value of an extra feature defined on every structure. The second uses the same library but with one attribute for every "color" of feature as described above, rather than for every feature. This takes advantage of the high modularity of the grammar. The third uses undocumented SICStus internal predicates to manipulate those attributes directly in order to exploit the existence of appropriateness conditions. The fourth is not a proper Prolog encoding — it uses a Prolog data structure that must be dereferenced before unification. This data structure is the one found in ALE 3.2, the most recent public release of the ALE system. Both the third and the fourth use the exact feature arity of every type for its representation, so no coloring is needed. For non-statically typable modules, these two are the best alternatives available. The advantage of the third is that it can be used together with Prolog-term-encoded static modules because of the availability of the `verify_attributes/3` unification hook for attributed variables.

The last three are proper Prolog term encodings as elaborated upon here. The fifth was obtained from ProFIT 1.54 with its tree encoding method — the list and set modules are tree encodable. ProFIT comes with a port of the naive HPSG grammar that strips out polymorphic lists so that Prolog lists can be used at the abstract-machine level. As a result, it operates at a significant advantage. The last two alternatives use an optimal tree encoding on modules with no join reducible types. These are so easy to detect and the encoding is so quickly derived that no other choice makes sense. The sixth was obtained using the approximate method presented in Section 8.1.2 on the two other modules, but without feature graph coloring. The seventh uses the optimal method presented in Section 8.1.3 with feature coloring. The sixth and seventh bound the performance of the four possible permutations of encoding method with feature coloring; and, as can be seen, it makes very little difference. For this grammar, a simple, completely polynomial approximation with Colmerauer's method is worthwhile, and both are even slightly faster than pure tree encoding with no polymorphic lists.

Memory consumption ranged from 86 MB, by the optimal term encoding method plus feature coloring, to 161 MB, by the alternative that makes direct use of SICStus attributes. SICStus Prolog occupies 4.9 MB itself, between 6.9 MB and 8.5 MB once the respective versions of the feature structure compilation and run-time parsing code have been compiled, between 8.5 MB and 10 MB after the HPSG grammar has been compiled, and approximately 24 MB after the corpus has been loaded.

The second grammar (Figure 8.8) is a categorial grammar from Bell Laboratories encoded in typed feature logic, designed to have similar coverage to the naive HPSG grammar, while avoiding recursive types, having taken the lessons of Chapter 4 to heart. Its signature is also an example of one that is not tree-encodable, which means that ProFIT could not be tested on it. It has a total of 209 types and 27 features. It has only five modules, however, with the largest containing 119 of the types, including 88 meet irreducible types, but having an optimal encoding of arity 6 — in fact, there is only one join reducible type in it. Another module has 17 of the features, with a 6-colorable feature graph. This grammar's corpus consists of 100,002 sentences, of which 79,786 are grammatical, with sentence lengths ranging from 2 to 18 words, and substring table sizes ranging from 14 to 6,342 edges. The sentences are again presented in ascending order by parse time with optimal encoding plus

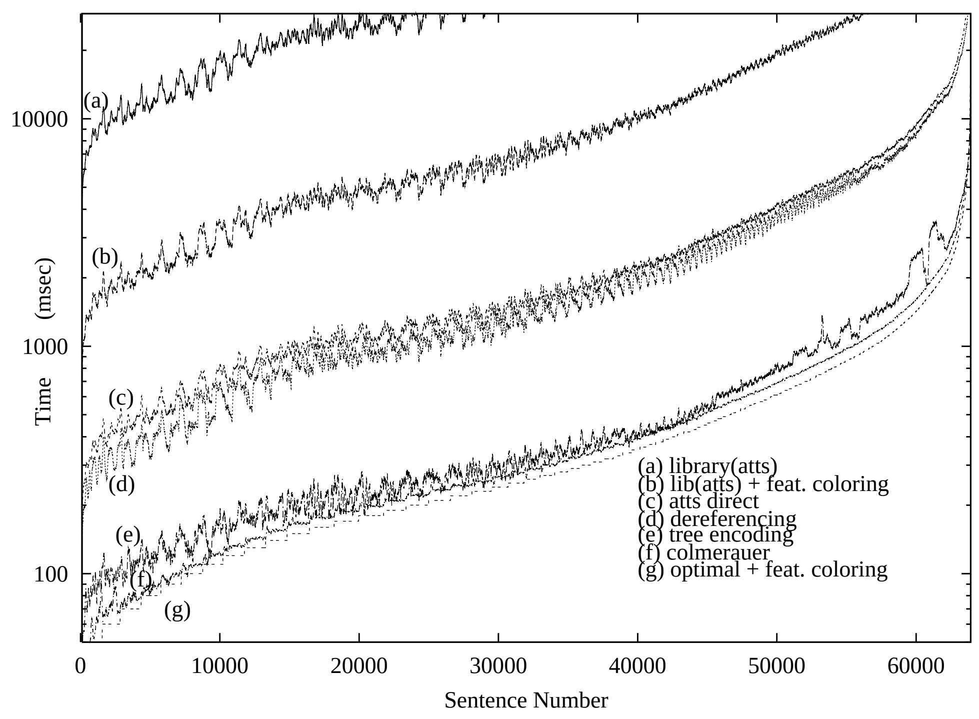

Figure 8.7: Evaluation on the ALE HPSG grammar.

