Chapter 1

Introduction

In this monograph we motivate and provide theoretical foundations for the specification and implementation of systems employing data structures which have come to be known as *feature structures*. Feature structures provide a record-like data structure for representing partial information that can be expressed in terms of features or attributes and their values. Feature structures are inherently associative in nature, with features interpreted as associative connections between domain objects. This leads to natural graph-based representations in which the value of a feature or attribute in a feature structure is either undefined or another feature structure. Under our approach, feature structures are ideally suited for use in unification-based formalisms, in which unification, at its most basic level, is simply an operation that simultaneously determines the consistency of two pieces of partial information and, if they are consistent, combines them into a single result. The standard application of unification is to efficient hierarchical pattern matching, a basic operation which has applications to a broad range of knowledge representation and automated reasoning tasks.

Our notion of feature structures, as well as its implications for processing, should be of interest to anyone interested in the behavior of systems that employ features, roles, attributes, or slots with structured fillers or values. We provide a degree of abstraction away from concrete feature structures by employing a general attribute-value logic. We extend the usual attribute-value logic of Rounds and Kasper (1986, Kasper and Rounds 1986) to allow for path inequations, but do not allow general negation or implication of the sort studied by Johnson (1987, 1988), Smolka (1988), or King (1989). On the other hand, we extend the axiomatization of Rounds and Kasper to account not only for path inequations, but also type inclusion, appropriateness, and extensionality specifications.

Feature structures, as presented here, have immediate applications to linguistic formalisms based on unification, including the basic phrase structure formalisms, Functional Unification Grammar (FUG) (Kay 1979, 1985) and Parse and Translate II (PATR-II) (Shieber et al. 1983, Shieber 1984), equation-based formalisms such as Lexical Functional Grammar (LFG) (Kaplan and Bresnan 1982), and also constraint-based formalisms such as Head-driven Phrase Structure Grammar (HPSG) (Pollard and Sag 1987, in press). For a general introduc-

tion to feature structures in linguistic theories, see Shieber (1986). Because our feature structures provide a natural generalization of first-order terms, they also have applications to the more traditional logic grammar formalisms, which are based on the unification of first-order terms (Pereira and Warren 1980, Pereira and Shieber 1987). Another obvious application is to logic programming languages themselves, in which feature structures can replace first-order terms as the data structure out of which definite clauses are constructed (Aït-Kaci and Nasr 1986, Höhfeld and Smolka 1988). Our notion of feature structures is related to the tree-based data structures used in Prolog II (Colmerauer 1984, 1987) and Prolog III (Guenthner 1987, Lehner in press). Finally, we believe that our feature structures also have applications to the so-called *terminological* knowledge representation systems such as KL-ONE (Brachman and Schmolze 1985) and its descendants, especially LOOM (Mac Gregor 1988, 1990) and CLASSIC (Borgida et al. 1989, Brachman et al.1991). These terminological formalisms are themselves descendants of the more procedurally oriented frame-based representation systems (Minsky 1975). The basis of the terminological components of these systems is an inheritance network of concepts defined in terms of their features and values. It is often argued that it is the associative nature of inheritance and frame-based representation systems and their facility for pattern-matching that leads to their efficiency (see Brachman (1979) for a summary of associative network reasoning systems).

We introduce a number of new and borrowed notions that have not been previously considered in the context of feature structure unification systems. These extensions form a proper generalization of both the more commonly found feature structure unification systems used in linguistic formalisms and the term representations found in logic programming languages. We consider these extensions in turn in an attempt to place our system in the perspective of previous studies of feature structures and unification.

Many researchers have chosen to study algebraic models of attribute-value logics, in which features are interpreted as partial functions over a domain of objects (for instance, Smolka 1988, Johnson 1988, King 1989). We feel free to restrict our attention to feature structures in our formal development, as Smolka (1988, 1989) has shown that they play a canonical role as models for attribute-value logics in much the same way that initial algebras play a canonical role with respect to a collection of first-order equational axioms (Cohn 1965, Goguen et al. 1978, Goguen 1988).

We choose to impose a particular kind of object-oriented type discipline on the collection of feature structures. We incorporate an ordered notion of types organized into a multiple inheritance hierarchy based on type inclusion. We also impose appropriateness conditions that specify the features which are appropriate for each type and the admissible types of their values. The benefits of typed programming languages are well known (see Cardelli and Wegner 1985). In particular, they allow for compile- and run-time error detection and efficient memory management. The benefits of multiple inheritance in type systems is especially prominent in reasoning systems, in which a significant amount of in-

formation can be efficiently coded in terms of the types, which eliminates a large degree of redundancy and allows information to be encoded at the appropriate level of abstraction.

