Feature Constraint Grammars

von

Thilo Götz

Philosophische Dissertation
angenommen von der Neuphilologischen Fakultät
der Universität Tübingen

am 21.12. 1999

Yorktown Heights, NY

2000

Gedruckt mit der Genehmigung der Neuphilologischen Fakultät
der Universität Tübingen

Haupberichterstatter:	Prof. Dr. Erhard Hinrichs
Mitberichterstatter:	Prof. Dr. Uwe Mönnich
Mitberichterstatter:	Priv. Doz. Dr. Fritz Hamm
Mitberichterstatter:	Prof. Dr. Robert T. Kasper (Ohio State University, USA)
Dekan:	Prof. Dr. Bernd Engler

Contents

Acknowledgments

I would like to thank my advisors Erhard Hinrichs and Uwe Mönnich for their advice and encouragement. Without their support after I left Tübingen to take a job with IBM, this dissertation would not be finished today.

I would also like to thank the other members of my committee, Bob Kasper and Fritz Hamm, who worked under extreme time pressure to provide valuable feedback on my dissertation.

Most of the work reported in this thesis was carried out in the context of SFB 340 of the Deutsche Forschungsgemeinschaft. I am grateful to Erhard Hinrichs and Dale Gerdemann for keeping me in bread and butter for several years, and for providing such a stimulating research environment.

I am also grateful to my colleagues at the IBM Watson Research lab for giving me time and encouragement to work on my dissertation: David Johnson, Fred Damerau, Frank Oles and Warren Plath.

A special thanks goes to the people who have had the most profound influence on my way of thinking and working; my fellow students Detmar Meurers and Frank Morawietz, and my teachers Dale Gerdemann and Paul King.

Last not least, I would like to thank Sigrid Beck for encouragement, rock climbing, and other good things.

Chapter 1

Introduction

This thesis is concerned with the logical foundations and computational modeling of constraint-based grammar formalisms. By computational modeling I understand the testing of the empirical predictions of a given grammar on a computer. This subsumes, e.g., the parsing problem: given a grammar and a string, does the grammar predict that the string is grammatical, or ungrammatical? However, I see this only as an instance of a more general problem: given a (partial) description of a linguistic structure (a tree, say) and a grammar, does the grammar predict that linguistic structures of that description exist? And if such structures exist, what do they look like?

The grammar formalism that I'll be looking at specifically is Head-driven Phrase Structure Grammar (HPSG, Pollard and Sag 1994). However, the general points are valid for other feature logic constraint grammar formalisms. It is important to distinguish between *constraint* formalisms (grammar formalisms, programming languages) and *rule-based* formalisms. The constraint-based idea is that any structure is well-formed according to the grammar or program *unless* it is specifically ruled out by a constraint. A constraint grammar is thus a set of constraints that *rule out non well-formed (ungrammatical) structures*. In a rule-based formalisms, on the other hand, some structure is well-formed *only* if generated by some rule in the grammar (or program). A rule-based grammar is thus a set of rules that *generate the well-formed (grammatical) structures*. I will discuss the differences between the two approaches in more detail in Chapter 3. There I will also show that HPSG is to be understood as a constraint formalism in that sense.

HPSG grammars are expressed as sets of implications in an attribute-value language. Attribute-value expressions have a history that is almost as long as modern linguistics itself. Their more recent introduction into computational linguistics is usually credited to Kay (1979). However, it was not until much later that the first investigation into the formal properties of such systems was published (Rounds and Kasper 1986). In their seminal work, Rounds and Kasper (1986) defined what came to be known as a Rounds-Kasper calculus. They defined attribute-value expressions as a formal language with a semantics based on partial record structures. Partial record structures were used at the time as a device to explicate the semantics of records in denotational semantics of programming languages.

A few years later, Smolka (1988) proposed a different approach to the semantics of attribute-value expression. His work was based on classical model-theory of first-order logic. This split the researchers into two opposing camps: the partial objects camp and the classical logic camp. The classical logic followers argue that since a subset of first-order logic is sufficient to account for the semantics of attribute-value expressions, this is the natural way to go. Since first-order logic is very well understood, it will serve as an excellent basis for feature logic. According to this view, the tools of denotational semantics, developed to account for the semantics of imperative programming languages, are complete overkill for a simple problem like the semantics of attribute-value expressions. Among the proponents of this view are Smolka (1988), Johnson (1988) and King (1989). The partial objects followers, on the other hand, argue that since we're very much interested computation, we should avail ourselves of the powerful domain-theoretic tools developed in the denotational semantics community. Proponents of this view include Rounds and Kasper (1986), Carpenter, Pollard, and Franz (1991) and Carpenter (1992).

This thesis falls into the classical logic camp. It is one of the main goals of this thesis to show that with a classical semantics for feature logic, one can compute as well as or better than with partial objects-based account. However, most of the time, the differences are minimal and only of philosophical or aesthetic interest. Where there are major differences between the approaches, I will be careful to point them out.

The more or less informal notions of HPSG were given a foundation based on classical logic in King (1994). Yet although the logical notions are clear enough, there is no operational semantics to go with the logic. Thus, in NLP practice, HPSG-based grammars are implemented as phrase structure grammars or (constraint) logic programs. HPSG principles are then encoded as relations (e.g., ID-schemata) or added as feature logic constraints to relations wherever appropriate (e.g., the Head-Feature Principle). The translation from HPSG principles to rules and constraints needs to be done by hand.

From a linguistic perspective, this is an undesirable state of affairs. Linguistic theories can not be tested directly, they have to be recoded into something which bears a more or less close similarity to the original theory. Moreover, this has to be done by hand. In this thesis, I propose a computational mechanism that works directly on HPSG grammars.

The thesis is organized as follows:

- Chapter 2 introduces our feature logic and examines its properties. I will relate my approach to other feature logics and examine possible alternatives. In the computational part of the chapter, I will show how the descriptive logic can be understood as a constraint logic. I will provide a terminating rewrite system for constraint set normalisation, thus showing that the satisfiability problem for our logic is decidable.

- Chapter 3 considers the notion of grammar and how a grammar defines the well-formed linguistic structures. We call this the prediction problem. Prediction will be shown to be fundamentally different for constraint and rule-based grammars. We examine the abstract computational properties of prediction by giving an undecidability result and by showing its relation to problems of first-order predicate logic. The last part of the chapter will be dedicated to defining a normal form for grammars that makes them more amenable to automatic processing.

- Chapter 4 defines a translation from feature constraint grammars to constraint logic programs, i.e., a rule-based grammar formalism. The soundness and completeness properties of this translation will make the relation between the two approaches more transparent.

- Chapter 5 defines the procedural interpretation of HPSG grammars. We show that our method is complete for the negative case, i.e., when the grammar does not admit a given term. The results of ch. 3 tell us that we can't do any better.

- In Chapter 6, we consider extending our logic with relational constructs. Relations can be defined in a clausal language similar to constraint logic programming. We consider the semantic issues involved and extend our procedural semantics to deal with the added complexity. As it turns out, we obtain the same completeness results as for the simpler logic without relations.

The theoretical ideas presented in this thesis have been implemented to a large extent in the ConTroll system in the context of a research project "Efficient Compilation of HPSG Grammars" at the University of Tübingen, Germany. This project was part of the SFB 340, funded by the German Research Council (Deutsche Forschungsgemeinschaft). The effort was lead by Erhard Hinrichs and Dale Gerdemann. Contributors to ConTroll include: Björn Aldag, Natali Alt, Dale Gerdemann, John Griffith, Carsten Hess, Detmar Meurers, Guido Minnen, Frank Morawietz and Stephan Kepser.

In the final chapter, I will detail which parts of the thesis have been implemented, and which not. See (Götz and Meurers 1997a) and (Götz et al. 1997) for more background on ConTroll.

Chapter 2

Feature terms and feature constraints

2.1 Introduction

In this chapter, I will discuss different *feature description languages*, the basic component of the grammar architecture(s) to be introduced in later chapters. There are many different feature description languages, but they all have some points in common.

- *Features* (or attributes) are used to denote the fact that objects have certain properties. In the notation I will use, the expression $f : \phi$ describes all objects that have property f (with value ϕ).
- A notion of *reentrancy* (or structure sharing) is used to denote the fact that certain (distinct) properties of some object have identical values. For example, $f : X \wedge g : X$ describes all objects that have the properties f and g, whose values must be identical.
- *Types* (or sorts) are often used to classify objects.

I will consider alternative languages in Sec. 2.4, also from a historical perspective. In Sec. 2.2, I will introduce the logic that will be used throughout the rest of this thesis. An important computational problem for our description language is satisfiability. In Sec. 2.3, I will give a terminating algorithm to solve this problem, thus showing it is decidable.

The use of feature-value pairs in linguistics and computational linguistics has a long history. It is probably fair to say that it is as old as modern linguistics itself. Some versions of feature-value pairs are used in many, if not most, theories of phonology, morphology, and syntax. For example, in syntax, many authors represent a syntactic category as a combination of the features N and V, both of which can take $+$ or $-$ as value. So instead of noun, verb, preposition and adjective, we use $\langle N : +, V : -\rangle$, $\langle N : -, V : +\rangle$, $\langle N : -, V : -\rangle$ and $\langle N : +, V : +\rangle$, respectively. Now suppose some grammatical constraints apply to both verbs and adjectives. Since we can talk about a category $\langle V : +\rangle$, the fact that verbs and adjectives fall into a natural class is built into our representation. This technique is called underspecification. We simple don't talk about the V part of the category. Underspecification should be contrasted with disjunction, where we explicitly list all possibilities. So instead of $\langle V : +\rangle$, we could also say $verb \vee adjective$. For the finite case, disjunction and underspecification are clearly equivalent, and it is a question of aesthetics which one prefers. There are, however, cases where underspecification can finitely express an infinite number of possibilities, which can not be achieved with a (finite) disjunction.

It computational linguistics, the use of feature-value pairs may be traced back to Martin Kay's Functional Grammar (FG, Kay (1979)). The first logic-based approach to feature-value pairs is generally attributed to (Rounds and Kasper 1986). Their contribution was the insight that there should be a strict separation between a *description language* and the *models* of that language. However, their model theory was not based on classical logic. Instead, they chose as models finite partial graph structures. The motivation for this step would appear to be mainly computational.

The partial models can be seen to stand in a subsumption ordering, with the more general (i.e., unspecific) models subsuming more general ones. It was relatively simple to find a (finite set of) most general satisfier(s) for a given description. That is, a model that is more general than all other models satisfying the description. Conversely, it is also the case that every model subsumed by the most general satisfier also satisfies the description. Furthermore, given two descriptions A and B, we can find the most general satisfier for $A \wedge B$ by finding the most general satisfiers for A and B, and then finding the most general model that is subsumed by both of them. This operation is generally called unification and can be efficiently computed.

In the late 80s, Smolka (1988) and King (1989) independently realized a model theory based on classical first order logic. This split the feature logic camp into two halves. One half maintained that one should stick with partial models, since they worked well for computation. The other half claimed that one could do computation with a classical description language just as well. Furthermore, a classical model theory gave one access to the host of results from that field, without having to reinvent the wheel. I will argue for the latter position in this dissertation.

Types[1] were, to the best of my knowledge, introduced into feature logic with the first HPSG book (Pollard and Sag 1987). King (1989) gave the first formal account of a typed feature logic. Computation with a typed feature logic was first considered in (Carpenter, Pollard, and Franz 1991), and later in Carpenter's book (Carpenter 1992).

The logic I will introduce in the next section is not yet another feature logic. The contribution is to show how to compute with a logic with a standard classical model theory.

2.2 A feature term language: $\mathcal{FL}$

The logic $\mathcal{FL}$ described in this section is based on those of (King 1994), (Smolka 1992) and (Dörre 1994). We begin by defining what the interpretation structures of our language are. The basic properties of the language and its interpretations are fixed in a logical signature. As usual, the signature defines the non-logical symbols of the language. In our case, those are a set of type symbols, and a set of feature symbols. The type symbols are arranged into a finite join semi-lattice (i.e., a partial order with a greatest symbol, usually denoted as $\top$). Particular importance is attached to those types that do not subsume any other types. Since we have to refer to this subset very often, we use a special symbol for it: $\mathcal{V}$.

Definition 2.1 (signature)
A signature is a quadruple $\langle \mathcal{T}, \preceq, \mathcal{F}, approp \rangle$ *s.t.*

- $\mathcal{T}$ *is a finite set of types, and* $\langle \mathcal{T}, \preceq \rangle$ *is a join semi-lattice,*
- $\mathcal{V} = \{t \in \mathcal{T} \mid \textit{if } t' \preceq t \textit{ then } t' = t\}$ *is the set of minimal types,*

[1] Some people say types, others say sorts. Given that the distinction is not very clear-cut to begin with, it doesn't really matter. One might argue that sorts are subsets of the domain, whereas types can also be higher-order. Since we do not talk about higher-order types, we might just as well use the term sorts. However, we will stick with types, since that is what most people use.

- *$\mathcal{F}$ is a finite set of feature names, and*
- *approp* : $\mathcal{V} \times \mathcal{F} \rightharpoonup \mathcal{T}$ *is a partial function from pairs of minimal types and features to types*

The join semi-lattice $\langle \mathcal{T}, \preceq \rangle$ is called the *type hierarchy*. The whole signature is usually depicted graphically as a graph structure. An example is shown in Fig. 2.1. In this example, we use SMALL CAPS FONT for feature names and *lower case italics* for type names.

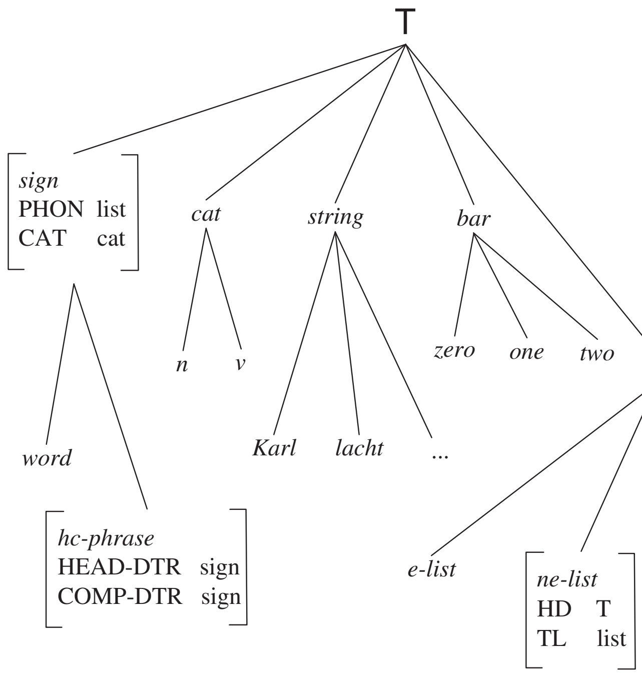


Figure 2.1: An example signature

Clearly, we can read off all the information of the signature from such a graph. The *approp* function is shown in attribute value matrix (AVM) notation with the respective types. In the example, *approp*(*word*,HD) is undefined, whereas *approp*(*word*,PHON) is defined, and *approp*(*word*, PHON) = *list*. Notice that in Def. 2.1, *approp* was only defined for types in $\mathcal{V}$, but the example shows appropriate features for the type *sign*, which is not in $\mathcal{V}$. This is so because intuitively, we would like to provide non-redundant information in the graph. The definition of *approp* on the set of minimal types naturally gives rise to a function *approp**, defined for all types. We will use the following notation: if f is a partial function, then we write $f(x)\downarrow$ for "f is defined on x" and $f(x)\uparrow$ for "f is undefined on x".

Definition 2.2 (*approp)**
Let $\langle \mathcal{T}, \preceq, \mathcal{F}, approp\rangle$ *be a signature.* $approp^*(t,f)\downarrow$ *iff for all* $t' \in \mathcal{V}$, *if* $t' \preceq t$, *then* $approp(t',f)\downarrow$. *If defined, then* $approp^*(t,f) = \bigvee\{t'' \mid t' \in \mathcal{V},\ t' \preceq t \text{ and } approp(t',f) = t''\}$.

We can thus depict *approp** in our graphs, and we can furthermore restrict ourselves to showing non-redundant information only, leaving the readers to infer inherited appropriateness themselves.

We're now ready to define interpretation structures. Interpretations are simply sets of objects, together with functions that interpret the features, and a function that assigns a type to each object. The functions interpreting the features have to respect the appropriateness conditions set forth in the signature.

Definition 2.3 (interpretation)
Let $S = \langle \mathcal{T}, \preceq, \mathcal{F}, approp\rangle$ *be a signature. An S-interpretation is a triple* $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}\rangle$ *s.t.*

- $\mathcal{U}$ *is a set of objects, the domain of* $\mathcal{I}$
- $\mathcal{S} : \mathcal{U} \to \mathcal{V}$ *is a total function from the set of objects to the set of minimal types*
- $\mathcal{A} : \mathcal{F} \to \mathcal{U}^{\mathcal{U}}$ *is an attribute interpretation function s.t.*
  - *for each* $u \in \mathcal{U}$, *for each* $f \in \mathcal{F}$, *if* $approp(\mathcal{S}(u), f)$ *is defined and* $approp(\mathcal{S}(u), f) = t$, *then* $\mathcal{A}(f)(u)$ *is defined and* $\mathcal{S}(\mathcal{A}(f)(u)) \preceq t$
  - *for each* $u \in \mathcal{U}$, *for each* $f \in \mathcal{F}$, *if* $\mathcal{A}(f)(u)$ *is defined, then* $approp(\mathcal{S}(u), f)$ *is defined and* $\mathcal{S}(\mathcal{A}(f)(u)) \preceq approp(\mathcal{S}(u), f)$

The function $\mathcal{V}$ associates a *minimal* type with every object. This is different in alternative logics to be discussed in Sec. 2.4. Note also how the attribute interpretation function $\mathcal{A}$ is restricted by *approp*. If some object u is of (minimal) type t, then $\mathcal{A}(f)(u)$ is defined iff $approp(t, f)$ is defined, and the object $\mathcal{A}(f)(u)$ must be of appropriate type. Let us consider an example, which we will also depict as a graph (Figure 2.2 on p. 14).

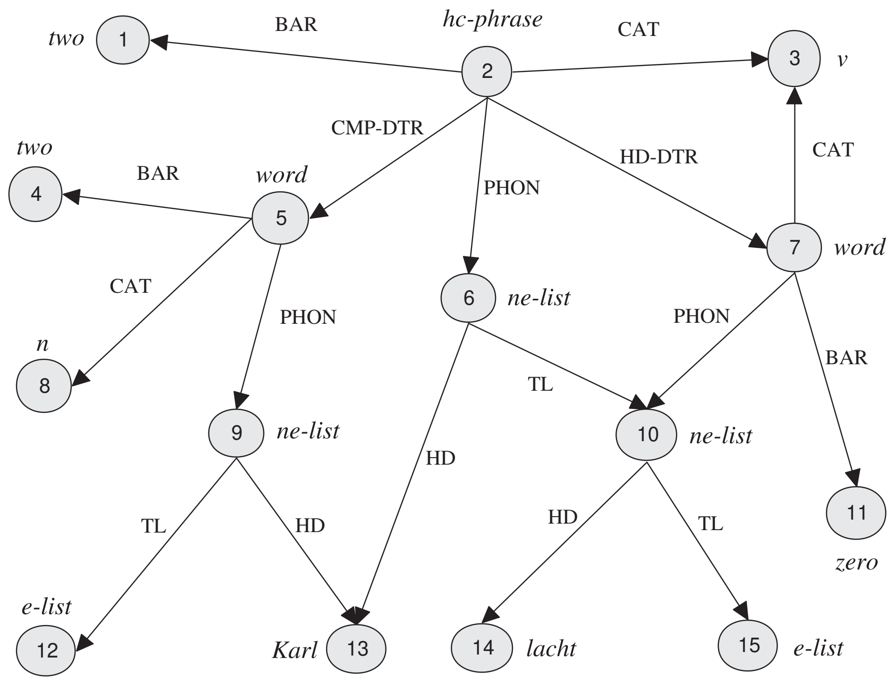


Figure 2.2: An example interpretation

The objects are the nodes of the graph, and the labeling of the nodes indicates the typing function: object 1 is of type *two*, object 2 is of type *hc-phrase*, and so on. The edges of the graph show the attribute interpretation function. $\mathcal{A}(\text{BAR})$ maps object 2 to object 1, 5 to 4 and 7 to 11. Notice that the objects that $\mathcal{A}(\text{BAR})$ is defined on are exactly those of type sign.

We now turn to the definition of our formal description language. We will assume a countably infinite set VAR of variables. Variable names will always be *UPPER CASE ITALICS* letters. Feature terms are built from the symbols defined in a signature, variables, and the logical connectives $\wedge$, $\vee$ and $\neg$.

Definition 2.4 (S-terms)
Let $S = \langle \mathcal{T}, \preceq, \mathcal{F}, approp \rangle$ *be a signature.*

- *X is a S-term* — *if $X \in$ VAR*
- *t is a S-term* — *if $t \in \mathcal{T}$*
- *$f : \phi$ is a S-term* — *if $f \in \mathcal{F}$ and ϕ is a S-term.*
- *$\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\neg\phi_1$ are S-terms* — *if ϕ_1 and ϕ_2 are S-terms.*

We leave out reference to the signature if it is clear from the context. Additionally, we allow the term $\phi_1 \rightarrow \phi_2$ to stand for $\neg\phi_1 \vee \phi_2$. Parentheses are used to disambiguate terms. In the absence of parentheses, we use the following operator precedence conventions (in decreasing order):

$$: \quad \neg \quad \wedge \quad \vee \quad \rightarrow$$

As mentioned above, feature terms denote sets of objects in an interpretation. There is no explicit quantification, variables are generally assumed to be existentially quantified. Feature terms are interpreted using a variable assignment function.

Definition 2.5 (variable assignment)
Let $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ be an interpretation. A variable assignment is a total function $\alpha :$ VAR $\rightarrow \mathcal{U}$. Write ASS *for the set of all assignments for some given interpretation.*

Next, we define term interpretation.

Definition 2.6 ($[\![\cdot]\!]^{\mathcal{I}}_{\alpha}$)
Term interpretation is defined with respect to an interpretation $\mathcal{I}$ and a variable assignment $\alpha \in$ ASS:

- $[\![X]\!]^{\mathcal{I}}_{\alpha} = \{\alpha(X)\}$ — *if $X \in$ VAR*
- $[\![t]\!]^{\mathcal{I}}_{\alpha} = \{u \in \mathcal{U} \mid \mathcal{S}(u) \preceq t\}$ — *if $t \in \mathcal{T}$*
- $[\![f : \phi]\!]^{\mathcal{I}}_{\alpha} = \{u \in \mathcal{U} \mid (\mathcal{A}(f))(u)\downarrow \wedge (\mathcal{A}(f))(u) \in [\![\phi]\!]^{\mathcal{I}}_{\alpha}\}$
- $[\![\phi_1 \wedge \phi_2]\!]^{\mathcal{I}}_{\alpha} = [\![\phi_1]\!]^{\mathcal{I}}_{\alpha} \cap [\![\phi_2]\!]^{\mathcal{I}}_{\alpha}$
- $[\![\phi_1 \vee \phi_2]\!]^{\mathcal{I}}_{\alpha} = [\![\phi_1]\!]^{\mathcal{I}}_{\alpha} \cup [\![\phi_2]\!]^{\mathcal{I}}_{\alpha}$
- $[\![\neg\phi]\!]^{\mathcal{I}}_{\alpha} = \mathcal{U} \backslash [\![\phi]\!]^{\mathcal{I}}_{\alpha}$

We're often not interested in the denotation of a term with respect to a specific variable assignment. Instead, we want to know which objects can be described by a term given a *suitable* variable assignment. We thus abstract away from the variable assignment.

Definition 2.7 ($[\![\cdot]\!]^{\mathcal{I}}$)
$[\![\phi]\!]^{\mathcal{I}} = \bigcup_{\alpha \in \mathrm{ASS}} [\![\phi]\!]^{\mathcal{I}}_{\alpha}$

Let $\mathcal{I}$ be the interpretation defined in Fig. 2.2. We then get the equalities shown in Fig. 2.3.

$$
\begin{array}{rcl}
[\![\mathit{word}]\!]^{\mathcal{I}} & = & \{5,7\} \\
[\![\mathit{sign}]\!]^{\mathcal{I}} & = & \{2,5,7\} \\
[\![\textsc{cat} : \mathit{v}]\!]^{\mathcal{I}} & = & \{2,7\} \\
[\![\mathit{word} \wedge \textsc{bar} : \mathit{two}]\!]^{\mathcal{I}} & = & \{5\} \\
[\![\mathit{word} \vee \textsc{bar} : \mathit{two}]\!]^{\mathcal{I}} & = & \{2,5,7\} \\
[\![\neg \mathit{word}]\!]^{\mathcal{I}} & = & \{1-4,6,8-15\} \\
[\![\textsc{bar} : \neg \mathit{two}]\!]^{\mathcal{I}} & = & \{7\} \\
[\![\textsc{cat} : X \wedge \textsc{head-dtr} : \textsc{cat} : X]\!]^{\mathcal{I}} & = & \{2\} \\
[\![\mathit{word} \wedge \mathit{hc\text{-}phrase}]\!]^{\mathcal{I}} & = & \emptyset \text{ (inconsistent)}
\end{array}
$$

Figure 2.3: Example term denotations

We conclude this section with a definition of what it means for two feature terms to be equivalent.

Definition 2.8 (equivalence of feature terms)
For each feature term ϕ, ψ, $\phi \equiv \psi$ *iff for each interpretation* $\mathcal{I}$, $[\![\phi]\!]^{\mathcal{I}} = [\![\psi]\!]^{\mathcal{I}}$.

2.3 Checking satisfiability: constraint solving

We can now consider the first practical problem: determining if a given term is satisfiable or not.

Definition 2.9
A feature term ϕ *is satisfiable iff there is an interpretation* $\mathcal{I}$ *s.t.* $[\![\phi]\!]^{\mathcal{I}} \neq \emptyset$.

This problem has been shown to be decidable for several languages closely related to one presented here. The trick is to bring a term into some normal form that displays (un)satisfiability. Since later chapters will depend on what exactly the normal form looks like, we will investigate this in some detail.

Following (Smolka 1988), the transformation of a term into normal form proceeds in three steps. First, the term is brought into disjunctive normal form, pushing negation down to variables. This term in disjunctive normal form is then transformed into a set of sets of what Smolka called *feature constraints*. In a last step, these sets will then be brought into a normal form displaying (un)satisfiability. Furthermore, we can compact the normal form down into a smaller but equivalent representation. This is called *unfilling*. We begin with a definition of what we mean by disjunctive normal form (DNF).

Definition 2.10 (DNF)
We say that a feature term is a simple term iff it contains no disjunction and the only occurrence of negation is in front of variables. A feature term is in DNF iff it is a disjunction of simple terms.

Using the following equivalences, we can eliminate negation from our terms everywhere except in front of variables. We will use $\bot$ as an abbreviation[2] for $X \wedge \neg X$, for some variable X. This is the simplest way of expressing inconsistency in our language.

- $\neg\neg\phi \equiv \phi$
- $\neg(\phi \wedge \psi) \equiv (\neg\phi \vee \neg\psi)$
- $\neg(\phi \vee \psi) \equiv (\neg\phi \wedge \neg\psi)$
- $\neg t \equiv \begin{cases} \bigvee\{t' \mid t' \in \mathcal{V} \wedge t' \not\preceq t\} & \text{if } \{t' \mid t' \in \mathcal{V} \wedge t' \not\preceq t\} \neq \emptyset \\ \bot & \text{otherwise} \end{cases}$
- $\neg(f{:}\,\phi) \equiv \begin{cases} f{:}\,\neg\phi, \ \textit{if } \{t \mid t \in \mathcal{V}, \textit{approp}(t,f)\uparrow\} = \emptyset \\ f{:}\,(\neg\phi) \vee \bigvee\{t \mid t \in \mathcal{V}, \textit{approp}(t,f)\uparrow\}, \ \textit{otherwise} \end{cases}$

[2]Note that there is no $\bot$ symbol in the type hierarchy. We use it simply to denote a false or inconsistent term. $\top$, on the other hand, is both a type symbol and our way of denoting a necessarily true term. This is because, by our definition of a signature and its interpretation, the *term* $\top$ will always be true, under any interpretation and assignment.

The second to last equivalence needs to be so complicated because the empty disjunction is not part of our language. As usual, the empty disjunction is inconsistent.

The last equivalence is not completely obvious. From the definition of term interpretation it is clear what an expression of the form $\neg(f\colon\phi)$ must denote: it's true of all objects that don't have f defined on them or that have f defined and mapping onto an object not in the denotation of ϕ. We can describe all objects that don't have f defined on them with the disjunction of all minimal types t s.t. $\mathit{approp}(t, f)$ is undefined. That gives us the following result.

Proposition 2.1
Let $S = \langle \mathcal{T}, \preceq, \mathcal{F}, \mathit{approp}\rangle$ be a signature s.t. $\exists t \in \mathcal{V}.\ \mathit{approp}(t, f)\uparrow$. For every S-interpretation $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}\rangle$,

$$[\![\neg f : \phi]\!]^{\mathcal{I}} = [\![f\colon(\neg\phi) \vee \bigvee\{t \mid \mathit{approp}(t, f)\uparrow\}]\!]^{\mathcal{I}}$$

Proof

$$\begin{aligned}
[\![\neg f\colon\phi]\!]^{\mathcal{I}} &= \bigcup_{\alpha\in\mathrm{ASS}} [\![\neg f\colon\phi]\!]^{\mathcal{I}}_{\alpha} \\
&= \bigcup_{\alpha\in\mathrm{ASS}} \mathcal{U} \setminus [\![f\colon\phi]\!]^{\mathcal{I}}_{\alpha} \\
&= \bigcup_{\alpha\in\mathrm{ASS}} \mathcal{U} \setminus \{u \in \mathcal{U} \mid \mathcal{A}(f)(u)\downarrow \wedge \mathcal{A}(f)(u) \in [\![\phi]\!]^{\mathcal{I}}_{\alpha}\} \\
&= \bigcup_{\alpha\in\mathrm{ASS}} (\{u \in \mathcal{U} \mid \mathcal{A}(f)(u)\uparrow\} \cup \{u \in \mathcal{U} \mid \mathcal{A}(f)(u) \notin [\![\phi]\!]^{\mathcal{I}}_{\alpha}\}) \\
&= \bigcup_{\alpha\in\mathrm{ASS}} (\{u \in \mathcal{U} \mid \mathit{approp}(\mathcal{S}(u), f)\uparrow\} \cup \{u \in \mathcal{U} \mid \mathcal{A}(f)(u) \in [\![\neg\phi]\!]^{\mathcal{I}}_{\alpha}\}) \\
&= \bigcup_{\alpha\in\mathrm{ASS}} (\bigcup\{[\![t]\!]^{\mathcal{I}}_{\alpha} \mid \mathit{approp}(t, f)\uparrow\} \cup [\![f\colon\neg\phi]\!]^{\mathcal{I}}_{\alpha}) \\
&= \bigcup_{\alpha\in\mathrm{ASS}} [\![f\colon(\neg\phi) \vee \bigvee\{t \mid \mathit{approp}(t, f)\uparrow\}]\!]^{\mathcal{I}}_{\alpha} \\
&= [\![f\colon(\neg\phi) \vee \bigvee\{t \mid \mathit{approp}(t, f)\uparrow\}]\!]^{\mathcal{I}}
\end{aligned}$$

■

We thus know that every term can be transformed into an equivalent term where negation occurs only in front of variables in a finite number of steps. Using the distributive laws and the fact that $f\colon(\phi \vee \psi) \equiv f\colon\phi \vee f\colon\psi$, we can now bring such a term into disjunctive normal form.

2.3.1 Variables and negation

We digress at this point to reflect on the interaction of our quantifier-less approach to variables, and negation. The unrestricted use of variables and negation has undesired consequences. Although we would like negation to behave classically, this is not the case in our system.

Proposition 2.2
It is not the case that for all interpretations $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$, *for all terms* ϕ, $[\![\neg\phi]\!]^{\mathcal{I}} = \mathcal{U} \setminus [\![\phi]\!]^{\mathcal{I}}$

Since we've defined $[\![\neg\phi]\!]^{\mathcal{I}}_{\alpha} = \mathcal{U} \setminus [\![\phi]\!]^{\mathcal{I}}_{\alpha}$, this can only have to do with variable assignments, i.e., variables.

Example 2.1
Let $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ *be an interpretation s.t.* $|\mathcal{U}| \geq 2$ *and* X *be a variable. We have the following:*

$$[\![X]\!]^{\mathcal{I}} \;=\; \bigcup_{\alpha \in ASS} [\![X]\!]^{\mathcal{I}}_{\alpha} \;=\; \{\alpha(X) \mid \alpha \in \mathrm{ASS}\} \;=\; \mathcal{U},$$

but also

$$[\![\neg X]\!]^{\mathcal{I}} \;=\; \bigcup_{\alpha \in ASS} [\![\neg X]\!]^{\mathcal{I}}_{\alpha} \;=\; \bigcup_{\alpha \in ASS} \mathcal{U} \setminus \{\alpha(X)\} \;=\; \mathcal{U}$$

Notice that for this equation to hold, $\mathcal{U}$ *has to contain at least two elements.*

The easiest way out of this is to disallow negative occurrences of variables, or even make do without variables. As (King 1989) shows, there are no logical reasons to use variables, everything can be expressed using path equations. However, variables are a useful notational convenience, and some form of variables is necessary when using relations, as we will do later on. We will therefore describe a class of feature terms where negation has the desired effect. This is done by restricting the use of variables, in a way that arguably does not restrict the expressive power of the formalism. We begin by augmenting our language with *path equations* and *path inequations*.

Definition 2.11 (path equations and inequations)

1. *A path is a member of* $\mathcal{F}^*$. *Write* ε *for the empty path.*

2. *If* π_1, π_2 *are paths, then* $\pi_1 \approx \pi_2$ *is a feature term (path equation).*

3. *If* π_1, π_2 *are paths, then* $\pi_1 \not\approx \pi_2$ *is a feature term (path inequation).*

Definition 2.12 (path interpretation)
If $\langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ *is an interpretation, then*

1. $\mathcal{A}^*(\varepsilon) = id_{\mathcal{U}}$*, the identity function on* $\mathcal{U}$*.*
2. $\mathcal{A}^*(f\pi) = \mathcal{A}(f) \circ \mathcal{A}^*(\pi)$

Definition 2.13 (denotations of path equations and inequations)

1. $[\![\pi \approx \pi']\!]^{\mathcal{I}}_{\alpha} =$
   $\{u \in \mathcal{U} \mid \mathcal{A}^*(\pi)(u)\!\downarrow,\ \mathcal{A}^*(\pi')(u)\!\downarrow \text{ and } \mathcal{A}^*(\pi)(u) = \mathcal{A}^*(\pi')(u)\}$
2. $[\![\pi \not\approx \pi']\!]^{\mathcal{I}}_{\alpha} =$
   $\{u \in \mathcal{U} \mid \mathcal{A}^*(\pi)(u)\!\downarrow,\ \mathcal{A}^*(\pi')(u)\!\downarrow \text{ and } \mathcal{A}^*(\pi)(u) \neq \mathcal{A}^*(\pi')(u)\}$

It is intuitively clear that feature terms that can be equivalently expressed without the use of variables, i.e., using only path equations and inequations, behave classically with respect to negation. We will first show that this is indeed so, and then define a class of feature terms that are "safe" to use in this sense.

Definition 2.14 (FV)
We write $\mathrm{FV}(\phi)$ *for the set of free variables in* ϕ*.*

Of course, there are not bound variables in our terms. Thus, the term "free variables" will always mean the set of all variables.

We give the next proposition without its easy proof.

Proposition 2.3
For every interpretation $\mathcal{I}$*, for every term* ϕ*, for every assignment* α *and* β*:*

If for every $X \in \mathrm{FV}(\phi)$*,* $\alpha(X) = \beta(X)$*, then* $[\![\phi]\!]^{\mathcal{I}}_{\alpha} = [\![\phi]\!]^{\mathcal{I}}_{\beta}$*.*

Corollary 2.4
If $\mathrm{FV}(\phi) = \emptyset$*, then for each* $\alpha \in \mathrm{ASS}$*,* $[\![\phi]\!]^{\mathcal{I}} = [\![\phi]\!]^{\mathcal{I}}_{\alpha}$

Corollary 2.5
If $\mathrm{FV}(\phi) = \emptyset$*, then for each interpretation* $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$*,* $[\![\neg\phi]\!]^{\mathcal{I}} = \mathcal{U} \setminus [\![\phi]\!]^{\mathcal{I}}$

Definition 2.15 (var-terms)

1. *For each variable X, X is a (positive) var-term.*

2. *For each variable X, $\neg X$ is a (negative) var-term.*

3. *If ϕ is a (positive/negative) var-term and f is a feature, then $f{:}\phi$ is a (positive/negative) var-term.*

4. *Nothing else is a var-term.*

Definition 2.16 (VNF)
$\psi = \bigvee \psi_i$ is in variable normal form (VNF) iff each $\psi_i = (\psi_{i_1} \wedge \psi_{i_2})$ s.t. ψ_{i_1} is a conjunction of var-terms and $\mathrm{FV}(\psi_{i_2}) = \emptyset$.

Proposition 2.6
Let ϕ be in DNF. For each disjunct ϕ_i in ϕ, there is a $\psi_i = (\psi_{i_1} \wedge \psi_{i_2})$ s.t. ψ_{i_1} is a conjunction of var-terms, $\mathrm{FV}(\psi_{i_2}) = \emptyset$ and $\phi_i \equiv \psi_i$.

Proof Using the fact that $f{:}(\phi \wedge \psi) \equiv f{:}\phi \wedge f{:}\psi$, we can bring each disjunct into the required format. ■

Definition 2.17 (negation safe terms)
ϕ in VNF is called negation safe iff for each disjunct ϕ_i in ϕ and each variable X, if X occurs negatively in ϕ_i, then it also occurs positively in ϕ_i.

Proposition 2.7
If ϕ is negation safe, then there is a term ψ s.t. $\mathrm{FV}(\psi) = \emptyset$ and $\phi \equiv \psi$.

Proof Let $\phi = \phi_1 \vee \ldots \vee \phi_l$, where for each ϕ_i, $\phi_i = (\phi_{i1} \wedge \phi_{i2} \wedge \phi_{i3})$ s.t. ϕ_{i1} is a conjunction $\phi_{i1,1} \wedge \ldots \wedge \phi_{i1,n}$ of positive var-terms, ϕ_{i2} is a conjunction $\phi_{i2,1} \wedge \ldots \wedge \phi_{i2,m}$ of negative var-terms and $\mathrm{FV}(\phi_{i3}) = \emptyset$. For each ϕ_i, define $\psi_i = \bigwedge\{\pi \approx \pi' \mid 1 \leq j, k \leq n,\ \phi_{i1,j} = \pi : X,\ \phi_{i1,k} = \pi' : X\} \wedge \bigwedge\{\pi \not\approx \pi' \mid 1 \leq j \leq n,\ 1 \leq k \leq m,\ \phi_{i1,j} = \pi : X,\ \phi_{i2,k} = \pi' : \neg X\} \wedge \phi_{i3}$. Clearly, $\phi_i \equiv \psi_i$. Thus, we get that $\psi = \bigvee_{1 \leq i \leq l} \psi_i$. ■

Unfortunately, this does not mean that every negation safe term can itself be negated with the expected result. It only means that an equivalent term exists that can be negated. A reasonable example would be the term $f{:}X \wedge g{:}X$. When negated, it turns out to be[3]

$$\begin{aligned} \neg(f{:}X \wedge g{:}X) &\equiv \neg f{:}X \vee \neg g{:}X \\ &\equiv f\uparrow \vee f{:}\neg X \vee g\uparrow \vee g{:}\neg X \\ &\equiv \top \end{aligned}$$

[3] We write $f\uparrow$ for $\bigvee\{t \mid t \in \mathcal{V} \wedge \mathit{approp}(t, f)\uparrow\}$

for all interpretations with more than one domain object. However, we also have that

$$f{:}X \wedge g{:}X \;\equiv\; f \approx g \quad \text{and} \quad \neg f \approx g \;\equiv\; f{\uparrow} \;\vee g{\uparrow} \;\vee f \not\approx g$$

which is exactly the desired result. This example should also make it clear that it is really the absence of explicit quantification that causes the problem. Suppose we encoded our feature descriptions as first order logic formulae. For our example above, we would then get something like $\exists x, y.\ f(y) = x \wedge g(y) = x$. The negation of this sentence is $\forall x, y. \neg\ f(y) = x \vee \neg\ g(y) = x$ which, if we reformulate it slightly as $\forall x, y.\ f(y) = x \rightarrow \neg\ g(y) = x$, can be seen to posses the intended meaning. What we actually get, however, when negating $f{:}X \wedge g{:}X$ is something like $\exists x, y. \neg(f(y) = x \wedge g(y) = x)$. Because all our variables are implicitly quantified from the outside, our negation turns up *inside* the existential quantification and provides us with unintended meanings.