feature coloring. A moving average was again used for smoothing, but with a window of 1000 sentences. In this grammar, optimal type encoding is of much greater significance — even the direct use of SICStus attributes and the dereferencing method of ALE 3.2 are better than Colmerauer's method here. Colmerauer's method failed to allocate enough memory after 78,546 sentences, in fact, because the term encodings were too large. Colmerauer's method plus feature coloring failed after 79,552 sentences, and optimal term encoding (without feature coloring) failed after 79,370 sentences.

Memory consumption ranged from 142 MB by the direct use of SICStus attributes to over 256 MB (the maximum amount that SICStus Prolog's tag-pointer indexing scheme can allocate) on those alternatives that failed to complete the test suite. The largest memory consumption by an alternative to complete the test suite was 160 MB, by optimal encoding plus feature coloring. SICStus Prolog plus the feature structure compilation, run-time parsing code and the compiled Bell Labs grammar occupies between 9.3 MB

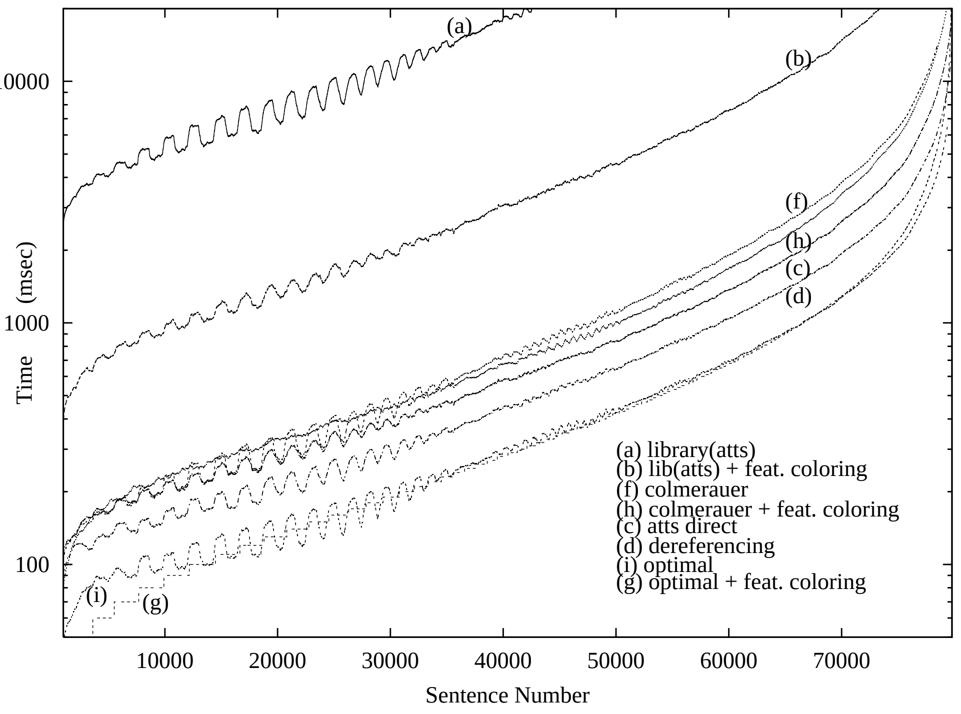

Figure 8.8: Evaluation on the Bell Labs Categorial Grammar.

and 19 MB, and after the corpus has been loaded, between 27 MB and 38 MB.

Among large scale sentences that can be parsed both by the Bell Labs grammar and naive HPSG, the Bell Labs grammar is approximately 323 times faster than naive HPSG on large-scale (8000 or more edges) parses. Combined with optimal term encoding, the improved static analysis necessary for term encoding (which is included in all of the alternatives measured above), and better indexing for parsing, its performance is slightly over 113,000 times faster than naive HPSG running on ALE 3.0, the version of ALE that served as the starting point for this study.

The question also arises how any Prolog term encoding might compare with a logic programming language whose abstract machine was designed specifically for typed feature structures. The first abstract machine architecture proposed for an attribute-value logic was described in Aït-Kaći and Di Cosmo, 1993, although the variant of attribute-value logic assumed there allowed for infinite-branching terms.