We contrast two different forms of typing, one of which is more restrictive than the other. In the weaker version, a feature structure is well-typed if all of its features and their values are appropriate; in the stronger, the converse must also hold: every appropriate feature must be defined. The weaker version is well suited to the representation of sparse feature structures, while the strong approach lends itself well to efficient precompilation.

Both the strong and weak versions of our type system are well behaved in that they admit straightforward type inference algorithms. More precisely, we can define an effective procedure which takes a feature structure with underspecified types and extends it to a unique minimal well-typed feature structure if it is typable. Type inference is necessary to show that unification is well defined; we want to maintain a system in which two pieces of partial information represented by feature structures can always be combined into a single unified whole. The general picture of unification in our typed setting amounts to a standard process of unification followed by a type inference operation. For the unification portion of this process, Aït-Kaci (1984) and Moshier (1989) have shown how to modify an efficient (quasi-linear) term unification algorithm (Martelli and Montanari 1982, Jaffar 1984) to deal with type ordering information.

We also show how to efficiently (linearly) compute the subsumption or information containment relation between feature structures. This operation is often employed in linguistic applications relying on chart parsing techniques. Subsumption also plays a fundamental role in knowledge representation formalisms in the KL-ONE family, in which it is applied to both reasoning and knowledge acquisition tasks (Brachman and Schmolze 1985, Schmolze and Lipkis 1983, Mac Gregor 1988).

We study the conditions on type specifications under which a static type discipline can be maintained. A type system is said to be *static* if all of the type checking and type inference operations can be computed at compile time. The standard way to show that a type system is static is to define a type inference procedure and then show that all of the basic operations of the system preserve well-typedness. In our particular case, to achieve a fully static notion of typing, we need to ensure that the unification of two well-typed feature structures results in a well-typed feature structure. If well-typedness is preserved by unification, then all of the necessary type inference can be carried out at compile time.

An additional benefit of ordering our types into a multiple inheritance hierarchy surfaces at run time. Information which might otherwise be encoded by inference steps, or even in terms of feature structure unification, can be recast in terms of efficient inheritance operations. This factorization of inheritance, structural unification, and inference allows many expensive unification and inference steps to be reduced to table look-ups and has proved invaluable in automatic theorem proving systems (Walther 1985, Cohn 1987) and hybrid

reasoning systems such as KRYPTON (Brachman et al. 1983), KL-TWO (Vilain 1985), BACK (Luck et al. 1987, Nebel 1988), CLASSIC (Borgida et al. 1989, Brachman et al. 1991), and LOOM (Mac Gregor 1988, 1990).

In addition to a type discipline, we allow a specialized form of negation that can be expressed by means of path inequations. Our inequations are the feature structure analogues of the inequations found in Prolog II (Colmerauer 1987). The intuition is that two feature structures can be explicitly constrained to be non-identical even if they do not contain conflicting information. In terms of our motivating application to natural language grammar formalisms, inequality provides a natural interpretation for disjoint reference conditions imposed by non-reflexive pronouns (Pereira 1987). What separates this brand of negation from others that have been applied to feature structures, with the notable exceptions of the intuitionistic approach of Moshier and Rounds (1987) and the three-valued approach of Dawar and Vijay-Shanker (1989), is that it preserves the monotonicity of the model theory. More specifically, if a feature structure models a description, then every informational extension of the feature structure also models the description. A further benefit of our version of negative information is that it allows the unification of two consistent feature structures to result in a unique well-defined feature structure, which is absolutely crucial if we want to base our systems on a notion of efficient unification (Pereira 1987). In Prolog II and Prolog III, it has been found that many of the practical applications of negation, including the cut operation, can be cast in terms of inequations rather than highly problematic forms of predicate negation (Colmerauer 1987).