To conclude, we have shown that we can freely use negation, as long as the result is negation safe. We didn't show that if some term is not negation safe, then it has an undesirable denotation. However, we conjecture that this is the case. The problem is that the denotation of an isolated negated variable is $[\![\neg X]\!]^{\mathcal{I}} = \mathcal{U}$ if $|\mathcal{U}| \geq 2$, and $[\![\neg X]\!]^{\mathcal{I}} = \emptyset$ if $|\mathcal{U}| < 2$. There is no term that has exactly the complementary denotation.

This concludes the digression.

2.3.2 Feature constraints

We now turn back to the question of how to determine satisfiability of feature terms. As we saw in the previous section, if a term is negation safe, we can transform it into one without variables. Since our language without variables is a notational variant of the logic defined in (King 1989), we can use the decision algorithm of (Kepser 1994) to test for satisfiability. However,

- this doesn't work for terms that are not negation safe,
- we also want to talk about the notion of unfilling (Götz 1994), which is not considered in Kepser's work, and
- we're not satisfied with a theoretical decidability result. We want a normal form that's at least close to what's actually being used in the implementation.

Thus, we will devise a new normal form and decision algorithm, combining (Smolka 1988), (Carpenter 1992), (Götz 1994) and (Kepser 1994). We start by defining the notion of *feature constraints*, following (Smolka 1988).

Definition 2.18 (feature constraints)
A feature constraint is of the form $X \,|\, Y$, $X \,|\, \neg Y$, $X \,|\, t$ *or* $X \,|\, f{:}Y$, *where* X, Y *are variables,* t *is a type symbol and* f *is a feature symbol.*

Definition 2.19 (feature constraint satisfaction)
Let $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ *be an interpretation and* α *an assignment. Define satisfaction of feature constraints as follows.*

- $\mathcal{I}, \alpha \models X \,|\, Y$ *iff* $\alpha(X) = \alpha(Y)$
- $\mathcal{I}, \alpha \models X \,|\, \neg Y$ *iff* $\alpha(X) \neq \alpha(Y)$
- $\mathcal{I}, \alpha \models X \,|\, t$ *iff* $\mathcal{S}(\alpha(X)) \preceq t$
- $\mathcal{I}, \alpha \models X \,|\, f{:}Y$ *iff* $\mathcal{A}(f)(\alpha(X))\downarrow$ *and* $\mathcal{A}(f)(\alpha(X)) = \alpha(Y)$

We will often need to talk about sets of constraints and sets of sets of constraints. We call a (not necessarily finite) set of feature constraints a feature clause, and a set of clauses a matrix. Feature clauses are interpreted conjunctively, matrices disjunctively.

Definition 2.20
An interpretation $\mathcal{I}$ *satisfies a feature clause* Σ *iff there is a variable assignment* α *s.t.* $\mathcal{I}, \alpha \models \Sigma$.

Definition 2.21 (clause entailment)
For each set of feature constraints Σ, Σ', $\Sigma \models \Sigma'$ *iff for each interpretation* $\mathcal{I}$*,if* $\mathcal{I} \models \Sigma$*, then* $\mathcal{I} \models \Sigma'$*. We say that* Σ *entails* Σ'.

Formally, we have introduced a new feature logic language. Whereas so far, we've only talked about description languages, we've now introduced a relational language, a language that talks about relations between variables. A feature constraint does not denote a set of objects in a given interpretation, it is true or false with respect to a given interpretation and assignment. Intuitively, sets of feature constraints are very similar to feature structures: the variables in feature constraints correspond to nodes in a (concrete) feature structure. That is also the computational role of constraints: whereas in a unification-based approach, we look for a most general feature structure satisfying a description, in a constraint-based approach, we translate a description into an equivalent set of constraints and then check to see if the resulting set is satisfiable. A very important difference though is that sets of feature constraints need not be rooted, they don't even need to be connected in the sense that the underlying graph of a feature structure needs to be connected.

Definition 2.22
Let Σ be a feature clause, Γ a matrix, $\mathcal{I}$ an interpretation and α an assignment.

- $\mathcal{I}, \alpha \models \Sigma$ *iff for each* $\sigma \in \Sigma,\ \mathcal{I}, \alpha \models \sigma$
- $\mathcal{I}, \alpha \models \Gamma$ *iff for some* $\Sigma \in \Gamma,\ \mathcal{I}, \alpha \models \Sigma$

This clearly gives us the standard interpretation of conjunction and disjunction. Notice in particular that for no $\mathcal{I}, \alpha:\ \mathcal{I}, \alpha \models \emptyset$, if $\emptyset$ is interpreted as a matrix. For the empty clause, on the other hand, we have that for each $\mathcal{I}, \alpha:\ \mathcal{I}, \alpha \models \emptyset$.

Next we define an auxiliary notion that allows us to classify the variables occurring in a feature clause.

Definition 2.23
Let Σ be a set of feature constraints. If $X|Y \in \Sigma$, $X \neq Y$ and Y occurs nowhere else in Σ, then Y is an auxiliary variable in Σ. If $Z \in \mathrm{FV}(\Sigma)$ and Z is not an auxiliary variable, then it is a main variable in Σ. Define $main(\Sigma) = \{X \in \mathrm{FV}(\Sigma) \mid X \text{ is a main variable in } \Sigma\}$.

Notice that the distinction between auxiliary and main variables makes no intuitive sense for arbitrary clauses. For clauses in normal form as we define them in the next definition, however, there is an intuitive difference. Main variables correspond to nodes in a feature structure. They are the ones that are really constrained. All the auxiliary variables do is point to a main variable.

Definition 2.24
A feature clause Σ is in normal form iff it has the following properties:

1. *If X is a main variable in Σ, then for some $a \in \mathcal{T}$, $X|a \in \Sigma$.*

2. *If* $X|\,f:Y, X|\,f:Z \in \Sigma$*, then* $Y = Z$.

3. *If* $X|\,a,\ X|\,b \in \Sigma$*, then* $a = b$.

4. *If* $X|\,Y \in \Sigma$ *and* $X \neq Y$*, then* Y *occurs nowhere else in* Σ.

5. *If* $X|\,\neg Y \in \Sigma$*, then* $X \neq Y$.

6. *If* $X|\,a,\ X|\,f\!:Y \in \Sigma$*, then for each* $t, t' \in \mathcal{V}$ *s.t.* $t \preceq a$ *and* $t' \preceq a$*,* $approp(t,f)\!\downarrow$*,* $approp(t',f)\!\downarrow$ *and* $approp(t,f) = approp(t',f)$.

7. *If* $X|\,a,\ X|\,f\!:Y,\ Y|\,b \in \Sigma$*, then for each* $t \in \mathcal{V}$ *such that* $t \preceq a$*,* $approp(t,f)\!\downarrow$ *and* $b \preceq approp(t,f)$.

Condition 1. says that each variable that is constrained in some way must be typed. Condition 2. requires features to be functional. In cond. 3., we say the typing information on each type must be unique. Condition 4. says that if two variables are required to be identical, then the right-hand one must be an auxiliary variable. This is to ensure that all information on a variable is collected in one place. Condition 5. states that inequated variables must be distinct. Notice that by definition, negated variables are always *main* variables. Conditions 6. and 7. ensure that the typing information conforms to the appropriateness conditions. In 6. we demand that if a feature is defined on a variable, then the feature must be appropriate for the type on the variable. The additional conditions ensure that our type discipline is static. We will take up this point in a later section (see prop. 2.16 on p. 38). Finally, cond. 7. requires that the variable a feature points to is appropriately typed.

We will now show that each feature clause in normal form is satisfiable. We begin by defining an interpretation that has exactly one object for each minimal type in the signature. We need to make the unproblematic assumption that the set of types is totally ordered in some arbitrary way. We will assume standard lexicographic order for our purposes.

Definition 2.25
For each signature $S = \langle \mathcal{T}, \preceq, \mathcal{F}, approp \rangle$*, define an interpretation* $\mathcal{I}_S = \langle \mathcal{U}_S, \mathcal{S}_S, \mathcal{A}_S \rangle$ *with*

- $\mathcal{U}_S = \{u_t \mid t \in \mathcal{V}\}$
- $\mathcal{S}_S(u_t) = t$

- $\mathcal{A}_S(f)(u_t) = \begin{cases} u_{t'} & \textit{if approp}(t,f)\downarrow \textit{ and } t' \textit{ is the lex. smallest minimal type s.t. } t' \preceq \textit{approp}(t,f) \\ \textit{undefined} & \textit{otherwise} \end{cases}$

Proposition 2.8
For each signature S, $\mathcal{I}_S$ is an S-interpretation.

Proposition 2.9
Let $S = \langle \mathcal{T}, \preceq, \mathcal{F}, approp \rangle$ be a signature. If Σ is a feature clause in normal form, then Σ is satisfiable.

Proof

First, let Σ' be exactly like Σ except that each constraint $X \mid a \in \Sigma$ is replaced by $X \mid t$ in Σ', where $t \in \mathcal{V}$ is the lex. smallest minimal type s.t. $t \preceq a$. Clearly, it suffices to show that Σ' is satisfiable.

We now construct an interpretation $\mathcal{I}$ and assignment α s.t. $\mathcal{I}, \alpha \models \Sigma'$. Define $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$, where

- $\mathcal{U} = \mathcal{U}_S \cup main(\Sigma)$
- $\mathcal{S}(u) = \begin{cases} \mathcal{S}_S(u) & \text{if } u \in \mathcal{U}_S \\ t & \text{if } u = X \text{ for some } X \in \mathrm{FV}(\Sigma') \text{ and } X \mid t \in \Sigma' \end{cases}$
- $\mathcal{A}(f)(u) = \begin{cases} \mathcal{A}_S(f)(u) & \text{if } u \in \mathcal{U}_S \text{ and } \mathcal{A}_S(f)(u)\downarrow \\ Y & \text{if } u = X \text{ for some } X \in \mathrm{FV}(\Sigma') \text{ and } X \mid f{:}Y \in \Sigma' \\ u_{t'} & \text{if } u = X \text{ for some } X \in \mathrm{FV}(\Sigma'), X \mid t \in \Sigma', \text{ for no } Y \text{ is } X \mid f : Y \in \Sigma', \textit{approp}(t,f)\downarrow \text{ and } t' \in \mathcal{V} \text{ is the lex. smallest minimal type s.t. } t' \preceq \textit{approp}(t,f) \\ \text{undefined} & \text{otherwise} \end{cases}$

We need to make sure that $\mathcal{I}$ is a S-interpretation. Suppose $X \mid f : Y \in \Sigma$. By condition 1. on normal form sets, we know that for some $a, b \in \mathcal{T}$, $X \mid a$, $Y \mid b \in \Sigma$ and thus, for some $t, t' \in \mathcal{V}$, $X \mid t$, $Y \mid t' \in \Sigma'$. By conditions 6. and 7. of a normal clause and construction of Σ', we know that $approp(t,f)\downarrow$ and $t' \preceq approp(t,f)$. Conversely, if $X \mid t \in \Sigma'$, $approp(t,f)\downarrow$ and for no Y is $X \mid f{:}Y \in \Sigma'$, then $\mathcal{A}(f)(X)\downarrow$ and $\mathcal{A}(f)(X) = u_{t'}$ for some $t' \preceq approp(t,f)$. Thus, $\mathcal{I}$ is a S-interpretation.

Let β be some variable assignment on $\mathcal{U}$. Define

$$\alpha(X) = \begin{cases} X & \text{if } X \in \mathit{main}(\Sigma') \\ Y & \text{if } X \notin \mathit{main}(\Sigma') \text{ and } Y \,|\, X \in \Sigma' \\ \beta(X) & \text{otherwise} \end{cases}$$

It remains to be shown that for each $c \in \Sigma'$, $\mathcal{I}, \alpha \models c$.

- If $X \,|\, Y \in \Sigma'$ then $\alpha(X) = \alpha(Y)$, by definition of α.
- If $X \,|\, \neg Y \in \Sigma'$ then $X \neq Y$ and $X \,|\, Y \notin \Sigma'$. Thus, $\alpha(X) \neq \alpha(Y)$.
- If $X \,|\, t \in \Sigma'$ then $\mathcal{S}(\alpha(X)) = t$.
- If $X \,|\, f{:}Y \in \Sigma'$ then $\alpha(X) = X$, $\alpha(Y) = Y$ and $\mathcal{A}(f)(X) = Y$.

Thus, $\mathcal{I}, \alpha \models \Sigma'$, and furthermore, $\mathcal{I}, \alpha \models \Sigma$.

■

Normal form sets have a stronger property than just satisfiability: if Σ is in normal form, then we can replace any $X \,|\, a \in \Sigma$ with $X \,|\, b$ if $b \preceq a$, and the new Σ is still satisfiable. This is particularly useful in practice, since it means that the normalisation of a conjunction of two normal sets can be computed very efficiently.

Sometimes, a normal form set contains information that is redundant and can be removed. This concerns information that can be deduced from the signature. We call removing such redundant information "unfilling".

Proposition 2.10 (Unfilling)
Let $S = \langle \mathcal{T}, \preceq, \mathcal{F}, \mathit{approp} \rangle$ be a signature. Furthermore, let $\Sigma_1 = \Sigma_2 \uplus \{X \,|\, f : Y,\ Y \,|\, b\}$ s.t. $X \,|\, a \in \Sigma_2$ be in normal form and $Y \notin \mathrm{FV}(\Sigma_2)$. If $\mathit{approp}^*(a, f) \equiv b$, then $\Sigma_1 \equiv \Sigma_2$.

Proof

Let $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ be an S-interpretation and $\alpha \in \mathrm{ASS}$.

1. Suppose $\mathcal{I}, \alpha \models \Sigma_1$. Then $\mathcal{I}, \alpha \models \Sigma_2$, since $\Sigma_2 \subset \Sigma_1$.
2. Suppose $\mathcal{I}, \alpha \models \Sigma_2$. Since $\mathit{approp}^*(a, f)\downarrow$ and $\mathit{approp}^*(a, f) \equiv b$, we know that $\mathcal{A}(f)(\alpha(X))\downarrow$ and $\mathcal{S}(\mathcal{A}(f)(\alpha(X))) \preceq b$. Since $Y \notin \mathrm{FV}(\Sigma_2)$, $\mathcal{I}, \alpha_{[Y \mapsto \mathcal{A}(f)(\alpha(X))]} \models \Sigma_1$.

■

The intuitive content of this proposition is that information that's not more informative than what's in the signature can be removed without changing the denotation. See (Götz 1994) for examples.

2.3.3 From terms to constraints

We've seen in the last section that feature constraints have a normal form that exhibits satisfiability. What we're interested in, of course, is to get feature terms into that normal form. We will begin this section by defining an algorithm to rewrite a feature term into a set of constraints. Since we've already seen how to bring a feature term into DNF, we will only consider conjunctive terms without negation, except in front of variables. We begin by defining a translation procedure that takes a purely conjunctive feature term with the negations pushed down, and translates it into an equivalent feature clause. This translation is a slight variant of the one given in (Dörre and Dorna 1993).

Definition 2.26 (trans)

1. $\mathbf{trans}(X, a) = \{X \mid a\}$, *if a is a type*
2. $\mathbf{trans}(X, Y) = \{X \mid Y\}$, *if Y is a variable*
3. $\mathbf{trans}(X, \neg Y) = \{X \mid \neg Y\}$, *if Y is a variable*
4. $\mathbf{trans}(X, f{:}T) = \{X \mid f{:}Y\} \cup \mathbf{trans}(Y, T)$, *where Y is a new variable*
5. $\mathbf{trans}(X, S \wedge T) = \mathbf{trans}(X, S) \cup \mathbf{trans}(X, T)$

Notice that **trans** is not a function, since it introduces arbitrary new variables. One could turn it into a function by the standard trick of well-ordering the variables. However, since we don't rely on the functionality of **trans** for anything, we avoid this technical complication.

We will now show that the translation actually preserves the denotation. Since the notion of denotation is not defined for feature clauses, we relate the denotation of an input term ϕ in an interpretation $\mathcal{I}$ with the set of objects that can be assigned to X so that $\mathcal{I}$ satisfies $\mathbf{trans}(X, \phi)$.

Proposition 2.11 (Correctness of trans)
Let $D = D_1 \vee \ldots \vee D_n$ be in DNF. For each D_i, $1 \leq i \leq n$, for each interpretation $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$, if $X \notin \mathrm{FV}(D)$ then

$$[\![D_i]\!]^{\mathcal{I}} = \{\alpha(X) \mid \mathcal{I}, \alpha \models \mathbf{trans}(X, D_i)\}$$

Proof

$\supseteq$: if $\mathcal{I}, \alpha \models \mathbf{trans}(X, D_i)$, then $\alpha(X) \in [\![D_i]\!]^{\mathcal{I}}$. Proof by induction on the definition of **trans**.

- $\mathcal{I}, \alpha \models \mathbf{trans}(X, a)$
  $\Rightarrow\ \mathcal{I}, \alpha \models X \mid a$
  $\Rightarrow\ \mathcal{S}(\alpha(X)) \leq a$
  $\Rightarrow\ \alpha(X) \in [\![a]\!]^{\mathcal{I}}_{\alpha}$
- $\mathcal{I}, \alpha \models \mathbf{trans}(X, Y)$
  $\Rightarrow\ \mathcal{I}, \alpha \models X \mid Y$
  $\Rightarrow\ \alpha(X) = \alpha(Y)$
  $\Rightarrow\ \alpha(X) \in [\![Y]\!]^{\mathcal{I}}_{\alpha}$
- $\mathcal{I}, \alpha \models \mathbf{trans}(X, \neg Y)$
  $\Rightarrow\ \mathcal{I}, \alpha \models X \mid \neg Y$
  $\Rightarrow\ \alpha(X) \neq \alpha(Y)$
  $\Rightarrow\ \alpha(X) \in [\![\neg Y]\!]^{\mathcal{I}}_{\alpha}$
- $\mathcal{I}, \alpha \models \mathbf{trans}(X, f{:}T)$
  $\Rightarrow\ \mathcal{I}, \alpha \models X \mid f{:}Y \cup \mathbf{trans}(Y, T)$
  $\Rightarrow\ \mathcal{A}(f)(\alpha(X))\downarrow$ and $\mathcal{A}(f)(\alpha(X)) = \alpha(Y)$ and, by induction, $\alpha(Y) \in [\![T]\!]^{\mathcal{I}}_{\alpha}$
  $\Rightarrow\ \alpha(X) \in [\![f{:}T]\!]^{\mathcal{I}}_{\alpha}$
- $\mathcal{I}, \alpha \models \mathbf{trans}(X, S \wedge T)$
  $\Rightarrow\ \mathcal{I}, \alpha \models \mathbf{trans}(X, S) \cup \mathbf{trans}(X, T)$
  $\Rightarrow\ \alpha(X) \in [\![S]\!]^{\mathcal{I}}_{\alpha}$ and $\alpha(X) \in [\![T]\!]^{\mathcal{I}}_{\alpha}$ (by induction)
  $\Rightarrow\ \alpha(X) \in [\![S \wedge T]\!]^{\mathcal{I}}_{\alpha}$

Since $\alpha(X) \in [\![D_i]\!]^{\mathcal{I}}_{\alpha}$, it follows that $\alpha(X) \in [\![D_i]\!]^{\mathcal{I}}$.

$\subseteq$: for each $u \in [\![D_i]\!]^{\mathcal{I}}$ there is an α s.t. $\alpha(X) = u$ and $\mathcal{I}, \alpha \models \mathbf{trans}(X, D_i)$.

We prove a stronger result by induction on D_i: if $u \in [\![D_i]\!]^{\mathcal{I}}_{\alpha}$ where $\alpha(X) = u$, then there is an α' s.t. $\alpha'(X) = u$, for each $Y \in \mathrm{FV}(D)$, $\alpha'(Y) = \alpha(Y)$ and $\mathcal{I}, \alpha' \models \mathbf{trans}(X, D_i)$.

- $u \in [\![a]\!]^{\mathcal{I}}_{\alpha}$ where $\alpha(X) = u$

  $\Rightarrow\ \mathcal{I}, \alpha \models X \,|\, a$

  $\Rightarrow\ \mathcal{I}, \alpha \models \mathbf{trans}(X, a)$

- $u \in [\![Y]\!]^{\mathcal{I}}_{\alpha}$ where $\alpha(X) = u$

  $\Rightarrow\ \alpha(Y) = u = \alpha(X)$

  $\Rightarrow\ \mathcal{I}, \alpha \models X \,|\, Y$

  $\Rightarrow\ \mathcal{I}, \alpha \models \mathbf{trans}(X, Y)$

- $u \in [\![\neg Y]\!]^{\mathcal{I}}_{\alpha}$ where $\alpha(X) = u$

  $\Rightarrow\ \alpha(Y) \neq u = \alpha(X)$

  $\Rightarrow\ \mathcal{I}, \alpha \models X \,|\, \neg Y$

  $\Rightarrow\ \mathcal{I}, \alpha \models \mathbf{trans}(X, \neg Y)$

- $u \in [\![f : T]\!]^{\mathcal{I}}_{\alpha}$ where $\alpha(X) = u$ and $\mathbf{trans}(X, f : T) = \{X \,|\, f : Y\} \cup \mathbf{trans}(Y, T)$

  $\Rightarrow\ \exists \beta.\ \beta(Y) = \mathcal{A}(f)(u)$, f. a. $Z \in FV(D)$, $\beta(Z) = \alpha(Z)$ and $\mathcal{I}, \beta \models \mathbf{trans}(Y, T)$

  (by induction, since $Y \notin \mathrm{FV}(T)$)

  $\Rightarrow\ \mathcal{I}, \beta_{[X \mapsto u]} \models \{X \,|\, f{:}Y\} \cup \mathbf{trans}(Y, T)$

  (since $X \notin \mathrm{FV}(\mathbf{trans}(Y, T))$)

  $\Rightarrow\ \mathcal{I}, \beta_{[X \mapsto u]} \models \mathbf{trans}(X, f{:}T)$

- $u \in [\![S \wedge T]\!]^{\mathcal{I}}_{\alpha}$ where $\alpha(X) = u$

  $\Rightarrow\ u \in [\![S]\!]^{\mathcal{I}}_{\alpha}$ and $u \in [\![T]\!]^{\mathcal{I}}_{\alpha}$

  $\Rightarrow\ \exists \beta.\ \mathcal{I}, \beta \models \mathbf{trans}(X, S)$, $\beta(X) = u$ and for each $Z \in FV(D)$, $\beta(Z) = \alpha(Z)$, and$\exists \beta'.\ \mathcal{I}, \beta' \models \mathbf{trans}(X, T)$, $\beta'(X) = u$ and for each $Z \in FV(D)$, $\beta'(Z) = \alpha(Z)$

  (by induction)

  $\Rightarrow$ if $\alpha' = \beta_{[Y_1 \mapsto \beta'(Y_1), \ldots, Y_n \mapsto \beta'(Y_n)]}$, where $\{Y_1, \ldots, Y_n\} = \mathrm{FV}(\mathbf{trans}(X, T)) \backslash (\{X\} \cup \mathrm{FV}(D))$, then $\mathcal{I}, \alpha' \models \mathbf{trans}(X, S \wedge T)$, since $(\mathrm{FV}(\mathbf{trans}(X, S)) \backslash (\{X\} \cup \mathrm{FV}(D))) \cap$

$(\mathrm{FV}(\mathbf{trans}(X,T))\backslash(\{X\}\cup \mathrm{FV}(D))) = \emptyset$ and
for each $Y \in \mathrm{FV}(D)$, $\beta(Y) = \beta'(Y)$

Since $X \notin \mathrm{FV}(D_i)$, $u \in [\![D_i]\!]^{\mathcal{I}}_{\alpha[X\mapsto u]}$ and thus, by the induction above, α' exists with $\alpha'(X) = u$ and $\mathcal{I}, \alpha' \models \mathbf{trans}(X, D_i)$.

■

2.3.4 Normalizing constraint matrices

We will now define a rewrite system to transform an arbitrary (finite) set of constraints into one in normal form. However, since the result is sometimes not unique, the rewrite system will not operate on sets of constraints, but on constraint matrices (sets of sets of constraints).

Definition 2.27 (variable substitution)
Write $\Sigma_{[X/Y]}$ for Σ with every occurrence of Y replaced by X.

The main work in normalizing constraint matrices consists in type checking. This was first discussed, for a somewhat different logic, in (Carpenter et al. 1991). We use the following definition to express our typing rules more succinctly.

Definition 2.28 (dom)
For every signature $\langle \mathcal{T}, \preceq, \mathcal{F}, approp\rangle$, $\mathbf{dom}$ is a total function $\mathcal{T}\times\mathcal{F} \to \mathcal{P}(\mathcal{T})$ s.t.

$\mathbf{dom}(a, f) = max(\{b \mid b \leq a,\ approp(b, f)\downarrow$ *and for each* $t, t' \in \mathcal{V}$ *s.t.* $t \leq b$ *and* $t' \leq b$, *it is the case that* $approp(t, f) = approp(t', f)\})$

A rewrite system for bringing feature matrices into normal form is shown in Fig. 2.4. Clearly, a set of constraints Σ is in normal form iff none of the rewrite rules applies to $\{\Sigma\}$. This can be seen by an easy case analysis.

Proposition 2.12
Let Σ be a set of feature constraints. Σ is in normal form iff $\{\Sigma\} \not\to$.

The intuition behind the rules is straightforward in most cases. Rule 1 ensures that typing information is present for all main variables. This is necessary for the typing rules to fire. Rule 2 makes sure that a feature defined on a variable does not point to two distinct variables. Rules 3a

1) $\Gamma \cup \{\Sigma\} \longrightarrow \Gamma \cup \{\Sigma \cup \{X|\top\}\}$,
if X is a main variable in Σ and for no $a \in \mathcal{T},\ X|t \in \Sigma$

2) $\Gamma \cup \{\Sigma \cup \{X|f{:}Y, X|f{:}Z\}\} \longrightarrow \Gamma \cup \{\Sigma \cup \{X|f{:}Y, Y|Z\}\}$, if $Y \neq Z$

3a) $\Gamma \cup \{\Sigma \cup \{X|a, X|b\}\} \longrightarrow \Gamma \cup \{\Sigma \cup \{X|c\}\}$,
if $a \neq b$, $glb(a,b)$ exists and $glb(a,b) = c$

3b) $\Gamma \cup \{\Sigma \cup \{X|a, X|b\}\} \longrightarrow \Gamma$, if $glb(a,b)$ does not exist

4) $\Gamma \cup \{\Sigma \cup \{X|Y\}\} \longrightarrow \Gamma \cup \{\Sigma_{[X/Y]} \cup \{X|Y\}\}$,
if $X \neq Y$ and $Y \in \mathrm{FV}(\Sigma)$

5) $\Gamma \cup \{\Sigma \cup \{X|\neg X\}\} \longrightarrow \Gamma$

6) $\Gamma \cup \{\Sigma \cup \{X|a, X|f{:}Y\}\} \longrightarrow \Gamma \cup \{\Sigma \cup \{X|c, X|f{:}Y\} \mid c \in \mathbf{dom}(a,f)\}$,
if $\mathbf{dom}(a,f) \neq \{a\}$

7a) $\Gamma \cup \{\Sigma \cup \{X|a, X|f{:}Y, Y|b\}\} \longrightarrow \Gamma \cup \{\Sigma \cup \{X|a, X|f{:}Y, Y|c\}\}$,
if $\mathbf{dom}(a,f) = \{a\}$, $approp^*(a,f) = d$, $glb(d,b)\downarrow$, $glb(d,b) = c$ and $c \neq b$

7b) $\Gamma \cup \{\Sigma \cup \{X|a, X|f{:}Y, Y|b\}\} \longrightarrow \Gamma$,
if $\mathbf{dom}(a,f) = \{a\}$ and $glb(approp^*(a,f), b)\uparrow$

Figure 2.4: Feature matrix rewriting

and 3b take care of multiple typing information on one variable. Either the types are consistent (3a), then they are replaced by their greatest lower bound in the type lattice. Or they they are inconsistent (3b), then the whole clause is inconsistent and is removed. Rule 4 makes sure that if we have two equated variables, all constraining information on these variables is collected on *one* of them. Effectively, it makes one of the variables into an auxiliary variable. If a clause contains the information that some variable is not equal to itself, this clause is clearly inconsistent, and rule 5 removes it. Rule 6 ensures that if a feature f is defined on some variable X, then the type a on X is in the domain of f. Furthermore, every type subsumed by a must have the same type appropriate for f as a. As mentioned before, this ensures that our type discipline is static. We will make this notion precise in Prop. 2.16 on p. 38. Rules 7a and 7b, finally, ensure that the constraints on the ranges of features specified in the appropriateness conditions are met. Either the typing on a variable that a feature points to is consistent with appropriateness conditions (7a), or inconsistent (7b), in which case the clause is removed from the clause.

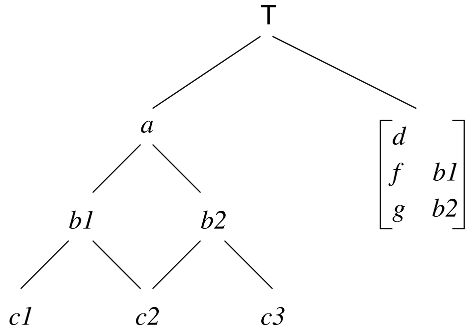


Figure 2.5: Example signature

A small example will illustrate the mechanics we've set up so far. We will normalize the term $f{:}Z \wedge g{:}Z$ with respect to the signature in fig. 2.5. Since the term is already in disjunctive normal from, we can directly apply the **trans** procedure.

$$
\begin{array}{rcl}
\mathbf{trans}(X, f{:}Z \wedge g{:}Z) & = & \mathbf{trans}(X, f{:}Z) \cup \mathbf{trans}(X, g{:}Z) \\
& = & \{X \,|\, f{:}Y\} \cup \mathbf{trans}(Y, Z) \cup \mathbf{trans}(X, g{:}Z) \\
& = & \{X \,|\, f{:}Y, Y \,|\, Z\} \cup \mathbf{trans}(X, g{:}Z) \\
& = & \{X \,|\, f{:}Y, Y \,|\, Z, X \,|\, g{:}Y'\} \cup \mathbf{trans}(Y', Z) \\
& = & \{X \,|\, f{:}Y, Y \,|\, Z, X \,|\, g{:}Y', Y' \,|\, Z\}
\end{array}
$$

One possible normalisation is shown below. The individual steps are annotated with the rules that will be applied, and the constraints that are used to derive the next stage are underlined. Since we're only dealing with a single clause here, we ignore the fact that the rewrite system really applies to matrices.

$$
\begin{array}{lr}
\{X \mid f{:}Y, Y \mid Z, X \mid g{:}Y', \underline{Y' \mid Z}\} & (4) \\
\longrightarrow\ \{X \mid f{:}Y, \underline{Y \mid Y'}, X \mid g{:}Y', Y' \mid Z\} & (4) \\
\longrightarrow\ \{X \mid f{:}Y, Y \mid Y', X \mid g{:}Y, Y \mid Z\} & (1) \\
\longrightarrow\ \{X \mid f{:}Y, Y \mid Y', X \mid g{:}Y, Y \mid Z, X \mid \top\} & (1) \\
\longrightarrow\ \{\underline{X \mid f{:}Y}, Y \mid Y', X \mid g{:}Y, Y \mid Z, \underline{X \mid \top}, Y \mid \top\} & (6) \\
\longrightarrow\ \{\underline{X \mid f{:}Y}, Y \mid Y', X \mid g{:}Y, Y \mid Z, \underline{X \mid d}, \underline{Y \mid \top}\} & (7) \\
\longrightarrow\ \{X \mid f{:}Y, Y \mid Y', \underline{X \mid g{:}Y}, Y \mid Z, \underline{X \mid d}, \underline{Y \mid b1}\} & (7a) \\
\longrightarrow\ \{X \mid f{:}Y, Y \mid Y', X \mid g{:}Y, Y \mid Z, X \mid d, Y \mid c2\} &
\end{array}
$$

We next consider termination of the rewrite system on finite sets of finite sets of constraints. Notice that the rewrite rules can be strongly ordered: a matrix can be closed under rule 1, and no application of another rule can make rule 1 become applicable again (since no rule introduces new variables). We will make use of this fact in our termination proof. The rules 2 through 4 also form a subsystem of their own. Once a matrix is closed under rules 1 through 4, no application of the other rules can make application of rules 1 through 4 necessary. Rule 5 comes next. The typing rules 6, 7a and 7b are also strictly ordered, rule 6 having to apply before one of the others can.

Proposition 2.13
If Γ is a finite matrix of finite sets, then there are a Γ', Γ'' and natural numbers n and m s.t. $\Gamma \xrightarrow{n} \Gamma' \xrightarrow{m} \Gamma'' \not\longrightarrow$, where rule 1) does not apply to Γ'.

Proof

Suppose $\Gamma = \{\Sigma_1, \dots, \Sigma_k\}$. Γ' can be derived in at most $2 \times (|\Sigma_1| \times \ldots \times |\Sigma_k|)$ steps (since for each i, $|main(\Sigma_i)| \leq 2 \times |\Sigma_i|$). Since none of the other rules introduce any new variables, we can consider the rest of the rules without rule 1).

We proceed by associating a weight with each set of constraints. Since each rule application generates finitely many sets of constraints Σ' from a set Σ (which is removed from the matrix), it suffices to show that for each of those new Σ', $weight(\Sigma') < weight(\Sigma)$. A derivation defines a finitely branching tree, and to show that the tree is finite, we need to show that every branch is finite.

- $weight(X \mid a) = |\{b \mid b \preceq a\}|$

- $weight(X \,|\, Y) = 1$
- $weight(X \,|\, \neg Y) = 1$
- $weight(X \,|\, f{:}Y) = 2$
- $weight(\Sigma) = \left(\sum_{\sigma \in \Sigma} weight(\sigma) \right) \times |main(\Sigma)|$

Now consider rule application. Rules 3b, 5 and 7b eliminate the clauses they're applied to, since these clauses are inconsistent. Their application thus constitutes a terminal node in the derivation tree.

- If $\Gamma \cup \Sigma \longrightarrow \Gamma \cup \Sigma'$ by application of rule 2), then $weight(\Sigma') \leq weight(\Sigma) - 1$, since $|main(\Sigma)| = |main(\Sigma')|$.

- Suppose rule 3a) is applied: $\Gamma \cup \{\Sigma \cup \{X \,|\, a, X \,|\, b\}\} \longrightarrow \Gamma \cup \{\Sigma \cup \{X \,|\, c\}\}$, where $a \neq b$, $glb(a,b)\!\downarrow$ and $glb(a,b) = c$. Let $A = \{a' \mid a' \preceq a\}$, $B = \{b' \mid b' \preceq b\}$ and $C = \{c' \mid c' \preceq c\}$. Assume w.l.o.g. that $|A| \geq |B|$. Since $a \neq b$ and $glb(a,b)\!\downarrow$, we know that $|A| \geq 2$. Thus, $weight(X \,|\, a) + weight(X \,|\, b) = |A| + |B| \geq 2 + |B| > |B| \geq |C| = weight(X \,|\, c)$.

- Suppose rule 4) applies: $\Gamma \cup \{\Sigma \cup \{X \,|\, Y\}\} \longrightarrow \Gamma \cup \{\Sigma_{[X/Y]} \cup \{X \,|\, Y\}\}$, with $X \neq Y$ and $Y \in \mathrm{FV}(\Sigma)$. Now $\sum_{\sigma \in (\Sigma \cup \{X \,|\, Y\})} weight(\sigma) = \sum_{\sigma \in (\Sigma_{[X/Y]} \cup \{X \,|\, Y\})} weight(\sigma)$, but $|main(\Sigma \cup \{X \,|\, Y\})| = |main(\Sigma_{[X/Y]} \cup \{X \,|\, Y\})| - 1$. Thus, $weight(\Sigma_{[X/Y]} \cup \{X \,|\, Y\}) < weight(\Sigma \cup \{X \,|\, Y\})$.

- Suppose rule 6) is applied:
  $\Gamma \cup \{\Sigma \cup \{X \,|\, a, X \,|\, f{:}Y, Y \,|\, b\}\} \longrightarrow \Gamma \cup \{\Sigma \cup \{X \,|\, c, X \,|\, f{:}Y, Y \,|\, b\} \mid c \in \mathbf{dom}(a,f)\}$, where $\mathbf{dom}(a,f) \neq \{a\}$. But we know that for each $c \in \mathbf{dom}(a,f)$, $c \preceq a$ and $c \neq a$. Thus, for each $c \in \mathbf{dom}(a,f)$, $weight(\Sigma \cup \{X \,|\, a, X \,|\, f{:}Y, Y \,|\, b\}) > weight(\Sigma \cup \{X \,|\, c, X \,|\, f{:}Y, Y \,|\, b\})$.

- Suppose rule 7a) is applied:
  $\Gamma \cup \{\Sigma \cup \{X \,|\, a, X \,|\, f{:}\,Y, Y \,|\, b\}\} \longrightarrow \Gamma \cup \{\Sigma \cup \{X \,|\, a, X \,|\, f{:}\,Y, Y \,|\, c\}\}$, where there is a d s.t. for all $t \in \mathcal{V}$ s.t. $t \leq a$, it is the case that $approp(t,f)\!\downarrow$, $approp(t,f) = d$, $glb(d,b)\!\downarrow$,
  $glb(d,b) = c$ and $c \neq b$. Since $c \preceq b$ and $c \neq b$, we know that $weight(\Sigma \cup \{X \,|\, a, X \,|\, f{:}Y, Y \,|\, b\}) > weight(\Sigma \cup \{X \,|\, a, X \,|\, f{:}Y, Y \,|\, c\})$.

■

We can roughly estimate what the complexity of this procedure is. Closing a clause under rule 1 is linear in the size of the clause. The same holds for rules 2, 4 and 5, if we take variable substitution to be an atomic operation. We can then close the clause under rules 3a and 3b, again in time linear to the size of the (original) clause. Rule 6 introduces the real complexity. In the worst case, each application of rule 6 can split up the the clause into as many clauses as there are minimal types in the type hierarchy ($|\mathcal{V}|$). Since this may happen once for each feature selection constraint $X|\,f{:}\,Y$ in the clause, application of rule 6 is exponential in the number of such constraints. Finally, closing under rule 7a and 7b is again linear in the size of the clause.

It remains to be shown that the rewrite system preserves the denotation of a matrix.

Proposition 2.14
For each matrix Γ, Γ', *interpretation* $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ *and assignment* α, *if* $\Gamma \longrightarrow \Gamma'$, *then*

$$\mathcal{I}, \alpha \models \Gamma \textit{ iff } \mathcal{I}, \alpha \models \Gamma'$$

Proof

Rule 1: obvious, since $\mathcal{I}, \alpha \models X|\top$ is always true.

Rule 2: $\mathcal{I}, \alpha \models \{X|\,f{:}Y, X|\,f{:}Z\} \;\Leftrightarrow\; \mathcal{A}(f)(\alpha(X))\downarrow,\ \mathcal{A}(f)(\alpha(X)) = \alpha(Y)$ and $\mathcal{A}(f)(\alpha(X)) = \alpha(Z) \;\Leftrightarrow\; \mathcal{A}(f)(\alpha(X))\downarrow,\ \mathcal{A}(f)(\alpha(X)) = \alpha(Y)$ and $\alpha(Y) = \alpha(Z) \;\Leftrightarrow\; \mathcal{I}, \alpha \models \{X|\,f{:}Y, X|\,Y\}$

Rule 3a: obvious, since for each $u \in \mathcal{U}$, $\mathcal{S}(u) \preceq \mathit{glb}(a,b)$ iff $\mathcal{S}(u) \preceq a$ and $\mathcal{S}(u) \preceq b$.

Rule 3b: obvious, since if $\mathit{glb}(a,b)\uparrow$, then for each $u \in \mathcal{U}$, $\mathcal{S}(u) \not\preceq a$ or $\mathcal{S}(u) \not\preceq b$.