Several such architectures have also been proposed for fragments of ALE, beginning with that of Wintner and Francez [1994], which did not support disjunctive descriptions nor non-statically typable signatures, and whose implementation, AMALIA [Wintner, 1997], did not include Prolog-style SLD resolution over relational predicates — only bottom-up parsing. Carpenter and Qu [1995] proposed one that does handle disjunctive descriptions and non-statically typable signatures. Its implementation, LiLFeS [Makino et al., 1998] includes true SLD resolution, having combined it with a feature-structure-based re-implementation of Aquarius Prolog based on the Berkeley Abstract Machine [Van Roy, 1990]. There is, in fact, a small cottage industry of abstract machines for feature-structure-based natural language processing now, largely due to the influence of these two original ones, abetted by careless, inaccurate benchmarking that exaggerated their improvement relative to Prolog-based systems such as ALE and ProFIT. That includes, for example, ignoring that different parsing algorithms and/or chart-indexing strategies were used, using very small test corpora (often fewer than 10 sentences) and using test sentences of such very small complexities that the initialization routines are more computationally significant than the parsing routine itself.

Because the naive HPSG grammar makes heavy use of SLD resolution, it cannot be tested on AMALIA. The naive HPSG grammar has been ported to LiLFeS, however, and a comparison between LiLFeS 0.88 compact code and the hybrid Colmerauer/optimal encoding from above in ALE on naive HPSG is presented in Figure 8.9. There is also a LiLFeS native code compiler for Pentium processors, which could not be tested at the time that these tests were made. Obtaining a large number of execution times in LiLFeS is not quite as simple, so a smaller corpus was used, consisting of eleven classes of approximately twenty parses each, distributed evenly across the range of parsing complexities found in the larger corpus above.

For the purposes of the comparison, ALE's parser was rewritten, so that it is exactly like the one distributed with LiLFeS for its port of naive HPSG. The LiLFeS port is actually a port of the ProFIT port of the naive HPSG grammar, so it too operates at an advantage because of the absence of polymorphic lists. At the closest separation, ALE running on SICStus compact code is 2.5 times faster, with LiLFeS's performance slowly degrading to slightly over 10 times slower. The eleventh test class resulted in a memory allocation failure on LiLFeS because the computer would not allocate over 700 MB of memory at run-time to the process. In fact, when ProFIT uses the same parsing

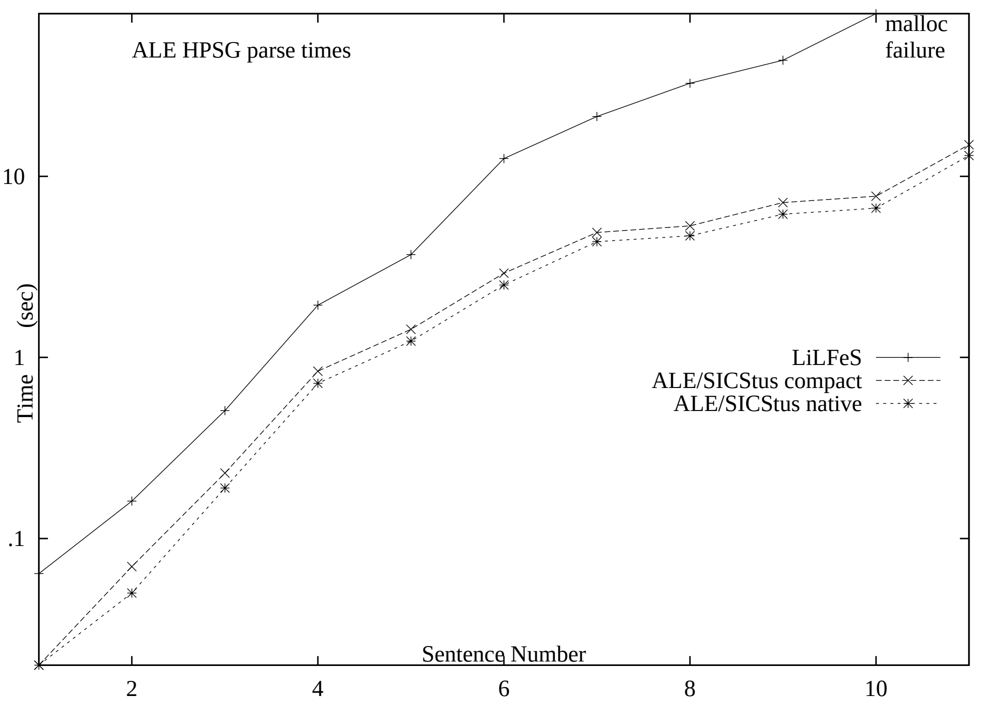

Figure 8.9: Comparison of LiLFeS and the ALE Colmerauer/optimal encoding on naive HPSG.

algorithm, even ProFIT is slightly faster than LiLFeS.

This test, of course, involves a number of different components: parsing, SLD resolution, tabulation of asserted predicates (the parsing algorithm was again a bottom-up chart parser) as well as unification. For a more controlled test of SLD resolution, LiLFeS was compared to ALE running on SICStus Prolog on the naive reverse benchmark, in which a non-polymorphic list (encoded as typed feature structures) of 30 elements is reversed without an accumulator 10,000 times. This is a standard benchmark of speed on Prolog-like systems. Table 8.1 shows the results. ALE running on compact code operates slightly faster, with almost a factor of 10 smaller memory usage. If lists are declared to be extensional, which allows ALE to use Prolog lists to encode ALE list-typed feature structures in a manner similar to the internal encoding used in LiLFeS, then ALE compact code is over 10 times faster.