One of the major differences between feature structures and logical terms is in the notion of identity. In logic programming systems, two terms are taken to be identical if they have the same function or relation symbol and all of their arguments are identical. Somewhat surprisingly, this is not the natural notion of identity that is applicable to feature structures or terminological knowledge representation systems. It has been almost universally assumed (for example, by Pereira and Shieber 1984, Aït-Kaci 1984, Kasper and Rounds 1986, Pollard and Moshier 1990, Johnson 1987, and Smolka 1988) that two feature structures can have an identical type and have identical values for all of their features without themselves being identical, thus building a fair degree of intensionality into feature structure systems. Pollard and Sag (1987) argue for an intensional treatment of identity in which token identity cannot be inferred from type identity in feature structures. Concepts in terminological knowledge representation systems are also assumed to be intensional in the sense that structural isomorphism is not enough to derive token identity (Brachman et al. 1991). We generalize the treatment of identity in both first-order terms and feature structures by requiring each type to be identified as either extensional or intensional. Two feature structures of the same extensional type are identified if they have identical values for all of their features. On the other hand, there is no way to derive identities between feature structures of an intensional type. The distinction between token identity and type identity is similar to the distinction between the predicates *EQ* and *EQUAL* in the Lisp programming language; the

EQ relations holds between two objects only if they occupy the same location in memory, whereas *EQUAL* checks to see if they have identical structure componentwise. A similar distinction is made in Prolog, in which the predicate == tests for true equality in the underlying representation rather than alphabetic variance. It is interesting to note that there is no built-in check for alphabetic variance in Prolog, as it is much more difficult to compute than ==, which can be determined by simply chasing pointers.

Another aspect of our feature structures that is worth mentioning is that we allow cycles. Thus it is possible to have a feature structure whose value for some non-empty sequence of features is itself. In Prolog II, cyclic terms are allowed and are identified semantically by their infinite, albeit rational, tree unfoldings. The rational tree representation is not adequate for our purposes, as it assumes that the types are all extensional. On the other hand, when we consider extensionality, we show how to represent both Prolog and Prolog II terms as a special case of typed feature structures. One of the major processing benefits of cyclic terms is that they arise naturally during unification and the so-called *occurs check* that would rule them out turns practical quasi-linear unification algorithms into quadratic ones (Martelli and Montanari 1982, Jaffar 1984). For the sake of completeness, we also present a treatment of acyclic feature structures, in which a single additional axiom scheme completely characterizes the notion of acyclicity.

The informational content of a term in a logic programming language can be characterized uniquely by the set of its instances which are ground terms (terms without variables). Two terms are alphabetic variants (informationally identical) if and only if they can be instantiated to exactly the same set of ground terms. Ground terms are informationally maximal in the sense that there are no terms that properly extend them. A ground feature structure would have a subsumption-maximal type (that is, a type with no proper subtypes) and a ground value for each of its features and would furthermore specify the identity or inequality of each of its substructures. What is interesting is that it is only with the addition of typing and inequality that the notion of ground feature structure begins to make sense. In the standard treatments of feature structures without inequality or types, maximal feature structures require maximal amounts of structure sharing; a feature structure could always be extended informationally to a properly more specific feature structure by unifying two compatible substructures. With inequality, a feature structure can be extended by either unifying two substructures or explicitly adding an inequality between them. Without a notion of typing, maximal feature structures must provide values for every feature, even those not intended to occur together in the application. A similar problem arises in Prolog with inappropriate values. Mycroft and O'Keefe (1984) point out that the standard program for *append* admits $append(nil, 3, 3)$ as a solution, where *append* was clearly only intended to be applied to lists. Even in our extended setting, it is quite difficult to characterize feature structures in terms of their maximal or ground extensions; the interaction of inequations, extensional types and finite type hierarchies leads to subtle inconsistencies which neither our logic nor our models are powerful enough to

detect. For instance, we later see that there are some feature structures which cannot be extended to any ground feature structure.

After considering the status of maximal feature structures and the nature of partial information in our representational system, we go on to consider two "extensions" to the system, of which one enriches the description language, and the other generalizes the class of models in an algebraic direction. However, it turns out that neither of these moves truly adds any power to the system, which is why they are put off until after our study of feature structures and Rounds-Kasper-like description language. First, we consider the addition of variables to the description language, along with equations and inequations between variables. We then go on to axiomatize the enlarged collection of descriptions with variables. This allows us to prove a normal form theorem which shows how descriptions involving variables can be eliminated in favor of path equations and inequations. After variables, we tackle the problem of providing more general algebraic models of our description language. In particular, we consider a class of models consisting of an arbitrary collection of domain objects and treat features as partial functions over this domain. It is straightforward to cast the collection of feature structures as an instance of this extended model scheme. We next define satisfaction and other logical notions for these models and see that our description logic is sound over the extended interpretations. Soundness, in turn, allows us to prove, following Smolka (1988, 1989), that the feature structure model is canonical in the sense that logical equivalence, satisfiability and validity are the same notions when defined over the whole class of models as when restricted solely to the feature structure model.