Rule 4): $\mathcal{I}, \alpha \models \Sigma \cup \{X|\,Y\} \;\Leftrightarrow\; \mathcal{I}, \alpha \models \Sigma$ and $\alpha(X) = \alpha(Y) \;\Leftrightarrow\; \mathcal{I}, \alpha \models \Sigma_{[X/Y]} \cup \{X|\,Y\}$.

Rule 5: obvious.

Rule 6: We need to show that
$\mathcal{I}, \alpha \models \{X|\,a, X|\,f{:}Y, X|\,b\} \;\Leftrightarrow\; \exists c \in \mathbf{dom}(a,f).\ \mathcal{I}, \alpha \models \{X|\,c, X|\,f{:}Y, X|\,b\}$
Right to left is obvious, since for each $c \in \mathbf{dom}(a,f)$, $c \preceq a$.

$\mathcal{I}, \alpha \models \{X|\,a, X|\,f{:}Y, X|\,b\}$

$\Rightarrow\ \mathcal{S}(\alpha(X)) \preceq a,\ \mathcal{A}(f)(\alpha(X))\downarrow,\ \mathcal{A}(f)(\alpha(X)) = \alpha(Y)$ and $\mathcal{S}(\alpha(Y)) \preceq b$

$\Rightarrow\ \exists c \in \mathbf{dom}(a, f).\ \mathcal{S}(\alpha(X)) \preceq c,\ \mathcal{A}(f)(\alpha(X))\downarrow,\ \mathcal{A}(f)(\alpha(X)) = \alpha(Y)$ and $\mathcal{S}(\alpha(Y)) \preceq b$

$\Rightarrow\ \exists c \in \mathbf{dom}(a, f).\ \mathcal{I}, \alpha \models \{X \,|\, c, X \,|\, f{:}Y, X \,|\, b\}$

Rule 7a: again, right to left is obvious, since $c \preceq b$. For left to right, we have:

$\mathcal{I}, \alpha \models \{X \,|\, a, X \,|\, f{:}Y, Y \,|\, b\}, \exists d.\ \forall t \preceq a,\ t \in \mathcal{V}.\ \mathit{approp}(t, f)\downarrow, \mathit{approp}(t, f) = d,\ \mathit{glb}(b, d)\downarrow$ and $\mathit{glb}(b, d) = c$

$\Rightarrow\ \mathcal{S}(\alpha(X)) \preceq a, \mathcal{A}(f)(\alpha(X))\downarrow, \mathcal{A}(f)(\alpha(X)) = \alpha(Y),$
$\mathcal{S}(\alpha(Y)) \preceq b, \mathcal{S}(\alpha(Y)) \preceq d,\ \mathit{glb}(b, d)\downarrow$ and $\mathit{glb}(b, d) = c$

$\Rightarrow\ \mathcal{S}(\alpha(X)) \preceq a,\ \mathcal{A}(f)(\alpha(X))\downarrow,\ \mathcal{A}(f)(\alpha(X)) = \alpha(Y)$ and $\mathcal{S}(\alpha(Y)) \preceq c$

$\Rightarrow\ \mathcal{I}, \alpha \models \{X \,|\, a, X \,|\, f{:}Y, Y \,|\, c\}$

Rule 7b: If $\mathit{approp}^*(a, f)\downarrow$ and $\mathit{glb}(\mathit{approp}^*(a, f), b)\uparrow$, then $\{X \,|\, a, X \,|\, f{:}Y, Y \,|\, b\}$ is inconsistent.

■

The interpretation constructed in the proof of prop. 2.9 is certainly finite for finite clauses. However, if the clause was derived from a term, then the interpretation is actually almost a feature structure. We will make this precise by first defining what a feature structure is.

Definition 2.29 (feature structure)
Let S be a signature. A feature structure over S is a quadruple $FS = \langle \mathcal{U}, \bar{u}, \mathcal{S}, \mathcal{A} \rangle$ s.t.

1. *$\langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ is a S-interpretation,*
2. *$\mathcal{U}$ is finite,*
3. *$\bar{u} \in \mathcal{U}$, and*
4. *for each $u \in \mathcal{U}$, for some path π, $\mathcal{A}^*(\pi)(\bar{u})\downarrow$ and $\mathcal{A}^*(\pi)(\bar{u}) = u$.*

Corollary 2.15
Every satisfiable feature term is satisfied by a feature structure.

Proof

W.l.o.g., let $\phi = \phi_1 \vee \ldots \vee \phi_n$ be a term in DNF. Now apply the **trans** procedure to one of the satisfiable disjuncts ϕ_i and let $\Sigma \in \mathrm{NF}(\{\mathbf{trans}(X, \phi_i)\})$. Since Σ is in normal form, we can use the construction in the proof of prop. 2.9 to give us an interpretation $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ and assignment α s.t. $\mathcal{I}, \alpha \models \Sigma$. Since Σ is finite, we know that $\mathcal{U}$ is finite. Our root object will be $\bar{u} = \alpha(X)$, but there may be objects that are not reachable from $\bar{u}$. We thus have to restrict our domain to $\mathcal{U}' = \{u \in \mathcal{U} \mid \mathcal{A}^*(\pi)(\bar{u})\downarrow$ and $\mathcal{A}^*(\pi)(\bar{u}) = u\}$. If we now restrict $\mathcal{S}$ with respect to $\mathcal{U}'$ to $\mathcal{S}'$ and $\mathcal{A}$ to $\mathcal{A}'$, then $\mathcal{I} = \langle \mathcal{U}', \bar{u}, \mathcal{S}', \mathcal{A}' \rangle$ is a feature structure that satisfies ϕ.

■

As a final point of this section, we will examine how the rewrite system behaves at run-time. We will generally find ourselves in the situation that we have two sets of constraints Σ and Σ', both in normal form, and two variables $X \in \mathrm{FV}(\Sigma)$ and $X' \in \mathrm{FV}(\Sigma')$[4] we wish to identify. That is, we need to normalize the matrix $\Gamma = \{\Sigma \cup \Sigma' \cup \{X \mid X'\}\}$. The next proposition will show that our type discipline is static, i.e., we can normalize Γ without using the type inference rules. This means that the sources of complexity we identified for the rewrite system are not there at run-time. It also means that Γ will not grow in size. It is either inconsistent, in which case its normal form is the empty set, or it is consistent with just a single clause.

Proposition 2.16 (static typing)
Let Σ, Σ' be clauses in normal form, $X \in \mathrm{FV}(\Sigma)$ and $X' \in \mathrm{FV}(\Sigma')$. Then the matrix $\Gamma = \{\Sigma \cup \Sigma' \cup \{X \mid X'\}\}$ can be normalized using rules 2–5 only.

Proof

Clearly, rule 1 can never apply.

Assume w.l.o.g. that $\{Y \mid a, Y \mid f{:}Y'\} \subseteq \Sigma$ and that $\Gamma \xrightarrow{*} \Gamma' \supseteq \{Y \mid b, Y \mid f{:}Y'\}$, where $a \neq b$. For the sake of clarity, we consider the derivation modulo variable renaming. Obviously, we have that $b \prec a$. Also, $\mathbf{dom}(a, f) = \{a\}$, thus $\mathbf{dom}(b, f) = \{b\}$, and therefore, rule 6 doesn't apply.

For rule 7a and 7b, assume w.l.o.g. that $\{Y \mid a, Y \mid f{:}Y', Y \mid b\} \subseteq \Sigma$. Suppose further that $\Gamma \xrightarrow{*} \Gamma' \supseteq \{Y \mid c, Y \mid f{:}Y', Y \mid d\}$. We know that $\mathbf{dom}(a, f) = \{a\}$ and thus, *approp**(a, f) = *approp**(c, f). Therefore, since $d \preceq b$ and

[4] Σ and Σ' may or may not be standardized apart ($\mathrm{FV}(\Sigma) \cap \mathrm{FV}(\Sigma') = \emptyset$). This makes no difference for the rewrite system.

$glb(approp^*(a,f),b) = b$, $glb(approp^*(c,f),d) = d$, and rule 7a doesn't apply. Since $glb(approp^*(c,f),d)\downarrow$, rule 7b is also not applicable.

■

2.4 Alternative feature description languages

There are numerous other approaches to feature description languages, differing in both syntax and semantics. We will first give a non-exhaustive list of possible variations, and then look at a specific instance in more detail.

2.4.1 Models of total information vs. models of partial information

The logic we presented in the preceding sections takes the view that an interpretation provides total information about the real objects that we want to talk about. In fact, our interpretations *are* the things that we want to talk about. Another possibility is to consider the models to provide partial information themselves, to be partial models of the real world. This view induces the possibility of imposing an ordering on models: some provide more information than others. This ordering is usually called feature structure subsumption, since models in this setup are generally feature structures. We say that a feature structure A subsumes feature structure B iff A contains the same or less information than B.

In the total information approach, partiality resides solely with the descriptions, our feature terms. Underspecification is syntactic, the semantic objects are totally specified. The partial information approach introduces a second level of underspecification, namely the models.

Computationally, the partial information approach offers the possibility to compute with the models. This works by representing a description ϕ by a finite set of feature structures $\mathbf{MGS}(\phi)$ (the most general satisfiers of ϕ) with the property that for any feature structure B described by ϕ there is a feature structure A $\in \mathbf{MGS}(\phi)$ s.t. A subsumes B. Conversely, any feature structure subsumed by some A $\in \mathbf{MGS}(\phi)$ must be described by ϕ. Notice that $\mathbf{MGS}(\phi)$ must be *finite*. In fact, it should be as small as possible, for practical purposes. This is often not feasible under the total information approach. As we've seen in the previous section, we use normal form descriptions.

We can call computing with models unification-based, since the most important operation is the merging of the information in two feature structures, called unification. More precisely, a unifier of two feature structures A and B is a greatest (with respect to the subsumption ordering) feature structure C that contains all the information of A and B. For the unification approach to work, we need to require that for every feature logic expression ϕ there is a finite set S of feature structures (the most general satisfiers of ϕ) s.t. for each feature structure A, if A satisfies ϕ, then for some B$\in$S, B subsumes A.

So for the unification approach to work, we require that

- the class of models is ordered by subsumption, and
- that the most general satisfier property obtains.

We will later see that for one of the logics we present, the most general satisfier property doesn't hold.

The important difference between unification and constraint solving is that constraints are syntactic entities that *restrict* the class of possible structures, whereas in a unification approach, explicit models are *constructed*. To determine satisfiability of a given expression in a constraint-based approach, it is thus not necessary to compute explicit unifiers, it is sufficient to determine if a given set of constraints is consistent. In general, one can therefore treat a larger class of languages with constraint solving techniques than with unification (Baader and Siekmann 1994).

2.4.2 Closed world vs. open world

In the context of typed feature logic, the distinction closed world vs. open world interpretation can refer to two entirely different matters.

Interpretation of the type hierarchy

Closed world interpretation of the type hierarchy expresses the view that all types of objects that exist are mentioned in the type hierarchy. Specifically, this means that the domain of an interpretation is partitioned by the minimal types. This means that each object is of exactly one minimal type. Also, the denotation of each type is the union of the denotations of its minimal subtypes.

An open world interpretation makes no such assumptions. It is usually required that the denotations of the minimal types are disjoint, but the union of the denotations of the minimal types does not cover the whole universe of an interpretation. A non-minimal type type can have objects in its denotation that are not in the denotation of any of its subtypes.

Consider a type hierarchy with a type *sign* with *phrase* and *word* as its only immediate subtypes. A closed world interpretation requires that any object of type *sign* is also a *phrase* or *word*. Under an open world interpretation, objects of type *sign* that are neither of type *phrase* nor *word* are possible.

There are two major differences between the two approaches. One concerns negation. In a closed world approach, full classical negation is (theoretically) no problem. An open world approach has in general considerable problems with negation. There are several ways to get around this problem, the easiest one being to give up on classical negation. Another possibility will be mentioned later on.

The other difference has to do with appropriateness conditions and is of a mainly practical nature. The type inferencing required to test descriptions for satisfiability is in some sense a lot messier for a closed world approach than for an open world approach. Even for purely conjunctive descriptions, closed world inference is, as we've discussed, NP hard. That closed world inference is NP hard was first shown by Bob Carpenter (personal communication). We will see in the next section that under certain conditions, type inference for open world reasoning can be more efficient.

Open world normal forms are also smaller (their representations) and their unification is generally cheaper. Closed world inference, however, is completely static, whereas some dynamic type inferencing is necessary under an open world approach.

It appears that the ease of type inference is the reason that many practical system chose an open world approach to type hierarchy interpretation. The CUF (Dörre and Dorna 1993) system (among others) however is so powerful as to allow both kinds of inferences on different parts of the hierarchy.

Choosing the domain of interpretation

Our logic is completely free as to what the objects in the domain of an interpretation are. This is called an open world approach, since a user of the logic is completely free to apply it to a domain of his/her choosing. The domain just has to satisfy the general requirements of an interpretation. This is the standard approach in mathematical logic. A different approach is to fix a basic domain of interpretation, and require the domain of each interpretation to be a subset of the basic domain. This approach is useful if there is an intended interpretation one wants to talk about. Instead of carefully defining ones interpretations to have all the properties of the intended interpretation, one simply talks about the intended interpretation in the first place. In the context of feature logic, this could be some set of suitably well-formed feature structures. *For our purposes, there is no decisive difference between the two approaches.* A practical system will compute with some internal data structure that can be viewed either as some normal form description or as objects in the fixed domain or constraints on objects or whatever else is most convenient.

The same differences can be found in logic programming literature. For example, one can view standard logic programming as making statements about first order logic expressions (the "traditional" view, see (Lloyd 1984)) or as constraint logic programming over the universe of Herbrand terms. There is no practical difference whatsoever (but there are some subtle theoretical ones), it's just a question of perspective.

We've chosen an open world approach in this sense for two reasons: it's more flexible and we don't have an intended interpretation.

2.4.3 Other parameters

A big difference between feature logics is often their type system. Types where introduced into feature logic in (Smolka 1988).

Many feature logics differ in the expressive power of their syntax. Most logics, e.g., do not allow the unrestricted use of negation (and thus, implication). Others are more expressive still. Smolka (1992) allows explicit quantification over variables, thus rendering the satisfiability problem for his logic undecidable. Some logics do not use variables, but path equations. As we've seen before, this gets around some problems with unbound variables in the scope of negation.

Some authors have relaxed the condition that features must be functional. Manandhar (1995) used this construction to model linear precedence constraints. Another extension is named or distributed disjunction (Dörre and Eisele 1991), a construct that allows the compact computational representation of disjunctive information.

2.5 Open world reasoning and negation

Since open world reasoning is of considerable interest for computational linguistics, we will in this section consider a feature logic with open world type inference and full negation. This logic was first developed in (Carpenter 1992). However, Carpenter (1992) used a unification approach to his logic. This led to problems with negation since the most general satisfier property doesn't hold. We will give an example of this later. First, however, we need to define the logic.

Definition 2.30 (open world signature)
A signature is a quadruple $\langle \mathcal{T}, \preceq, \mathcal{F}, approp \rangle$ s.t.

- *$\mathcal{T}$ is a finite set of types, and $\langle \mathcal{T}, \preceq \rangle$ is a join semi-lattice*
- *$\mathcal{F}$ is a finite set of feature names*
- *$approp : \mathcal{T} \times \mathcal{F} \rightharpoonup \mathcal{T}$ is a partial function from pairs of types and features to types s.t. if $t' \preceq t$ and $approp(t, f)\downarrow$, then $approp(t', f)\downarrow$ and $approp(t', f) \preceq approp(t, f)$.*

Notice that the only difference between an open world signature and the signatures as we've defined them before lies in the appropriateness conditions. What we've previously defined as a closure property of *approp*, namely *approp**, is now part of the signature proper. That is, *approp* is defined for *all* types, not just minimal ones. The consistency condition imposed on *approp* simply ensures correct inheritance of features and their values.

Definition 2.31 (open world interpretation)
Let $S = \langle \mathcal{T}, \preceq, \mathcal{F}, approp \rangle$ be a signature. An S-interpretation is a quadruple $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ s.t.

- *$\mathcal{U}$ is a set of objects, the domain of $\mathcal{I}$*
- *$\mathcal{S} : \mathcal{U} \to \mathcal{T}$ is a total function from the set of objects to the set of types*
- *$\mathcal{A} : \mathcal{F} \to \mathcal{U}^{\mathcal{U}}$ is an attribute interpretation function s.t.*
    1. *for each $u \in \mathcal{U}$, for each $f \in \mathcal{F}$, if $(\mathcal{A}(f))(u)$ is defined, then $approp(\mathcal{S}(u), f)$ is defined and $\mathcal{S}((\mathcal{A}(f))(u)) \preceq approp(\mathcal{S}(u), f)$*
    2. *for each $u \in \mathcal{U}$, for each $f \in \mathcal{F}$, if $approp(\mathcal{S}(u), f)$ is defined and $approp(\mathcal{S}(u), f) = t$, then $\mathcal{A}(f)(u)$ is defined and $\mathcal{S}(\mathcal{A}(f)(u)) \preceq t$*

Notice that the type assignment function no longer assigns a minimal type to each object, but rather some arbitrary type. Following the terminology of (Carpenter 1992), we can call a structure obeying condition 1. of the definition of $\mathcal{A}$ well-typed. A structure obeying both conditions is called totally well-typed.

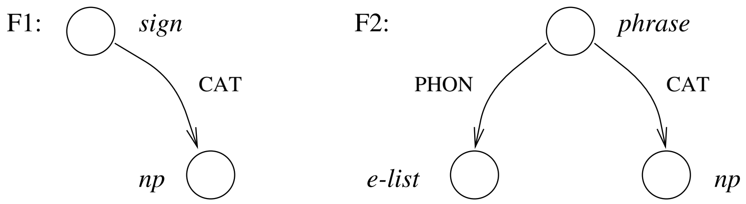


Figure 2.6: Example feature structures

Term interpretation is exactly identical to the closed world case. We can now see why the most general satisfier property does not hold for this logic. We will use the signature given in Fig. 2.1 on p. 12 and consider the feature structures in Fig. 2.6. The term we're interested in is $\neg$*phrase*. Feature structure F1 of Fig. 2.6 is described by this, whereas F2 is not. However, F1 subsumes F2. In our most general satisfier set for $\neg$*phrase* we would need a feature structure that subsumes F1, but not F2, which is clearly impossible. Under the usual definition of feature structure, this logic is thus not amenable to a unification approach. There are several options one can pursue at this point.

1. One can simply throw out negation. If a given application doesn't require the use of negation, that is the easiest option.

2. One can enrich the notion of model to accommodate negative concepts. Carpenter (1992) introduced the notion of inequated feature structures to be able to handle inequations, a restricted form of negation, in his logic. One can do the same thing for negated type expressions. However, the feature structures would then appear to lose the intuitive appeal that they had before.

3. The possibility we will pursue here is to use a constraint solving approach, instead of unification. This also lets us compare closed and open world typing more easily.

We have to consider that not all tautologies that held under a closed world interpretation also hold under an open world one. Clearly, nothing changes for the boolean tautologies. What changes is the interaction of negation with feature selection, and the negation of types. Under open world reasoning, it is not generally possible to replace a statement $\neg a$, where a is a type symbol, by another statement that doesn't involve negation. Before we can examine this issue further, we need an auxiliary definition.

Definition 2.32 (*intro*)
Let $S = \langle \mathcal{T}, \preceq, \mathcal{F}, \mathit{approp} \rangle$ be a signature. $\mathit{intro} : \mathcal{F} \to 2^{\mathcal{T}}$ is a total function from the set of features to the power set of types where for each $f \in \mathcal{F}$, $\mathit{intro}(f) = \max(\{t \mid \mathit{approp}(t, f)\downarrow\})$.

If $|\mathit{intro}(f)| \leq 1$ for each $f \in \mathcal{F}$, we say that S obeys the *feature introduction condition* (Carpenter 1992). As we will see later, the feature introduction condition has ramifications for type inference. For a discussion of the feature introduction condition, see (King and Götz 1993).

Proposition 2.17
Let $S = \langle \mathcal{T}, \preceq, \mathcal{F}, \mathit{approp} \rangle$ be a signature, $f \in \mathcal{F}$ and ϕ be a term. Then for each open world interpretation $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ of S,

$$[\![\neg f{:}\,\phi]\!]^{\mathcal{I}} = [\![(\bigwedge_{t \in \mathit{intro}(f)} \neg t) \vee f{:}\,\neg\phi]\!]^{\mathcal{I}}$$

Proof

$[\![\neg f\!:\phi]\!]^{\mathcal{I}}$

$$
\begin{array}{rl}
= & \bigcup_{\alpha\in\mathrm{ASS}} [\![\neg f\!:\phi]\!]^{\mathcal{I}}_{\alpha} \\
= & \bigcup_{\alpha\in\mathrm{ASS}} \mathcal{U} \setminus [\![f\!:\phi]\!]^{\mathcal{I}}_{\alpha} \\
= & \bigcup_{\alpha\in\mathrm{ASS}} \mathcal{U} \setminus \{u \in \mathcal{U} \mid (\mathcal{A}(f))(u)\!\downarrow \wedge\, \mathcal{A}(f))(u) \in [\![\phi]\!]^{\mathcal{I}}_{\alpha}\} \\
= & \bigcup_{\alpha\in\mathrm{ASS}} (\{u \in \mathcal{U} \mid (\mathcal{A}(f))(u)\!\uparrow\} \cup \{u \in \mathcal{U} \mid \mathcal{A}(f))(u) \notin [\![\phi]\!]^{\mathcal{I}}_{\alpha}\}) \\
= & \bigcup_{\alpha\in\mathrm{ASS}} (\{u \in \mathcal{U} \mid \mathit{approp}(\mathcal{S}(u), f)\!\uparrow\} \cup \{u \in \mathcal{U} \mid \mathcal{A}(f))(u) \in [\![\neg\phi]\!]^{\mathcal{I}}_{\alpha}\}) \\
= & \bigcup_{\alpha\in\mathrm{ASS}} (\{u \in \mathcal{U} \mid \neg\exists t \in \mathit{intro}(f).\ u \in [\![t]\!]^{\mathcal{I}}_{\alpha}\} \cup \{u \in \mathcal{U} \mid \mathcal{A}(f))(u) \in [\![\neg\phi]\!]^{\mathcal{I}}_{\alpha}\}) \\
= & \bigcup_{\alpha\in\mathrm{ASS}} [\![(\bigwedge_{t\in \mathit{intro}(f)} \neg t) \vee f\!:\neg\phi]\!]^{\mathcal{I}}_{\alpha} \\
= & [\![(\bigwedge_{t\in \mathit{intro}(f)} \neg t) \vee f\!:\neg\phi]\!]^{\mathcal{I}}
\end{array}
$$

■

We can now define what it means for a feature term to be in (open world) disjunctive normal form.

Definition 2.33 (DNF)
A feature term is in DNF if it is a disjunction of conjunctions (with no disjunctions "inside"), and the only place where negation occurs is in front of variables and type names.

It is obvious that each feature term can be transformed into an equivalent one in DNF in finite time. Next, we look at feature constraints.

Definition 2.34 (feature constraints)
A feature constraint is of the form $X|Y$, $X|\neg Y$, $X|t$, $X|\neg t$ or $X|f:Y$, where X, Y are variables, t is a type symbol and f is a feature symbol.

As we said above, we can not in general eliminate negation in front of types. Thus, we had to introduce an additional feature constraint that we didn't need in the closed world case. Constraint satisfaction is also identical, except that we need to say what the condition for the new constraint is.

Definition 2.35 (constraint satisfaction)

- $\mathcal{I}, \alpha \models X|\,Y$ *iff* $\alpha(X) = \alpha(Y)$
- $\mathcal{I}, \alpha \models X|\,\neg Y$ *iff* $\alpha(X) \neq \alpha(Y)$
- $\mathcal{I}, \alpha \models X|\,t$ *iff* $\mathcal{S}(\alpha(X)) \preceq t$
- $\mathcal{I}, \alpha \models X|\,\neg t$ *iff* $\mathcal{S}(\alpha(X)) \not\preceq t$
- $\mathcal{I}, \alpha \models X|\,f{:}Y$ *iff* $\mathcal{A}(f)(\alpha(X))\downarrow$ *and* $\mathcal{A}(f)(\alpha(X)) = \alpha(Y)$

We now define a normal form for feature clauses and show that every clause in normal form is satisfiable.

Definition 2.36 (open world normal form)
A feature clause Σ is in normal form iff it has the following properties:

1. *If X is a main variable in Σ, then for some $a \in \mathcal{T}$, $X|\,a \in \Sigma$.*
2. *If $X|\,f : Y, X|\,f : Z \in \Sigma$, then $Y = Z$.*
3. *If $X|\,a,\ X|\,b \in \Sigma$, then $a = b$.*
4. *If $X|\,a,\ X|\,\neg b \in \Sigma$, then $a \neq b$ and $b \preceq a$.*
5. *If $X|\,\neg a,\ X|\,\neg b \in \Sigma$, then $b \not\preceq a$.*
6. *If $X|\,Y \in \Sigma$, then Y occurs nowhere else in Σ.*
7. *If $X|\,\neg Y \in \Sigma$, then $X \neq Y$.*
8. *If $X|\,a,\ X|\,f{:}Y,\ Y|\,b \in \Sigma$, then approp$(a, f)\downarrow$ and $b \preceq$ approp(a, f).*

Condition 4. requires that negative type information must be more specific than positive one. Again with reference to our example type hierarchy on p. 12, consider the feature clause $\{X|\,\neg \textit{sign}, X|\,\textit{word}\}$. This is clearly inconsistent, since if something is a *word*, then it is necessarily also a *sign*. In $\{X|\,\neg \textit{list}, X|\,\textit{word}\}$, the information that the object denoted by X is not a *list* is redundant, because a *word* can never be a *list*, anyway. These cases are ruled out by condition 4. The clause $\{X|\,\neg \textit{word}, X|\,\textit{sign}\}$, on the other hand, is allowed. It says that the object denoted by X can be a *sign* or a *hc-phrase*, but not a *word*.

Condition 5. is not strictly necessary for satisfiability, it just keeps the constraint clauses smaller. Consider the clause $\{X \mid \neg \mathit{list}, X \mid \neg \mathit{e\text{-}list}\}$. The information that the object denoted by X is not an *e-list* is redundant, since we also know that it may not be a *list* of any kind.

Finally, note that we only need a single condition (cond. 8) to ensure that the appropriateness conditions are met. This is so because objects need not be of a maximally specific type. The appropriateness conditions can thus be checked with the types that are actually constraining the variables, with no need to "look down" at the minimal types.

Before we show that every feature clause in normal form is satisfiable, we define a useful standard interpretation. It has exactly one object of each type. This is very similar to $\mathcal{I}_S$ we defined in Sec. 2.3.2.

Definition 2.37 ($\mathcal{I}_S$)
For each signature $\Sigma = \langle \mathcal{T}, \preceq, \mathcal{F}, \mathit{approp} \rangle$, define an interpretation $\mathcal{I}_S = \langle \mathcal{U}_S, \mathcal{S}_S, \mathcal{A}_S \rangle$ with

- $\mathcal{U}_S = \{u_t \mid t \in \mathcal{T}\}$
- $\mathcal{S}_S(u_t) = t$
- $\mathcal{A}_S(f)(u_t) = \begin{cases} u_{t'} & \text{if } \mathit{approp}(t, f)\downarrow \text{ and } t' = \mathit{approp}(t, f) \\ \text{undefined} & \text{otherwise} \end{cases}$

Proposition 2.18
If $S = \langle \mathcal{T}, \preceq, \mathcal{F}, \mathit{approp} \rangle$ is a signature and Σ is a feature clause in normal form, then Σ is satisfiable.

Proof

We construct an interpretation $\mathcal{I}$ and assignment α s.t. $\mathcal{I}, \alpha \models \Sigma$. Define $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$, where

- $\mathcal{U} = \mathcal{U}_S \cup \{X \mid X \in \mathit{main}(\Sigma)\}$
- $\mathcal{S}(u) = \begin{cases} \mathcal{S}_S(u) & \text{if } u \in \mathcal{U}_S \\ t & \text{if } u = X \text{ for some } X \in \mathrm{FV}(\Sigma) \text{ and } X \mid t \in \Sigma \end{cases}$

- $\mathcal{A}(f)(u) = \begin{cases} \mathcal{A}_S(f)(u) & \text{if } u \in \mathcal{U}_S \text{ and } \mathcal{A}_S(f)(u)\downarrow \\ Y & \text{if } u = X \text{ for some } X \in \mathrm{FV}(\Sigma) \text{ and } \\ & X|\, f{:}Y \in \Sigma \\ u_{t'} & \text{if } u = X \text{ for some } X \in \mathrm{FV}(\Sigma), \\ & X|\, t \in \Sigma, \text{ for no } Y \text{ is } X|\, f : Y \in \Sigma, \\ & \mathit{approp}(t, f)\downarrow \text{ and } t' = \mathit{approp}(t, f) \\ \text{undefined} & \text{otherwise} \end{cases}$

Let β be some variable assignment. Define

$$\alpha(X) = \begin{cases} X & \text{if } X \text{ is a main variable in } \Sigma \\ Y & \text{if } X \text{ is an auxiliary variable and } Y|\, X \in \Sigma \\ \beta(X) & \text{otherwise} \end{cases}$$

It remains to be shown that for each $c \in \Sigma$, $\mathcal{I}, \alpha \models c$.

- If $X|\, Y \in \Sigma$ then $\alpha(X) = \alpha(Y)$, by definition of α.
- If $X|\, \neg Y \in \Sigma$ then $X \neq Y$ and $X|\, Y \notin \Sigma$. Thus, $\alpha(X) \neq \alpha(Y)$.
- If $X|\, a \in \Sigma$ then $\mathcal{S}(\alpha(X)) = a$.
- If $X|\, \neg a \in \Sigma$ then for some b, $X|\, b \in \Sigma$, $\mathcal{S}(\alpha(X)) = b$, $a \neq b$ and $a \preceq b$. Thus, $\mathcal{S}(\alpha(X)) \not\preceq a$.
- If $X|\, f{:}Y \in \Sigma$ then $\alpha(X) = X$, $\alpha(Y) = Y$ and $\mathcal{A}(f)(X) = Y$.

Thus, $\mathcal{I}, \alpha \models \Sigma$.

■

At this point, we need to consider how we get from terms to constraint matrices. However, since the definition of **trans** for this logic is almost identical to the closed world case, we leave it out. The same holds for the correctness proof, which is only a minor variation of the one given for Prop. 2.11.

We now turn to the definition of the rewrite system for open world reasoning, shown in Fig. 2.7 on p. 50. In large parts, it is identical to the one given for closed world reasoning.

We have already discussed the intuition behind rule 4a in the example above. Rule 4b takes care of the case when the negated type provides non-redundant information. 4b can only apply when we have a multiple inheritance hierarchy.

1) $\Gamma \cup \{\Sigma\} \longrightarrow \Gamma \cup \{\Sigma \cup \{X | \top\}\}$,
if X is a main variable in Σ and for no $a \in \mathcal{T},\ X | t \in \Sigma$

2) $\Gamma \cup \{\Sigma \cup \{X | f{:}Y, X | f{:}Z\}\} \longrightarrow \Gamma \cup \{\Sigma \cup \{X | f{:}Y, Y | Z\}\}$,
if $Y \neq Z$

3a) $\Gamma \cup \{\Sigma \cup \{X | a, X | b\}\} \longrightarrow \Gamma \cup \{\Sigma \cup \{X | c\}\}$,
if $b \neq a,\ glb(a, b)$ exists and $glb(a, b) = c$

3b) $\Gamma \cup \{\Sigma \cup \{X | a, X | b\}\} \longrightarrow \Gamma$, if $glb(a, b)$ does not exist

4a) $\Gamma \cup \{\Sigma \cup \{X | a, X | \neg b\}\} \longrightarrow \Gamma$, if $a \preceq b$

4b) $\Gamma \cup \{\Sigma \cup \{X | a, X | \neg b\}\} \longrightarrow \Gamma \cup \{\Sigma \cup \{X | a, X | \neg c\}\}$,
if $a \not\preceq b,\ glb(a, b)$ exists, $glb(a, b) = c$ and $b \neq c$

4c) $\Gamma \cup \{\Sigma \cup \{X | a, X | \neg b\}\} \longrightarrow \Gamma \cup \{\Sigma \cup \{X | a\}\}$, if $glb(a, b)\!\uparrow$

5) $\Gamma \cup \{\Sigma \cup \{X | \neg a, X | \neg b\}\} \longrightarrow \Gamma \cup \{\Sigma \cup \{X | \neg a\}\}$,
if $a \neq b$ and $b \preceq a$

6) $\Gamma \cup \{\Sigma \cup \{X | Y\}\} \longrightarrow \Gamma \cup \{\Sigma_{[X/Y]} \cup \{X | Y\}\}$,
if $X \neq Y$ and $Y \in \mathrm{FV}(\Sigma)$

7) $\Gamma \cup \{\Sigma \cup \{X | \neg X\}\} \longrightarrow \Gamma$

8) $\Gamma \cup \{\Sigma \cup \{X | a, X | f{:}Y, Y | b\}\} \longrightarrow$
$\Gamma \cup \{\Sigma \cup \{X | c, X | f{:}Y, Y | b\} \mid c \in \max(\{glb(a, d) \mid d \in intro(f)\})\}$,
if $approp(a, f)\!\uparrow$

9a) $\Gamma \cup \{\Sigma \cup \{X | a, X | f{:}Y, Y | b\}\} \longrightarrow \Gamma \cup \{\Sigma \cup \{X | a, X | f{:}Y, Y | c\}\}$,
if $approp(a, f)\downarrow,\ glb(b, approp(a, f))\downarrow,\ glb(b, approp(a, f)) = c$
and $c \neq b$

9b) $\Gamma \cup \{\Sigma \cup \{X | a, X | f{:}Y, Y | b\}\} \longrightarrow \Gamma$,
if $approp(a, f)\!\downarrow$ and $glb(b, approp(a, f))\!\uparrow$

Figure 2.7: Matrix rewriting for open world reasoning

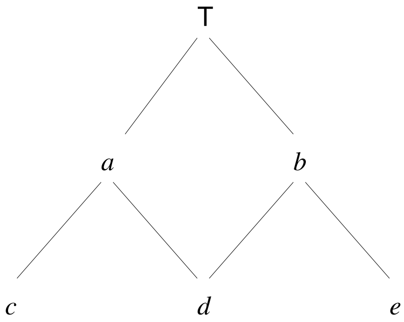


Figure 2.8: An example signature

Assume we have the signature in Fig. 2.8 and the constraints $\{X|\,a, X|\,\neg b\}$. The constraint $X|\,a$ tells us that the object denoted by X may be of type a, c or d. The constraint $X|\,\neg b$ tells us that the object denoted by X may be neither of type b, nor d, nor of type e. We thus know that the object denoted by X must be of type a or of type c. This is non-redundantly expressed with the set $\{X|\,a, X|\,\neg d\}$, which is exactly what we get from applying rule 4b to $\{X|\,a, X|\,\neg b\}$. Rules 4c and 5 take care of the cases when a negated type does not provide constraining information, as discussed with the example on p. 47 above.

Rules 8, 9a and 9b ensure the correctness of the typing information with respect to the appropriateness conditions. These rules are a straightforward encoding of the procedures *TypDom* and *TypRan* of (Carpenter 1992). Notice that rule 8 makes use of the set $intro(f)$. This is the only place where some constraint clause may be split up into several. Thus, if a signature obeys the feature introduction condition, consistency can be determined much more efficiently. Rule 8 makes sure that the variable the feature f is applied to is appropriately typed. The object denoted by X must be in the domain of f. Rules 9a and 9b do the same for the variable f maps to, ensuring that the object denoted by Y is in the range of f. Note that rule 8 requires that $approp(a, f)$ is undefined, whereas rules 9a and 9b require the opposite. This means that these rules apply in a strict sequence.

Again, we will give no proof that a set of constraints Σ is in normal form iff none of the rewrite rules applies to $\{\Sigma\}$. This can be seen by an easy case analysis.

We next consider termination of the rewrite system on finite sets of finite sets of constraints. We use the same techniques as in Sec. 2.3.2, but the weights are somewhat different in this proof.

Proposition 2.19
If Γ is a finite matrix of finite sets, then there are Γ', Γ'' and natural numbers n and m s.t. $\Gamma \xrightarrow{n} \Gamma' \xrightarrow{m} \Gamma'' \not\longrightarrow$, where rule 1) does not apply to Γ'.

Proof

Suppose $\Gamma = \{\Sigma_1, \ldots, \Sigma_k\}$. Γ' can be derived in at most $2 \times (|\Sigma_1| \times \ldots \times |\Sigma_k|)$ steps (since for each i, $|main(\Sigma_i)| \leq 2 \times |\Sigma|$). Since none of the other rules introduce any new variables, we can consider the rest of the rules without rule 1).

We define the weight of each set of constraints.

- $weight(X \,|\, a) = |\{b \mid b \preceq a\}|$
- $weight(X \,|\, \neg a) = |\{b \mid b \preceq a\}|$
- $weight(X \,|\, Y) = 1$
- $weight(X \,|\, \neg Y) = 1$
- $weight(X \,|\, f{:}Y) = 2$
- $weight(\Sigma) = \left(\sum_{\sigma \in \Sigma} weight(\sigma) \right) \times |main(\Sigma)|$

Now consider rule application, ignoring the obvious cases:

- If $\Gamma \cup \Sigma \longrightarrow \Gamma \cup \Sigma'$ by application of rule 2), then $weight(\Sigma') \leq weight(\Sigma) - 1$ $(|main(\Sigma)| \leq |main(\Sigma')|)$.
- Suppose rule 3a) is applied: $\Gamma \cup \{\Sigma \cup \{X \,|\, a, X \,|\, b\}\} \longrightarrow \Gamma \cup \{\Sigma \cup \{X \,|\, c\}\}$, where $a \neq b$, $glb(a,b)\downarrow$ and $glb(a,b) = c$. Let $A = \{a' \mid a' \preceq a\}$, $B = \{b' \mid b' \preceq b\}$ and $C = \{c' \mid c' \preceq c\}$. Assume w.l.o.g. that $|A| \geq |B|$. Since $a \neq b$ and $glb(a,b)\downarrow$, we know that $|A| \geq 2$. Thus, $weight(X \,|\, a) + weight(X \,|\, b) = |A| + |B| \geq 2 + |B| > |B| \geq |C| = weight(X \,|\, c)$.
- If rule 4b) applies, then $c \preceq b$ and $c \neq b$. Thus, $weight(X \,|\, \neg c) < weight(X \,|\, \neg b)$.

- Suppose rule 6) applies: $\Gamma \cup \{\Sigma \cup \{X \,|\, Y\}\} \longrightarrow \Gamma \cup \{\Sigma_{[X/Y]} \cup \{X \,|\, Y\}\}$, with $X \neq Y$ and $Y \in \mathrm{FV}(\Sigma)$. Now $\sum_{\sigma \in (\Sigma \cup \{X \,|\, Y\})} \mathit{weight}(\sigma) = \sum_{\sigma \in (\Sigma_{[X/Y]} \cup \{X \,|\, Y\})} \mathit{weight}(\sigma)$, but $|\mathit{main}(\Sigma \cup \{X \,|\, Y\})| = |\mathit{main}(\Sigma_{[X/Y]} \cup \{X \,|\, Y\})| - 1$. Thus, $\mathit{weight}(\Sigma_{[X/Y]} \cup \{X \,|\, Y\}) < \mathit{weight}(\Sigma \cup \{X \,|\, Y\})$.

- If rule 8) applies, then for each new c, $c \preceq b$ and $c \neq b$. Thus, $\mathit{weight}(X \,|\, c) < \mathit{weight}(X \,|\, b)$.

- Rule 9a): same as rule 7).

■

As before, we can roughly estimate the complexity of this procedure. If the feature introduction condition does not hold, the cost is also exponential in the size of the clause, due to rule 8. If feature introduction *does* hold, however, rule 8 will never split up the clause into several ones. The number of times we have to consider combinations of constraints is then bounded by a constant factor (determined by the signature), the the cost is linear in the size of the clause. Notice also that under feature introduction, the number of clauses we get out of the rewrite system is less than or equal to the number we put in.