LiLFeS's slower performance is mostly due to the fact that SICStus's

System	Time	LIPS[4]	Image Size
LiLFeS compact	34.21s	144,969	55M
ALE/SICStus compact	26.64s	186,186	6.4M
ALE/SICStus native	10.38s	477,842	6.4M
ALE/SICStus compact / list cells	2.61s	1,900,383	6.4M
ALE/SICStus native / list cells	0.42s	11,809,524	6.4M

Table 8.1: Comparison of LiLFeS and ALE on the `nrev30x10K` benchmark.

memory management and predicate compilation are simply much better. On the other hand, this is one of the main reasons for using a Prolog-based implementation to begin with: avoiding redundant problem-solving and utilizing the last sixteen years' worth of research on optimizing the Warren Abstract Machine. ALE's only challenge was to find an encoding of typed feature structures that allowed for maximal transparency and minimal term size. That challenge was met through a proper understanding of the algebraic structure induced by attributed type signatures.

8.4 Summary

Two methods have been presented to encode statically typable type signatures as flat Prolog terms, which provide an improvement in speed over other general representation methods, including abstract-machine-based ones, and a competitive performance with tree encoding, in addition to its more general applicability. A few of the results, such as the graph-coloring reduction and the selection matrix reduction, are independently of theoretical interest.

What remains now is to find heuristic methods that, for linguistically prevalent type signatures, can constrain the compile-time parametric search for optimal flat-term type encodings, and hybrids of the encoding strategies considered here that can provide the best performance to the empirically realistic processing needs of the knowledge representation and computational linguistics communities.

[4]logical inferences per second.

Chapter 9

Conclusion

This dissertation presents formal definitions of signature subsumption and equivalence that have several applications to understanding, extending, and computing with the logic of typed feature structures. In the process, a better abstraction of join-preserving encodings was formulated that encompasses the classical definition as a special case. The difference between the two abstractions was critical in finding a Prolog term encoding of statically typable attributed type signatures that is robust enough to be used with an extra-logical relational extension (which is arguably required to be useful at all).

The view of logic programming with typed feature structures that one can assemble from the results presented here is that it is a task that can be decomposed into essentially three areas that have already been well-studied: logic programming with Prolog terms, (sparse) matrix multiplication, and various graph-theoretic algorithms, such as the graph coloring reduction of minimum feature position allocation. That reduction has been shown to yield a significant improvement in both coverage and speed on the two grammars it was tested with when compared to other Prolog and customized-abstract-machine-based approaches.

The areas of immediate interest for future research in light of the results presented here seem to be mostly practical. One is the further development of the view of encoding as a matrix multiplication problem. The optimal flat-term encoding problem still requires the discovery of a better class of polynomial-time heuristic methods. Matrices also lie at the heart of the view of signature specifications presented in Chapter 7, which also requires further development, particularly of sparse algorithms that can exploit the proper algebraic structure of closed semi-rings.

By far the most compelling open problem suggested by the work presented here is the use of statistical methods to optimize term encodings. A practical encoding algorithm would weigh the importance of assigning a more terse or flatter term to individual types by the likelihood that those types will be encountered by a unification algorithm in the course of its use on typical input. An optimal encoding, given these weights, may actually assign larger term encodings to some types than the method presented here, but with an overall gain in efficiency because those types are only rarely encountered. Preliminary empirical attempts date back to the work of Schöter [1992], who proposed to reorder the arguments of a Prolog term encoding based on the likelihood that unification of those arguments would fail. Specifically, the arguments should be re-ordered from left-to-right in decreasing order of the probability of failure, because this is the order in which Prolog conventionally unifies its arguments. Those probabilities, however, were estimated by a structural analysis of the signature, in which types with more join-incompatible subtypes were assumed to be more likely to cause a unification failure than types with fewer. Empirically, the probabilities can deviate significantly from that estimate — to the extent of preferring the exact opposite ordering. A properly empirical estimation of these probabilities was attempted in the context of a more general consideration of optimizing don't-care-non-determinism by Penn [1999b]. Of course, the structure of attributed type signatures is now well-enough understood that it makes sense to begin to apply statistical methods to the general encoding problem more globally than by simply reordering arguments, as well as to other problems such as indexing which can play a very important role in the efficiency of large-scale logic programming or parsing systems.

On the more theoretical side, the expressive power of parametric type signatures have still not been adequately characterized with respect to non-parametric signatures in terms of true signature equivalence, although the more practical weak equivalence in the form of symmetric subsumption has been addressed here. A more fine-grained analysis of the equivalences that must certainly exist between bounded unfoldings of recursive features (whose unfolding was simply written off here as being infinite and therefore impossible) is definitely in order. In all likelihood, there is a more elegant category-theoretic treatment of signature equivalence and subsumption that would perhaps shed more light on parametric types as well as the other equivalences treated here in a more classical fashion. One can cite Moshier, 1997a,b as an initial step in this direction.