The primary difference between the general algebraic models and the feature structure models of our description language is that the algebraic models have a degree of infiniteness that is not present in the feature structures. Thus our final step in characterizing the logic of feature structures is to consider the class of feature structures with possibly infinite sets of nodes. This is standard in the logical treatments of Johnson (1987) and Smolka (1988), and was originally treated in the feature structure case by Pereira and Shieber (1984), who provided a denotational semantics of PATR-II. Our treatment follows Moshier (1988) and Pollard and Moshier (1990), who make connections to the description language we employ and study the type inheritance situation. Just as in the algebraic case, this extension does not affect our logical notions, such as satisfiability and logical equivalence, because our infinite feature structure models form a particular algebraic model. The purpose of studying countably infinite feature structures is that the collection of them forms a domain in the denotational semantics sense (see Gunter and Scott in press). In fact, the collection of possibly infinite feature structures modulo alphabetic variance form a Scott domain (that is, arbitrary consistent joins and arbitrary meets are well-defined). One nice property of this domain model is that the finite or compact domain elements correspond to the finite feature structures; infinite feature structures can be treated as the limits of their finite approximations. Another nice point about the domain-theoretic model is that it allows solutions to be constructed that correspond to non-terminating computations in the finite case.

Finally, following Pollard and Moshier (1990), we show how to apply the upper powerdomain construction of Smyth (1978) to model disjunction in our logic.

After completing our study of feature structures, we provide three applications to phrase structure grammars, definite clause programs and general constraint resolution systems. Our treatment of phrase structure grammars generalizes context-free grammars to allow feature structures to act as categories. This is the standard treatment of unification grammars in the feature structure literature and dates back to Kay's (1979) Functional Unification Grammar, Kaplan and Bresnan's (1982) Lexical Functional Grammar and the generic PATR-II formalism of Shieber et al. (1983). Because our feature structures can be used to model first-order terms, our treatment of phrase structure grammars also provides a generalization of the logic grammars, which originated with Colmerauer's (1970, 1978) Metamorphosis Grammars. The treatment of logic grammars we provide is most like Pereira and Warren's (1980) Definite Clause Grammars (also see Pereira and Shieber 1987, Gazdar and Mellish 1989, and sources cited therein). We later show that the phrase structure formalism is able to characterize arbitrary recursively enumerable languages, and is thus undecidable. We also consider restrictions put forward by Kaplan and Bresnan (1982), which ensure decidability.

Our second application is to definite clause programs. In this application, we treat feature structures much like the logical terms of a definite clause logic programming language like Prolog (Lloyd 1984, Sterling and Shapiro 1986). In fact, because our feature structures can be used to model logical terms and even Prolog II terms with cycles and inequations, our treatment of definite clause programs reduces to Prolog and Prolog II in the relevant cases. Our treatment does not quite fall into the constraint logic programming paradigm put forward by Jaffar and Lassez (1987) and applied to feature structures by Höhfeld and Smolka (1988). In a constraint logic programming approach, we would take our feature structure description language to provide the constraints. This was the approach adopted by Aït-Kaci and Nasr (1986) in their definite clause programming language LOGIN. Instead of defining definite relations using our attribute-value description language, we treat feature structures as terms themselves. Thus we do not have an analogue of relations at all. Rather, our treatment is much more similar to that of Mukai (1985, 1987, in press, Mukai and Yasukawa 1985), which simply replaces first-order terms with feature structures. The only real difference is that we allow a much more general notion of typed feature structure with inequations and extensionality than was considered by Mukai. We provide the standard analysis of both the operational and denotational semantics of our definite clause programming language (van Emden and Kowalski 1976, Jaffar and Lassez 1987). We show that our system is Turing-complete in the sense that any Turing-computable function can be captured by a definite clause program over the feature structures. The primary benefit of our definite clause system is that it allows inheritance-based reasoning as well as logical reasoning by means of definite clause. It shares this property with LOGIN (Aït-Kaci and Nasr 1986) and also the order-sorted logical programming language developed by Smolka (1988b). The benefit of adding inheritance-based reasoning to

a logic programming language is the same as that of augmenting first-order theorem provers with inheritance (Walther 1985, 1988); expensive inference steps that might otherwise be carried out by structural unification or chains of logical inferences can be replaced by efficient inheritance operations. This partitioning of information between inheritance and logic has become commonplace in artificial intelligence applications of terminological reasoning, beginning with KRYPTON (Brachman et al. 1983).