The run-time behaviour under open world type inference is not quite so nice. The typing system is not static. We will give a simple example involving rule 9b. Rule 9a may also need to be applied at run-time. Rule 8, on the other hand, does not. Consider the signature in Fig. 2.9. Suppose we have the clauses $\Sigma = \{X \,|\, a, X \,|\, f\!: Y, Y \,|\, \mathit{true}\}$ and $\Sigma' = \{Z \,|\, c\}$, both in normal form. Now we try to normalise $\{\Sigma \cup \Sigma' \cup \{X \,|\, Z\}\}$, yielding the following derivation.

$$\{\{X \,|\, a, X \,|\, f\!:\! Y, Y \,|\, \mathit{true}, X \,|\, Z, Z \,|\, c\}\}$$

$$\xrightarrow{1} \quad \{\{X \,|\, a, X \,|\, c, X \,|\, Z, X \,|\, f\!:\! Y, Y \,|\, \mathit{true}\}\} \qquad (6)$$

$$\xrightarrow{1} \quad \{\{X \,|\, c, X \,|\, Z, X \,|\, f\!:\! Y, Y \,|\, \mathit{true}\}\} \qquad (3a)$$

$$\xrightarrow{1} \quad \emptyset \qquad (9b)$$

We already mentioned above that under the feature introduction condition, $|\mathrm{NF}(\{\Sigma\})| \leq 1$. Carpenter (1992) shows that under certain conditions, $\mathrm{NF}(\{\Sigma\})$ gives us a unique (up to renaming) most general satisfier.

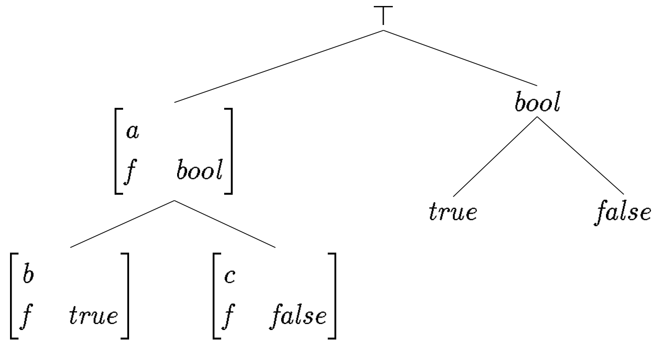


Figure 2.9: An example signature

Definition 2.38 (*approp*$^+$)
Define approp$^+$ as the smallest relation s.t.

- *approp*$^+(t, t')$ *if approp*$(t, f) = t'$ *for some* f
- *approp*$^+(t, t'')$ *if approp*$(t, f) = t'$ *for some* f *and approp*$^+(t', t'')$

Please note that *approp*$^+$ has nothing to do with *approp**.

Proposition 2.20
For each feature term ϕ, there is a most general satisfier if the following conditions are met (Carpenter 1992):

1. *The feature introduction condition holds.*
2. *ϕ contains no disjunction and negation.*
3. *approp$^+$ is anti-symmetric.*

The third condition ensures that the trivial interpretation $\mathcal{I}_S$ contains no cycles.

Finally, we show that the rewrite system preserves the denotation of a matrix.

Proposition 2.21
For each matrix Γ, Γ', interpretation $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ and assignment α, if $\Gamma \longrightarrow \Gamma'$, then

$\mathcal{I}, \alpha \models \Gamma$ *iff* $\mathcal{I}, \alpha \models \Gamma'$

Proof

Rule 1): obvious, since $\mathcal{I}, \alpha \models X \,|\, \top$ is always true.

Rule 2):

$\mathcal{I}, \alpha \models \{X \,|\, f{:}Y, X \,|\, f{:}Z\}$

$\Leftrightarrow\ \mathcal{A}(f)(\alpha(X))\!\downarrow,\ \mathcal{A}(f)(\alpha(X)) = \alpha(Y)$ and $\mathcal{A}(f)(\alpha(X)) = \alpha(Z)$

$\Leftrightarrow\ \mathcal{A}(f)(\alpha(X))\!\downarrow,\ \mathcal{A}(f)(\alpha(X)) = \alpha(Y)$ and $\alpha(Y) = \alpha(Z)$

$\Leftrightarrow\ \mathcal{I}, \alpha \models \{X \,|\, f{:}Y, Y \,|\, Z\}$

Rule 3a): obvious, since for each $u \in \mathcal{U}$, $\mathcal{S}(u) \preceq glb(a, b)$ iff $\mathcal{S}(u) \preceq a$ and $\mathcal{S}(u) \preceq b$.

Rule 3b): obvious, since if $glb(a, b)\!\uparrow$, then for each $u \in \mathcal{U}$, $\mathcal{S}(u) \not\preceq a$ or $\mathcal{S}(u) \not\preceq b$.

Rule 4a): if $a \preceq b$ then for no $\mathcal{I}, \alpha$: $\mathcal{I}, \alpha \models \{X \,|\, a, X \,|\, \neg b\}$.

Rule 4b): $\mathcal{I}, \alpha \models \{X \,|\, a, X \,|\, \neg b\}$, $glb(a, b)\!\downarrow$ and $glb(a, b) = c$

$\Leftrightarrow\ \mathcal{S}(\alpha(X)) \preceq a,\ \mathcal{S}(\alpha(X)) \not\preceq b$ and $glb(a, b) = c$

$\Leftrightarrow\ \mathcal{S}(\alpha(X)) \preceq a$ and $\mathcal{S}(\alpha(X)) \not\preceq c$

$\Leftrightarrow\ \mathcal{I}, \alpha \models \{X \,|\, a, X \,|\, \neg c\}$

Rule 4c): $\mathcal{I}, \alpha \models \{X \,|\, a\}$ and $glb(a, b)\!\uparrow$

$\Rightarrow\ \mathcal{S}(\alpha(X)) \preceq a$ and $\mathcal{S}(\alpha(X)) \not\preceq b$

$\Rightarrow\ \mathcal{I}, \alpha \models \{X \,|\, a, X \,|\, \neg b\}$

Rule 5): $\mathcal{I}, \alpha \models \{X \,|\, \neg a\}$ and $b \preceq a$

$\Rightarrow\ \mathcal{S}(\alpha(X)) \not\preceq a$ and $\mathcal{S}(\alpha(X)) \not\preceq b$

$\Rightarrow\ \mathcal{I}, \alpha \models \{X \,|\, \neg a, X \,|\, \neg b\}$

Rule 6): $\mathcal{I}, \alpha \models \Sigma \cup \{X \,|\, Y\}\ \Leftrightarrow\ \mathcal{I}, \alpha \models \Sigma$ and $\alpha(X) = \alpha(Y)\ \Leftrightarrow\ \mathcal{I}, \alpha \models \Sigma_{[X/Y]} \cup \{X \,|\, Y\}$.

Rule 7): obvious.

Rule 8): right to left is obvious. Consider left to right:

$\mathcal{I}, \alpha \models \{X \,|\, a, X \,|\, f{:}Y, X \,|\, b\}$

$\Rightarrow\ \mathcal{S}(\alpha(X)) \preceq a$ and for some $d \in \mathit{intro}(f),\ \mathcal{S}(\alpha(X)) \preceq d$

$\Rightarrow$ for some $c \in \max(\{\mathit{glb}(a,d) \mid d \in \mathit{intro}(f)\}),\ \mathcal{S}(\alpha(X)) \preceq c$

$\Rightarrow\ \mathcal{I}, \alpha \models \{X \mid c\}$

Rule 9a): again, right to left is obvious. For left to right, we have:

$\mathcal{I}, \alpha \models \{X \mid a, X \mid f{:}Y, Y \mid b\}$

$\Rightarrow\ \mathcal{S}(\alpha(X)) \preceq a,\ \mathcal{A}(f)(\alpha(X))\!\downarrow,\ \mathcal{A}(f)(\alpha(X)) = \alpha(Y)$ and $\mathcal{S}(\alpha(Y)) \preceq b$

$\Rightarrow\ \mathit{approp}(a,f)\!\downarrow,\ \mathit{glb}(b, \mathit{approp}(a,f))\!\downarrow$ and $\mathcal{S}(\alpha(Y)) \preceq \mathit{glb}(b, \mathit{approp}(a,f))$

$\Rightarrow\ \mathcal{I}, \alpha \models \{X \mid a, X \mid f{:}Y, Y \mid \mathit{glb}(b, \mathit{approp}(a,f))\}$

Rule 9b): this is also obvious, since the clause that is removed is clearly inconsistent.

■

2.6 Conclusion

In this chapter, we have investigated the foundations of feature logics. We defined what signatures, terms and interpretations are, and examined the computational properties of the satisfiability problem.

We chose to address the satisfiability problem with a constraint-based approach. We feel that a constraint-based approach improves upon a unification-based approach by keeping semantics and computation strictly separate. It obviates the need for things like, e.g., inequated feature structures, which are strictly unnecessary from a semantic point of view and were only introduced to make unification work in the presence of path inequations.

The constraint-based approach, on the other hand, is very flexible and can easily accommodate language extensions. It is also straightforward to use in as a relational constraint language, as we will see in ch 6. There is no need to invent things like multi-rooted feature structures, or encode relations (like phrase structure rules) as feature structures (cf. Shieber 1989). These properties are built into the constraint approach.

Chapter 3

Constraint grammars

In this chapter, we will examine the notion of grammar in our feature logic setting. In particular, we will be interested in how a grammar defines the grammatical structures, and how it rules out the ungrammatical ones. Apart from details of the underlying formalism, it is these properties that can help us distinguish between different grammar formalisms. Grammar formalisms seem to fall into two basic categories, namely *constraint grammars* and *rule-based grammars*. Constraint grammars are characterized by the fact that their expressions are interpreted as constraints that rule out non-well formed structures. That is, any structure is considered well-formed unless specifically ruled out by a grammar expression. In rule-based grammars, on the other hand, grammar expressions are considered as *production rules*. No linguistic structure is considered well-formed unless it is generated by some grammar expression.

(1)	S	::=	NP VP
(2)	VP	::=	IV | SV S
(3)	IV	::=	sleeps
(4)	SV	::=	know
(5)	NP	::=	Arthur | knights

Figure 3.1: A context free grammar

As an example, let us consider context free grammars as sets of expressions in Backus-Naur form (BNF). Context free grammars are traditionally inter-

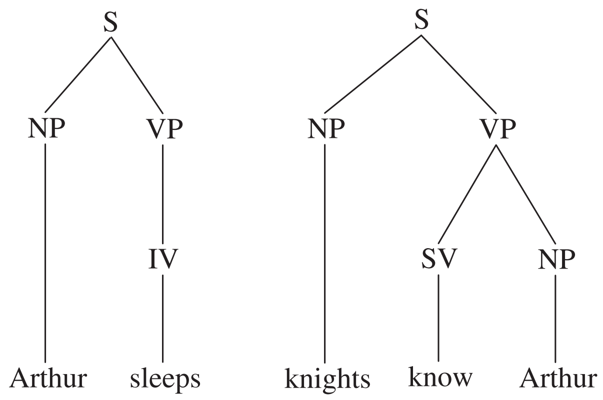


Figure 3.2: Example trees

preted as rule-based grammars. However, one can equally well define them as constraint grammars, as we shall see in the following example. Consider the CFG in fig. 3.1 and the trees in fig. 3.2. Under a rule-based interpretation, every local tree must be generated by one of the rules in the grammar. Thus, the first tree is grammatical, since the local trees are generated by rules (1), (5), (2) and (3), respectively. The second tree is not well-formed, since the local tree with VP as mother an SV and NP as daughters is not generated by any of the productions. Under a constraint view, we interpret the BNF expressions as implications. If the mother of a local tree is labeled with the category to the left of the ::=, then the daughters must be labeled with the categories to the right of the ::=. Under this view, the first tree is well-formed, since all local trees obey all the constraints. The second tree is non-well formed, since the offending local tree is ruled out by constraint (2).

The way we have described it, the rule-based view and the constraint view seem to be equivalent for context free grammar. However, this is not quite true. Suppose there is a non-terminal symbol X that does not appear as the left-hand side of a BNF expression. The rule-based view holds then that *no* local tree whose mother is labeled with X is well-formed, since there is no rule to generate such a tree. The constraint view, on the other hand, says that *every* local tree with X labeling the mother is well-formed, since there is no constraint to rule it out. For CFGs, the two approaches can be made to coincide exactly[1], but in general, this exemplifies again the main difference between the two views. In the absence of constraints, everything is well-formed, whereas in the absence of rules, nothing is.

As far as computation is concerned, at first glance it would appear that a constraint approach must be much less efficient. To check some structure for well-formedness, every substructure (local tree in the case of CFGs) must be checked against *every* constraint. Under a rule-based approach, one only needs to check each substructure against *some* rule. We will see later that with appropriate grammar transformation and indexing techniques, computation with constraint grammars can be just as efficient as with rule-based grammars.

3.1 Grammar formalisms

Having drawn the basic distinction between constraint and rule-based grammars, we can now see where some of the current grammar formalisms fit into the scheme. What, for example, is the status of so-called *constraint-based* grammar formalisms? Let us begin by considering a simple example from computational linguistics, PATR-II (Shieber 1989).

$$
\begin{array}{l}
\mathrm{S} \;\rightarrow\; \mathrm{NP}\ \mathrm{VP}, \\
\quad \begin{array}{lcl}
\mathrm{NP{:}agr} & \approx & \mathrm{VP{:}agr}, \\
\mathrm{S{:}cat} & \approx & \mathrm{VP{:}cat}, \\
\mathrm{VP{:}cat} & \sim & \mathrm{v}, \\
\mathrm{NP{:}cat} & \sim & \mathrm{n}.
\end{array}
\end{array}
$$

Figure 3.3: A PATR-II rule

Figure 3.3 shows an example PATR-II rule. It is to be understood as a phrase structure rule generating a local tree. The nodes of the tree are not simply labeled with atomic symbols, but with complex feature structures. The first line of the rule says that it generates a tree with two daughters. The following lines place constraints on the possible feature structures the nodes of the tree can be labeled with. It is possible for feature structures on distinct nodes to share feature values. In the example, the agr value of the

[1] E.g., with an appropriate completion operation that adds a constraint for each non-terminal symbol not occurring on the left-hand side of a grammar expression.

NP node is required to be identical to the agr value of the vp node. The $\sim$ operator assigns atomic values to features (or more generally, paths).

Notice how the constraints in this setup work only locally. They restrict the local trees generated by the rule. It is not possible in PATR-II to state *universal* constraints that apply to all local trees. For example, it is not possible to state the restriction that every non-terminal node must have a head, a daughter that shares its cat value, for example. Thus, under the distinction drawn in the previous section, PATR-II falls into the class of rule-based formalisms.

A similar observation can be made about lexical-functional grammar (LFG, Kaplan and Bresnan 1982). Intuitively, LFG and PATR-II are more or less notational variants. Figure 3.4 shows an example of a LFG rule.

$$\begin{array}{lccc} \mathrm{S} & \rightarrow & \mathrm{NP} & \mathrm{VP} \\ & & (\uparrow \mathrm{subj}) = \downarrow & \uparrow = \downarrow \end{array}$$

Figure 3.4: A LFG rule

Apart from the fact that LFG uses the symbol $\uparrow$ to refer to the mother of the local tree, and the symbol $\downarrow$ to refer to the annotated category, the expressive means of both systems are very similar, though not identical. What is of interest here is that the LFG notion of grammaticality is one of generation, just as with PATR-II. There is no way in LFG to place universal constraints on structures. However, there is one universal constraint that is part of every LFG grammar, namely the off-line parsability constraint. Using the formulation of Johnson (1988), we can define the off-line parsability constraint as follows.

Definition 3.1 (off-line parsability constraint)
Every constituent structure must obey the following constraints.

1. *It must not include a non-branching dominance chain with the same category labeling two distinct nodes.*
2. *The empty string must not label any terminal node.*

If we take the off-line parsability constraint to be part of every LFG grammar, then LFG is not a pure rule-based system, but rather a mixed formalism, with both rules and constraints. However, since this constraint is part of *every* grammar, it seems much more plausible to interpret it as a constraint on LFG model structures in general, much like the constraint that all trees must be finite. We can thus claim with some justification that LFG is a rule-based formalism.

Let us now turn to another generative grammar formalism, namely Generalised Phrase Structure Grammar (GPSG, Gazdar et al. 1985). In GPSG, many different mechanisms interact to form a complex architecture: immediate dominance (ID) rules, linear precedence (LP) statements, meta rules, feature co-occurrence restrictions (FCRs), feature specification defaults (FSDs) and universal feature instantiation principles. The universal feature instantiation principles are supposed to be the same for every grammar, and thus have a similar status as the off-line parsability constraint. Let us see how we can fit the rest of GPSG's grammatical apparatus into our rules vs. constraints distinction.

- ID rules are clearly rules, just as the phrase structure rules of PATR-II or LFG.
- LP statements are constraints. Their interpretation is that no local tree may violate an LP statement. In other words, every LP statement must be true of every local tree.
- Meta rules fall out of our schema, since they produce new ID rules from given ones, and do not directly generate or constrain linguistic structures. We thus consider them part of the ID component.
- FCRs are also global constraints. Every "feature structure" that labels a node in a tree must obey all the FCRs. In fact, the appropriateness conditions as we introduced them play a role very similar to FCRs. See (Gerdemann and King 1993) for discussion.
- The interpretation of FSDs is somewhat tricky. However, for our purposes it is enough to say that, everything else being equal, a local tree must satisfy the FSDs. Thus, FSDs are also interpreted as global constraints.

Of the grammar frameworks considered so far, GPSG is the only one that is constraint-based in the sense of constraint as we use it here. It is clearly a mixed system with a set of rules (ID rules) and a variety of global constraints. If one considers this setup with a view to processing, the ID rules play a primary role. They generate structures which then have to pass the filters of the global constraints. This is even made explicit in the formal definition of tree admissibility given in (Gazdar et al. 1985).

1. rule system
   (a) lexicon
   (b) syntax
      i. categorial component
      ii. transformational component
   (c) PF-component
   (d) LF-component
2. principles
   (a) bounding theory
   (b) government theory
   (c) θ-theory
   (d) binding theory
   (e) Case theory
   (f) control theory

Figure 3.5: Components of a GB grammar (Chomsky 1981)

As far as it is possible to determine, this is similar in another grammatical framework, namely Government and Binding theory (GB, Chomsky 1981). Figure 3.5 shows the components of a GB grammar, which is subdivided into rules and principles. The rules can be interpreted to be rules in our sense. The lexicon and the categorial component of syntax together generate D-structures. The transformational component consists only of the rule Move-α, generating S-structures from D-structures. From S-structures, the PF- and LF-components then generate PF- and LF-representations, respectively.

The principles, on the other hand, can be construed as constraints. This is not totally clear-cut, since some principles are formulated as constraints on, for example, Move-α. Now this is not a constraint in our sense, since by constraint we mean restrictions on objects, not rules. However, it seems that principles constraining rules in GB can generally be reinterpreted as constraints on structures, i.e., trees. We can thus conclude that GB, like

GPSG, is a mixed framework, relying on both rules and constraints to generate grammatical structures.

It is worth pointing out that this is not the only possible way to characterize GB. It has also variously been cast as a purely constraint-based formalism. See (Rogers 1999) and references cited therein.

In HPSG (Pollard and Sag 1994) finally, grammars generally appear to be constraint-based. This seems to be clear from the original formulation in (Pollard and Sag 1987), although there is less formal elaboration in the 1994 book. Still, the general idea is that potential grammatical structures are filtered out by constraints. There is some confusion as to the use of relations. It is generally assumed that some logic programming like extension exists for HPSG grammars. We will return to this issue in ch. 6.

3.2 Validity vs. satisfiability-based approaches

After the intuitive considerations of the last section, we will now approach grammatical frameworks from a more formal angle. Following much recent work, we will consider a grammar to be a set of formulae (a *theory*) of some logic. The grammar framework may then determine the exact shape these formulae can take, but the interpretation is taken care of by the logic. The grammatical structures are simply the models of the theory. This approach has two decisive advantages.

- The meaning of any given grammar is precisely defined. It is, at least in principle, possible to falsify a purported grammar of a given language by showing that it wrongly excludes grammatical sentences, or admits ungrammatical ones.
- If one wants to provide a computational model of the grammar theory, it is sufficient to provide a model of the underlying logic, if that is possible. If the grammatical theory then changes, as it invariably does, the computational model is still appropriate if the underlying logic hasn't changed.

The disadvantage of this approach is that only formulae of the chosen logic are allowed as grammar statements. This can mean that an intuitively simple statement may have to be spelled out as a long and complicated formula. It is thus the responsibility of the computational linguist or logician to come up with a logic that not only provides the expressive power to state all grammatical rules and constraints, but that allows one to do so in a concise and intuitive way.

We said above that in this logic-based approach, the grammatical structures are the models of the grammatical theory, where model here means model in the logical sense. However, we are normally interested in more general questions. We will want our system to at least be able to tell us if there are grammatical structures for a given string in our language, without having to tell the system what the analysis should look like. In other words, we want to solve the parsing problem. Or we may want to give the system a partial description of some ungrammatical structure, and see if this structure is actually excluded by our grammar. In general, given a theory Θ and a single expression ϕ, we want to know if Θ admits ϕ or not.

Following Johnson (1994), we can identify at least two approaches how this informal notion can be made precise for any given theory of grammar. The first one is based on the standard notion of logical consequence. Given a theory Θ and an expression ϕ, we ask whether ϕ is a logical consequence of Θ, i.e., whether ϕ is true in *all* models of Θ. Johnson (1994) calls this the validity-based approach, since for finite theories and logics with a deduction theorem, determining logical consequence is equivalent to determining validity. The alternative is to ask if ϕ is true in *some* model of Θ, a satisfiability problem.

Let us consider this distinction from a computational point of view. Suppose our underlying logic is standard first order predicate logic (FOL). We know that the validity problem for FOL is undecidable. However, by the completeness theorem, we know that we can enumerate the valid FOL formulae. Combining the two results tells us that we can not enumerate the ones that are not valid. Thus, if we express the admission problem (does Θ admit ϕ) as a validity problem, we know that an algorithm exists that

- will terminate, if Θ does in fact admit ϕ, and
- may not terminate, if Θ does not admit ϕ.

Now to determine satisfiability is in some sense the co-problem to determining validity. To be precise, we can express the question if ϕ is valid equivalently as the question if $\neg\phi$ is *unsatisfiable*. From this fact, we can conclude that a notion of admission based on satisfiability has the following properties.

- It is undecidable.

- The grammatical expressions are *not* recursively enumerable. That is, there is no algorithm that is guaranteed to terminate if Θ admits ϕ.

- The *ungrammatical* expressions *are* recursively enumerable.

It does not seem particularly important that the set of grammatical expressions is not recursively enumerable. Since practically, we can not tell if a query will never terminate, or it simply hasn't terminated yet, but will in a short while, the real difference is between decidable and undecidable formalisms. For undecidable grammar formalisms, it would not appear to make a practical difference whether the admissibility problem itself is enumerable, or its co-problem.

An interesting question is what, if any, is the connection between rules vs. constraints and the validity vs. satisfiability distinction. We can observe for now that Johnson's example for a validity-based approach, namely Definite Clause Grammars (DCGs), is a rule-based formalism in our sense. His example for a satisfiability-based approach, a system very similar to the one we shall define shortly, is constraint based.

3.3 Feature constraint grammars

In this section, we will define grammars in our feature logic and the Generalised parsing problem. We will then explore how this architecture relates to FOL and what this tells us about the complexity of our grammars.

Definition 3.2 (grammar)
A grammar is a pair $\langle S, \Theta\rangle$, where S is a signature and Θ is a finite set of S-terms.

Definition 3.3 (model)
An interpretation $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}\rangle$ is a model of a set Θ of feature terms with respect to a signature S iff for every $u \in \mathcal{U}$ and for every $\phi \in \Theta$, $u \in [\![\phi]\!]^{\mathcal{I}}$.

This is equivalent to saying that for every $\phi \in \Theta$, $[\![\phi]\!]^{\mathcal{I}} = \mathcal{U}$. The definition as we've given it emphasizes the constraint nature of grammar terms: every constraint must be true of every object.

Grammaticality is a relation between grammars and linguistic structures. However, as we pointed out above, a more interesting relation is generally one between grammars and terms (or descriptions). Informally, we called this the admission problem. We will continue to use admission as a general term for all grammar formalisms. Following King (1995), we will call the technical notion for our grammars *prediction.*

Definition 3.4 (prediction)
$\mathcal{G} = \langle S, \Theta \rangle$ predicts a term ϕ iff there is a model $\mathcal{I}$ of $\mathcal{G}$ s.t. $[\![\phi]\!]^{\mathcal{I}} \neq \emptyset$. We call ϕ a query.

This definition clearly makes our approach satisfiability based. $\mathcal{G}$ predicts ϕ just in case there is at least one model of $\mathcal{G}$ that makes ϕ true. This will become even clearer when we consider how to translate the prediction problem into a FOL satisfiability problem.

3.3.1 Translating to first order logic

It is clear that the feature logic we've defined is closely related to (a subset of) FOL. Johnson (1991) showed this for the untyped case. A translation for Speciate Re-entrant Logic (SRL, King 1994) was recently proposed and proven correct in (Aldag 1997). Aldag's work carries over almost unchanged to our case.

We begin by defining a FOL signature for every $\mathcal{FL}$ signature. We simply encode types as unary predicates, and features as binary ones. No other symbols are needed. Next, we define a set of axioms to encode the properties of the type hierarchy and the appropriateness conditions. Given a signature $S = \langle \mathcal{T}, \preceq, \mathcal{F}, \mathit{approp} \rangle$, define the translation S' of S as the union of the following sets.

1. $\{\forall x. \bigvee\{v(x) \mid v \in \mathcal{V}\}\}$
2. $\{\forall x. v(x) \rightarrow \neg v'(x) \mid v, v' \in \mathcal{V}, v \neq v'\}$
3. $\{\forall x. t(x) \rightarrow \bigvee\{v(x) \mid v \in \mathcal{V}, v \preceq t\} \mid t \in (\mathcal{T} \setminus \mathcal{V})\}$
4. $\{\forall xyz. (f(x,y) \wedge f(x,z)) \rightarrow y = z \mid f \in \mathcal{F}\}$
5. $\{\forall x. v(x) \rightarrow \exists y. f(x,y) \wedge t(y) \mid v \in \mathcal{V},\ t \in \mathcal{T},\ f \in \mathcal{F},\ \mathit{approp}(v,f)\downarrow,$ $\mathit{approp}(v,f) = t\}$
6. $\{\forall x. v(x) \rightarrow \neg\exists y. f(x,y) \mid v \in \mathcal{V}, f \in \mathcal{F}, \mathit{approp}(v,f)\uparrow\}$

The first axiom encodes the fact that each object is of a minimal type. The second one says that no object can be of two different minimal types. Axiom set 3 encodes inheritance of the type hierarchy. If an object is of a non-minimal type t, then it is also of some minimal type subsumed by t. The other axiom sets encode the properties of features. Set 4 says that features are functional. Set 5 says that for each object u of some minimal type v, the feature f must be defined on u if f is appropriate for v, and the object u is mapped to by f must be of the correct type. Finally, axiom set 6 requires that all features that are not appropriate for minimal type v must be undefined on each object of type v.

We now turn to encoding $\mathcal{FL}$ terms (see sec. 2.2) in FOL. Since our terms look rather different from SRL descriptions, we have to use a modified version of Aldag's translation. In fact, our translation will be very similar to the **trans** procedure we use for converting terms into constraint sets. This highlights the close relation between our approach to constraint solving and the translation to FOL. Given the translation of signatures above, we can view constraint clauses as FOL formulae. For example, the clause $\{X \,|\, a, X \,|\, f\colon Y, Y \,|\, b\}$ can be written as the formula $\exists X. \, (a(X) \wedge \exists Y. \, (f(X,Y) \wedge b(y)))$. However, the translation to FOL is more general as it can also handle terms that are not in disjunctive normal form. Below, we give the definition of the translation procedure, which we'll call **fol**. Like **trans**, **fol** takes two arguments: a variable and a term.

- $\mathbf{fol}(X, Y) := (X = Y)$, if Y is a variable.
- $\mathbf{fol}(X, t) := t(X)$, if $t \in \mathcal{T}$.
- $\mathbf{fol}(X, f : \phi) := \exists Y. \, (f(X,Y) \wedge \mathbf{fol}(Y, \phi))$, where Y is new.
- $\mathbf{fol}(X, (\phi \wedge \psi)) := (\mathbf{fol}(X, \phi) \wedge \mathbf{fol}(X, \psi))$.
- $\mathbf{fol}(X, (\phi \vee \psi)) := (\mathbf{fol}(X, \phi) \vee \mathbf{fol}(X, \psi))$.
- $\mathbf{fol}(X, (\neg\phi)) := (\neg\mathbf{fol}(X, \phi))$.

Notice that **fol** produces open formulae with exactly one free variable. We will make use of this fact to encode the prediction problem.

Definition 3.5
Let S be a signature, Θ a set of S-terms and ϕ a S-term. Define the translation

$$\mathbf{fol}(S, \Theta, \phi) = S' \cup \{\forall X. \, \mathbf{fol}(X, \theta) \mid \theta \in \Theta\} \cup \{\exists X. \, \mathbf{fol}(X, \phi)\},$$

where S' is the translation of S as defined above, and $X \notin \mathrm{FV}(\Theta \cup \{\phi\})$.

For every constraint θ, the root variable of its translation is universally quantified. This has the effect of making θ true of every object in the domain, which is exactly what we want for a constraint. The root variable of ϕ, on the other hand, is existentially quantified. It is sufficient for *some* object in the domain to make the translation of ϕ true.

We're now ready to state the main result of this section.

Proposition 3.1
Let S be a signature, Θ a set of S-terms and ϕ a S-term.

$$\Theta \textit{ predicts } \phi \textit{ iff } \mathbf{fol}(S, \Theta, \phi) \textit{ is satisfiable.}$$

Proof

See the proof in (Aldag 1997). It carries over with minor modifications due to the different syntax.

■

Given our discussion of the complexity of the satisfiability problem above, we thus conclude that the non-prediction problem is recursive, i.e., there is an algorithm that will terminate in finite time if a theory Θ does not predict some term ϕ. Unfortunately, this result tells us nothing about the positive case, i.e., prediction. If by some chance we've hit upon a fragment of FOL that is decidable, then the prediction problem would be recursive, as well. However, we shall see in the next section that this is not the case.

3.3.2 An example

Let us return for a moment to the constraint vs. rule-based and satisfiability vs. validity distinctions. We saw that the definition of model that we gave makes our grammars constraint grammars. Given our definition of models, only a satisfiability-based approach to prediction makes sense. To see this, let us consider an example. We will encode the equivalent of the context free grammar from the beginning of this section. The relevant signature is shown in fig. 3.6. NT stands for non-terminal, BR for branching, NBR for non-branching and PT for pre-terminal. We need two different VPs (VP1 and VP2), since we have both a branching and a non-branching VP rule. The theory encoding the grammar is shown in fig 3.7.

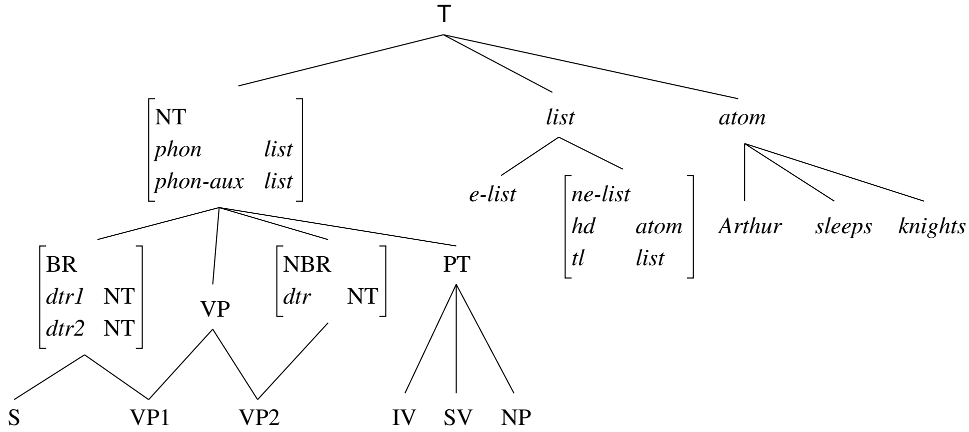


Figure 3.6: An example signature

1. $\mathrm{BR} \rightarrow \mathit{phon} : P0 \wedge \mathit{phon\text{-}aux} : P \wedge$
   $\mathit{dtr1} : (\mathit{phon} : P0 \wedge \mathit{phon\text{-}aux} : P1) \wedge$
   $\mathit{dtr2} : (\mathit{phon} : P1 \wedge \mathit{phon\text{-}aux} : P)$
2. $\mathrm{NBR} \rightarrow \mathit{phon} : P0 \wedge \mathit{phon\text{-}aux} : P \wedge$
   $\mathit{dtr} : (\mathit{phon} : P0 \wedge \mathit{phon\text{-}aux} : P)$
3. $\mathrm{PT} \rightarrow \mathit{phon} : [\mathit{atom} \mid T] \wedge \mathit{phon\text{-}aux} : T$
4. $\mathrm{S} \rightarrow \mathit{dtr1} : \mathrm{NP} \wedge \mathit{dtr2} : \mathrm{VP}$
5. $\mathrm{VP} \rightarrow (\mathit{dtr} : \mathrm{IV} \;\vee (\mathit{dtr1} : \mathrm{SV} \wedge \mathit{dtr2} : \mathrm{S}))$
6. $\mathrm{IV} \rightarrow \mathit{phon} : \mathit{hd} : \mathit{sleeps}$
7. $\mathrm{SV} \rightarrow \mathit{phon} : \mathit{hd} : \mathit{know}$
8. $\mathrm{NP} \rightarrow \mathit{phon} : \mathit{hd} : (\mathit{knights} \vee \mathit{Arthur})$

Figure 3.7: Theory encoding a CFG

We use the usual conventions for abbreviating list expressions. $[\,]$ stands for *e-list*, $[\phi_1, \ldots, \phi_n]$ stands for $(\mathit{ne\text{-}list} \wedge \mathit{hd} : \phi_1 \wedge \mathit{tl} : (\ldots(\mathit{ne\text{-}list} \wedge \mathit{hd} : \phi_n \wedge \mathit{tl} : \mathit{e\text{-}list})\ldots)$ and $[\phi_1, \ldots, \phi_n \mid \psi]$ abbreviates the term $(\mathit{ne\text{-}list} \wedge \mathit{hd} : \phi_1 \wedge \mathit{tl} :$

$(\ldots(\textit{ne-list} \wedge hd : \phi_n \wedge tl : \psi)\ldots)$.

Our example uses a difference list encoding of the phonological string. Constraint 1 rules the distribution of the phonological string on binary branching trees, constraint 2 on unary branching ones. Constraint 3 says that each preterminal takes exactly one element off the string. Constraints 4 and 5 are the grammar rules, and 5-7 the lexical entries.

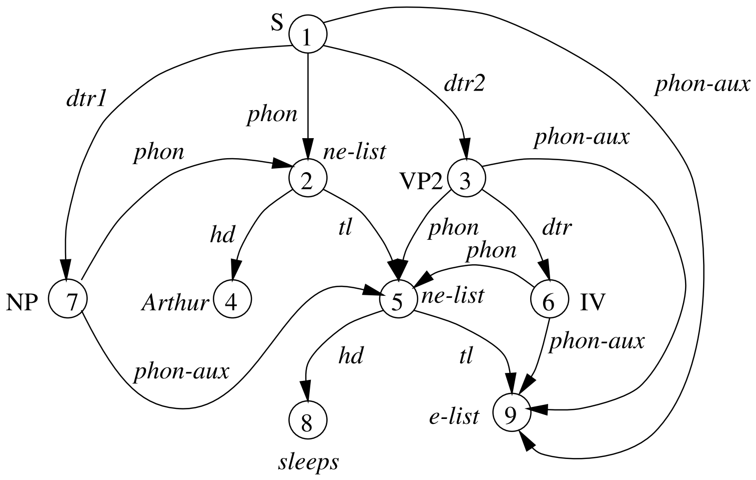


Figure 3.8: An example model

A query for the sentence "Arthur sleeps" has the form $\phi = (\mathrm{S} \wedge \textit{phon} : [\textit{Arthur}, \textit{sleeps}] \wedge \textit{phon-aux} : [\,])$. Figure 3.8 shows an example of an interpretation that makes ϕ true. With a little patience, one can verify that fig. 3.8 is in fact an interpretation according to the signature in fig. 3.6. The interpretation is also a model of our theory in fig. 3.7. Every constraint is true of every object. Thus, we know that ϕ is predicted by our theory.

Let us now consider the sentence that was excluded by our grammar, namely "knights know Arthur". As query, we have the feature term $\phi = (\mathrm{S} \wedge \textit{phon} : [\textit{knights}, \textit{know}, \textit{Arthur}] \wedge \textit{phon-aux} : [\,])$. An interpretation satisfying ϕ is given in fig. 3.9. However, this interpretation is not a model of our theory, since object 3 does not satisfy constraint 5. That constraint says that a VP is either a unary branching structure, which object 3 isn't, or that it's a binary branching structure with the second daughter an S. Since the second daughter of 3 is 10, which is an NP, we have a contradiction to constraint 5.

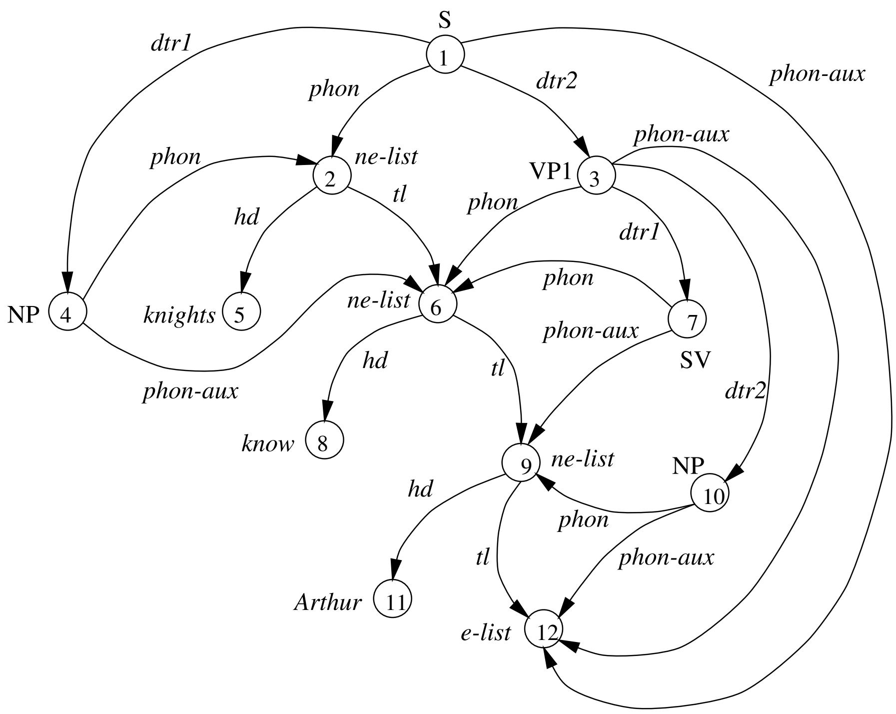


Figure 3.9: An interpretation which is not a model

Of course, one counterexample is not enough to show that our theory does not predict ϕ. To show that, we would have to prove that there is no model of our theory that satisfies ϕ. In chapter 5, we will provide a proof method that shows non-prediction in the general case.

3.4 Undecidability of prediction

In this section, we will give a proof of the undecidability of the prediction problem. We show that one can encode a well-known undecidable problem, the word problem for Thue systems, as a prediction problem. Our proof is a variation of the one given in (Aït-Kaci et al. 1993). Our contribution is a proof of the correctness of the encoding, which was omitted in (Aït-Kaci et al. 1993). An undecidability proof for prediction in the SRL of (King 1994) can be found in (King et al. 1999). That proof is based on the encoding of a tiling problem.

We begin by giving a definition of Thue systems. Next, we give a translation from word problems in Thue systems to prediction problems, and finally show that this translation is correct. Our exposition of Thue systems is based on (Lewis and Papadimitriou 1981).