Bibliography

N. Ach. Determining tendencies; awareness. In D. Rapaport, editor, *Organization and Pathology of Thought*, pages 15–38. Columbia University Press, 1951. English translation of chapter 4 of Ach [1905].

N.K. Ach. *Über die Willenstätigkeit und das Denken.* Vandenhoeck & Ruprecht, Göttingen, 1905.

A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

H. Aït-Kaći. *A Lattice-theoretic Approach to Computation based on a Calculus of Partially Ordered Type Structures.* PhD thesis, University of Pennsylvania, 1984.

H. Aït-Kaći. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

H. Aït-Kaći, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, 1989.

H. Aït-Kaći and R. Di Cosmo. Compiling order-sorted feature term unification. Technical Report 7, Digital Equipment Corporation Paris Research Lab (DEC PRL), 1993.

K. Bertet, M. Morvan, and L. Nourine. Lazy 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.

L. Bolc, K. Czuba, A. Kupść, M. Marciniak, A. Mykowiecka, and A. Przepiórkowski. A survey of systems for implementing HPSG grammars. Technical Report 814, Institute of Computer Science, Polish Academy of Sciences, 1996.

A. Borgida, R. J. Brachman, D. L. McGuinness, and L. A. Resnick. Classic: A structural data model for objects. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, 1989.

R. J. Brachman. *A Structural Paradigm for Representing Knowledge.* PhD thesis, Harvard University, 1977.

R. J. Brachman. What IS-A is and isn't: An analysis of taxonomic links in semantic networks. *IEEE Computer*, 16(10):30–36, 1983.

R. J. Brachman, R. E. Fikes, and H. J. Levesque. KRYPTON: A functional approach to knowledge representation. *IEEE Computer*, 16(10): 67–73, 1983.

R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

J. S. Bruner, J. J. Goodnow, and G. A. Austin. *A Study of Thinking.* Wiley, 1956.

B. Carpenter. *The Logic of Typed Feature Structures.* Cambridge, 1992.

B. Carpenter. An attribute-value logic for sets. In *Third ASL/LSA Conference on Logic and Language*, Ohio State University, 1993a.

B. Carpenter. Skeptical and credulous unification with applications to lexical templates and inheritance. In T. Briscoe, A. Copestake, and V. de Paiva, editors, *Default Reasoning and Lexical Organization.* Cambridge University Press, 1993b.

B. Carpenter and P.J. King. The complexity of closed world reasoning in constraint-based grammar theories. In *Fourth Meeting on the Mathematics of Language*, University of Pennsylvania, 1995.

B. Carpenter and G. Penn. Negation vs. inequation and typing for linguistic applications. Available from Gerald Penn's homepage: `http://www.cs.cmu.edu/∼gpenn`, 1993.

B. Carpenter and G. Penn. Compiling typed attribute-value logic grammars. In H. Bunt and M. Tomita, editors, *Recent Advances in Parsing Technologies*, pages 145–168. Kluwer, 1996.

B. Carpenter and C. Pollard. Inclusion, disjointness and choice: The logic of linguistic classification. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics (ACL-91)*, pages 9–16, 1991.

B. Carpenter and Y. Qu. An abstract machine architecture for typed attribute-value grammars. In *Proceedings of the 4th International Workshop on Parsing Technology*, 1995.

N. Chomsky and M. Halle. *The Sound Pattern of English.* Harper & Row, 1968.

A. Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 85–99, 1984.

A. Colmerauer. Theoretical model of Prolog II. In M. van Canegham and D. H. Warren, editors, *Logic Programming and its Application*, pages 1–31. Ablex, Norwood, New Jersey, 1987.

D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progression. *Journal of Symbolic Computation*, 9(3):251–280, 1990. Special Issue on Computational Algebraic Complexity.

T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

V. Dahl. Un systeme deductif d'interrogation de banques de donnees en espagnol. Technical report, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille II, 1977.

V. Dahl. On database systems development through logic. *ACM Transactions on Database Systems*, 7(1), 1982.

B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

R. Dietrich and F. Hagl. A polymorphic type system with subtypes for Prolog. In *Proceedings of the 2nd European Symposium on Programming*, number 300 in LNCS, pages 79–93. Springer, 1988.

M. Dorna. Erweiterung der Constraint-Logiksprache CUF um ein Typsystem. Diplomarbeit, Universität Stuttgart, 1992.

C.C Douglas, M.A. Heroux, G. Slishman, and R. Smith. Gemmw: A portable level 3 blas winograd variant of strassen's matrix-matrix multiply algorithm. *Journal of Computational Physics*, 110:1–10, 1994.