Our final application, which is to constraint solving, is one that we believe is unique to feature structure-based approaches and is based on the KBL knowledge representation system of Aït-Kaci (1984, 1986) and on the HPSG system of Pollard and Sag (1987). More specifically, our approach is based directly on the denotational semantics of Pollard and Moshier (1990), which formalizes a restricted version of Pollard and Sag's (1987) informal presentation of the mechanisms underlying HPSG. Pollard and Moshier's system is a generalization of Aït-Kaci's KBL system to possibly cyclic structures, and they clean up Aït-Kaci's semantics to the point where their system can be proven complete. We generalize Pollard and Moshier's treatment by allowing types, extensionality specifications, and inequations. The motivation for constraint-based grammars is the overwhelming trend in theoretical linguistics toward grammar formalisms which eschew rule-based analyses of individual constructions, as in a phrase structure grammar, in favor of interacting collections of general constraints, as in Chomsky's (1981, 1988) Government-Binding theory or in HPSG. In the present context, a system of constraints is realized as a set of descriptions of the admissible objects of each type. These descriptions are culled from our general attribute-value descriptive language. A feature structure is a solution to a constraint system if all of its substructures satisfy the constraints placed on their types and supertypes. One nice feature of the constraint system we present is that it can be smoothly integrated with the inheritance hierarchy so that constraints can be placed at the appropriate level of generalization. In some ways, our constraints are similar to the rules in terminological knowledge representation systems such as LOOM (Mac Gregor 1988, 1990) or CLASSIC (Brachman et al. 1991). Rules in these systems correspond to implicational constraints between a concept and a concept description; if an object is determined to match a concept, all of the rules matching the concept are fired in a forward-chaining fashion and their consequences added to the object description. The reason that rules are separated from the other constraints in CLASSIC is that they lead to systems in which classification (subsumption checking) is not only intractable, but undecidable. The type theory we present restricts the well-formed objects of a given type in a very weak way and is easily decidable. Allowing arbitrary descriptions to be attached to types which might involve disjunction, structure sharing, and type restrictions at arbitrary depths in a structure, leads to a system in which arbitrary recursively enumerable languages can be represented. On the other hand, we are able to show how to effectively generate the collection of solutions to an arbitrary constraint system using a method similar to the SLD-resolution technique employed for enumerating the solutions to a query with respect to a definite clause logic program (van Emden and Kowalski 1976).

Part I

Basics

Chapter 2

Types and Inheritance

From the outside, our feature structures look much like the Ψ-terms of Aït-Kaci (1984, 1986) or the feature structures of Pollard and Sag (1987), Moshier (1988) or Pollard and Moshier (1990). In particular, a feature structure is modeled by a possibly cyclic directed graph with labels on all of the nodes and arcs. Each node is labeled with a symbol representing its type, and the arcs are labeled with symbols representing features. We think of our types as organizing feature structures into natural classes. In this role, our types are doing the same duty as concepts in a terminological knowledge representation system (Brachman and Schmolze 1985, Brachman, Fikes, and Levesque 1983, Mac Gregor 1988) or abstract data types in object-oriented programming languages (Cardelli and Wegner 1985). Thus it is natural to think of the types as being organized in an inheritance hierarchy based on their generality. Feature structure unification is then modified so that two feature structures can only be unified if their types are compatible according to the primitive hierarchy of types.

In this chapter, we discuss how type inheritance hierarchies can be specified and the restrictions that we impose on them that allow us to define an adequate notion of type inference, which is necessary during unification. These restrictions were first noted by Aït-Kaci (1984) in his unification-based reasoning system. The polymorphism allowed in our type system is based on inheritance in which a subtype inherits information from all of its supertypes. The possibility of more than one supertype for a given type allows for multiple inheritance.

Type Inheritance Hierarchies

To begin with, we assume that we are dealing with a finite set Type of types ordered according to their specificity, and we think of a type τ as being more specific than another type σ if τ inherits information from σ. We write $\sigma \sqsubseteq \tau$ for types $\sigma, \tau \in \mathsf{Type}$, if τ inherits from σ and say that σ *subsumes* or is more *general* than τ (or inversely, τ is subsumed by or more *specific* than σ). If $\sigma \sqsubseteq \tau$, then σ is a *supertype* of τ, or inversely, τ is a *subtype* of σ.

The standard procedure for specifying subsumption in inheritance-based approaches to knowledge representation has been to allow the specification of a finite number of so-called ISA arcs which link subtypes to supertypes. The full