Definition 3.6 (Thue systems)
A Thue system $T = \{\{u_1, v_1\}, \ldots, \{u_n, v_n\}\}$ is a finite set of unordered pairs of strings over an alphabet (a finite set of symbols) Σ. A Thue system T determines a relation $\sim_T$ on strings s.t. $x \sim_T y$ iff $x = z_1 u_i z_2$ and $y = z_1 v_i z_2$ or $x = z_1 v_i z_2$ and $y = z_1 u_i z_2$, for some $z_1, z_2 \in \Sigma^*$ and $\{u_i, v_i\} \in T$. The reflexive transitive closure $\equiv_T$ of $\sim_T$ is the equivalence relation on strings induced by T. The word problem for Thue systems is to determine, given alphabet Σ, Thue system T and two strings $x, y \in \Sigma^*$, whether $x \equiv_T y$.

A proof of the general undecidability of the word problem for Thue systems can be found in (Lewis and Papadimitriou 1981).

We can now turn to the encoding of Thue systems as feature constraint grammars. The idea is very simple. There will be only a single type in the signature, and the symbols of the alphabet Σ will be encoded as features. For every pair of strings in the Thue system, the theory will require the corresponding paths to be equated for every object in the universe.

Definition 3.7 (translation of Thue systems)
Let Σ be an alphabet and T a Thue system over Σ. The signature $S = \langle \mathcal{T}, \preceq, \mathcal{F}, approp \rangle$ encoding Σ is defined as follows.

- $\mathcal{T} = \{\top\}$,
- $\top \preceq \top$,
- $\mathcal{F} = \Sigma$, *and*
- $approp(\top, f) = \top$ *for each* $f \in \mathcal{F}$.

We need a function :: from pairs of strings and terms to terms. We will use :: as an infix operator, defined as follows.

$$v :: \phi \ := \begin{cases} \phi & \text{if } v = \varepsilon \\ f : (u :: \phi) & \text{if } v = fu,\ f \in \Sigma \end{cases}$$

Now define the theory Θ encoding T as $\Theta \ := \ \{(\top \rightarrow u :: X \wedge v :: X) \mid \{u, v\} \in T\}$.

We will now show that the word problem for Thue systems can be encoded as a *non-prediction* problem for feature constraint grammars. It is sufficient to show this, since the undecidability of non-prediction entails the undecidability of prediction, its co-problem. We will show that for each Thue system T, its encoding theory Θ and strings u, v, $u \equiv_T v$ iff Θ does not predict $(u :: X \wedge v :: \neg X)$. We will break up the proof of this into two propositions. Proposition 3.2 will prove the direction from left to right, and Prop. 3.3 the other one.

Proposition 3.2
Let Σ be an alphabet, T a Thue system, Θ the encoding of T and v, w strings over Σ. If $v \equiv_T w$, then for each model $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ of Θ, for each $u \in \mathcal{U}$, $\mathcal{A}^*(v)(u) = \mathcal{A}^*(w)(u)$.

Proof

By induction on the number of $\sim_T$ steps from v to w.

Suppose $v = w$. Then $\mathcal{A}^*(v)(u) = \mathcal{A}^*(w)(u)$.

Now suppose $v \equiv_T v'$, $v' \sim_T w$ and for each model $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ of Θ, for each $u \in \mathcal{U}$, $\mathcal{A}^*(v)(u) = \mathcal{A}^*(v')(u)$. But then $v' = z_1 v_i z_2$ and $w = z_1 w_i z_2$ or $v' = z_1 w_i z_2$ and $w = z_1 v_i z_2$ for some $z_1, z_2 \in \Sigma^*$ and $\{v_i, w_i\} \in T$. Thus, $(\top \rightarrow v_i :: X \wedge w_i :: X) \in \Theta$ and for all $u' \in \mathcal{U}$, $\mathcal{A}^*(v_i)(u') = \mathcal{A}^*(w_i)(u')$. Thus, $\mathcal{A}^*(v)(u) = \mathcal{A}^*(w)(u)$.

■

Proposition 3.3
Let Σ be an alphabet, T a Thue system, $S = \langle \mathcal{T}, \preceq, \mathcal{F}, approp \rangle$ the signature encoding Σ, Θ the encoding of T and v, w strings over Σ. If $v \not\equiv_T w$, then there is a model $\mathcal{I}$ of Θ s.t. $[\![v :: X \wedge w :: \neg X]\!]^{\mathcal{I}} \neq \emptyset$.

Proof

We define a model that has the desired property. Define $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ with

- $\mathcal{U} = \Sigma^*_{\equiv_T}$ (the set of strings modulo $\equiv_T$)
- $\mathcal{S}(u) = \top$, for each $u \in \mathcal{U}$
- $\mathcal{A}([u]_{\equiv_T})(f) = [uf]_{\equiv_T}$, for each $[u]_{\equiv_T} \in \mathcal{U}$ and $f \in \mathcal{F}$

First, we show that $\mathcal{I}$ is an S-interpretation. Clearly, the appropriateness conditions are satisfied. To see that $\mathcal{A}$ is well-defined, note that $v \equiv_T w \Rightarrow v\sigma \equiv_T w\sigma$.

Next, we show that $\mathcal{I}$ is a model of Θ. For each constraint $\theta = (\top \to u' :: X \wedge v' :: X) \in \Theta$, for each $[w]_{\equiv_T} \in \mathcal{U}$:

$\theta \in \Theta$

$\Rightarrow \{u', v'\} \in T$

$\Rightarrow wu' \equiv_T wv'$

$\Rightarrow \mathcal{A}^*([w]_{\equiv_T})(u') = [wu']_{\equiv_T} = [wv']_{\equiv_T} = \mathcal{A}^*([w]_{\equiv_T})(v')$

$\Rightarrow [w]_{\equiv_T} \in [\![\theta]\!]^{\mathcal{I}}$

■

We can now combine these results to give us our undecidability result.

Proposition 3.4
Let Σ be an alphabet, T a Thue system, $S = \langle \mathcal{T}, \preceq, \mathcal{F}, approp \rangle$ the signature encoding Σ, Θ the encoding of T and v, w strings over Σ.

$v \equiv_T w$ iff Θ does not predict $v :: X \wedge w :: \neg X$.

Proof

Right to left is prop. 3.3. For left to right, we have:

$v \equiv_T w$

$\Rightarrow$ for each model $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ of Θ, for each $u \in \mathcal{U}$, $\mathcal{A}^*(u)(v) = \mathcal{A}^*(u)(w)$ (prop. 3.2)

$\Rightarrow$ for each model $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ of Θ, for each $u \in \mathcal{U}$, $u \in [\![v :: X \wedge w :: X]\!]^{\mathcal{I}}$

$\Rightarrow$ for each model $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ of Θ, for each $u \in \mathcal{U}$, $u \notin [\![v :: X \wedge w :: \neg X]\!]^{\mathcal{I}}$

$\Rightarrow \Theta$ does not predict $v :: X \wedge w :: \neg X$

■

3.5 Normal form grammars

Although the constraints in a grammar can in principle take any form, only a certain form of term really makes sense as a constraint. Those have the form $\phi \to \psi$. In fact, usually ϕ will not be a complex term at all, but simply a type name. Notice that any term can be trivially transormed into this form when we note the following equivalence.

$$\phi \ \equiv \ \top \to \phi$$

In the later, computational chapters, we will require all grammar constraints to be of the form $t \rightarrow \phi$, where t is some type name. Since the above mentioned method does not generally yield a very compact grammar representation, we will now discuss a method how to transform constraints of the form $\phi \rightarrow \psi$ to an equivalent constraint of the form $t \rightarrow \psi'$ in a more intelligent manner.

Before we do this, let's look at a method that is sometimes found in the linguistics literature. As an example consider as an example HPSG's head feature principle. This example was adapted from (Meurers 1994). See the same paper for linguistically motivated arguments for why it is not a good idea to manipulate the type hierarchy in the way described below.

We use the following notational conventions here: feature names are written in upper case roman font, type names in lower case italics and variables in upper case italics.

Example 3.1
DTRS : *headed-struc* $\rightarrow$

$$\begin{array}{l} \mathrm{SYNSEM : LOC : CAT : HEAD :} X \ \& \\ \mathrm{DTRS : HD\text{-}DTR : SYNSEM : LOC : CAT : HEAD :} X \end{array}$$

A method often employed to get rid of the complex antecedent is to introduce new types into the type hierarchy. In this particular example, that's fairly easy: we simply introduce two new sub-types of *phrase*, *headed-phrase* and *non-headed-phrase* (assume for the moment that there were no previous sub-types of *phrase*). *headed-phrase* and *non-headed-phrase* inherit all their appropriateness specifications from *phrase*, except DTRS, which is *headed-struc* on *headed-phrase*, and *non-headed-struc* on *non-headed-phrase*. Having done this, we can now restate the head-feature principle as follows.

Example 3.2
headed-phrase $\rightarrow$

$$\begin{array}{l} \mathrm{SYNSEM : LOC : CAT : HEAD :} X \ \& \\ \mathrm{DTRS : HD\text{-}DTR : SYNSEM : LOC : CAT : HEAD :} X \end{array}$$

Of course, this scheme can not always be applied so easily. Consider a principle of the form

SYNSEM : LOC : CAT : HEAD : *verb* $\rightarrow$ ϕ, where ϕ is some term. To turn the antecedent into a type, we would have to introduce a verbal sign, e.g., *v-sign* (and its dual, *non-v-sign*). But that is not enough, since in the type hierarchy, we can not express that a *v-sign* takes the value *verb* under the path SYNSEM : LOC : CAT : HEAD, since the appropriateness specifications only always go one level down. So to give *v-sign* the desired properties, we have to introduce intermediate types *v-synsem*, *v-loc*, *v-cat*, *v-head* and their respective duals. But that's not all: since *sign* has other sub-types, say *word* and *phrase*, these have to be cross-classified with the new types, yielding yet more types *v-word*, *non-v-word*, *v-phrase* and *non-v-phrase*. We won't even consider what happens when the antecedent gets yet more complex or contains variables.

There is a different, logical approach to this problem that does not affect the type hierarchy. We will first explain the method, then consider an example.

1. Suppose we have an implication $\phi \rightarrow \psi$. Bring ϕ into disjunctive normal form $\phi_1 \vee \ldots \vee \phi_n$. This expands $\phi \rightarrow \psi$ to a set of constraints of the form $\phi_i \rightarrow \psi$, since $(\phi_1 \vee \phi_2) \rightarrow \psi \;\equiv\; (\phi_1 \rightarrow \psi) \wedge (\phi_2 \rightarrow \psi)$.

2. Suppose we have an implication $\phi \rightarrow \psi$, where ϕ is purely conjunctive. Since ϕ is conjunctive, we can find out which type t all objects that satisfiy ϕ must have, and we can transform $\phi \rightarrow \psi$ to $(t \wedge \phi) \rightarrow \psi$.

3. $(t \wedge \phi) \rightarrow \psi \;\equiv\; t \rightarrow (\neg\phi \vee \psi)$

In principle, we're done now, but we have a negated formula on the right hand side. If the negated expression does not contain any variables, we can actually transform it into a term not containing any negations (and if it does, we can at least push negation down to the variables):

- $\neg t \;\equiv\; \bigvee_{t' \in \mathcal{V}, t' \not\preceq t} t'$
- $\neg \mathrm{F} : \psi \;\equiv\; \mathrm{F} : \neg\psi \vee \bigvee\{t \mid t \in \mathcal{V}, \mathit{approp}(t, \mathrm{F})\uparrow\}$
- $\neg(\phi \wedge \psi) \;\equiv\; \neg\phi \vee \neg\psi$

This looks very disjunctive, i.e., computationally unattractive. Yet there is one piece of information that we can still use to our advantage: an inferred type. We will explain what we mean by that with an example. We abbreviate SYNSEM as SS, LOC as LC, CAT as CT, and HEAD as HD.

Example 3.3

$$\mathrm{SS}:\mathrm{LC}:\mathrm{CT}:\mathrm{HD}:\mathit{verb} \;\rightarrow\; \phi$$

$$\equiv\; \mathit{sign} \wedge \mathrm{SS}:\mathrm{LC}:\mathrm{CT}:\mathrm{HD}:\mathit{verb} \;\rightarrow\; \phi$$

$$\textit{since all and only signs have a SYNSEM attribute}$$

$$\equiv\; \mathit{sign} \;\rightarrow\; (\neg\mathrm{SS}:\mathrm{LC}:\mathrm{CT}:\mathrm{HD}:\mathit{verb} \vee \phi)$$

$$\equiv\; \mathit{sign} \;\rightarrow\; \mathit{sign} \wedge (\neg\mathrm{SS}:\mathrm{LC}:\mathrm{CT}:\mathrm{HD}:\mathit{verb} \vee \phi)$$

$$\equiv\; \mathit{sign} \;\rightarrow\; ((\mathit{sign} \wedge \neg\mathrm{SS}:\mathrm{LC}:\mathrm{CT}:\mathrm{HD}:\mathit{verb}) \vee (\mathit{sign} \wedge \phi))$$

$$\equiv\; \mathit{sign} \;\rightarrow\; ((\mathit{sign} \wedge (\textstyle\bigvee\{t \mid t \in \mathcal{V}, \mathit{approp}(t,\mathrm{SS})\uparrow\} \vee \mathrm{SS}:\neg\mathrm{LC}:\mathrm{CT}:\mathrm{HD}:\mathit{verb})) \vee (\mathit{sign} \wedge \phi))$$

$$\equiv\; \mathit{sign} \;\rightarrow\; ((\mathit{sign} \wedge \mathrm{SS}:(\neg\mathrm{LC}:\mathrm{CT}:\mathrm{HD}:\mathit{verb})) \vee (\mathit{sign} \wedge \phi))$$

$$\textit{since } \mathit{sign} \wedge (\textstyle\bigvee\{t \mid t \in \mathcal{V}, \mathit{approp}(t,\mathrm{SS})\uparrow\}) \textit{ is inconsistent}$$

$$\equiv\; \mathit{sign} \;\rightarrow\; ((\mathit{sign} \wedge \mathrm{SS}:(\mathit{synsem} \wedge \neg\mathrm{LC}:\mathrm{CT}:\mathrm{HD}:\mathit{verb})) \vee (\mathit{sign} \wedge \phi))$$

$$\vdots$$

$$\equiv\; \mathit{sign} \;\rightarrow\; ((\mathit{sign} \wedge \mathrm{SS}:\mathrm{LC}:\mathrm{CT}:\mathrm{HD}:\; (\mathit{head} \wedge \neg\mathit{verb})) \vee (\mathit{sign} \wedge \phi))$$

$$\equiv\; \mathit{sign} \;\rightarrow\; (\mathit{sign} \wedge (\mathrm{SS}:\mathrm{LC}:\mathrm{CT}:\mathrm{HD}:\; (\mathit{noun} \vee \mathit{adj} \vee \mathit{prep} \vee \mathit{func}) \vee \phi))$$

$$\equiv\; \mathit{sign} \;\rightarrow\; (\mathrm{SS}:\mathrm{LC}:\mathrm{CT}:\mathrm{HD}:\; (\mathit{noun} \vee \mathit{adj} \vee \mathit{prep} \vee \mathit{func}) \vee \phi)$$

The idea is clear: when we have to deal with a term $(\neg\phi)$, we look for a most specific type t s.t. $(\neg\phi)$ is equivalent to $t \wedge (\neg\phi)$ *in the given context*. By the context, we mean which feature the term occurs under. I.e., for a given term ϕ, it makes a difference if it occurs as SYNSEM: ϕ or as LOC: ϕ, since those two features have different types appropriate for them.

Then we expand ϕ to a possibly disjunctive term, and check each of the disjuncts for consistency with t. This way, we can eliminate many of the disjuncts generated while expanding $(\neg\phi)$.

It is not so clear if the second approach is more efficient in the sense that it generates less disjunctive information than the first one. It certainly localizes the disjunctive information in a different part of the grammar, namely in the set of constraints instead of in the type hierarchy. Whether this is good or bad or indifferent is a practical question that we will not further persue here.

The second approach has one big advantage over the first one, though: if the transformation is done automatically (as it should), the second approach doesn't change the signature. This means that the grammar writers will not see any types in their solutions that they never defined in the first place.

Since we have shown how to transform arbitrary implicational constraints into ones that have just types as antecedents, we can now restrict our attention to the simpler variety.

Definition 3.8 (grammar)
A grammar is a pair $\langle S, R\rangle$ s.t. $S = \langle \mathcal{T}, \preceq, \mathcal{F}, approp\rangle$ is a signature, R is a finite set of S-terms s.t. $R \subseteq \{t \rightarrow \phi \mid t \in \mathcal{T}$ and ϕ is an S-term$\}$

We now define an inheritance function IC that will further simplify the computational treatment of the grammaticality problem. We will require our grammars to have the following form. For every maximal type $t \in \mathcal{V}$ there is at most one constraint $t \rightarrow \phi$. There are no other constraints.

Definition 3.9 (IC)
$IC : \mathcal{T} \rightarrow \mathsf{Term}$ is a total function from the set of types to the set of terms s.t. $IC(t) = t \wedge \bigwedge_{t \preceq t', t' \rightarrow \phi \in R} \phi$, where the empty conjunction is $\top$. Let $\mathcal{G} = \langle S, R\rangle$ be a grammar s.t. the sets of free variables of the terms in R are pairwise disjoint. Then $\mathcal{G}_{IC} = \langle S, \{t \rightarrow IC(t) \mid t \in \mathcal{V} \wedge IC(t) \neq \top\}\rangle$

Clearly, $\mathcal{G}_{IC}$ is a grammar equivalent to $\mathcal{G}$. We can thus assume that grammars will always be in this format. For normal form grammars, we additionally demand that the consequents of the implicational constraints be in DNF.

Definition 3.10 (normal form grammar)
We say that a grammar $\mathcal{G} = \langle S, R\rangle$ is in normal form iff $\mathcal{G} = \mathcal{G}_{IC}$ and for each $t \rightarrow \phi \in R$, ϕ is in DNF.

3.6 Conclusion

In this chapter, we investigated how we can use our feature logic to define a grammar formalism. We started out be considering the difference between constraint formalisms and rule-based formalisms. We argued that the intuitive difference between the two is as follows. In a rule-based formalism, rules will generate individual structures. Each part of a structure must be generated by *some* rule. In a constraint-based formalism, on the other hand, each part of a structure must pass *every* constraint. We then went on to consider some contemporary grammar formalisms from the linguistics literature. We found that most formalisms mix constraints and rules in some form or other.

We then discussed the notion of validity vs. satisfiabiltity-based approaches to formalizing grammar formalisms (Johnson 1994). This distinction applies particularly to grammar formalisms based on classical logic. We concluded that in that setting, validity-based approaches can be identified with rule-based ones, and satisfiability-based approaches with constraint-based ones.

The grammar formalism we then defined is satisfiability/constraint-based. We chose this approach because it follows naturally from the way grammars are expressed in HPSG. We showed how the prediction problem for such grammars can be translated into a first-order logic satisfiability problem, and gave an undecidability result for prediction.

Finally, we considered how general feature logic grammars can be transformed into a normal form that is more amenable to computation.

Chapter 4

From grammars to logic programs

In this chapter[1], we will consider a first approach to solving the prediction problem for feature logic grammars. Our approach will be based on logic programming, or more generally, constraint logic programming. We will define a translation from HPSG constraint grammars into constraint logic programs that preserves the prediction problem. We will show that there can be no complete translation, yet we will argue that for theoretical as well as practical reasons, it is interesting to see how closely one can approximate HPSG grammars with logic programs. We will thus examine the properties of the translation in detail and come up with a restriction on HPSG grammars that ensures the completeness of the translation.

Implementations of HPSG grammars are generally based on logic programming. Now logic programs are also first-order theories, but the parsing problem is different. Given a logic program P and a definite goal G (which again codes the string we want to parse), we ask if P entails G, i.e., if every model of P satisfies G. These two different ways of encoding the parsing problem have recently been discussed in (Johnson 1994).

It would be desirable to have an automatic translation from HPSG grammars to logic programs that preserves the parsing problem for two obvious reasons: it would ensure the faithfulness of the encoding, and it would still enable us to use the wealth of results that have been produced over the

[1]This chapter is a revised and expanded version of (Götz and Meurers 1995) and (Götz 1995).

last two decades or so for optimizing the execution of logic programs. We could also use existing efficiently implemented logic programming systems without having to custom build a new system for the prediction problem. However, completely abstract considerations tell us that there can be no completely faithful, decidable translation. Since the prediction problem is undecidable, yet can be reduced to a first-order satisfiability problem, we know that we can enumerate all negative instance, i.e., complete algorithms for solving the *non-prediction* problem exist (Boolos and Jeffrey 1974). For logic programming, on the other hand, well-known results tell us that only for the positive problem instance do complete algorithms exist (cf. Lloyd 1984). We can therefore only give a translation that works in many, but not all cases. In this paper, we consider a candidate translation and show for which class of grammars it is a completely faithful coding. Our approach is a direct encoding of grammar constraints as definite clauses. Finally, we will argue that for most linguistically relevant grammars, the translation does indeed give the desired results.

4.1 Adding relation symbols: $\mathcal{R}(\mathcal{FL})$

The target language of our translation will be an instance of the constraint logic programming scheme by Höhfeld and Smolka (1988). What we need first is an appropriate constraint language. In principle, we already have one: the feature constraints developed in ch. 2 would work. However, those would be a bit cumbersome to use for the present purpose. Thus, we will introduce a new constraint language that is closer to the notation used in (Höhfeld and Smolka 1988) and easier to use. However, it should be obvious that this is not much more than a notational variant of what we've developed before.

Definition 4.1 (formulae)

- *$X = \phi$ is a formula if X is a variable and ϕ is a term.*
- *$F_1 \& F_2$ is a formula if F_1 and F_2 are formulae.*

Definition 4.2
Formulae are assigned sets of variable assignments as denotations:

- $[\![X = \phi]\!]^{\mathcal{I}} = \{\alpha \in \mathrm{ASS} \mid \alpha(X) \in [\![\phi]\!]^{\mathcal{I}}_{\alpha}\}$ *if $X \in \mathsf{VAR}$ and ϕ is a term*
- $[\![F_1 \& F_2]\!]^{\mathcal{I}} = [\![F_1]\!]^{\mathcal{I}} \cap [\![F_2]\!]^{\mathcal{I}}$ *if F_1 and F_2 are formulae*

Note that the denotation of an equation is asymmetric, since we always know that the left part is a variable. Yet this is only a more specific instance of the general approach:

$[\![\phi = \psi]\!]^{\mathcal{I}} = \{\alpha \in \text{ASS} \mid \exists u \in \mathcal{U}.u \in [\![\phi]\!]^{\mathcal{I}}_{\alpha} \wedge u \in [\![\psi]\!]^{\mathcal{I}}_{\alpha}\}$ if ϕ and ψ are terms.

(Smolka 1992) has both constraints, calling the more specific ones membership constraints. (Dörre 1994) uses a different, maybe more intuitive notation for membership constraints. He writes $T(X)$ instead of $X = T$. This captures the intuition that the term T denotes a unary predicate. However, if term T is big, this notation is very hard to read. We will therefore stick with our less intuitive, yet hopefully more readable notation.

Our feature logic fulfills all the requirements of (Höhfeld and Smolka 1988) for a constraint language.

- it is decidable,
- closed under intersection,
- closed under renaming and
- compact.

We can therefore simply apply their schema to extend our language with relation symbols and obtain a sound and complete operational semantics for definite clause programs.

Definition 4.3 ($\mathcal{R}(\mathcal{FL})$)

1. *$\mathcal{R}$ is a decidable set of relation symbols*
2. *the variables of $\mathcal{R}(\mathcal{FL})$ are the variables of $\mathcal{FL}$*
3. *the constraints of $\mathcal{R}(\mathcal{FL})$ are defined inductively as follows*
   (a) *every $\mathcal{FL}$ constraint is an $\mathcal{R}(\mathcal{FL})$ constraint*
   (b) *if $r \in \mathcal{R}$ is a relation symbol with arity n and $\vec{x}$ is an n-tuple of pairwise distinct variables, then $r(\vec{x})$ is an $\mathcal{R}(\mathcal{FL})$ constraint called an atom*
   (c) *if F and G are $\mathcal{R}(\mathcal{FL})$ constraints, then so are*
      - *$\emptyset$ (empty conjunction)*
      - *F & G (conjunction)*
      - *$F \rightarrow G$ (implication)*

4. *a definite clause is an $\mathcal{R}(\mathcal{FL})$ constraint $A \leftarrow \phi \ \& \ B_1 \ \& \ \ldots \ \& \ B_n$, where $A, B_1, \ldots, B_n$ are atoms and ϕ is an $\mathcal{FL}$ constraint*

5. *an interpretation $\mathcal{A}$ of $\mathcal{R}(\mathcal{FL})$ is obtained from an $\mathcal{FL}$-interpretation $\mathcal{I}$ with domain $\mathcal{U}$ by choosing for every $r \in \mathcal{R}$ a relation $r^{\mathcal{A}}$ on $\mathcal{U}$ with the same arity, and by defining:*

   (a) *the domain of $\mathcal{A}$ is $\mathcal{U}$*
   (b) $[\![\phi]\!]^{\mathcal{A}} = [\![\phi]\!]^{\mathcal{I}}$ *if ϕ is an $\mathcal{FL}$ formula*
   (c) $[\![r(\vec{x})]\!]^{\mathcal{A}} = \{\alpha \in \mathrm{ASS} \mid \alpha(\vec{x}) \in r^{\mathcal{A}}\}$
   (d) $[\![\emptyset]\!]^{\mathcal{A}} = \mathrm{ASS}, \qquad [\![F \ \& \ G]\!]^{\mathcal{A}} = [\![F]\!]^{\mathcal{A}} \cap [\![G]\!]^{\mathcal{A}}$
   (e) $[\![F \rightarrow G]\!]^{\mathcal{A}} = (\mathrm{ASS} - [\![F]\!]^{\mathcal{A}}) \cup [\![G]\!]^{\mathcal{A}}$

A goal in this setup is a conjunction of atoms and $\mathcal{FL}$ constraints. If $\mathcal{P}$ is a set of definite clauses and G is a goal, then a $\mathcal{FL}$ constraint ϕ is a $\mathcal{P}$-answer of G just in case ϕ is satisfiable and $\phi \rightarrow G$ is valid in every model of $\mathcal{P}$. (Höhfeld and Smolka 1988) prove two propositions showing that the minimal model properties of conventional logic programs extend to their schema for arbitrary constraint languages. We will make use of the $\mathcal{FL}$-instances of these propositions in our soundness proof later on.

Proposition 4.1 (Höhfeld and Smolka 1988, theorem 4.4, p. 12) *Let $\mathcal{P}$ be a set of definite clauses in $\mathcal{R}(\mathcal{FL})$ and let $\mathcal{I}$ be an $\mathcal{FL}$ interpretation. Then the equations*

$$r^{\mathcal{A}_0} := \emptyset, \qquad r^{\mathcal{A}_{i+1}} := \{\alpha(\vec{x}) \mid (r(\vec{x}) \leftarrow G) \in \mathcal{P} \wedge \alpha \in [\![G]\!]^{\mathcal{A}_i}\}$$

define a chain $\mathcal{A}_0 \subseteq \mathcal{A}_1 \subseteq \ldots$ of $\mathcal{R}(\mathcal{FL})$ interpretations extending $\mathcal{I}$. Moreover, the union $\bigcup_{i \geq 0} \mathcal{A}_i$ is the least model of $\mathcal{P}$ extending $\mathcal{I}$.

Proposition 4.2 (Höhfeld and Smolka 1988, proposition 4.5, p. 12) *Let $\mathcal{P}$ be a set of definite clauses in $\mathcal{R}(\mathcal{FL})$, let G be a goal and ϕ an $\mathcal{FL}$ constraint. Then $\phi \rightarrow G$ is valid in every model of $\mathcal{P}$ iff it is valid in every minimal model of $\mathcal{P}$.*

We now have a formal language for HPSG with a notion of grammaticality, and we have $\mathcal{R}(\mathcal{FL})$ with an operational semantics. We just somehow need to connect the two. The basic idea is very simple: for each grammar $\mathcal{G}$ we generate an $\mathcal{R}(\mathcal{FL})$-program $\mathcal{P}$ that defines the unary relation *gram* encoding grammaticality. We can then compute if ϕ is predicted by $\mathcal{G}$ by asking if the goal $gram(X) \ \& \ X = \phi$ has a $\mathcal{P}$-answer.

4.2 Translating to $\mathcal{R}(\mathcal{FL})$

To be able to illustrate the translation, we will work with a continuous example throughout this section. We will use the CFG example from the previous chapter, which we repeat here for convenience. The only thing we've changed is the type appropriate for feature *hd* on *ne-list*. To make the example a little more interesting, we change this type from *atom* to $\top$. The signature is shown in fig. 4.1, and the theory in fig. 4.2.

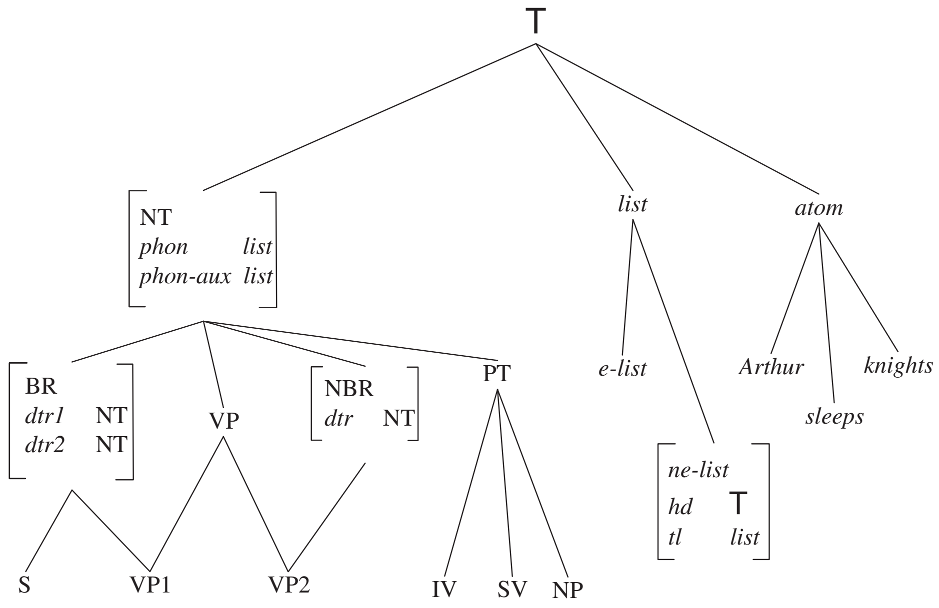


Figure 4.1: An example signature

We begin our definition of the translation by stating what we consider to be a correct translation.

Definition 4.4 (correct translation)
Let $\mathcal{P}$ be a logic program that defines the relation gram and $\mathcal{G}$ a grammar. $\mathcal{P}$ is a correct translation of $\mathcal{G}$ iff

$\mathcal{G}$ *predicts term* ϕ *iff the goal gram*(X) & $X = \phi$ *has a* $\mathcal{P}$-answer.

1. $\mathrm{BR} \rightarrow phon : P0 \wedge phon\text{-}aux : P \wedge$
   $dtr1 : (phon : P0 \wedge phon\text{-}aux : P1) \wedge$
   $dtr2 : (phon : P1 \wedge phon\text{-}aux : P)$

2. $\mathrm{NBR} \rightarrow phon : P0 \wedge phon\text{-}aux : P \wedge$
   $dtr : (phon : P0 \wedge phon\text{-}aux : P)$

3. $\mathrm{PT} \rightarrow phon : [atom \mid T] \wedge phon\text{-}aux : T$

4. $\mathrm{S} \rightarrow dtr1 : \mathrm{NP} \wedge dtr2 : \mathrm{VP}$

5. $\mathrm{VP} \rightarrow (dtr : \mathrm{IV} \ \vee (dtr1 : \mathrm{SV} \wedge dtr2 : \mathrm{S}))$

6. $\mathrm{IV} \rightarrow phon : hd : sleeps$

7. $\mathrm{SV} \rightarrow phon : hd : know$

8. $\mathrm{NP} \rightarrow phon : hd : (knights \vee Arthur)$

Figure 4.2: Theory encoding a CFG

We now partition the set of types $\mathcal{T}$ into three subsets: constrained, hiding and simple types. The motivation for the distinctions is computational and in some sense already an optimization. In short, it obviates the need for some rather complex partial evaluation later on and reduces the number of choice points.

Here's the intuitive meaning of the three sets:

- **constrained types** are exactly those types that unify with some antecedent of the grammar, and whose structure is thus constrained by the conditions stated in the grammar.

- **hiding types** are types that are not constrained themselves but "hide" constrained types somewhere in their structure (loosely speaking).

- **simple types** are neither constrained themselves nor can they hide constrained types anywhere. In other words, the grammar doesn't say anything about them.

Definition 4.5 (type interaction)
We say that two types t and t' interact if they have a common subtype, i.e., $\exists t'' \in \mathcal{T}.\ t'' \preceq t \wedge t'' \preceq t'$.

Definition 4.6 (directly constrained type)
A directly constrained type is a type that serves as antecedent of an implicational constraint in the grammar, i.e., the set of directly constrained types is $\{t \mid t \Rightarrow \phi \in R\}$.

In our example, $\{\mathrm{BR}, \mathrm{NBR}, \mathrm{PT}, \mathrm{S}, \mathrm{VP}, \mathrm{IV}, \mathrm{SV}, \mathrm{NP}\}$ is the set of directly constrained types.

Definition 4.7 (constrained type)
A constrained type is a type that interacts with a directly constrained type. Write C *for the set of constrained types.*

In addition to the directly constrained types, the set of constrained types for our example contains $\{\top, \mathrm{NT}, \mathrm{VP1}, \mathrm{VP2}\}$.

Definition 4.8 (hiding type)
The set of hiding types is the smallest set $H \subseteq \mathcal{T}$ *s.t.*

1. *if* t *is not a constrained type and* $t' \preceq t$*, where* t' *is not a constrained type s.t.* $approp(t', f)$ *is defined and* $approp(t', f)$ *is a constrained type, then* $t \in H$
2. *if* t *is not a constrained type and* $t' \preceq t$*, where* t' *is not a constrained type s.t.* $approp(t', f)$ *is defined and* $approp(t', f) \in H$ *, then* $t \in H$

4.8 is a recursive definition, the base case of which is a type that has a constrained type appropriate for at least one of its features. The only hiding types in our example are *ne-list* and *list*.

Definition 4.9 (simple type)
A simple type is a type that is neither a constrained nor a hiding type.

{*atom*, *Arthur*, *sleeps*, *knights*, *e-list*} is the list of simple types for our example.

Definition 4.10 (hiding feature)
If t *is a constrained or hiding type, then* f *is a hiding feature on* t *iff* $approp(t, f)\downarrow$ *and* $approp(t, f)$ *is a constrained or hiding type.*

The hiding features are the computationally interesting features on a type, as we will see later on. As it turns out, all features in our example are hiding features. If we hadn't changed the type hierarchy from the previous chapter,

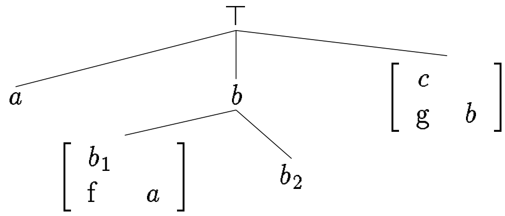


Figure 4.3: An example type hierarchy

then neither would *list* or *ne-list* be hiding types, nor would the feature *hd* and *tl* on *ne-list* be hiding features.

As another example, consider the type hierarchy in fig. 4.3. Suppose a is the only directly constrained type. Then the set of constrained types is $\{a,\top\}$. By clause 1. of def. 4.8, we know that b_1 is a hiding type, since it interacts with itself and has a constrained type (a) appropriate for feature f. By the same clause, b is a hiding type, since it interacts with b_1. $\top$ also interacts with b_1, but it's a constrained type, and thus cannot be a hiding type. Finally, by clause 2. of def. 4.8, we know that c is a hiding type, since it embeds a hiding type under feature g. So the only simple type in this hierarchy is b_2.

We will define three different sets of definite clauses defining the *gram*-relation, one for simple types, one for hiding types and one for constrained types. Notice that we only define clauses for minimal types since under our definition of interpretation, every object is of some minimal type.

Definition 4.11 ($\mathcal{P}_s(\mathcal{G})$)
Let $\mathcal{G}$ be a grammar and S the set of simple types in $\mathcal{G}_{IC}$. Define

$$\mathcal{P}_s(\mathcal{G}) = \{\mathit{gram}(X) \ \leftarrow X = t \mid t \in S \wedge t \in \mathcal{V}\}$$

All the clauses for simple types represent base cases of the *gram* relation. From our definition of simple types, it is intuitively clear that any term of a simple type is admissible (if it's satisfiable). A proof of this fact is given in prop. 4.3. The clauses for the simple types of our continuing example are shown in fig. 4.4 on p. 89.

Definition 4.12 ($\mathcal{P}_h(\mathcal{G})$)
Let $\mathcal{G}$ be a grammar and H the set of hiding types in $\mathcal{G}_{IC}$. For each type $t \in H$, define Clause$(t) := \mathit{gram}(X) \ \leftarrow X = (t \ \wedge \ f_1 \colon Y_1 \ \wedge \ldots \wedge \ f_n \colon$

Y_n) & $gram(Y_1)$ & ... & $gram(Y_n)$, where $f_1 \dots f_n$ are the hiding features of t and $X, Y_1, \dots, Y_n$ are pairwise distinct variables. Then let

$$\mathcal{P}_h(\mathcal{G}) = \{Clause(t) \mid t \in H \wedge t \in \mathcal{V}\}$$

This definition introduces a program clause for each minimal hiding type by saying that an object of a hiding type is grammatical just in case the objects under the type's hiding features are also grammatical. The non-hiding features play no role, since they only map to objects of simple types, which are grammatical in any case. For our continuing "Arthur sleeps" example, the clause for *ne-list* is shown in fig. 4.4.

For constrained types, the definition is exactly parallel, except that additionally we need to add the information from the implicational constraints.

Definition 4.13 ($\mathcal{P}_c(\mathcal{G})$)
Let $\mathcal{G}$ be a grammar and C the set of constrained types in $\mathcal{G}_{IC}$. If $t \in C$ then $Clause(t)$:= $gram(X) \leftarrow X = (t \wedge IC(t) \wedge f_1 : Y_1 \wedge \dots \wedge f_n : Y_n)$ & $gram(Y_1)$ & ... & $gram(Y_n)$, where $f_1 \dots f_n$ are the hiding features of t and $X, Y_1, \dots, Y_n$ are pairwise distinct variables not occuring in $IC(t)$. Then let

$$\mathcal{P}_c(\mathcal{G}) = \{Clause(t) \mid t \in C \wedge t \in \mathcal{V}\}$$

As an example, consider the clause we would generate for the type IV in our example. Note that IV inherits from PT, which means that the constraint on PT needs to be conjoined to the one on IV:

$$\begin{aligned} gram(X) \leftarrow X = (\mathrm{IV} \wedge phon : (Y_1 \wedge [sleeps \mid T]) \;\& \\ phon\text{-}aux : (T \wedge Y_2)) \;\& \\ gram(Y_1) \;\&\; gram(Y_2) \end{aligned}$$

Note that the clause has been somewhat simplified to make it easier to read. All clauses for constrained types in the "Arthur sleeps" example are shown in fig. 4.5 on p. 90.

The complete program is simply the union of the three sets we've just defined.

Definition 4.14 ($\mathcal{P}(\mathcal{G})$)
$\mathcal{P}(\mathcal{G}) = \mathcal{P}_s(\mathcal{G}) \cup \mathcal{P}_h(\mathcal{G}) \cup \mathcal{P}_c(\mathcal{G})$

We can now give the complete grammar as a logic program. The individual clauses have been simplified for better readability. The clauses for the simple and hiding types are shown in fig. 4.4, and the ones for constrained types are shown in fig. 4.5.

$$
\begin{array}{lcl}
gram(X) & \leftarrow & X = e\text{-}list \\
gram(X) & \leftarrow & X = Arthur \\
gram(X) & \leftarrow & X = sleeps \\
gram(X) & \leftarrow & X = knights \\
gram(X) & \leftarrow & X = (ne\text{-}list \wedge hd : Y_1 \wedge tl : Y_2) \ \&\ gram(Y_1) \ \&\ gram(Y_2)
\end{array}
$$

Figure 4.4: Clauses for simple and hiding types

Note how the translation is in general insensitive to the form of the input constraints. The relevant information is taken from the type hierarchy, plus some extra information about types taken from the grammar. This means that the specific properties of the constraint language (e.g., whether it allows for negation or not) are irrelevant for the compilation process. The important thing is that the constraint language provided by the target logic programming language must be at least as powerful as the input constraint language.