G. Erbach. Multi-dimensional inheritance. In *Proceedings of KONVENS 94*. Springer, 1994.

G. Erbach. ProFIT: Prolog with features, inheritance and templates. In *Proceedings of EACL-95*, 1995.

G. Erbach. *Bottom-Up Earley Deduction for Preference-Driven Natural Language Processing*. PhD thesis, Universität des Saarlandes, 1996.

S. E. Fahlman. *A System for Representing and Using Real-world Knowledge*. PhD thesis, MIT, 1977.

S. E. Fahlman. *NETL: A System for Representing and Using Real-World Knowledge*. MIT Press, 1979.

A. Fall. *Reasoning with Taxonomies*. PhD thesis, Simon Fraser University, 1996. Repaginated as a single-spaced document with minor corrections, and available from **http://www.cs.sfu.ca/cs/people/GradStudents/fall/personal/pub/thesis.ps**.

D.D. Ganguly, C.K. Mohan, and S. Ranka. A space-and-time-efficient coding algorithm for lattice computations. *IEEE Transactions on Knowledge and Data Engineering*, 6(5):819–829, 1994.

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Co., 1979.

G. Gazdar, E. Klein, G.K. Pullum, and I.A. Sag. *Generalized Phrase Structure Grammar*. Basil Blackwell, 1985.

D. Gerdemann. Open and closed world types in NLP. In J. Kilbury and R. Wiese, editors, *Integrative Ansätze in der Computerlinguistik: Proceedings der 5. Fachtagung der Sektion Computerlinguistik der DGfS*, pages 25–30, 1995a.

D. Gerdemann. Term encoding of typed feature structures. In *Proceedings of the 4th International Workshop on Parsing Technologies*, pages 89–97, 1995b.

D. Gerdemann and P. J. King. The correct and efficient implementation of appropriateness specifications for typed feature structures. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING-94)*, 1994.

M. Habib and L. Nourine. Bit-vector encoding for partially ordered sets. In V. Bouchitté and M. Morvan, editors, *Orders, Algorithms, Applications: International Workshop ORDAL '94 Proceedings*, pages 1–12. Springer-Verlag, 1994.

P. J. Hayes. The logic of frames. In D. Metzing, editor, *Frame Conceptions and Text Understanding*, pages 46–61. Walter de Gruyter and Co., 1979.

R. Henschel. Traversing the labyrinth of feature logics for a declarative implementation of large scale systemic grammars. In *Proceedings of the 1995 Workshop on Computational Logic for Natural Language Processing (CLNLP-95)*, 1995.

M. Höhfeld and G. Smolka. Definite relations over constraint languages. LILOG Report 53, IBM Deutschland, 1988.

C. Holzbaur. *Specification of Constraint Based Inference Mechanism through Extended Unification.* PhD thesis, University of Vienna, 1990.

C. Holzbaur. Metastructures vs. attributed variables in the context of extensible unification. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, number 631 in LNCS, pages 260–268. Springer Verlag, 1992.

C. I. Hovland. A "communication analysis" of concept learning. *Psychological Review*, 59:461–472, 1952.

R. A. Hudson. Systemic generative grammar. In M. A. K. Halliday and J. R. Martin, editors, *Readings in Systemic Linguistics*. Batsford Academic, 1981.

G. Humphrey. *Thinking: An Introduction to its Experimental Psychology.* Methuen & Co., 1951.

E.B. Hunt. *Concept Learning: An Information Processing Problem.* Wiley and Sons, 1962.

M. Johnson. *Attribute-Value Logic and the Theory of Grammar.* CSLI Publications, 1988.

R. 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.

R. Kaplan and A. Zaenen. Long-distance dependencies, constituent structure and functional uncertainty. In M. Baltin and A.S. Kroch, editors, *Alternative Conceptions of Phrase Structure*, pages 17–42. University of Chicago Press, 1986.

L. Karttunen. Features and values. In *Proceedings of the Tenth International Conference on Computational Linguistics (COLING-84)*, pages 28–33, 1984.

R.T. Kasper. Systemic grammar and Functional Unification Grammar. In J. Benson and W. Greaves, editors, *Proceedings of the Twelfth International Systemics Workshop*, 1986.

R.T. Kasper. *Feature Structures: A Logical Theory with Application to Language Analysis.* PhD thesis, University of Michigan, 1987a.

R.T. Kasper. A unification method for disjunctive feature structures. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics (ACL-87)*, pages 235–242, 1987b.

R.T. Kasper. An experimental parser for systemic grammars. In *Proceedings of the Twelfth International Conference on Computational Linguistics (COLING-88)*, 1988.

R.T. Kasper. Unification and classification: An experiment in information-based parsing. In *Proceedings of the First International Workshop on Parsing Technologies (IWPT)*, pages 1–7, 1989.

R.T. Kasper and W.C. Rounds. A logical semantics for feature structures. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics (ACL-86)*, pages 235–242, 1986.

R.T. Kasper and W.C. Rounds. The logic of unification in grammar. *Linguistics and Philosophy*, 13(1):35–58, 1990.