4.3 Soundness

Having defined the translation, we can now investigate it's correctness. We start by proving a strong soundness result showing that the minimal model construction for the definite clause programs can be used to build models for the corresponding grammar.

Definition 4.15
Let $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ be an interpretation.

If $\mathcal{U}' \subseteq U$, then $\mathcal{I}_{|\mathcal{U}'} = \langle \mathcal{U}', \mathcal{S}_{|\mathcal{U}'}, \mathcal{A}_{|\mathcal{U}'}, [\![\cdot]\!]^{\mathcal{I}_{|\mathcal{U}'}} \rangle$ where for each $f \in \mathcal{F}$, for each $u \in \mathcal{U}'$, $\mathcal{A}_{|\mathcal{U}'}(f)(u)\downarrow$ iff $\mathcal{A}(f)(u)\downarrow$ and $\mathcal{A}(f)(u) \in \mathcal{U}'$, and if $\mathcal{A}_{|\mathcal{U}'}(f)(u)\downarrow$, then $\mathcal{A}_{|\mathcal{U}'}(f)(u) = \mathcal{A}(f)(u)$.

$$
\begin{aligned}
\mathit{gram}(X) \ \leftarrow\ & X = (\mathrm{S}\ \wedge \mathit{phon} : P_0 \wedge \mathit{phon\text{-}aux} : P \wedge \\
& \qquad \mathit{dtr1} : (Y_1 \wedge \mathrm{NP} \wedge \mathit{phon} : P_0 \wedge \mathit{phon\text{-}aux} : P_1) \wedge \\
& \qquad \mathit{dtr2} : (Y_2 \wedge \mathrm{VP} \wedge \mathit{phon} : P_1 \wedge \mathit{phon\text{-}aux} : P)) \ \& \\
& \mathit{gram}(P_0)\ \&\ \mathit{gram}(P_1)\ \&\ \mathit{gram}(P)\ \&\ \mathit{gram}(Y_1)\ \&\ \mathit{gram}(Y_2) \\
\mathit{gram}(X) \ \leftarrow\ & X = (\mathrm{VP1}\ \wedge \mathit{phon} : P_0 \wedge \mathit{phon\text{-}aux} : P \wedge \\
& \qquad \mathit{dtr1} : (Y_1 \wedge \mathrm{SV} \wedge \mathit{phon} : P_0 \wedge \mathit{phon\text{-}aux} : P_1) \wedge \\
& \qquad \mathit{dtr2} : (Y_2 \wedge \mathrm{S} \wedge \mathit{phon} : P_1 \wedge \mathit{phon\text{-}aux} : P)) \ \& \\
& \mathit{gram}(P_0)\ \&\ \mathit{gram}(P_1)\ \&\ \mathit{gram}(P)\ \&\ \mathit{gram}(Y_1)\ \&\ \mathit{gram}(Y_2) \\
\mathit{gram}(X) \ \leftarrow\ & X = (\mathrm{VP2}\ \wedge \mathit{phon} : P_0 \wedge \mathit{phon\text{-}aux} : P \wedge \\
& \qquad \mathit{dtr} : (Y \wedge \mathrm{IV} \wedge \mathit{phon} : P_0 \wedge \mathit{phon\text{-}aux} : P)) \ \& \\
& \mathit{gram}(P_0)\ \&\ \mathit{gram}(P)\ \&\ \mathit{gram}(Y) \\
\mathit{gram}(X) \ \leftarrow\ & X = (\mathrm{IV} \wedge \mathit{phon} : (Y_1 \wedge [\mathit{sleeps} \mid T])\ \& \\
& \qquad \mathit{phon\text{-}aux} : (T \wedge Y_2))\ \& \\
& \mathit{gram}(Y_1)\ \&\ \mathit{gram}(Y_2) \\
\mathit{gram}(X) \ \leftarrow\ & X = (\mathrm{SV} \wedge \mathit{phon} : (Y_1 \wedge [\mathit{knows} \mid T])\ \& \\
& \qquad \mathit{phon\text{-}aux} : (T \wedge Y_2))\ \& \\
& \mathit{gram}(Y_1)\ \&\ \mathit{gram}(Y_2) \\
\mathit{gram}(X) \ \leftarrow\ & X = (\mathrm{NP} \wedge \mathit{phon} : (Y_1 \wedge [\mathit{Arthur} \mid T])\ \& \\
& \qquad \mathit{phon\text{-}aux} : (T \wedge Y_2))\ \& \\
& \mathit{gram}(Y_1)\ \&\ \mathit{gram}(Y_2)
\end{aligned}
$$

Figure 4.5: Clauses for constrained types

This definition takes an interpretation and "makes it smaller". Note that $\mathcal{I}_{|\mathcal{U}'}$ is not always an interpretation.

Proposition 4.3
Let $\mathcal{G} = \langle \Sigma, R \rangle$ be a grammar, $S_\mathcal{G}$ be the set of simple types in $\mathcal{G}$, $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ be an interpretation and $\mathcal{W} \subseteq \mathcal{U}$ s.t. $\mathcal{W} = \{u \in \mathcal{U} \mid \mathcal{V}(u) \in S_\mathcal{G}\}$. Then $\mathcal{J} = \langle \mathcal{W}, \mathcal{S}_{|\mathcal{W}}, \mathcal{A}_{|\mathcal{W}}, [\![\cdot]\!]^\mathcal{J} \rangle$ is a model of $\mathcal{G}$.

Proof

To show that $\mathcal{J}$ is a model of $\mathcal{G}$, we first need to show that $\mathcal{J}$ is an interpretation. For this we need to show in particular that $\mathcal{A}_{|\mathcal{W}}$ meets definition 2.3:

for each $u \in \mathcal{U}$, for each $f \in \mathcal{F}$, if $approp(\mathcal{S}(u), f)\downarrow$ and $approp(\mathcal{S}(u), f) = t$, then $(\mathcal{A}(f))(u)$ is defined and $\mathcal{S}((\mathcal{A}(f))(u)) \preceq t$

$\Rightarrow$ f.a. $u \in \mathcal{W}$ and $f \in \mathcal{F}$, if $approp(\mathcal{S}_{|\mathcal{W}}(u), f)\downarrow$ and $approp(\mathcal{S}_{|\mathcal{W}}(u), f) = t$, then $(\mathcal{A}_{|\mathcal{W}}(f))(u)$ is defined and $\mathcal{S}_{|\mathcal{W}}((\mathcal{A}_{|\mathcal{W}}(f))(u)) \preceq t$, since t must be a simple type

and

for each $u \in \mathcal{U}$, for each $f \in \mathcal{F}$, if $(\mathcal{A}(f))(u)$ is defined, then $approp(\mathcal{S}(u), f)$ is defined and $\mathcal{S}((\mathcal{A}(f))(u)) \preceq approp(\mathcal{S}(u), f)$

$\Rightarrow$ for each $u \in \mathcal{W}$, for each $f \in \mathcal{F}$, if $(\mathcal{A}_{|\mathcal{W}}(f))(u)\downarrow$, then $approp(\mathcal{S}_{|\mathcal{W}}(u), f)$ is defined and $\mathcal{S}_{|\mathcal{W}}((\mathcal{A}_{|\mathcal{W}}(f))(u)) \preceq approp(\mathcal{S}_{|\mathcal{W}}(u), f)$

Thus, $\mathcal{J}$ is an interpretation.

Furthermore,

for each $u \in \mathcal{W}$, $\mathcal{S}_{|\mathcal{W}}(u)$ is a simple type

$\Rightarrow$ for each $u \in \mathcal{W}$, for each $(t \Rightarrow T) \in R$, for each $\alpha \in$ ASS, $u \in [\![t \Rightarrow T]\!]^{\mathcal{J}}_{\alpha}$ (since $[\![t]\!]^{\mathcal{J}}_{\alpha} \cap [\![\mathcal{S}_{|\mathcal{W}}(u)]\!]^{\mathcal{J}}_{\alpha} = \emptyset$)

$\Rightarrow$ for each $u \in \mathcal{W}$, for each $(t \Rightarrow T) \in R$, for some $\alpha \in$ ASS, $u \in [\![t \Rightarrow T]\!]^{\mathcal{J}}_{\alpha}$

$\Rightarrow$ $\mathcal{J}$ is a model of $\mathcal{G}$

■

This result says that as far as simple types are concerned, satisfiability and prediction are equivalent. In other words, if there's a term that we know to be of a simple type with respect to some grammar, then that grammar predicts that term iff the term is at all satisfiable.

The next proposition makes a related claim about hiding types. Intuitively, it says that whether an object of a hiding type is part of a model of a certain grammar only depends on whether the objects it's mapped to are part of a model. Note that this is not generally true of all objects. For constrained types, this property also depends on what the mapped to objects "look like". This will be made explicit in proposition 4.5.

Proposition 4.4
Let $\mathcal{G} = \langle \Sigma, R \rangle$ be a grammar, $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ be an interpretation and $\mathcal{W} \subseteq \mathcal{U}$ s.t. $\mathcal{I}_{|\mathcal{W}}$ is a model of $\mathcal{G}$. If t is a minimal hiding type and $u \in \mathcal{U}$ s.t. $\mathcal{S}(u) = t$ and for each feature f, if $\mathcal{A}(f)(u)\downarrow$ then $\mathcal{A}(f)(u) \in \mathcal{W}$, then $\mathcal{I}_{|\mathcal{W} \cup \{u\}}$ is a model of $\mathcal{G}$.

Proof

Clearly, $\mathcal{I}_{|\mathcal{W} \cup \{u\}}$ is an interpretation. Furthermore,

t is a hiding type

$\Rightarrow$ for each $t' \Rightarrow T \in R$, for each $\alpha \in$ ASS, $[\![t]\!]_{\alpha}^{\mathcal{I}_{|\mathcal{W}\cup\{u\}}} \cap [\![t']\!]_{\alpha}^{\mathcal{I}_{|\mathcal{W}\cup\{u\}}} = \emptyset$

$\Rightarrow$ for each $t' \Rightarrow T \in R$, for each $\alpha \in$ ASS, $u \in [\![t' \Rightarrow T]\!]_{\alpha}^{\mathcal{I}_{|\mathcal{W}\cup\{u\}}}$

$\Rightarrow$ $\mathcal{I}_{|\mathcal{W} \cup \{u\}}$ is a model of $\mathcal{G}$

$\blacksquare$

Proposition 4.5
Let $\mathcal{G} = \langle \Sigma, R \rangle$ be a grammar, $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ be an interpretation and $\mathcal{W} \subseteq \mathcal{U}$ s.t. $\mathcal{I}_{|\mathcal{W}}$ is a model of $\mathcal{G}$. If t is a minimal constrained type and $u \in \mathcal{U}$ s.t. $\mathcal{S}(u) = t$, for each feature f, if $\mathcal{A}(f)(u)\downarrow$ then $\mathcal{A}(f)(u) \in \mathcal{W}$ and if $t \Rightarrow T \in IC(R)$ then for some $\alpha \in$ ASS, $u \in [\![T]\!]_{\alpha}^{\mathcal{I}_{|\mathcal{W}\cup\{u\}}}$, then $\mathcal{I}_{|\mathcal{W} \cup \{u\}}$ is a model of $\mathcal{G}$.

Proof

Let $\mathcal{J} = \mathcal{I}_{|\mathcal{W} \cup \{u\}}$. Clearly, $\mathcal{J}$ is an interpretation. Furthermore,

t is a minimal constrained type and if $t \Rightarrow T \in IC(R)$, then for some α, $u \in [\![T]\!]^{\mathcal{J}}$

$\Rightarrow$ for each t', for some α, if $t' \Rightarrow T' \in IC(R)$ then $u \in [\![t' \Rightarrow T']\!]^{\mathcal{J}}$

$\Rightarrow$ $\mathcal{J}$ is a model of $\mathcal{G}$

$\blacksquare$

Proposition 4.6
Let $\mathcal{G}$ be a grammar, $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$, $X \subseteq \wp(\mathcal{U})$ s.t. for each $\mathcal{U}' \in X$, $\mathcal{I}_{|\mathcal{U}'}$ is a model of $\mathcal{G}$, and let $\mathcal{W} = \bigcup X$. Then $\mathcal{I}_{|\mathcal{W}}$ is a model of $\mathcal{G}$.

Proposition 4.7
If $\mathcal{G}$ is a grammar, $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ a $\mathcal{FL}$-interpretation and $\mathcal{M}$ the minimal model of $\mathcal{P}(\mathcal{G})$ extending $\mathcal{I}$, then $\mathcal{I}_{|\{u \in \mathcal{U} | \langle u \rangle \in [\![gram]\!]^{\mathcal{M}}\}}$ is a model of $\mathcal{G}$.

Proof

Firstly, since for each simple type t, $(gram(X) \leftarrow X = t) \in \mathcal{P}(\mathcal{G})$, we know that for each $i \geq 1$, $\mathcal{U}_s = \{u \in \mathcal{U} \mid \mathcal{S}(u) \text{ is a simple type}\} \subseteq \{u \in \mathcal{U} \mid \langle u \rangle \in [\![gram]\!]^{\mathcal{M}_i}\}$ ... (1)

We can now do an induction on the construction of $\mathcal{M}$.

Clearly, $\mathcal{I}_{|\{u \in \mathcal{U} | \langle u \rangle \in [\![gram]\!]^{\mathcal{M}_0}\}} = \mathcal{I}_{|\emptyset}$ is a model of $\mathcal{G}$.

Now suppose $i \geq 1$. Let $\mathcal{W} = \{u \in \mathcal{U} \mid \langle u \rangle \in [\![gram]\!]^{\mathcal{M}_{i-1}}\}$ and $\mathcal{W}' = \{u \in \mathcal{U} \mid \langle u \rangle \in [\![gram]\!]^{\mathcal{M}_i}\}$.

For each $u \in \mathcal{U}$,

$u \in \mathcal{W}'$

$\Rightarrow$ $\mathcal{S}(u)$ is a simple type

$\Rightarrow$ $u \in \mathcal{U}_s$, $\mathcal{U}_s \subseteq \mathcal{W}'$ and $\mathcal{I}_{|\mathcal{U}_s}$ is a model of $\mathcal{G}$ by (1) and prop. 4.3

$\mathcal{S}(u)$ is a hiding type

$\Rightarrow$ There is a $(gram(X) \leftarrow X = (\mathcal{S}(u) \wedge f_1 : Y_1 \wedge \ldots) \;\&\; gram(Y_1) \;\&\; \ldots) \in \mathcal{P}(\mathcal{G})$. $\exists \alpha \in$ ASS. $\alpha \in [\![X = (\mathcal{S}(u) \wedge f_1 : Y_1 \wedge \ldots) \;\&\; gram(Y_1) \;\&\; \ldots]\!]^{\mathcal{M}_{i-1}}$ and $\alpha(X) = u$

$\Rightarrow$ $\mathcal{I}_{|\{u\} \cup \mathcal{W} \cup \mathcal{U}_s}$ is a model of $\mathcal{G}$ by (1) and propositions 4.3, 4.4 and 4.6

$\mathcal{S}(u)$ is a constrained type

$\Rightarrow$ There is a $(gram(X) \leftarrow X = (\mathcal{S}(u) \wedge IC(\mathcal{S}(u)) \wedge f_1 : Y_1 \wedge \ldots) \;\&\; gram(Y_1) \;\&\; \ldots) \in \mathcal{P}(\mathcal{G})$. $\exists \alpha \in$ ASS. $\alpha \in$ $[\![X = (\mathcal{S}(u) \wedge IC(\mathcal{S}(u)) \wedge f_1 : Y_1 \wedge \ldots) \;\&\; gram(Y_1) \;\&\; \ldots]\!]^{\mathcal{M}_{i-1}}$ and $\alpha(X) = u$

$\Rightarrow$ $\mathcal{I}_{|\{u\} \cup \mathcal{W} \cup \mathcal{U}_s}$ is a model of $\mathcal{G}$ by (1) and propositions 4.3, 4.5 and 4.6

$\Rightarrow$ $\mathcal{I}_{|\mathcal{W}'}$ is a model of $\mathcal{G}$

Thus, by induction and prop. 4.6, $\mathcal{I}_{|\{u \in \mathcal{U} | \langle u \rangle \in [\![gram]\!]^{\mathcal{M}}\}}$ is a model of $\mathcal{G}$.

■

Proposition 4.8 (soundness)
Let $\mathcal{G}$ be a grammar and ϕ a term. If $gram(X)$ & $X = \phi$ has a $\mathcal{P}(\mathcal{G})$-answer then $\mathcal{G}$ predicts ϕ.

Proof *By props. 4.1, 4.2 and 4.7.* ∎

This completes our soundness result: every answer computed by a program we automatically compile from a grammar is a correct answer. We will now investigate the opposite direction, i.e., if some term is predicted by the grammar, does the program also provide an answer for it?

4.4 Completeness

Our method can't be complete, for reasons outlined in the introduction to this chapter. For some queries in some grammars there can be no finite proofs (or refutations). E.g., adding the (reasonable) constraint

$$person \Rightarrow \text{father: } (person \wedge \text{gender: } male)$$

to our example, this would be translated to

$$gram(X) \leftarrow X = (person \wedge \text{father} : (Y \wedge person \wedge gender : male) \;\&\; gram(Y)$$

which goes into an infinite loop on the query

$$gram(X) \;\&\; X = person$$

This particular example could be handled using lazy evaluation as proposed in (Aït-Kaci et al. 1993), but there are really pathological examples. What we would expect is that we can prove completeness for some subset of the decidable grammars. And indeed, we will show in this section that the method is complete for grammars that exhibit a form of the finite model property.

Definition 4.16 ($Path_{\mathcal{I}}$)
If $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ and $u \in \mathcal{U}$, then $Path_{\mathcal{I}}(u) = \{\pi \mid \mathcal{A}^*(\pi)(u)\downarrow\}$[2]

[2] Where $\mathcal{A}^*$ is the reflexive transitive closure of $\mathcal{A}$

Definition 4.17 (semi-finite interpretation)
An interpretation $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ *is called semi-finite iff for each* $u \in \mathcal{U}$, $Path_{\mathcal{I}}(u)$ *is finite.*

Proposition 4.9
Let $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ *be an interpretation,* $\mathcal{G}$ *a grammar,* $\mathcal{P}(\mathcal{G})$ *the program associated with* $\mathcal{G}$ *and* $\mathcal{M}$ *the minimal model of* $\mathcal{P}(\mathcal{G})$ *extending* $\mathcal{I}$.

If $\mathcal{W} \subseteq \mathcal{U}$ *s.t.* $\mathcal{I}_{|\mathcal{W}}$ *is a semi-finite model of* $\mathcal{G}$ *and* $u \in \mathcal{W}$, *then* $u \in \{v \mid \langle v \rangle \in [\![gram]\!]^{\mathcal{M}}\}$.

Proof

The proof runs by induction on $|Path_{\mathcal{I}}(u)|$.

$|Path_{\mathcal{I}}(u)| = 1$

$\Rightarrow$ $Path_{\mathcal{I}}(u) = \{\varepsilon\}$ and

1) $\mathcal{S}(u)$ is a simple type
$\Rightarrow$ $gram(X) \leftarrow X = \mathcal{S}(u) \in \mathcal{P}(\mathcal{G})$
$\Rightarrow$ $u \in \{v \mid \langle v \rangle \in [\![gram]\!]^{\mathcal{M}}\}$

2) $\mathcal{S}(u)$ is not a hiding type, since hiding types have at least one feature defined on them

3) $\mathcal{S}(u)$ is a constrained type
$\Rightarrow$ $gram(X) \leftarrow X = (\mathcal{S}(u)\ \&\ IC(\mathcal{S}(u))) \in \mathcal{P}(\mathcal{G})$
$\Rightarrow$ $u \in \{v \mid \langle v \rangle \in [\![gram]\!]^{\mathcal{M}}\}$

Furthermore,

$i \geq 1$, for each $v \in \mathcal{W}$ s.t. $|Path_{\mathcal{I}}(v)| \leq i$, $\langle v \rangle \in [\![gram]\!]^{\mathcal{M}}$ and $|Path_{\mathcal{I}}(u)| = i + 1$

$\Rightarrow$

1) $\mathcal{S}(u)$ is a simple type
$\Rightarrow$ $gram(X) \leftarrow X = \mathcal{S}(u) \in \mathcal{P}(\mathcal{G})$
$\Rightarrow$ $u \in \{v \mid \langle v \rangle \in [\![gram]\!]^{\mathcal{M}}\}$

2) $\mathcal{S}(u)$ is a hiding type

$\Rightarrow$ $gram(X) \leftarrow X = (\mathcal{S}(u)\ \&\ f_1 : Y_1\ \&\ldots)\ \&\ gram(Y_1)\ldots \in \mathcal{P}(\mathcal{G})$

$\Rightarrow$ $\exists\alpha \in$ ASS. $\alpha \in [\![X = (\mathcal{S}(u)\ \&\ f_1 : Y_1\ \&\ldots)\ \&\ gram(Y_1)\ldots]\!]^{\mathcal{M}}$, $\alpha(X) = u$ and for each j, $\alpha(Y_j) = \mathcal{A}(f_j)(u)$, by the inductive hypothesis

$\Rightarrow$ $\exists\alpha.\ \alpha(X) = u$ and $\langle\alpha(X)\rangle \in [\![gram]\!]^{\mathcal{M}}$

$\Rightarrow$ $u \in \{v \mid \langle v\rangle \in [\![gram]\!]^{\mathcal{M}}\}$

3) $\mathcal{S}(u)$ is a constrained type

$\Rightarrow$ $gram(X) \leftarrow X = (\mathcal{S}(u)\ \&\ IC(\mathcal{S}(u))\ \&\ f_1 : Y_1\ \&\ldots)\ \&\ gram(Y_1)\ldots \in \mathcal{P}(\mathcal{G})$

$\Rightarrow$ $\exists\alpha.\ \alpha \in [\![X = (\mathcal{S}(u)\ \&\ IC(\mathcal{S}(u))\ \&\ f_1 : Y_1\ \&\ldots)\ \&\ gram(Y_1)\ldots]\!]^{\mathcal{M}}$, $\alpha(X) = u$ and for each j, $\alpha(Y_j) = \mathcal{A}(f_j)(u)$, by the inductive hypothesis

$\Rightarrow$ $\exists\alpha.\ \alpha(X) = u$ and $\langle\alpha(X)\rangle \in [\![gram]\!]^{\mathcal{M}}$

$\Rightarrow$ $u \in \{v \mid \langle v\rangle \in [\![gram]\!]^{\mathcal{M}}\}$

■

We thus conclude that our translation is complete for grammars $\mathcal{G}$ with the following property: if $\mathcal{G}$ predicts ϕ, then ϕ is true in some semi-finite model of $\mathcal{G}$. One might argue that reasonable linguistic grammars should possess this property, anyway. However, two points are worth mentioning here. Firstly, in semi-finite models, we not only exclude "infinite" objects, but also cyclic ones. We will be able to lift this restriction with the methods developed in the next chapter. Secondly, even for a given grammar, it will be extremely hard to prove that it has the finite model property. It would be extremely useful to have a simple, decidable characterization of an interesting sub-class of the decidable grammars. However, such a characterization is not known at this point.

4.5 Conclusion

In this chapter, we have presented a compilation scheme that translates feature logic constraint grammars into efficient logic programs by using a classification of types. The compilation is sound and finitely complete.

We know from previous discussions that we can't achieve a complete translation. Any translation that we come up with will be incomplete for undecidable subclasses of feature constraint grammars. We showed that the translation is complete for grammars that have the finite model property. Given that the translation is relatively straightforward, this is an interesting result. We could probably catch an even larger class of grammars with a more involved translation, but that is likely to be only of theoretical interest.

The question whether this translation is really practical is difficult to answer. It is relatively clear that a logic program resulting from the translation would have severe termination problems. It is a research area in and of itself to find strategies to automatically deal with such termination problems. The interested reader is referred to (Minnen 1998) and references cited therein.

Chapter 5

Lazy evaluation

In this section, we will look at an evaluation method for type constraint grammars that has been discussed under the heading of *lazy type evaluation*. But whereas this was normally used as an on-line optimization strategy, we will show that given suitable grammars, lazy evaluation can be compiled into an internal representation we generate from the grammar. We will base this work on ideas presented in (Aït-Kaci et al. 1993). This chapter builds on work described in (Götz and Meurers 1999).

The basic idea of lazy type evaluation is that nodes with *more information content* should be preferred in evaluation over nodes with less information content. In practice, this takes the form that the evaluation of nodes with no features defined on them is delayed. However, we would like to compute this delaying information off-line instead of on-line. As it turns out, this can be done quite easily in practice. Theoretically, on the other hand, lazy evaluation changes our perspective on program semantics. Whereas our previous programs had the property of *persistence* (any term subsumed by a solution was also a solution), we give up this property to be able to compute more efficiently and simply demand that if T is a solution and there are more specific terms, then *some* of these more specific terms must also be solutions. Recall that this is just our definition of grammaticality: a term T is grammatical with respect to a grammar just in case the grammar has a model that satisfies T. To be able to do this, we impose a well-formedness condition on our grammars. This idea is due to (Aït-Kaci et al. 1993), who imposed a strong syntactic restriction on their grammars. Given our model-theoretic approach, however, we will also use a (weaker) model-theoretic restriction.

Definition 5.1
Let $\Sigma = \langle \mathcal{T}, \preceq, \mathcal{F}, approp\rangle$ and $\mathcal{G}$ a grammar. $\mathcal{G}$ is called type consistent iff $\mathcal{G}$ has a model $\mathcal{I}$ s.t. for each $t \in \mathcal{V}$, $[\![t]\!]^{\mathcal{I}} \neq \emptyset$.

From a grammar writing perspective, this property is weaker than any syntactic property that would allow us to do the same thing, and thus more desirable. On the other hand, it is very hard to check in the general case. In fact, we have the following undesirable lemma.

Proposition 5.1
Given a grammar $\mathcal{G}$, it is undecidable if $\mathcal{G}$ is type consistent.

Proof
We show this by reducing the grammaticality problem to a type consistency problem. Let $\langle \mathcal{T}, \preceq, \mathcal{F}, approp\rangle$ be a signature, $\mathcal{V}$ the set of minimal types, $\mathcal{G}'$ a grammar and T a term. Let $\top$ be the top element of $\langle \mathcal{T}, \preceq\rangle$. Introduce a new minimal type $v \notin \mathcal{V}$ and a new top element $\top_{\mathbf{new}}$ s.t. $\mathcal{T}' = \mathcal{T} \cup \{v, \top_{\mathbf{new}}\}$ and $\langle \mathcal{T}', \preceq\rangle$ is exactly like $\langle \mathcal{T}, \preceq\rangle$ with the additional requirement that $\top \preceq \top_{\mathbf{new}}$ and $v \preceq \top_{\mathbf{new}}$. Also introduce a new feature $f \notin \mathcal{F}$ and modify *approp* so that

$$approp'(v', f') = \begin{cases} approp(v', f') & \text{if } v' \neq v \text{ and } f' \neq f \\ \top & \text{if } v' = v \text{ and } f' = f \end{cases}$$

Now let $\mathcal{G}' = \mathcal{G} \cup \{v \rightarrow f : T\}$. Clearly, $\mathcal{G}$ admits T iff $\mathcal{G}'$ is type consistent. Since this is undecidable, the type consistency problem is in general undecidable.

■

Like all undecidability results, this one also has to be taken with a grain of salt. It is to be expected that for many – if not most – actual grammars, a proof of type consistency (or inconsistency) can be automatically derived. We could do this by assuming that a given grammar *is* type consistent, and then running the derived program on every maximally specific type. If we get **no** as an answer, we know that the grammar is not type consistent, if we get **yes**, it is. By the above lemma, however, this test will not terminate in the general case.

5.1 Lazy resolution

Before we can define how to derive lazy programs from grammars, we need some auxiliary definitions. We begin by considering a semantic equivalence relation on variables in feature clauses.

Definition 5.2
Let Σ be a consistent set of constraints. Two variables $X, Y \in \mathrm{FV}(\Sigma)$ are equivalent in Σ ($X \equiv_\Sigma Y$) iff

$$\forall\, \mathcal{I}, \alpha.\ (\mathcal{I}, \alpha \models \Sigma \ \Rightarrow\ \alpha(X) = \alpha(Y))$$

Define $[X]_\Sigma = \{Y \mid X \equiv_\Sigma Y\}$.

Consider the feature clause $\Sigma = \{X|\,Y,\ Y|\,Z\}$. Since for any $\mathcal{I}, \alpha$ that satisfy Σ we have that $\alpha(X) = \alpha(Z)$, it follows that $X \equiv_\Sigma Z$. As another example, consider $\Sigma = \{X|\,f{:}Y,\ X|\,f{:}Z\}$. Since features are interpreted as functions, we know that for any $\mathcal{I}, \alpha$ that satisfy Σ, $\alpha(Y) = \alpha(Z)$ and thus, $Y \equiv_\Sigma Z$. Clearly, the normal form for feature clauses defined in definition 2.24 is designed exactly to make this equivalence relation explicit. We thus get the following lemma.

Proposition 5.2
Let Σ be a set of constraints. If Σ is in normal form, then for each $X, Y \in$ $\mathrm{FV}(\Sigma)$, $X \equiv_\Sigma Y$ iff $X|\,Y \in \Sigma$ or $Y|\,X \in \Sigma$.

We will sometimes be sloppy and talk about a variable X when we really mean $[X]_\Sigma$. However, this should always be clear from the context. When Σ is clear from the context, we will sometimes write $[X]$ instead of $[X]_\Sigma$. We now need one more definition before we can turn to lazy programs. The whole notion of a lazy derivation crucially builds upon the concepts of *branching* and *non-branching* variables. A branching variable is one for which a feature selection constraint is defined. A variable is called non-branching (or terminal) iff it is not branching. All variables in one equivalence class are either branching or non-branching. Consider the feature clause $\Sigma = \{X|\,f{:}Y,\ Y|\,a,\ X|\,Z\}$. Both X and Z are branching variables, and Y is non-branching.

Definition 5.3 (branching variables)
Let Σ be a feature clause and $X \in \mathrm{FV}(\Sigma)$. X is branching in Σ iff

$\exists f.\ \exists Y, Z.\ X \equiv_\Sigma Y$ and $Y | f{:}Z \in \Sigma$. Write $\mathrm{BV}(\Sigma)$ for the set of branching variables in Σ.

We now turn to the definition of a lazy program derived from a grammar $\mathcal{G}$. A lazy program is a set of triples $\langle \Sigma, X, S\rangle$, where Σ is a feature clause in normal form, X is a distinguished variable in Σ (the root variable) and S is a set of variable equivalence classes in Σ. The set S is comparable to the body of a clause in logic programming. It contains the information which variables need to be "checked" during a derivation, where we still need to apply constraints. This should become clear form an example later on.

Definition 5.4 (lazy program)
For each $t \rightarrow \phi \in \mathcal{G}_{IC}$, assume w.l.o.g. that ϕ is in disjunctive normal form. For each $t \in \mathcal{V}$ s.t. $t \rightarrow \phi \in \mathcal{G}_{IC}$ and $\Sigma \in \mathrm{NF}(\mathbf{trans}(X, \phi))$, $\langle \Sigma, X, \{[Y]_\Sigma \mid Y \in \mathrm{BV}(\Sigma),\ Y \not\equiv_\Sigma X,\ Y | a \in \Sigma, a \text{ constrained }\}\rangle \in \mathcal{P}(\mathcal{G})$. Nothing else is in $\mathcal{P}(\mathcal{G})$.

The definition of a lazy program is a lot simpler than the one for non-lazy programs of the last section. For example, the hiding types are completely gone. Also, all and only the maximally specific constrained types have clauses in the program.

We will use the example from the last chapter (see fig. 4.1 on p. 84 for the signature and fig. 4.2 on p. 85 for the theory) to illustrate lazy compilation. First, we need to translate the feature terms in the consequents of the implicational constraints into feature clauses. The clauses are shown in figures 5.1, 5.2, 5.3 and 5.4.

$$\Sigma_1 = \left\{ \begin{array}{lll} X|\,\mathrm{S}, & & \\ X|\,\mathit{phon}{:}\,P_0, & P_0|\,\mathit{list} & \\ X|\,\mathit{phon\text{-}aux}{:}\,P, & P|\,\mathit{list} & \\ X|\,\mathit{dtr1}{:}\,Y_1, & Y_1|\,\mathrm{NP} & \\ & Y_1|\,\mathit{phon}{:}\,P_0, & \\ & Y_1|\,\mathit{phon\text{-}aux}{:}\,P_1, & P_1|\,\mathit{list} \\ X|\,\mathit{dtr2}{:}\,Y_2, & Y_2|\,\mathrm{VP} & \\ & Y_2|\,\mathit{phon}{:}\,P_1, & \\ & Y_2|\,\mathit{phon\text{-}aux}{:}\,P & \end{array} \right\}$$

Figure 5.1: Clause for type S

$$\Sigma_2 = \left\{ \begin{array}{lll} X \mid \mathrm{VP1}, & & \\ X \mid phon\colon P_0, & P_0 \mid list & \\ X \mid phon\text{-}aux\colon P, & P \mid list & \\ X \mid dtr1\colon Y_1, & Y_1 \mid \mathrm{SV} & \\ & Y_1 \mid phon\colon P_0, & \\ & Y_1 \mid phon\text{-}aux\colon P_1, & P_1 \mid list \\ X \mid dtr2\colon Y_2, & Y_2 \mid \mathrm{S} & \\ & Y_2 \mid phon\colon P_1, & \\ & Y_2 \mid phon\text{-}aux\colon P & \end{array} \right\}$$

Figure 5.2: Clause for type VP1

$$\Sigma_3 = \left\{ \begin{array}{ll} X \mid \mathrm{VP2}, & \\ X \mid phon\colon P_0, & P_0 \mid list \\ X \mid phon\text{-}aux\colon P, & P \mid list \\ X \mid dtr\colon Y, & Y \mid \mathrm{IV} \\ & Y \mid phon\colon P_0, \\ & Y \mid phon\text{-}aux\colon P \end{array} \right\}$$

Figure 5.3: Clause for type VP2

$$\Sigma_4 = \left\{ \begin{array}{lll} X \mid \mathrm{IV}, & & \\ X \mid phon\colon P_0, & P_0 \mid ne\text{-}list & \\ & P_0 \mid hd\colon Y, & Y \mid sleeps \\ & P_0 \mid tl\colon P, & P \mid list \\ X \mid phon\text{-}aux\colon P & & \end{array} \right\}$$

$$\Sigma_5 = \left\{ \begin{array}{lll} X \mid \mathrm{SV}, & & \\ X \mid phon\colon P_0, & P_0 \mid ne\text{-}list & \\ & P_0 \mid hd\colon Y, & Y \mid knows \\ & P_0 \mid tl\colon P, & P \mid list \\ X \mid phon\text{-}aux\colon P & & \end{array} \right\}$$

$$\Sigma_6 = \left\{ \begin{array}{lll} X \mid \mathrm{NP}, & & \\ X \mid phon\colon P_0, & P_0 \mid ne\text{-}list & \\ & P_0 \mid hd\colon Y, & Y \mid Arthur \\ & P_0 \mid tl\colon P, & P \mid list \\ X \mid phon\text{-}aux\colon P & & \end{array} \right\}$$

Figure 5.4: Clause for types IV, SV and NP

Once we have the clauses, we can compile the program itself. The resulting program is shown in fig. 5.5.

$$\{\ \langle\Sigma_1, X, \{Y_1, Y_2\}\rangle,\ \langle\Sigma_2, X, \{Y_1, Y_2\}\rangle,\ \langle\Sigma_3, X, \{Y\}\rangle,$$
$$\langle\Sigma_4, X, \emptyset\rangle,\ \langle\Sigma_5, X, \emptyset\rangle,\ \langle\Sigma_6, X, \emptyset\rangle\ \}$$

Figure 5.5: Lazy program for "Arthur sleeps" grammar

Recall from the definition of a lazy program (def. 5.4 on p. 101) that a program item is a 3-tuple consisting of a clause, a variable from that clause, and a set of variables from the clause. The single variable is the root variable of the clause, and the set of variables are subgoals in the sense that they represent nodes that need to be checked further. Those goals correspond to the *gram* goals from the last chapter. If you compare the number of subgoals in the lazy version of our example in fig. 5.5 with that in the non-lazy version of the last chapter (fig. 4.5 on p. 90), you will notice that the number of subgoals is now significantly smaller. That is, we have not only simplified the compilation, we have also drastically reduced the search space.

We now turn to solving queries with respect to a lazy program. We need a few ancillary definitions.

Definition 5.5
If S is a set of types, then $\max(S) := \{a \in S \mid$ *for all* $b \in S$, *if* $a \preceq b$ *then* $a = b\}$.

Definition 5.6
$\mathbf{gst}(a) := \max(\{b \mid b \preceq a$ *and* b *is not constrained* $\})$ *is the set of greatest simple types below* a.

We can now specify what a one-step derivation is. We assume that there is a selection function that for each item $\langle\Sigma, S\rangle$ with non-empty S picks out an element of S.

Definition 5.7 (derivation item)
A derivation item is a pair $\langle\Sigma, S\rangle$, *where* Σ *is a feature clause and* S *is a subset of the equivalence classes of* $\mathrm{BV}(\Sigma)$.

During a derivation step, we will change the constraint clause by adding new information. This means that the variable equivalence classes in the derived clause may be different from the ones in the original one.

Definition 5.8
Let Σ, Σ' *be feature clauses s.t.* $\mathrm{FV}(\Sigma) \subseteq \mathrm{FV}(\Sigma')$ *and let* S *be a set of variable equivalence classes from* Σ*. Define* $S_{|\Sigma'} := \{[X]_{\Sigma'} \mid [X]_\Sigma \in S\}$.

Definition 5.9 (derivation)

$$\langle \Sigma, S\rangle \stackrel{1}{\mapsto} \langle \Sigma', S'\rangle \quad \text{if } [Y] \text{ is the selected variable (class) in } S,\ \langle \Lambda, X, S''\rangle \in \mathcal{P}(\mathcal{G}),\ \mathrm{FV}(\Lambda) \cap \mathrm{FV}(\Sigma) = \emptyset,\ \Sigma' = \mathrm{NF}(\Sigma \cup \Lambda \cup \{Y \mid X\}) \text{ is consistent and } S' = (S_{|\Sigma'} \cup S''_{|\Sigma'}) \setminus \{[Y]_{\Sigma'}\}.$$

$$\langle \Sigma, S\rangle \stackrel{1}{\mapsto} \langle \Sigma', S \setminus \{[Y]\}\rangle \quad \text{if } [Y] \text{ is the selected variable in } S,\ Y \mid a \in \Sigma,\ b \in \mathbf{gst}(a) \text{ and } \Sigma' = (\Sigma \setminus \{Y \mid a\}) \cup \{Y \mid b\}$$

A derivation is a non-empty sequence (finite or infinite) of items $\mathbf{D} = \langle I_1, I_2, I_3, \ldots\rangle$ *s.t. for each* $I_i, I_{i+1} \in \mathbf{D}$, $I_i \stackrel{1}{\mapsto} I_{i+1}$

Definition 5.10
A goal is an item $\langle \Sigma, \{[X] \mid X \in \mathrm{BV}(\Sigma),\ X \mid a \in \Sigma,\ a \text{ constrained }\}\rangle$*, where* Σ *is finite and in normal form.*

Due to our lazy approach, we can't simply submit a term ϕ as a query only selecting the root variable ($\langle \mathbf{trans}(X, \phi), \{[X]\}\rangle$), as we did in chapter 4. This would lead to incorrect results, as we will see when we prove the correctness of lazy resolution.

Definition 5.11
A successful lazy derivation is a finite derivation $\langle \Sigma, S\rangle \stackrel{*}{\mapsto} \langle \Sigma', \emptyset\rangle$*, where* $\langle \Sigma, S\rangle$ *is a goal.*

We can now consider some examples of goals and derivations. As our program, we will continue using the "Arthur sleeps" example. Consider the following trivial query.

$$\langle \{X \mid \mathrm{IV}\}, \emptyset\rangle$$

First note that this is indeed a legal goal. The set of variables is empty, since there are simply no branching variables in the constraint set. But this means that we don't need to prove anything, the goal is already the last item in a successful derivation. But why is this correct? Recall that we require grammars to be type consistent. Our example grammar is type consistent, as can be easily seen. That means that models of the grammar with IV objects in their domain are guaranteed to exist – and this is all that the example query asks: do objects of type IV exist in some model of the grammar? Note that the fact that IV is a constrained type does not matter in this case. Let's consider a slightly more complicated query.

$$\langle\{X \,|\, \mathrm{IV},\ X \,|\, \textit{phon-aux}{:}Y,\ Y \,|\, \textit{e-list}\}, \{[X]\}\rangle$$

This time, we have a non-empty variable set. X is assigned a constrained type (IV) and has a feature selection constraint defined for it. Thus, by the definition of a goal, it needs to be in the variable set. We get the following one-step derivation.

$$\begin{aligned}
&\langle\{X \,|\, \mathrm{IV},\ X \,|\, \textit{phon-aux}{:}P,\ P \,|\, \textit{e-list}\}, \{[X]\}\rangle \ \overset{1}{\mapsto} \\
&\langle\{X \,|\, \mathrm{IV},\ X \,|\, \textit{phon}{:}P_0,\ P_0 \,|\, \textit{ne-list},\ P_0 \,|\, \textit{hd}{:}Y, \\
&\quad Y \,|\, \textit{sleeps},\ P_0 \,|\, \textit{tl}{:}P,\ P \,|\, \textit{e-list},\ X \,|\, \textit{phon-aux}{:}P\},\ \emptyset\rangle
\end{aligned}$$

We have thus successfully applied the definition of the IV constraint. No other derivations are possible because

1. no other constraint is consistent with the goal, and
2. IV does not subsume an unconstrained type.

We now define the concept of a failed derivation. A derivation is failed if the last item has a non-empty variable set and if no transition is possible from the last item. If all derivations for a goal are failed, then the goal is failed and the grammar does not admit the goal. Again, we implicitly assume a goal selection function.

Definition 5.12
A goal $\langle\Sigma, S\rangle$ is finitely failed iff there are only finitely many derivations from $\langle\Sigma, S\rangle$, and for each derivation from $\langle\Sigma, S\rangle$ there exist Σ' and non-empty S' s.t. $\langle\Sigma, S\rangle \overset{*}{\mapsto} \langle\Sigma', S'\rangle \not\mapsto$

As an example, consider the following goal.

$$\langle\{X\,|\,\mathrm{IV},\ X\,|\,phon{:}Y,\ Y\,|\,e\text{-}list\},\{[X]\}\rangle \quad \not\mapsto$$

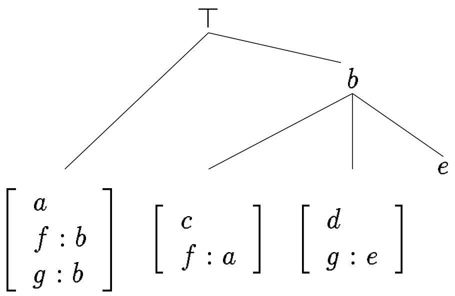
This goal is failed since no transition is defined for it. There is no IV object that has a *e-list* as *phon* value.

5.2 Soundness of finite success

The soundness proof for the success case proceeds in two steps. First (prop. 5.4), we show that if a feature clause has certain properties, then there is a model of the given grammar that satisfies the clause. Then (prop. 5.6), we show that the feature clause in the last item of a successful derivation has these properties. We begin by defining some auxiliary interpretations that we will use in the next proof. Since the relevant grammars are type consistent, we know that such models exist.

Definition 5.13
Let $\langle \mathcal{T}, \preceq, \mathcal{F}, approp\rangle$ be a signature and $\mathcal{G}$ a type consistent grammar. For every $a \in \mathcal{V}$, let $\mathcal{I}_a = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}\rangle$ be a model of $\mathcal{G}$ s.t. for some $u \in \mathcal{U}$, $\mathcal{S}(u) = a$. Furthermore, let $u(\mathcal{I}_a)$ denote some $u \in \mathcal{U}$ s.t. $\mathcal{S}(u) = a$.

The next proposition is the crucial result for the soundness of finite success. It says that if a clause Σ is such that for each branching variable of a constrained type, the constraints on that variable *entail* some item in the lazy program, then the grammar admits Σ. Intuitively this is the case since the clause satisfies all the grammar constraints. In the proof, we show the existence of a model of the grammar that satisfies Σ. We do this by constructing a partial model that contains objects for the branching variables. The partial model is constructed from some arbitrary interpretation that satisfies Σ. Such an interpretation exists since Σ is assumed to be consistent. This partial model then gets "filled in" with some generic models as we defined them in def. 5.13 – we have to make use of the fact that the grammar is type consistent. If the grammar were not type consistent, then the whole proof would break down and we could not actually show soundness of finite success. The system is unsound for grammars that are not type consistent! An example will illustrate this. We will use the following signature.



We use a trivial (type consistent) grammar.

$$\{\ a\ \rightarrow\ f : X\ \wedge\ g : X\ \}$$

For the grammar and derivations in this example, we will use a simplified AVM notation that is much more readable the constraint sets we use in the formal definitions and proofs. The correspondence between this notation and the underlying formal system should be immediately obvious. The lazy program will then contain a single item.

$$\langle\, \boxed{\mathrm{x}} \begin{bmatrix} a & \\ f & \boxed{\mathrm{y}}\ b \\ g & \boxed{\mathrm{y}} \end{bmatrix},\ \boxed{\mathrm{x}},\ \emptyset \rangle$$

Now if we pose the query $f : b$, we get the following derivation.

$$\langle\, \boxed{\mathrm{x}} \begin{bmatrix} a \\ f : b \end{bmatrix},\ \{\boxed{\mathrm{x}}\}\ \rangle \ \overset{1}{\mapsto}\ \langle \begin{bmatrix} a & \\ f & \boxed{\mathrm{y}}\ b \\ g & \boxed{\mathrm{y}} \end{bmatrix},\ \emptyset \rangle$$

That is, we've shown that the grammar admits our query (if the soundness result holds). Now consider adding a new constraint to the grammar.

$$\{\ a\ \rightarrow\ f : X\ \wedge\ g : X,\ \ b\ \rightarrow\ f : X\ \wedge\ g : X\ \}$$

Our grammar is not type consistent anymore. The new constraint gives inconsistent information about objects of type b: there can be no objects of type b that satisfy this constraint. Now consider what happens to the lazy program and the query. Since translation of the new grammar constraint yields an empty matrix (the consequence of the constraint is inconsistent), the lazy program derived from the new grammar is exactly identical with the program derived from the first one. Thus, the derivation on the query $f : b$ is also identical and thus incorrect. Since there are no objects of type b in any model of the grammar, the grammar does not admit $f : b$. In fact, there is no model of this grammar with a non-empty universe at all. Notice that the incorrectness of this derivation stems only from the fact that we've applied lazy derivation to a grammar that's not type consistent. Consider a different query, namely $f : (c \wedge f : a)$. That term will get translated into the following query.

$$\langle \boxed{\mathrm{x}} \begin{bmatrix} a & \\ f & \boxed{\mathrm{y}} \begin{bmatrix} c & \\ f & a \end{bmatrix} \end{bmatrix}, \{\boxed{\mathrm{x}}, \boxed{\mathrm{y}}\} \rangle$$

The variable set of this query also contains the tag $\boxed{\mathrm{y}}$, since c is a constrained type and $\boxed{\mathrm{y}}$ is a branching variable. The derivation then goes as follows.

$$\langle \boxed{\mathrm{x}} \begin{bmatrix} a & \\ f & \boxed{\mathrm{y}} \begin{bmatrix} c & \\ f & a \end{bmatrix} \end{bmatrix}, \{\boxed{\mathrm{x}}, \boxed{\mathrm{y}}\} \rangle \overset{1}{\mapsto}$$

$$\langle \begin{bmatrix} a & \\ f & \boxed{\mathrm{y}} \begin{bmatrix} c & \\ f & a \end{bmatrix} \\ g & \boxed{\mathrm{y}} \end{bmatrix}, \{\boxed{\mathrm{y}}\} \rangle \not\mapsto$$

For this query, we get the correct result since we require that some constraint be applied to $\boxed{\mathrm{y}}$. The example has therefore shown that lazy evaluation will not always work for grammars that are not type consistent. In fact, we can show that type consistency is a necessary precondition for the correctness of lazy evaluation.

Proposition 5.3
Let $\mathcal{G}$ be any grammar. If lazy evaluation is sound for $\mathcal{P}(\mathcal{G})$, then $\mathcal{G}$ is type consistent.

Proof

Suppose $\mathcal{G}$ is not type consistent and let t be an inconsistent type. Then the goal $\langle\{X \mid t\}, \emptyset\rangle$ is also a terminal item in a derivation of length 0, contradiction.

■

This shows that any restriction that guarantees soundness of lazy evaluation implies type consistency. It remains to be shown that type consistency is also sufficient.

Proposition 5.4
Let $S = \langle \mathcal{T}, \preceq, \mathcal{F}, approp \rangle$ *be a signature,* $\mathcal{G}$ *a type consistent grammar,* Σ *a consistent set of constraints,* $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ *an interpretation and* α *an assignment s.t.* $\mathcal{I}, \alpha \models \Sigma$*. If for each* $X \in \{X \in \mathrm{BV}(\Sigma) \mid \mathcal{S}(\alpha(X))$ *is constrained* $\}$ *there is a* $\langle \Sigma', X', S \rangle \in \mathcal{P}(\mathcal{G})$ *s.t.* $\Sigma \models \Sigma' \cup \{ X | X' \}$*, then* $\mathcal{G}$ *admits* Σ*.*

Proof

For each $a \in \mathcal{V}$, $[X]_\alpha \in \mathrm{FV}(\Sigma)$, let $\mathcal{I}_a^{[X]_\alpha} = \langle \mathcal{U}_a^{[X]_\alpha}, \mathcal{S}_a^{[X]_\alpha}, \mathcal{A}_a^{[X]_\alpha} \rangle$ be a "copy" of $\mathcal{I}_a$. Define a new interpretation $\mathcal{I}' = \langle \mathcal{U}', \mathcal{S}', \mathcal{A}' \rangle$ and assignment α' s.t.

$$\begin{aligned}\mathcal{U}' = \; & \{u \in \mathcal{U} \mid \exists X, Y \in \mathrm{FV}(\Sigma).\ \exists f \in \mathcal{F}.\ \alpha(X) = u,\ \mathcal{A}(f)(u)\downarrow \text{ and } \\ & \mathcal{A}(f)(u) = \alpha(Y)\} \cup \bigcup\{\mathcal{U}_a \mid a \in \mathcal{V}\} \cup \bigcup\{\mathcal{U}_a^X \mid a \in \mathcal{V} \text{ and } [X]_\alpha \in \Sigma\}\end{aligned}$$

$$\mathcal{S}'(u) = \begin{cases} \mathcal{S}(u) & \text{if } u \in \mathcal{U} \\ \mathcal{S}_a^{[X]_\alpha}(u) & \text{if } u \in \mathcal{U}_a^{[X]_\alpha} \\ \mathcal{S}_a(u) & \text{if } u \in \mathcal{U}_a \end{cases}$$

$$\alpha'(X) = \begin{cases} \alpha(X) & \text{if } X \in \mathrm{FV}(\Sigma) \text{ and } \exists Y \in \mathrm{FV}(\Sigma).\ \exists f \in \mathcal{F}. \\ & \mathcal{A}(f)(\alpha(X))\downarrow \text{ and } \mathcal{A}(f)(\alpha(X)) = \alpha(Y) \\ u(\mathcal{I}^{[X]_\alpha}_{\mathcal{S}(\alpha(X))}) & \text{if } X \in \mathrm{FV}(\Sigma) \text{ and } \neg(\exists Y \in \mathrm{FV}(\Sigma).\ \exists f \in \mathcal{F}. \\ & \mathcal{A}(f)(\alpha(X))\downarrow \text{ and } \mathcal{A}(f)(\alpha(X)) = \alpha(Y)) \\ u(\mathcal{I}_a) & \text{otherwise, where } a \text{ is the lex. smallest minimal} \\ & \text{type} \end{cases}$$

$$\mathcal{A}'(f)(u) = \begin{cases} u' & \text{if } \exists X, Y \in \mathrm{FV}(\Sigma).\ \alpha(X) = u,\ \mathcal{A}(f)(u)\downarrow, \\ & \mathcal{A}(f)(u) = \alpha(Y) \text{ and } \alpha'(Y) = u' \\ u_{\mathcal{S}(\mathcal{A}(f)(u))} & \text{if } \exists X \in \mathrm{FV}(\Sigma).\ \alpha(X) = u, \mathcal{A}(f)(u)\downarrow \\ & \text{and } \neg(\exists Y \in \mathrm{FV}(\Sigma).\ \alpha(Y) = \mathcal{A}(f)(u)) \\ \mathcal{A}_a(f)(u) & \text{if } u \in \mathcal{U}_a \text{ and } \mathcal{A}_a(f)(u)\downarrow \\ \mathcal{A}_a^{[X]_\alpha}(f)(u) & \text{if } u \in \mathcal{U}_a^{[X]_\alpha} \text{ and } \mathcal{A}_a^{[X]_\alpha}(f)(u)\downarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

We now show that 1. $\mathcal{I}'$ is an interpretation with respect to S, that 2. $\mathcal{I}', \alpha' \models \Sigma$ and that 3. $\mathcal{I}'$ models $\mathcal{G}$.

1. Clearly, we only need to consider objects $u \in (\mathcal{U} \cap \mathcal{U}')$. First note that

$\mathcal{A}'(f)(u)\downarrow$ iff $\mathcal{A}(f)(u)\downarrow$. Secondly, if $\mathcal{A}'(f)(u)\downarrow$ and $\mathcal{A}'(f)(u) \in (\mathcal{U} \cap \mathcal{U}')$, then $\mathcal{S}'(\mathcal{A}'(f)(u)) = \mathcal{S}(\mathcal{A}(f)(u))$, by definition of α'. And finally, if $\mathcal{A}'(f)(u)\downarrow$ and $\mathcal{A}'(f)(u) \notin (\mathcal{U} \cap \mathcal{U}')$, then $\mathcal{A}'(f)(u) = u_{\mathcal{S}(\mathcal{A}(f)(u))}$ and thus, $\mathcal{S}'(\mathcal{A}'(f)(u)) = \mathcal{S}(\mathcal{A}(f)(u))$.

2. By the definition of α', we know that for each variable $X, Y \in \mathrm{FV}(\Sigma)$, $\mathcal{S}(\alpha(X)) = \mathcal{S}'(\alpha'(X))$ and $\alpha'(X) = \alpha'(Y)$ iff $\alpha(X) = \alpha(Y)$. Therefore, $\mathcal{I}', \alpha' \models \Sigma$.

3. We need to show that for each $u \in \mathcal{U}'$, u satisfies all the constraints in $\mathcal{G}$. This is clearly true for each $u \in \mathcal{U}_a$ or $u \in \mathcal{U}_a^{[X]_\alpha}$, for all a and X. Again, we have to consider objects $u \in (\mathcal{U} \cap \mathcal{U}')$. We only need to worry if $\mathcal{S}(u)$ is constrained. But if $\mathcal{S}(u)$ is constrained, then for some $X \in \mathrm{FV}(\Sigma)$, $\alpha(X) = u$ and $X \in \{X \in \mathrm{FV}(\Sigma) \mid \mathcal{S}(\alpha(X))$ is constrained and $\exists Y \in \mathrm{FV}(\Sigma).\ \exists f \in \mathcal{F}.\ \mathcal{A}(f)(\alpha(X))\downarrow$ and $\mathcal{A}(f)(\alpha(X)) = \alpha(Y)\}$, and for some $\langle \Sigma', X', S \rangle \in \mathcal{P}(\mathcal{G})$, $\Sigma \models \Sigma' \cup \{X \mid X'\}$. Thus, every constraint in $\mathcal{G}$ is true of u.

We've shown that there is a model $\mathcal{I}'$ of $\mathcal{G}$ s.t. $[\![\Sigma]\!]^{\mathcal{I}} \neq \emptyset$. Therefore we've shown that $\mathcal{G}$ admits Σ.

■

We've shown that if every branching variable of a constrained type in a clause has the property that its constraints entail a program clause, the clause is admitted by the grammar. We will now show that during a derivation, each variable in a clause either already has the desirable property, or it is on the goal list. The two results together will give us the soundness result we're after.

Proposition 5.5
Let $\langle \Sigma, S \rangle$ be a goal and $\langle \Sigma, S \rangle \overset{n}{\mapsto} \langle \Sigma', S' \rangle$. For each branching $X \in \mathrm{FV}(\Sigma')$, if $X \mid a \in \Sigma'$ and a constrained, then either for some $Y \in [X]$, $Y \in S'$, or for some $\langle \Lambda, Z, T \rangle \in \mathcal{P}(\mathcal{G})$, $\Sigma' \models \Lambda \cup \{X \mid Z\}$.

Proof

We prove this result by induction on n.

$n = 0$: since $\langle \Sigma, S \rangle$ is a goal, every branching $X \in \mathrm{FV}(\Sigma)$ s.t. $X \mid a \in \Sigma$ and a constrained is in S.

$n \rightsquigarrow n+1$: suppose $\langle \Sigma, S \rangle \overset{n}{\mapsto} \langle \Sigma', S' \rangle \overset{1}{\mapsto} \langle \Sigma'', S'' \rangle$.

1. Suppose Y is the selected variable in S', $\langle \Lambda, Z, T \rangle \in \mathcal{P}(\mathcal{G})$, $\mathrm{FV}(\Lambda) \cap \mathrm{FV}(\Sigma) = \emptyset$, $\Sigma' \cup \Lambda \cup \{Y | Z\}$ is consistent, $\Sigma'' = \mathrm{NF}(\Sigma' \cup \Lambda \cup \{Y | Z\})$ and $S'' = (S'_{|\Sigma''} \cup T_{|\Sigma''}) \setminus [Y]_{\Sigma''}$. Suppose X is branching in Σ'', $X | a \in \Sigma''$ and a is constrained. If $X \in [Y]_{\Sigma''}$, then $\Sigma'' \models \Lambda \cup \{Y | Z\}$. Otherwise, for some $X' \in [X]_{\Sigma''}$, X' is branching in Σ' or in Λ. If X' is branching in Σ', then the condition holds by induction (either for some $X'' \in [X']_{\Sigma'}$, $X'' \in S'$ and thus, $X'' \in S''$, or for some $\langle \Lambda', Z', T' \rangle \in \mathcal{P}(\mathcal{G})$, $\Sigma' \models \Lambda' \cup \{X'' | Z'\}$ and thus, $\Sigma'' \models \Lambda' \cup \{X'' | Z'\}$). If X' is branching in Λ, then by definition of $\langle \Lambda, Z, T \rangle$, $X' \in T$ and thus, $X' \in S''$.

2. Suppose that Y is the selected variable in S', for some $X \in [Y]$, $X | a \in \Sigma'$, $t \preceq a$, t is simple, $\Sigma'' = \mathrm{NF}(\Sigma' \cup \{X | t\})$ and $S'' = S' \setminus \{Y\}$. Since nothing has changed except for the variables in $[Y]$, and those are not constrained, the condition holds for $\langle \Sigma'', S'' \rangle$.

■

We can now combine the two previous propositions to get the soundness result.

Theorem 5.6
Let $\mathcal{G}$ be a type consistent grammar and $\langle \Sigma, S \rangle$ a goal. If $\langle \Sigma, S \rangle \overset{*}{\mapsto} \langle \Sigma', \emptyset \rangle$, then $\mathcal{G}$ admits Σ.

Proof

$\langle \Sigma, S \rangle \overset{*}{\mapsto} \langle \Sigma', \emptyset \rangle$

$\Rightarrow$ Σ' is consistent, $\Sigma' \models \Sigma$ and for each branching $X \in \mathrm{FV}(\Sigma')$ s.t. $X | a \in \Sigma'$ and a is constrained, for some $\langle \Lambda, Y, T \rangle \in \mathcal{P}(\mathcal{G})$, $\Sigma' \models \Lambda \cup \{X | Y\}$ (by prop. 5.5)

$\Rightarrow$ $\mathcal{G}$ admits Σ' (by prop. 5.4)

$\Rightarrow$ $\mathcal{G}$ admits Σ (since $\Sigma' \models \Sigma$)

■

5.2.1 Soundness of finite failure

We will now show that if a goal $\langle \Sigma, S\rangle$ is finitely failed with respect to a grammar $\mathcal{G}$, then $\mathcal{G}$ does not admit Σ. The reason why we can't simply show completeness of the success case and infer soundness of finite failure from that should be clear from previous discussions: there can be no system that is complete for the success case. However, we will show completeness of finite failure in the next section.

The first proposition shows that if there is a derivation item $\langle \Sigma, S\rangle$ s.t. $S \neq \emptyset$ and $\mathcal{G}$ admits Σ, then we can derive at least one other item from $\langle \Sigma, S\rangle$. Or put the other way round, if no other item can be derived from $\langle \Sigma, S\rangle$ ($\langle \Sigma, S\rangle \not\mapsto$), then Σ is not admitted by $\mathcal{G}$.

Proposition 5.7
Let $\mathcal{G}$ be a grammar and $\langle \Sigma, S\rangle$ a goal with $S \neq \emptyset$. If $\mathcal{G}$ admits Σ, then there are Σ', S' s.t. $\langle \Sigma, S\rangle \stackrel{1}{\mapsto} \langle \Sigma', S'\rangle$ and $\mathcal{G}$ admits Σ'.

Proof
Let X be the selected variable in S and $X' \in \mathrm{FV}(\Sigma)$ s.t. $X \equiv_\Sigma X'$ and $X'|\, a \in \Sigma$. Suppose $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}\rangle$ is a model of $\mathcal{G}$ and $\mathcal{I}, \alpha \models \Sigma$.

1. If $\mathcal{S}(\alpha(X))$ is simple, then for some $b \in \mathbf{gst}(a)$, $\mathcal{S}(\alpha(X)) \preceq b$ and if $\Sigma' = \mathrm{NF}(\Sigma \cup \{X|\, b\})$, then $\langle \Sigma, S\rangle \stackrel{1}{\mapsto} \langle \Sigma', S \setminus \{X\}\rangle$. Furthermore, $\mathcal{I}, \alpha \models \Sigma'$ and thus, $\mathcal{G}$ admits Σ'.

2. If $\mathcal{S}(\alpha(X))$ is constrained, then for some $\langle \Lambda, Z, T\rangle \in \mathcal{P}(\mathcal{G})$, for some β, $\beta(Z) = \alpha(X)$ and $\mathcal{I}, \beta \models \Lambda$. Define $\alpha'(Y) := \begin{cases} \beta(Y) & \text{if } Y \in \mathrm{FV}(\Lambda) \\ \alpha(Y) & \text{otherwise} \end{cases}$
   Let $\Sigma' := \Sigma \cup \Lambda \cup \{X|\, Z\}$. By definition of α', we know that $\mathcal{I}, \alpha' \models \Sigma'$. Thus, Σ' is consistent, $\mathcal{G}$ admits Σ' and $\langle \Sigma, S\rangle \stackrel{1}{\mapsto} \langle \Sigma', (S \cup T) \setminus \{X\}\rangle$.

■

Definition 5.14
$\langle \Sigma, S\rangle$ is n-failed iff $\langle \Sigma, S\rangle$ is finitely failed and $n \geq \max(\{n' \mid \langle \Sigma, S\rangle \stackrel{n'}{\mapsto} \langle \Sigma', S'\rangle \not\mapsto\})$.

It is now easy to show that for any finitely failed goal $\langle \Sigma, S\rangle$, $\mathcal{G}$ does not admit Σ. Notice by the way that the results in this section do not require grammars to be type consistent. The correctness of failure does not depend upon type consistency.

Proposition 5.8
Let $\mathcal{G}$ be a grammar. If $\langle \Sigma, S \rangle$ is n-failed for some $n \in \mathbb{N}$, then $\mathcal{G}$ does not admit Σ.

Proof

If $n = 0$, then $\langle \Sigma, S \rangle \not\mapsto$. Suppose $\mathcal{G}$ admits Σ. Thus by prop. 5.7 some Σ', S' exist s.t. $\langle \Sigma, S \rangle \overset{1}{\mapsto} \langle \Sigma', S' \rangle$, contradiction. Therefore, $\mathcal{G}$ does not admit Σ.

Suppose the claim is true for n, and suppose $\langle \Sigma, S \rangle$ is $(n+1)$-failed. Thus, each $\langle \Sigma', S' \rangle$ s.t. $\langle \Sigma, S \rangle \overset{1}{\mapsto} \langle \Sigma', S' \rangle$ is n-failed and $\mathcal{G}$ does not admit Σ'. Thus again by prop. 5.7, $\mathcal{G}$ does not admit Σ.

■

5.2.2 Infinite proofs and completeness of finite failure

Definition 5.15
A derivation is fair *iff for every item $\langle \Sigma, S \rangle$ and every variable $X \in S$ there is an item $\langle \Sigma', S' \rangle$ s.t. $\langle \Sigma, S \rangle \overset{*}{\mapsto} \langle \Sigma', S' \rangle$ and $[X] \notin S'$.*

Proposition 5.9
If $\mathcal{G}$ is a grammar, $\langle \Sigma_0, S_0 \rangle$ a goal and $\mathbf{D} = \langle \Sigma_0, S_0 \rangle \overset{1}{\mapsto} \langle \Sigma_1, S_1 \rangle \overset{1}{\mapsto} \ldots$ an infinite fair derivation, then $\mathcal{G}$ admits Σ.

Proof

Consider $\Sigma_\infty = \bigcup\{\Sigma_i \mid i \in \mathbb{N}\}$. Since for each $i \in \mathbb{N}$, Σ_i is consistent and for each $i \in \mathbb{N}$, $\Sigma_{i+1} \models \Sigma_i$, we know that Σ_∞ is consistent (by compactness). Furthermore, since $\mathbf{D}$ is fair, we know that if X is branching in Σ_∞ and $X|\, a \in \Sigma_\infty$, a constrained, then $a \in \mathcal{V}$ and for some Γ, $a \to \Gamma \in \mathcal{G}$, for some ${}^Y\Sigma \in \Gamma$, $\Sigma_\infty \models \Sigma \cup \{X|\, Y\}$. Thus, by prop. 5.4, $\mathcal{G}$ admits Σ_∞, and thus, Σ_0.

■

This proposition has two major consequences:

1. We can use it to show that for every ungrammatical term, there is a finite proof that the term is in fact ungrammatical. Thus, our system is complete for showing ungrammaticality.

2. The other consequence is a practical one. If we can detect an infinite proof branch for some term, then we know that term is grammatical. Of course this is not possible in the general case, but it may be possible in some interesting sub-cases.

This concludes the completeness proof.

5.3 Conclusion

In this chapter, we provided a solution to the prediction problem based on lazy type evaluation/unfolding. We showed that lazy evaluation was sound, provided that the grammar is type consistent. We also showed that type consistency is a necessary condition for the soundness of lazy evaluation.

Completeness was shown in the sense that each query that is not predicted by the grammar will terminate. As before, we know that this is as good as we can get, given the general undecidability of prediction.

On p. 59, we claimed that computation with constraint-based grammars could be just as efficient as with rule-based ones. We showed that this is the case in this chapter, since we have effectively solved the constraint satisfaction problem through a rule application approach. In the system presented in this chapter, there is not notion of arbitrarily generating structures, and then running them through a constraint filter. Rather, through our grammar transformations, we can use the constraints themselves to drive the generation of appropriate structure.

Proof systems addressing related problems were described in (Carpenter 1992) and (Aït-Kaci et al. 1993). Carpenter's system is unification-based, and operates on the domain of finite feature structures. It uses an open-world type system, and does not allow for negation. That means that constraints of the form $\phi \rightarrow \psi$ are not allowed in general. Instead, Carpenter defines type constraint (i.e., constraints of the form $t \rightarrow \phi$) to have a special meaning that is outside of the normal term interpretation. We do not need to do that, due to our constraint-based approach, as opposed to unification of finite structures. Since Carpenter's system is based on finite structures, he obtains completeness for the prediction problem, and not the non-prediction one.

The system of (Aït-Kaci et al. 1993) is similar in the sense that they use an open world type interpretation, and don't allow any form of negation.

However, their domain of interpretation are the finite *and* infinite feature structures, which gives their system complexity properties similar to our own. Their main contribution was their idea to put lazy evaluation on a sound basis by restricting the form of grammars.

It is interesting to note a possible difference in proof systems for the two approaches. In a system based on finite feature structures, an infinite proof branch means a failure branch, since there are no infinite structures in the domain. Yet to show failure in such a system, *every* proof branch must be either infinite or failed. This means that by detecting infinite branches, which are success branches in our system, we can terminate on a properly bigger class of queries than a system based on finite structures.

5.3.1 Detecting infinite proof branches

If we could detect infinite proof branches in every case, then we would have a decision procedure for the grammaticality problem. Since we know that the problem is undecidable, we don't need to waste our time looking for a general infinite branch detection mechanism. Still, it is possible to detect infinite branches in some cases, and we will now consider ways to do that.

Consider the example we had before.

$$\textit{person} \Rightarrow \text{father: gender: } \textit{male}$$

Suppose we would like to prove grammaticality of $\textit{person} \wedge \text{gender:}\,\textit{male}$. We start out with the constraint set

$$\{X \,|\, \textit{person},\ X \,|\, \text{gender:}\,Y,\ Y \,|\, \textit{male}\}$$

Unifying in the constraint on person, we get something like

$$\{X \,|\, \textit{person},\ X \,|\, \text{gender:}\,Y,\ Y \,|\, \textit{male},\ X \,|\, \text{father:}\,Z,\ Z \,|\, \textit{person},\ Z \,|\, \text{gender:}\ Z',\ Z' \,|\, \textit{male}\}$$

The only variable that we can expand now is Z. We note that the constraint on Z is exactly like the one we started out with (up to variable renaming). We can therefore conclude that we could go on like this forever, expanding variables of type *person*. Notice that this is an infinite proof branch only because Z is reachable from X. If this were not the case, then we would simply have to prove the same thing in two different locations, so to speak, which of course does not necessarily lead to an infinite proof, just a redundant one.

We can generalize what we learned from this example to the following condition: if Y is reachable from X, X has already been expanded and the constraints on Y are more general than the ones on X, then Y does need to be expanded. This is so because we can generate a cyclic solution by identifying X and Y. Alternatively, we could also generate an infinite, periodic solution.

Actually, the proof above could also be terminated by introducing a cycle with $X|Y$. This should not lead one to conclude that term with infinite proofs generally have finite, cyclic solutions as well. A simple example will illustrate this.

$$\mathit{count} \rightarrow \text{current:} X \wedge \text{next:} (\mathit{count} \wedge \text{current:prev:} X)$$

Now consider the query $\mathit{count} \wedge \text{current:}\,\mathit{nil}$. There is no finite solution to this query, cyclic or acyclic. An infinite solution is shown in fig. 5.6.

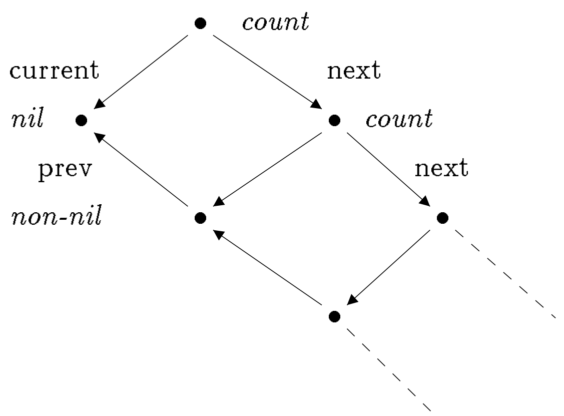


Figure 5.6: Infinite solution for $\mathit{count} \wedge \text{current:}\,\mathit{nil}$

Chapter 6

Adding relations

It is common usage among HPSG linguists to assume that the description language contains relation symbols in addition to the usual feature and type symbols. We will now consider a way how to formally integrate definite relations into feature constraint grammars as we've defined them in previous sections. This chapter builds on word described in (Götz and Meurers 1997b).

Notice first that it is not possible to simply apply the constraint logic programming scheme of (Höhfeld and Smolka 1988). If we simply wrapped the CLP scheme around our current notion of grammar, we would end up with a system that is not complete. Recall that (Höhfeld and Smolka 1988) requires that the constraint language be decidable (that is, the satisfiability problem for that language). As we saw in ch. 3, the prediction problem, which corresponds to the satisfiability problem, is undecidable. Not only is it undecidable, it is not even enumerable (only the co-problem, non-prediction, is). Thus, in applying the CLP scheme to our grammars, we would end up with a system where termination was guaranteed neither for queries that are consequences of the program, nor those that aren't. Since the whole point of the CLP scheme is to provide an off-the-shelf sound and complete procedural semantics for an arbitrary (decidable) constraint language, we will need to come up with a different, customized solution.

The first issue we need to consider is that normally, relational atoms (relation symbols applied to a proper number of arguments) are considered to have a different semantic type than feature terms. Feature terms, as we've defined them, denote sets of objects, whereas relational atoms denote truth values. An atom can be either true or false in a given interpretation. If we want a formalism where feature terms and relational atoms have the same syntactic status, we need to reconcile this apparent contradiction. There are several ways we could do this:

- Since for computational purposes, we translate our feature terms into a relational language anyway (feature constraints), we could consider this relational language as our primary language and thus have no problems with integrating relations. However, this would be rather awkward. Constraint matrices are in disjunctive normal form, and we would thus be required to express grammars that way. It might be possible to construct a different relational feature language that allows a more natural and compact notation, but we will not pursue this possibility any further here.
- Another possibility would be to change the notion of a truth value. We might say that if a relational atom is true in a given interpretation, then it denotes the whole domain $\mathcal{U}$. If it is false, on the other hand, it denotes the empty set $\emptyset$.
- The last option is the one originally proposed by (Dörre and Eisele 1991) and more recently described in (Dörre and Dorna 1993), namely to use a *functional* interpretation for relations: one argument is always the selected result argument, and the denotation of an atom is the denotation of that argument.

Although the last option is maybe the most elegant one, we will stick with the second alternative, since it simplifies the formal treatment. Notice also that (Dörre and Dorna 1993) treat the functional notation by syntactically translating it into the relational one. There is thus no profound difference between the two approaches.

The work in this chapter draws heavily on techniques developed for logic programming. I will presuppose an understanding of logic programming as presented in (Lloyd 1984). Other techniques and results will be introduced as necessary.

6.1 Syntax and semantics

Definition 6.1
A signature is a sextuple $\langle \mathcal{T}, \preceq, \mathcal{F}, approp, \mathcal{R}, Ar \rangle$ *s.t.*

- $\langle \mathcal{T}, \preceq \rangle$ *is a finite join semi-lattice*
- $\mathcal{V} = \{t \in \mathcal{T} \mid \text{if } t' \preceq t \text{ then } t' = t\}$
- $\mathcal{F}$ *is a finite set of feature names*
- $approp : \mathcal{V} \times \mathcal{F} \rightharpoonup \mathcal{T}$ *is a partial function from pairs of minimal types and features to types*
- $\mathcal{R}$ *is a finite set of relation symbols*
- $Ar : \mathcal{R} \to \mathbb{N}$ *is a total function from the set of relation symbols to the natural numbers (the arity function)*

Definition 6.2 (interpretation)
Let $S = \langle \mathcal{T}, \preceq, \mathcal{F}, approp, \mathcal{R}, Ar \rangle$ *be a signature. An* S*-interpretation is a quadruple* $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{P} \rangle$ *s.t.*

- $\mathcal{U}$ *is a set of objects, the domain of* $\mathcal{I}$
- $\mathrm{ASS} = \mathcal{U}^{\mathsf{VAR}}$ *is the set of variable assignments in* $\mathcal{I}$
- $\mathcal{S} : \mathcal{U} \to \mathcal{V}$ *is a total function from the set of objects to the set of minimal types*
- $\mathcal{A} : \mathcal{F} \to \mathcal{U}^{\mathcal{U}}$ *is an attribute interpretation function s.t.*
  - *for each* $u \in \mathcal{U}$, *for each* $f \in \mathcal{F}$, *if* $approp(\mathcal{S}(u), f)$ *is defined and* $approp(\mathcal{S}(u), f) = t$, *then* $(\mathcal{A}(f))(u)\!\downarrow$ *and* $\mathcal{S}((\mathcal{A}(f))(u)) \preceq t$
  - *for each* $u \in \mathcal{U}$, *for each* $f \in \mathcal{F}$, *if* $(\mathcal{A}(f))(u)$ *is defined, then* $approp(\mathcal{S}(u), f)$ *is defined and* $\mathcal{S}((\mathcal{A}(f))(u)) \preceq approp(\mathcal{S}(u), f)$
- *for each* $r \in \mathcal{R}$, $\mathcal{P}(r) \subseteq \mathcal{U}^{\mathrm{Ar}(r)}$

It will be useful to distinguish between ordinary feature terms, and relational ones. We will use this distinction to disallow embedding of relation term inside each other. Logically, there is no problem with this, but it would be inconvenient from a practical perspective. We can also disallow relational atoms in the scope of negation this way. We will discuss ways to relax this condition later in this chapter.

Definition 6.3 (relational atom)
$r(\phi_1, \ldots, \phi_n)$ *is a relational atom iff* $r \in \mathcal{R}$, $Ar(r) = n$ *and* $\phi_1, \ldots, \phi_n$ *are feature terms*

Definition 6.4 (relation terms)

- ϕ — *if ϕ is a feature term*
- A — *if A is an atom*
- $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$ — *if ϕ_1 and ϕ_2 are relation terms*

Definition 6.5 (term interpretation)
We extend the interpretation function to also handle relation terms. Let $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{P} \rangle$ be an interpretation.

- $[\![r(\phi_1, ..., \phi_n)]\!]^{\mathcal{I}}_{\alpha} = \begin{cases} \mathcal{U} & \text{if } \langle u_1, \ldots, u_n \rangle \in \mathcal{P}(r) \text{ and } u_1 \in [\![\phi_1]\!]^{\mathcal{I}}_{\alpha}, \ldots, u_n \in [\![\phi_n]\!]^{\mathcal{I}}_{\alpha} \\ \emptyset & \text{otherwise} \end{cases}$
- $[\![\phi_1 \wedge \phi_2]\!]^{\mathcal{I}}_{\alpha} = [\![\phi_1]\!]^{\mathcal{I}}_{\alpha} \cap [\![\phi_2]\!]^{\mathcal{I}}_{\alpha}$
- $[\![\phi_1 \vee \phi_2]\!]^{\mathcal{I}}_{\alpha} = [\![\phi_1]\!]^{\mathcal{I}}_{\alpha} \cup [\![\phi_2]\!]^{\mathcal{I}}_{\alpha}$

Definition 6.6 (clauses)
If A is an atom and ϕ is a relation term, then $A := \phi$ is a (definite) clause.

As an example of definite clauses, consider the encoding of the *append* relation. Notice how in the first clause, we use $\top$ to indicate an empty right-hand side.

$$\mathrm{append}(\mathit{elist}, X, X) := \top$$
$$\mathrm{append}(\mathit{hd}{:}\, H \wedge \mathit{tl}{:}\, T, X, \mathit{hd}{:}\, H \wedge \mathit{tl}{:}\, R) := \mathrm{append}(T, X, R)$$

Definition 6.7
Let $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{P} \rangle$ be an interpretation and C be a set of definite clauses. $\mathcal{I}$ models C just in case for each $r \in \mathcal{R}$, for all $u_1, \ldots, u_n \in \mathcal{U}$:

$\langle u_1, \ldots, u_n \rangle \in \mathcal{P}(r)$ *iff*
$\exists \alpha.\ \exists (r(\phi_1, \ldots, \phi_n) := \phi_0) \in C.\ \exists u_0 \in \mathcal{U}.\ u_0 \in [\![\phi_0]\!]^{\mathcal{I}}_{\alpha} \wedge \ldots \wedge u_n \in [\![\phi_n]\!]^{\mathcal{I}}_{\alpha}$

Notice how quantification works in this definition. All variables are existentially quantified (via the existential quantification of the variable assignment α). The n-tuples of objects, on the other hand, are universally quantified over. This gives us the same effect as Clark's completion semantics in logic programming (Clark 1978). We illustrate this with an example. Suppose we have the following logic program:

$\forall L.\ \mathrm{append}([], L, L).$

$\forall H, L, T1, T2.\ \mathrm{append}([H|T1], L, [H|T2]) \ \leftarrow\ \mathrm{append}(T1, L, T2).$

Then the completed program looks as follows:

$$
\begin{array}{l}
\forall X, Y, Z.\ \mathrm{append}(X, Y, Z) \ \leftrightarrow \\
\quad (\exists L.\ X = [] \ \wedge\ Y = L \ \wedge\ Z = L) \ \vee \\
\quad (\exists H, L, T1, T2.\ X = [H|T1] \ \wedge \\
\qquad\quad Y = L \ \wedge\ Z = [H|T2] \ \wedge\ \mathrm{append}(T1, L, T2))
\end{array}
$$

Together with an appropriate theory of equality, the completed program has stronger properties than the original program (Lloyd 1984). First of all, the original program is a logical consequence of its completion. The original motivation of Clark for this definition was the fact that more negative facts follow from a completed program. We have a similar motivation in that we want to exclude unwanted solutions in our definition of prediction. Notice that we only demand that there is *some* model of a grammar that satisfies a goal, whereas in logic programming, *every* model of a program must satisfy a query. This is too strong a requirement for our purposes, it wouldn't work with the rest of our framework. So instead we modify the meaning of definite clauses. Speaking in fixpoint terminology, all and only fixpoints are models for completed programs, which is exactly what we want. Notice that this is still different from standard logic programming, where only the least fixpoint is the intended model. However, since we're dealing with structures that can be cyclic or infinite (as opposed to Herbrand terms), a greatest fixpoint semantics may be more appropriate.