M. Kay. Functional grammar. In *Proceedings of the 5th Annual Meeting of the Berkeley Linguistics Society*, pages 142–158, 1979.

M. Kay. Functional Unification Grammar: A formalism for machine translation. In *Proceedings of the 10th International Conference on Computational Linguistics*, pages 75–78, 1984.

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.

B. Keller. *Feature Logics, Infinitary Descriptions and Grammar*. CSLI, 1993.

P.J. King. *A Logical Formalism for Head-driven Phrase Structure Grammar*. PhD thesis, University of Manchester, 1989.

P.J. King and T. Goetz. Eliminating the feature introduction condition by modifying type inference. Technical Report 31, Sonderforschungsbereich 340 (SFB 340), Tübingen, 1993.

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.

J.T. Kou, L.J. Stockmeyer, and C.K. Wong. Covering edges by cliques with regard to keyword conflicts and intersection graphs. *Communications of the ACM*, 21(2):135–139, 1978.

R. Kowalski. Predicate logic as a programming language. In *Proceedings of the 1974 Congress of the International Federation for Information Processing (IFIP)*, pages 569–574, 1974.

G. R. Kress, editor. *Halliday: System and Function in Language.* Oxford, 1976.

A. Lascarides and A. Copestake. Default representation in constraint-based frameworks. *Computational Linguistics*, 25(1):55–105, 1999.

H. J. Levesque. *A Formal Treatment of Incomplete Knowledge Bases.* PhD thesis, University of Toronto, 1981a.

H. J. Levesque. The interaction with incomplete knowledge bases: A formal treatment. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI-81)*, 1981b.

LinGO. The LinGO grammar and lexicon. Available on-line at **http://lingo.stanford.edu**, 1999.

R. MacGregor. Using a description classifier to enhance deductive inference. In *Proceedings of the Seventh IEEE Conference on AI Applications*, pages 141–147, 1991.

R. M. MacGregor. A deductive pattern matcher. In *Proceedings of AAAI-88*, pages 403–8, 1988.

T. Makino, K. Torisawa, and J. Tsuji. LiLFeS — practical unification-based programming system for typed feature structures. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and the 17th International Conference on Computational Linguistics (COLING/ACL-98)*, volume 2, pages 807–811, 1998.

S. Manandhar. An attributive logic of set descriptions and set operations. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics (ACL-94)*, pages 255–262, 1994.

W.C. Mann and C.M.I.M. Matthiessen. NIGEL: A systemic grammar for text generation. Technical Report RR-85-105, Information Sciences Institute, University of Southern California, 1983.

C. Manning. *Ergativity: Argument Structure and Grammatical Relations.* CSLI Publications, 1996.

C. Manning and I. Sag. Argument structure, valence, and binding. *Nordic Journal of Linguistics*, 21:107–144, 1998.

J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4), 1960.

C. Mellish. Implementing systemic classification by unification. *Computational Linguistics*, 14(1):40–51, 1988.

C. Mellish. Graph-encodable description spaces. Technical report, University of Edinburgh Department of Artificial Intelligence, 1991. DYANA Deliverable R3.2B.

C. Mellish. Term-encodable description spaces. In D.R. Brough, editor, *Logic Programming: New Frontiers*, pages 189–207. Kluwer, 1992.

M. Minsky. A framework for representing knowledge. In P.H. Winston, editor, *The Psychology of Computer Vision*, pages 211–277. McGraw-Hill, 1975.

M. A. Moshier. Featureless HPSG. In P. Blackburn and M. de Rijke, editors, *Specifying Syntactic Structures.* CSLI Publications, 1997a.

M. A. Moshier. Is HPSG featureless or unprincipled? *Linguistics and Philosophy*, 20(6):669–695, 1997b.

M. A. Moshier and C. J. Pollard. The domain of set-valued feature structures. *Linguistics and Philosophy*, 17:607–631, 1994.

M.A. Moshier. *Extensions to Unification Grammar for the Description of Programming Languages.* PhD thesis, University of Michigan, 1988.

M.A. Moshier and W. C. Rounds. A logic for partially specified data structures. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 156–167, 1987.

A. Newell and J.C. Shaw. Programming the logic theory machine. In *Proceedings of the 1957 Western Joint Computer Conference*, 1957.

A. Newell, J.C. Shaw, and H.A. Simon. Chess-playing programs and the problem of complexity. *IBM Journal of Research and Development*, 2: 230–335, 1958.

A. Newell and H.A. Simon. The logic theory machine: A complex information processing system. *Institute of Radio Engineers Transactions on Information Theory*, IT-2(3):61–79, 1956.

P. F. Patel-Schneider, L. A. Resnick, D. L. McGuinness, E. Weixelbaum, M. K. Abrahams, and A. Borgida. *NeoClassic Reference Manual: Version 1.0*, July 1998. Available from the NeoClassic Homepage: `http://www.bell-labs.com/project/classic/neo.html`.

G. Penn. Parametric types for typed attribute-value logic. Technical Report D-97-02, Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), 1997.

G. Penn. Parametric types for typed attribute-value logic. In *Proceedings of the 17th International Conference on Computational Linguistics and the 36th Annual Meeting of the Association for Computational Linguistics (COLING/ACL-98)*, volume 2, pages 1027–1033, 1998.

G. Penn. An optimised Prolog encoding of typed feature structures. Technical Report 138, Sonderforschungsbereich 340, Tübingen, 1999a.

G. Penn. Optimising don't-care non-determinism with statistical information. Technical Report 140, Sonderforschungsbereich 340, Tübingen, 1999b.

G. Penn. An optimized Prolog encoding of typed feature structures. In *Proceedings of the 16th International Conference on Logic Programming (ICLP-99)*, pages 124–138, 1999c.

G. Penn. A quasi-ring construction for compiling attributed type signatures. Technical Report 141, Sonderforschungsbereich 340 (SFB 340), Tübingen, 1999d.

G. Penn. A quasi-ring construction for compiling attributed type signatures. In *Proceedings of the 6th Meeting on the Mathematics of Language (MOL-6)*, pages 105–114, 1999e.

G. Penn and B. Carpenter. ALE for speech: A translation prototype. In *Proceedings of the 6th Conference on Speech Communication and Technology (EUROSPEECH-99)*, volume 2, pages 947–950, 1999.

F.C.N. Pereira and S.M. Shieber. The semantics of grammar formalisms seen as computer languages. In *Proceedings of the 10th International Conference on Computational Linguistics (COLING-84)*, 1984.

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.

C. Pollard and I. Sag. *Information-based Syntax and Semantics*. Number 13 in CSLI Lecture Notes. CSLI Publications, 1987.

C. Pollard and I. Sag. *Head-driven Phrase Structure Grammar*. Chicago, 1994.

C.J. 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.

R. T. Prosser. Applications of Boolean matrices to the analysis of flow diagrams. In *Proceedings of the 16th Eastern Joint Computer Conference*, pages 133–138, 1959.

M. R. Quillian. Semantic memory. In M. Minsky, editor, *Semantic Information Processing*, pages 227–270. MIT Press, 1968.

B. Raphael. A computer program which 'understands'. In *Proceedings of AFIPS Joint Computer Conference*, 1964.

B. Raphael. SIR: Semantic information retrieval. In M. Minsky, editor, *Semantic Information Processing*, pages 33–145. MIT Press, 1968.

J.C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*. Edinburgh, 1970.

F. Richter. *A Mathematical Formalism for Linguistic Theories with an Application in Head-driven Phrase Structure Grammar and a Fragment of German*. PhD thesis, Universität Tübingen, in prep.

J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

W.C. Rounds and R.T. Kasper. A complete logical calculus for record structures representing linguistic information. In *Proceedings of the 15th Annual IEEE Symposium on Logic and Computer Science*, pages 39–43, 1986.

I. A. Sag. English relative clause constructions. *Journal of Linguistics*, 33 (2):431–484, 1997.

A. Schöter. Term encoding of feature structures. University of Edinburgh, 1992.

L. K. Schubert. Extending the expressive power of semantic networks. *Artificial Intelligence*, 7(2):163–198, 1976.

N.K. Simpkins and M. Groenendijk. Multiple inheritance. Technical Report ALEP-1.3, Cray Systems, August 1994.

G. Smolka. A feature logic with subsorts. Technical Report LILOG Report 33, IBM Germany, Stuttgart, 1988.

G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types.* PhD thesis, Universität Kaiserslautern, 1989.

K. Steinicke and G. Penn. Compiling feature-based constraints with complex antecedents. Technical Report 136, Sonderforschungsbereich 340 (SFB 340), Tübingen, 1999.

V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California, Berkeley, 1990.

Stephen Warshall. A theorem on Boolean matrices. *Journal of the ACM*, 9 (1):11–12, January 1962.

P. H. Winston. Learning structural descriptions from examples. In P. H. Winston, editor, *The Psychology of Computer Vision*, pages 157–209. McGraw-Hill, 1975.

S. Wintner. *An Abstract Machine for Unification Grammars with Applications to an HPSG Grammar for Hebrew.* PhD thesis, Technion, 1997.

S. Wintner and N. Francez. Abstract machine for typed feature structures. In *Proceedings of the Conference on Natural Language Understanding and Logic Programming*, 1994.

S. Wintner and N. Francez. Off-line parsability and the well-foundedness of subsumption. *Journal of Logic, Language and Information*, 8(1):1–16, 1999.

W. A. Woods. Parallel algorithms for real time knowledge based systems. Technical Report 4181, Bolt Beranek and Newman (BBN), 1979.

W. A. Woods and J. G. Schmolze. The KL-ONE family. Technical Report TR-20-90, Aiken Computer Laboratory, Harvard University, 1990.

E. Yardeni, T. Früwirth, and E. Shapiro. Polymorphically typed logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 63–90. MIT Press, 1992.