As an example, suppose we would like to be able to say that all members of a list have a certain property p. We might encode this in a relation all_p.

$$
\begin{array}{r}
\mathrm{all\_p}(\mathit{elist}) := \top \\
\mathrm{all\_p}(\mathit{hd}{:}X \wedge \mathit{tl}{:}Y) := \mathrm{p}(X) \wedge \mathrm{all\_p}(Y)
\end{array}
$$

Since we're using a greatest fixpoint approach, all_p may be true even for infinite and cyclic lists. To see this, consider the example interpretation in fig. 6.1. Objects 1 and 2 form a cyclic list, with nodes 3 and 4 as members. Objects 3 and 4 are of some unspecified type a. Now assume that objects 3 and 4 are in the p relation, and 1 and 2 are in the all_p relation. It is easy to verify that this interpretation is a model for the all_p clauses defined above.

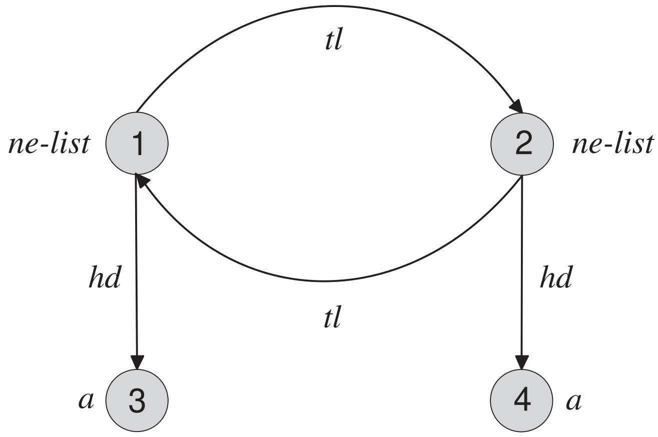


Figure 6.1: Example interpretation for all_p relation

However, this example would not work if we took a least fixpoint approach to the interpretation of definite clauses.

Notice that we consider definite clauses to be something different from our standard grammar constraints. The reason why this must be so should be clear from the semantics of definite clauses: there is some implicit universal quantification going on there.

We will now give a new definition of implicational grammar constraints. The purpose of this definition is to allow relational goals to be called in our standard constraints, but to disallow them in the scope of negation.

Definition 6.8 (implicational constraints)
$\phi \rightarrow \psi$ is an implicational constraint iff ϕ is a feature term and ψ is a relation term.

Definition 6.9 (grammar)
A grammar is a triple $\langle \Sigma, R, C \rangle$ s.t. $\Sigma = \langle \mathcal{T}, \preceq, \mathcal{F}, approp, \mathcal{R}, Ar \rangle$ is a signature, R is a finite set of implicational constraints and C is a finite set of definite clauses.

Definition 6.10 (model)
We define a model of a grammar $\langle \Sigma, R, C \rangle$ to be an interpretation $\mathcal{I}$ s.t. $\mathcal{I}$ models C and for every $u \in \mathcal{U}$ and for every $T \in R$, $u \in [\![T]\!]^{\mathcal{I}}$.

The definition of prediction remains unchanged.

6.2 Some properties of definite clauses

At this point, it is useful to take a step back and review the reasons for some of the decisions in this chapter. The intuitions for the use of relations in the HPSG literature seem to derive from logic programming. Why, then, do our definitions look so different from logic programming? Why can't we just add a definite clause component in a more straightforward manner?

Recall that our interest in definite clause theories in first order logic stems from the fact that there is an effective procedure to list the logical consequences of such theories. More precisely, only those consequences that are existential conjunctive formulae, so-called queries, are listed. The important point to note is that it's logical consequences we can compute. Satisfiability problems for definite clause programs are completely pointless. To see this, note that the full Herbrand interpretation is a model for any definite clause program. Thus, if we have some definite clause program P and some formula ϕ, then $P \wedge \phi$ is satisfiable iff ϕ is satisfiable. However, we saw that this is changed if we take the Clark completion of logic programs. The Clark completion severely restricts the possible models of definite clause programs, so that it makes in fact sense to consider satisfiability problems. The reason is that the completion strengthens the weak implications of definite clause programs to much stronger biimplications. But even with the Clark completion, we can only approximate logic programming, due to our satisfiability-based approach to prediction.

Another way in which first order logic programming differs from our case is the difference in the basic logic: equations between first order terms, on the one hand, and feature constraints on the other. Let us illustrate what we mean by an example. Consider the logic program P

$$\forall X.\ p(X)$$

P has as logical consequences, among other things, $p(a)$ (or equivalently, $\exists X.\ X = a \wedge p(X)$) and $p(f(a))$ ($\exists Y.\ Y = f(a) \wedge p(Y)$). This so because objects X and Y s.t. $X = a$ and $Y = f(a)$ *must* exist in every interpretation of the corresponding signature, by the definition of first order structures, and are thus contained in the p relation. Now suppose for a moment we weren't interested in implicational constraints, but only in logic programming over typed feature terms. Considering again the above program, are the expressions $\exists X.\ X|\,a \ \wedge \ p(X)$ and $\exists Y, Z.\ Y|\,f\colon Z \ \wedge \ Z|\,a \ \wedge \ p(Y)$ logical consequences of P? The answer is clearly "no", since the existence of objects X, Y and Z is not guaranteed. This is precisely the reason why in the CLP scheme of (Höhfeld and Smolka 1988) the central computational problem is not defined as one of determining logical consequence. Rather, they describe an algorithm that, given a query ψ, will produce a satisfiable solution constraint ϕ s.t. $\phi \to \psi$ is a logical consequence of the program. Note the difference to logic programming. There, we prove that the existential closure of a query is a logical consequence of a program, the existence of appropriate objects is guaranteed in all models. In CLP, we prove that the query is true in all models that contain the required objects, described by the solution ϕ. If a model doesn't contain the necessary objects, then ϕ will be false and thus, $\phi \to \psi$ is true.

After this brief comparison with logic programming, we now return to the practical problem of computing with the extended formalism we have defined. However, the issues discussed here should be kept in mind, as we will return to consider them towards the end of this chapter.

6.3 Translating feature terms with relations

We now have to go through the steps of chapter 2 again, this time with the addition of relation symbols: we need to define what constraints are, how to get from terms to constraints and how to check satisfiability. Fortunately, relations do not add much complexity here.

The first thing we need to worry about is how to get feature terms with relations into disjunctive normal form. Since we don't allow relational atoms in the scope of negation anywhere, the only thing we need to worry about is disjunction. We note the following.

Proposition 6.1
Let r be a relation symbol of arity n. Then

$$\begin{aligned} r(\phi_1, \ldots, (\phi_{i_1} \vee \phi_{i_2}), \ldots, \phi_n) \ \equiv\ & r(\phi_1, \ldots, \phi_{i_1}, \ldots, \phi_n) \vee \\ & r(\phi_1, \ldots, \phi_{i_2}, \ldots, \phi_n) \end{aligned}$$

With this additional equivalence, we can bring terms with relations into DNF.

We now turn to feature constraints.

Definition 6.11 (feature constraints)
Let r be a relation symbol of arity n and $X_1, \ldots, X_n$ be variables. Then $r(X_1, \ldots, X_n)$ is a feature constraint.

The only difference between relation terms and constraints is that the arguments of constraints are only variables.

Definition 6.12
Let $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{P} \rangle$ be an interpretation.

$$\mathcal{I}, \alpha \models r(X_1, \ldots, X_n) \textit{ iff } \langle \alpha(X_1), \ldots, \alpha(X_n) \rangle \in \mathcal{P}(r)$$

The normal form for feature clauses remains completely unchanged. Since we don't have to deal with negative literals, adding relational constraints can not make a clause inconsistent.[1] Trivially, one can assign $\mathcal{P}(r) = \mathcal{U}^n$ to every relation symbol r of arity n to get a satisfier (provided that the non-relational part is satisfiable).

We now extend the procedure **trans** that translates terms into constraints to also deal with relational atoms.

Definition 6.13 (trans)
$\mathbf{trans}(X, r(\phi_1, \ldots, \phi_n)) =$
$\quad \{r(X_1, \ldots, X_n)\} \cup \mathbf{trans}(X_1, \phi_1) \cup \ldots \cup \mathbf{trans}(X_n, \phi_n),$
where $X_1, \ldots, X_n$ are new variables.

The second argument of **trans** plays no role for relational atoms. This is simply due to the fact that where an atom occurs in a term is completely irrelevant.

Proposition 6.2 (correctness of trans)
Let $D = D_1 \vee \ldots \vee D_n$ be in DNF. For each D_i, $1 \leq i \leq n$, for each interpretation $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{P} \rangle$, if $X \notin \mathrm{FV}(D)$ then

$$[\![D_i]\!]^{\mathcal{I}} = \{\alpha(X) \mid \mathcal{I}, \alpha \models \mathbf{trans}(X, D_i)\}$$

[1] This would be different had we defined a typing scheme on arguments of relations, similar to the appropriateness conditions for types. That would be desirable in a real system, but we'll leave it out for reasons of simplicity.

Proof

Recall that in the proof of prop. 2.11, we showed two directions. The proof here is exactly identical, except for the added cases for relational atoms. We will therefore only show these, and not repeat all the other cases.

$\supseteq$: if $\mathcal{I}, \alpha \models \mathbf{trans}(X, D_i)$, then $\alpha(X) \in [\![D_i]\!]^{\mathcal{I}}$. Proof by induction on the structure of D_i.

- $\mathcal{I}, \alpha \models \mathbf{trans}(X, r(\phi_1, \ldots, \phi_n))$

  $\Rightarrow\ \mathcal{I}, \alpha \models \{r(X_1, \ldots, X_n)\} \cup \mathbf{trans}(X_1, \phi_1) \cup \ldots \cup \mathbf{trans}(X_n, \phi_n)$
  (by definition of **trans**)

  $\Rightarrow\ \langle \alpha(X_1), \ldots, \alpha(X_n) \rangle \in \mathcal{P}(r),\ \alpha(X_1) \in [\![\phi_1]\!]^{\mathcal{I}}_{\alpha},\ \ldots,\ \alpha(X_n) \in [\![\phi_n]\!]^{\mathcal{I}}_{\alpha}$
  (by induction)

  $\Rightarrow\ \alpha(X) \in [\![r(\phi_1, \ldots, \phi_n)]\!]^{\mathcal{I}}_{\alpha}$

Since $\alpha(X) \in [\![D_i]\!]^{\mathcal{I}}_{\alpha}$, it follows that $\alpha(X) \in [\![D_i]\!]^{\mathcal{I}}$.

$\subseteq$: for each $u \in [\![D_i]\!]^{\mathcal{I}}$ there is an α s.t. $\alpha(X) = u$ and $\mathcal{I}, \alpha \models \mathbf{trans}(X, D_i)$.

We prove a stronger result by induction: if $u \in [\![D_i]\!]^{\mathcal{I}}_{\alpha}$ where $\alpha(X) = u$, then there is an α' s.t. $\alpha'(X) = u$, for each $Y \in \mathrm{FV}(D)$, $\alpha'(Y) = \alpha(Y)$ and $\mathcal{I}, \alpha' \models \mathbf{trans}(X, D_i)$.

- $u \in [\![r(\phi_1, \ldots, \phi_n)]\!]^{\mathcal{I}}_{\alpha}$

  $\Rightarrow\ \exists u_1, \ldots, u_n.\ u_1 \in [\![\phi_1]\!]^{\mathcal{I}}_{\alpha}, \ldots, u_n \in [\![\phi_n]\!]^{\mathcal{I}}_{\alpha}$ and $\langle u_1, \ldots, u_n \rangle \in \mathcal{P}(r)$

  $\Rightarrow\ \exists \alpha_1, \ldots, \alpha_n.$ for each j, $1 \leq j \leq n$, $\alpha(X_j) = u_j$, for each $Y \in \mathrm{FV}(\phi_j)$,
  $\alpha(Y) = \alpha_j(Y)$ and $\mathcal{I}, \alpha_j \models \mathbf{trans}(X_j, \phi_j)$ (by induction)

  $\Rightarrow$ if $\alpha'(X) = u$, $\alpha'(Y) = \alpha_j(Y)$ for each $Y \in (\mathrm{FV}(trans(X_j, \phi_j)) \setminus \mathrm{FV}(D_i))$ and
  $\alpha'(Y') = \alpha(Y')$ elswhere,
  then $\mathcal{I}, \alpha' \models \mathbf{trans}(X, r(\phi_1, \ldots, \phi_n))$
  (for all $1 \leq k < l \leq n$, $(\mathrm{FV}(\mathbf{trans}(X_k, \phi_k)) \cap \mathrm{FV}(\mathbf{trans}(X_l, \phi_l))) \setminus \mathrm{FV}(D_i) = \emptyset$)

  $\Rightarrow\ \exists \alpha'.\ \alpha'(X) = u,\ \forall Y \in \mathrm{FV}(D_i).\ \alpha'(Y) = \alpha(Y)$ and
  $\mathcal{I}, \alpha' \models \mathbf{trans}(X, r(\phi_1, \ldots, \phi_n))$

Since $X \notin \mathrm{FV}(D_i)$, $u \in [\![D_i]\!]^{\mathcal{I}}_{\alpha[X \mapsto u]}$ and thus, by the induction above, α' exists with $\alpha'(X) = u$ and $\mathcal{I}, \alpha' \models \mathbf{trans}(X, D_i)$.

$\blacksquare$

6.4 Compiling grammars

Adding relations and definite clauses as we've done also necessitates some changes in the compilation of grammars and the procedural semantics discussed in section 5.1. Recall that resolution meant adding constraints from the grammar to a given clause. Where to add constraints was determined by a set of variables encoding where constraints from the grammar still needed to be applied. Now that we've enriched our notion of grammar by adding definite relations, we also need to make sure that the constraints placed on a relation by its definition are satisfied. We will have two different grammar items: implicational items (the ones we already know) and relational items. Common to both is the fact that what used to be a set of variables is now a set of variables and relational constraints.

Definition 6.14 (grammar item)
Let $\mathcal{G} = \langle S, R, C \rangle$ be a grammar s.t. $\langle S, R \rangle$ is in normal form. Grammar items in $\mathcal{P}(\mathcal{G})$ are triples defined as follows:

1. *For each $t \in \mathcal{V}$ s.t. $t \to \Psi \in R$, for each clause $\Sigma \in \mathrm{NF}(\mathbf{trans}(X, \Psi))$, define*
   - $V := \{Y \in \mathrm{FV}(\Sigma) \mid Y \neq X, Y|a \in \Sigma,$ *a is constrained and* $\exists Z, f.\ Y|f{:}Z \in \Sigma\}$
   - $P := \{A \mid A$ *is a relational atom in* $\Sigma\}$

   $\langle X,\ \Sigma \setminus P,\ V \cup P \rangle$ *is in* $\mathcal{P}(\mathcal{G})$.
2. *For each $r(\phi_1, \ldots, \phi_n) := \phi_0 \in C$, let $X_0, \ldots X_n$ be new variables. For each*
   $\Sigma \in \mathrm{NF}(\mathbf{trans}(X_0, \phi_0) \cup \ldots \cup \mathbf{trans}(X_n, \phi_n))$, *define*
   - $V := \{Y \in \mathrm{FV}(\Sigma) \mid Y|a \in \Sigma, a$ *constrained and* $\exists Z, f.\ Y|f{:}Z \in \Sigma\}$
   - $P := \{A \mid A$ *is a relational atom in* $\Sigma\}$

   $\langle r(X_0, \ldots, X_n),\ \Sigma \setminus P,\ V \cup P \rangle$ *is in* $\mathcal{P}(\mathcal{G})$.
3. *Nothing else is in $\mathcal{P}(\mathcal{G})$.*

The items for definite clauses look rather familiar from constraint logic programming. We have the clause head, a relational atom (first element), and the clause body (third element). The second element is a set of constraints on the variables occurring in the head and the body. What is unfamiliar is that in the body, we may also have a bunch of variables from the constraint. This is so because in the constraint, we may be talking about objects that need to satisfy implicational constraints from the grammar. The items for implicational constraints look rather similar, except that they don't have a clause head, but a root variable.

Since we've changed $\mathcal{P}(\mathcal{G})$, we now also need to augment the definition of a derivation. As before, goal variables are "cancelled" by application of an implicational grammar item. Additionally, we now also have to resolve relational calls against their grammar definitions.

Definition 6.15 (one step derivation)
$\langle \Sigma, S \rangle \mapsto \langle \Sigma', S' \rangle$, *if the selected element* $Y \in S$ *is a variable,* $\langle X,\ \Sigma'',\ S'' \rangle \in \mathcal{P}(\mathcal{G})$, $\Sigma' = \mathrm{NF}(\Sigma \cup \Sigma'' \cup \{Y | X\})$ *is consistent and* $S' = (S \cup S'') \setminus \{Y\}$.

$\langle \Sigma, S \rangle \mapsto \langle \Sigma', S \setminus \{Y\} \rangle$, *if the selected element* $Y \in S$ *is a variable,* $Y | a \in \Sigma$, $b \in \mathbf{gst}(a)$ *and* $\Sigma' = (\Sigma \setminus \{Y | a\}) \cup \{Y | b\}$.

$\langle \Sigma, S \rangle \mapsto \langle \Sigma', S' \rangle$, *if the selected element* $r(X_0, \ldots, X_n) \in S$ *is a relational atom,*
$\langle r(Y_0, \ldots, Y_N),\ \Sigma'',\ S'' \rangle \in \mathcal{P}(\mathcal{G})$, $\Sigma' = (\Sigma \cup \Sigma'' \cup \{X_0 | Y_0, \ldots, X_n | Y_n\})$ *is consistent and* $S'' = (S \cup S'') \setminus \{r(X_0, \ldots, X_n)\}$.

Definition 6.16 (goal)
Let Σ *be a feature clause in normal form. Define*

- $V := \{Y \in \mathrm{FV}(\Sigma) \mid Y | a \in \Sigma, a$ *constrained and* $\exists Z, f.\ Y | f{:}\, Z \in \Sigma\}$
- $P := \{A \mid A$ *is a relational atom in* $\Sigma\}$

Then $goal(\Sigma) = \langle \Sigma \setminus P,\ V \cup P \rangle$ *is a goal.*

6.5 Correctness of resolution

To obtain a correctness result for our new resolution procedure, we will employ essentially the same methods as in Ch. 5. However, the presence of relations complicates matters. We begin by considering the models for the relational part of a theory. Above, we stated without proof that all and only the fixpoints of a given base interpretation are models for the relational theory. For soundness, it will be sufficient to consider the greatest fixpoint. We will show that it always exists, and that it always produces a model.

First of all, though, we need to make precise what we are considering fixpoints of. We need a notion analogous to the T_P operator in logic programming.

For the rest of this section, unless otherwise stated, we assume an implicit signature.

Definition 6.17 (T_C)
Let C be a set of clauses. Define the operator T_C on interpretations $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{P} \rangle$ as

$$T_C(\mathcal{I}) = \mathcal{I}',$$

where $\mathcal{I}' = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{P}' \rangle$ s.t. for each n-place relation r,

$$\langle u_1, \ldots, u_n \rangle \in \mathcal{P}'(r) \text{ iff}$$

$$\exists \alpha.\ \exists (r(\phi_1, \ldots, \phi_n) := \phi_0) \in C.\ \exists u_0 \in \mathcal{U}.\ u_0 \in [\![\phi_0]\!]^{\mathcal{I}}_{\alpha} \wedge \ldots \wedge u_n \in [\![\phi_n]\!]^{\mathcal{I}}_{\alpha}$$

To be able to obtain standard fixpoint results, we need to show that the T_C operator is monotone. This is clearly the case, and we give the following proposition merely for the sake of completeness. We write $\mathcal{I} \subseteq \mathcal{I}'$ if $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{P} \rangle, \mathcal{I}' = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{P}' \rangle$ and for each relation symbol r, $\mathcal{P}(r) \subseteq \mathcal{P}'(r)$.

Proposition 6.3 (monotonicity of T_C)
Let C be a set of clauses. Then T_C is monotone, i.e., if $\mathcal{I} \subseteq \mathcal{I}'$, then $T_C(\mathcal{I}) \subseteq T_C(\mathcal{I}')$.

Proof Let $T_C(\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{Q} \rangle), T_C(\mathcal{I}' = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{Q}' \rangle)$ and $\langle u_1, \ldots, u_n \rangle \in \mathcal{Q}(r)$. Then, $\exists \alpha.\ \exists (r(\phi_1, \ldots, \phi_n) := \phi_0) \in C.\ \exists u_0 \in \mathcal{U}.\ u_0 \in [\![\phi_0]\!]^{\mathcal{I}}_{\alpha} \wedge \ldots \wedge u_n \in [\![\phi_n]\!]^{\mathcal{I}}_{\alpha}$. Since $\mathcal{I} \subseteq \mathcal{I}'$, $\langle u_1, \ldots, u_n \rangle \in \mathcal{Q}'(r)$. ∎

Since the space of interpretations with a fixed domain, type assignment and feature interpretation function forms a complete lattice under $\subseteq$, T_C has a greatest fixpoint (e.g., Lloyd (1984)). We fix our previous claim that all and only fixpoints are models for sets of clauses in a proposition.

Proposition 6.4
Let C be a set of clauses and $\mathcal{I}$ an interpretation. $\mathcal{I}$ models C iff $T_C(\mathcal{I}) = \mathcal{I}$.

Proof

Follows directly from def. 6.7 and def. 6.17.

■

For ease of notation, we define a function T_C, mapping interpretations and ordinal numbers to interpretations.

Definition 6.18
Let $\mathcal{I}$ be an interpretation, and C a set of clauses. Define

$$\begin{aligned} T_C(\mathcal{I}, 0) &= \mathcal{I} \\ T_C(\mathcal{I}, n+1) &= T_C(T_C(\mathcal{I}, n)) \\ T_C(\mathcal{I}, n) &= \bigcup_{n' < n} T_C(\mathcal{I}, n') \quad \textit{for } n \textit{ a limit ordinal} \end{aligned}$$

The following lemma will be useful in the soundness proof.

Definition 6.19
Let $\mathcal{G} = \langle S, R, C \rangle$ be a grammar and $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{P} \rangle$ an interpretation. If for each relation r of arity n, for each $\langle u_1, \ldots, u_n \rangle \in \mathcal{P}_r$ there is a clause $r(\phi_1, \ldots, \phi_n) := \phi_0 \in C$ and an $\mathcal{I}$-assignment α s.t. $u_1 \in [\![\phi_1]\!]^{\mathcal{I}}_{\alpha}$, ..., $u_n \in [\![\phi_n]\!]^{\mathcal{I}}_{\alpha}$ and $[\![\phi_0]\!]^{\mathcal{I}}_{\alpha} \neq \emptyset$, then we say that $\mathcal{I}$ is downward closed with respect to C.

Proposition 6.5
Let $\mathcal{G} = \langle S, R, C \rangle$ be a grammar and $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{P} \rangle$ an interpretation. If $\mathcal{I}$ is downward closed with respect to C, then $\mathcal{I} \subseteq gfp(T_C)$.

Proof

We will show that

- $T_C(\mathcal{I}, n)$ is monotone, and
- for each n, $T_C(\mathcal{I}, n)$ is downward closed.

$n = 0$: Obvious, since $T_C(\mathcal{I}, 0) = \mathcal{I}$.

n is a successor ordinal:

- Since $T_C(\mathcal{I}, n-1)$ is downward closed, $T_C(\mathcal{I}, n-1) \subseteq T_C(\mathcal{I}, n)$, and
- since $T_C(\mathcal{I}, n-1)$ is downward closed, so is $T_C(\mathcal{I}, n)$.

n is a limit ordinal:

- Obviously, $T_C(\mathcal{I}, n) \supseteq T_C(\mathcal{I}, n')$, for each $n' < n$.
- Since for each $n' < n$, $T_C(\mathcal{I}, n')$ is downward closed, so is $T_C(\mathcal{I}, n)$.

Since $T_C(\mathcal{I}, n)$ is monotone, it follows that $\mathcal{I} \subseteq gfp(T_C)$.

■

6.5.1 Success derivations

We can now turn to the soundness proof. The idea is as follows. Just as in the case without relations, we show that we can build a model from a successful derivation. The only difference is that this time, we need to make sure that the resulting interpretation models the definite clause part of the grammar.

We begin with a proposition similar to prop. 5.4. Since we still assume that our grammars are type consistent, we know that models with at least one object of each type exist. Note also that if we take two pairwise distinct models of a given grammar, then the union (in the obvious sense of union) of those two models is again a model of the grammar.

Proposition 6.6
Let $S = \langle \mathcal{T}, \preceq, \mathcal{F}, approp, \mathcal{R}, Ar \rangle$ be a signature, $\mathcal{G} = \langle S, R, C \rangle$ a type consistent grammar and Σ a consistent set of constraints. Let $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{P} \rangle$ be an S-interpretation and α an $\mathcal{I}$-assignment s.t. $\mathcal{I}, \alpha \models \Sigma$. Since Σ is consistent, such an interpretation is guaranteed to exist. If

- *for each $X \in \mathrm{BV}(\Sigma)$ s.t. $\mathcal{S}(\alpha(X))$ is constrained, there is a $\langle X', \Sigma', S \rangle \in \mathcal{P}(\mathcal{G})$ s.t. $\Sigma \models \Sigma' \cup \{ X | X' \}$, and*
- *for each $r(X_1, \ldots, X_n) \in \Sigma$ there is a $\langle r(Y_1, \ldots, Y_N), \Sigma', P \cup V \rangle \in \mathcal{P}(\mathcal{G})$ such that $\Sigma \models \Sigma' \cup P \cup \{X_1 | Y_1, \ldots, X_n | Y_n\}$,*

then $\mathcal{G}$ predicts Σ.

Proof

The proof proceeds in several steps. First, we construct a new interpretation $\mathcal{I}'$ that models the implicational constraints. Second, we construct another interpretation $\mathcal{I}''$ that also models the set of clauses. Finally, we show that Σ is satisfiable in $\mathcal{I}''$.

For each $a \in \mathcal{V}$, $[X]_\alpha \in \mathrm{FV}(\Sigma)$, let $\mathcal{I}_a^{[X]_\alpha} = \langle \mathcal{U}_a^{[X]_\alpha}, \mathcal{S}_a^{[X]_\alpha}, \mathcal{A}_a^{[X]_\alpha} \rangle$ be a copy of $\mathcal{I}_a$. Define a new interpretation $\mathcal{I}' = \langle \mathcal{U}', \mathcal{S}', \mathcal{A}', \mathcal{P}' \rangle$ and assignment α' s.t.

$$\begin{aligned}\mathcal{U}' = \ & \{u \in \mathcal{U} \mid \exists X, Y \in \mathrm{FV}(\Sigma).\ \exists f \in \mathcal{F}.\ \alpha(X) = u,\ \mathcal{A}(f)(u)\!\downarrow \text{ and} \\ & \mathcal{A}(f)(u) = \alpha(Y)\} \cup \bigcup\{\mathcal{U}_a \mid a \in \mathcal{V}\} \ \cup \ \bigcup\{\mathcal{U}_a^X \mid a \in \mathcal{V} \text{ and} \\ & [X]_\alpha \in \Sigma\}\end{aligned}$$

$$\mathcal{S}'(u) = \begin{cases} \mathcal{S}(u) & \text{if } u \in \mathcal{U} \\ \mathcal{S}_a^{[X]_\alpha}(u) & \text{if } u \in \mathcal{U}_a^{[X]_\alpha} \\ \mathcal{S}_a(u) & \text{if } u \in \mathcal{U}_a \end{cases}$$

$$\alpha'(X) = \begin{cases} \alpha(X) & \text{if } X \in \mathrm{FV}(\Sigma) \text{ and } \exists Y \in \mathrm{FV}(\Sigma).\ \exists f \in \mathcal{F}. \\ & \mathcal{A}(f)(\alpha(X))\!\downarrow \text{ and } \mathcal{A}(f)(\alpha(X)) = \alpha(Y) \\ u(\mathcal{I}_{\mathcal{S}(\alpha(X))}^{[X]_\alpha}) & \text{if } X \in \mathrm{FV}(\Sigma) \text{ and } \neg(\exists Y \in \mathrm{FV}(\Sigma).\ \exists f \in \mathcal{F}. \\ & \mathcal{A}(f)(\alpha(X))\!\downarrow \text{ and } \mathcal{A}(f)(\alpha(X)) = \alpha(Y)) \\ u(\mathcal{I}_a) & \text{otherwise, where } a \text{ is the lex. smallest minimal} \\ & \text{type} \end{cases}$$

$$\mathcal{A}'(f)(u) = \begin{cases} u' & \text{if } \exists X, Y \in \mathrm{FV}(\Sigma).\ \alpha(X) = u,\ \mathcal{A}(f)(u)\!\downarrow, \\ & \mathcal{A}(f)(u) = \alpha(Y) \text{ and } \alpha'(Y) = u' \\ u_{\mathcal{S}(\mathcal{A}(f)(u))} & \text{if } \exists X \in \mathrm{FV}(\Sigma).\ \alpha(X) = u, \mathcal{A}(f)(u)\!\downarrow \text{ and} \\ & \neg(\exists Y \in \mathrm{FV}(\Sigma).\ \alpha(Y) = \mathcal{A}(f)(u)) \\ \mathcal{A}_a(f)(u) & \text{if } u \in \mathcal{U}_a \text{ and } \mathcal{A}_a(f)(u)\!\downarrow \\ \mathcal{A}_a^{[X]_\alpha}(f)(u) & \text{if } u \in \mathcal{U}_a^{[X]_\alpha} \text{ and } \mathcal{A}_a^{[X]_\alpha}(f)(u)\!\downarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\begin{aligned}\mathcal{P}'(r) \ = \ & \{\langle u_1', \ldots, u_n' \rangle \mid \langle u_1, \ldots, u_n \rangle \in \mathcal{P}(r), \forall i, 1 \le i \le n. \exists X \in \mathrm{FV}(\Sigma) \textit{ s.t.} \\ & \qquad \alpha(X) = u_i, \textit{ and either } X \in \mathrm{BV}(\Sigma) \wedge u_i' = u_i \textit{ or} \\ & \qquad X \notin \mathrm{BV}(\Sigma) \wedge u_i' = u(\mathcal{I}_{\mathcal{S}(\alpha(X))}^{[X]_\alpha})\} \\ & \cup \bigcup_{a \in \mathcal{V}} \mathcal{P}_a(r) \ \cup \bigcup_{X \in \mathrm{FV}(\Sigma), a \in \mathcal{V}} \mathcal{P}_a^{[X]_\alpha}\end{aligned}$$

Clearly, the proof that this is indeed an S-interpretation is similar to the one for prop. 5.4. The added case of the relations is trivially true. Similarly, given prop. 5.4, it is easy to see that $\mathcal{I}', \alpha' \models \Sigma$. Now let $\mathcal{I}'' = gfp(T_C)$.

Since $\mathcal{I}'$ is downward closed, we know by prop. 6.5 that $\mathcal{I}' \subseteq \mathcal{I}''$. Thus, $\mathcal{I}'', \alpha' \models \Sigma$.

It remains to be shown that $\mathcal{I}''$ is a model of $\mathcal{G}$. Again, the non-relational part is the same as for prop. 5.4. The relational part is trivial, since $\mathcal{I}'' = \mathit{gfp}(T_C)$. Therefore we've shown that $\mathcal{G}$ predicts Σ.

■

We can use this proposition to show that successful derivations, finite or infinite, are sound. For infinite derivations, we need the additional proviso that the derivation be fair. All finite successful derivations are fair by definition. We give the proof below independently of the finiteness of the derivation.

Theorem 6.7 (Correctness of success derivations)
Let $\mathcal{G}$ be a grammar and ϕ a feature term. If there exists a successful fair derivation $D = \langle I_0, \ldots \rangle$ with respect to $\mathcal{P}(\mathcal{G})$ s.t. $I_0 = \mathit{goal}(\mathbf{trans}(X, \phi))$, then $\mathcal{G}$ predicts ϕ.

Proof

Let $\Sigma = \bigcup_{\langle \Sigma_i, \Gamma_i \rangle \in D} \Sigma_i \cup (\Gamma_i \setminus \mathrm{FV}(\Sigma_i))$. In addition to what we proved for prop. 5.9, we need to show that for each $r(X_1, \ldots, X_n) \in \Sigma$ there is a $\langle r(Y_1, \ldots, Y_N), \Sigma', P \cup V \rangle \in \mathcal{P}(\mathcal{G})$ s.t. $\Sigma \models \Sigma' \cup P \cup \{X_1 | Y_1, \ldots, X_n | Y_n\}$. Since D is fair, there is an i s.t. $r(X_1, \ldots, X_n) \in \Gamma_i$ is selected. Thus, there is a $\langle r(Y_1, \ldots, Y_N), \Sigma', P \cup V \rangle \in \mathcal{P}(\mathcal{G})$ such that $\Sigma_i \models \Sigma' \cup \{X_1 | Y_1, \ldots, X_n | Y_n\}$. Again, since D is fair, for each $G \in P$ there is an n s.t. G is selected after n steps. Let j be the largest such n. Then $\Sigma_{i+j+1} \models \Sigma' \cup \{X_1 | Y_1, \ldots, X_n | Y_n\} \cup P$. Now we can apply prop. 6.6, and together with the correctness of **trans** (prop. 6.2), this completes the proof.

■

6.5.2 Failed derivations

Again, we will parallel the results of ch. 5. We start out by showing that if at a given state in a derivation, we know that a model for this state exists, then this can not be the a failure state. The addition of relations to our framework does not add much complexity here. In particular, it is sufficient to consider a given item in a derivation, although we throw away relational atoms that we've already computed.

Proposition 6.8
Let $\mathcal{G}$ be a grammar and $\langle \Sigma, S \rangle$ a derivation item. If $\mathcal{G}$ predicts $\Sigma \cup (S \setminus \mathrm{FV}(\Sigma))$, then there are Σ', S' s.t. $\langle \Sigma, S \rangle \overset{1}{\mapsto} \langle \Sigma', S' \rangle$ and $\mathcal{G}$ predicts $\Sigma' \cup (S' \setminus \mathrm{FV}(\Sigma'))$.

Proof

Suppose $X \in \mathrm{FV}(\Sigma)$ is the selected item in S. Then the proof is the same as in prop. 5.7.

Now suppose the selected item in S is a relational atom $r(X_1, \ldots, X_n)$. Suppose further that $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{P} \rangle$ is a model of $\mathcal{G}$ and $\mathcal{I}, \alpha \models \Sigma$. Thus, a $\langle r(X_1, \ldots, X_N), \Sigma'', P \cup V \rangle \in \mathcal{P}(\mathcal{G})$ exists such that for some α', $\mathcal{I}, \alpha' \models \Sigma' \cup ((S \setminus \mathrm{FV}(\Sigma)) \setminus r(X_1, \ldots, X_N)) \cup P$. Thus, Σ' is consistent, $\mathcal{G}$ predicts Σ' and $\langle \Sigma, S \rangle \overset{1}{\mapsto} \langle \Sigma', S' \rangle$, where $S' = ((S \setminus \mathrm{FV}(\Sigma)) \setminus r(X_1, \ldots, X_N)) \cup P$.

■

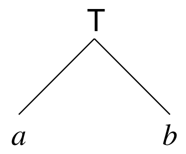
6.5.3 Conclusion

We have seen that by adding a "conservative" version of relational extension, we can obtain the same soundness and completeness result as in ch. 5. By conservative, I mean that the semantics of the relational part is a straightforward extension of the non-relational part. Our greatest fixpoint approach may have some non-intuitive consequences from a logic programming point of view, but fits better with our general approach. For finite, acyclic structures, it all boils down to the same, anyway.

A restriction that I imposed on grammars was that I did not allow for relational atoms in the scope of negation. I did this to be able to obtain a sound and complete procedural semantics. There is abundant literature to show that the computational properties of logic programs deteriorate very quickly when negation is added. However, it is interesting from a purely theoretical perspective what kind of semantics one would need to employ to be able to lift the restriction on negation. We will do just that for the rest of this chapter. However, I will make no attempt to provide a procedural semantics, since this attempt could only be fragmentary.

6.6 Adding full negation

It is well known that the Clark completion does not allow for as many negative inferences as one would wish, particularly for recursive relations. Although we don't allow relational atoms in the scope of negation, we're concerned with this problem due to our approach to prediction. We will illustrate the problem with a simple example adapted from (Wallace 1993). For an introduction to the use of negation in logic programming and problems with it, see (Shepherdson 1987) and (Kunen 1987).



Suppose we have the type hierarchy shown above with only three types and no features. We consider a grammar G with no implicational constraints and the following clauses.

$$p(a, a)$$

$$tp(X, Y) := p(X, Y)$$

$$tp(X, Z) := p(X, Y) \wedge tp(Y, Z)$$

The relation tp is supposed to encode the transitive closure of p. However, we get the following, somewhat surprising model: $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{P} \rangle$, where $\mathcal{U} = \{a', b'\}$, $\mathcal{S}(a') = a$, $\mathcal{S}(b') = b$, $\mathcal{P}(p) = \{\langle a', a' \rangle\}$ and $\mathcal{P}(tp) = \{\langle a', a' \rangle, \langle a', b' \rangle\}$. This means that G predicts $tp(a, b)$, which is certainly not in the transitive closure of p. What our intuitions about logic programming tell us is that the model that we're interested in is $\mathcal{I}' = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{P}' \rangle$, which is exactly like $\mathcal{I}$ except that $\mathcal{P}'(tp) = \{\langle a', a' \rangle\}$. In $\mathcal{I}'$, tp does in fact encode the (trivial) transitive closure of p.

Suppose that we actually allowed atoms in the scope of negation. For the example above this would mean that G predicts both $tp(a, b)$ and $\neg tp(a, b)$. Interpreted as a logic program under the Clark completion, we could infer neither the former nor the latter. That may not be desirable either, but it is clearly preferable to our situation. To be able to infer both a "ground" atom and its negation seems totally wrong.

The problems we're facing here seem to be caused by the fact that we have too many different interpretations that are all models of the given grammar. It seems that if we could reduce the number of models to a few or even a single one that is the intuitively correct one, then we wouldn't be in so much trouble. This is an approach that has been variously pursued in the logic programming literature. An interesting approach from our perspective is the stable model semantics of Gelfond and Lifschitz (1988). We will examine stable models in the next section, and then see how the basic idea can be implemented for our specific problem.

6.6.1 Stable models

The stable model semantics is not a completion semantics, but rather based on the notion of a fixed point on a program transformation operator. The semantics is not constructive, however. It requires, as Wallace puts it, an oracle to produce a candidate model, which can then be checked for being a fixed point and thus, a model. The property which distinguishes the stable model approach from other fixed point semantics approaches is the fact that the basic operation is not one on models, but on pairs of models and programs. Our presentation of the stable model semantics follows (Wallace 1993).

Finding a stable model for a logic program proceeds as follows. The first step is to replace the input program clauses with their ground instances, yielding a (usually infinite) variable free program. We then define the transformation GL (Gelfond-Lifschitz), taking as input a Herbrand model and a ground normal program. Let M be a Herbrand model and P a normal program. We write $inst(P)$ for the set of ground instances of clauses in P. The set $GL(M, P)$ is then defined to be all and only clauses $A \leftarrow B_1, \ldots, B_m$ such that $A \leftarrow B_1, \ldots, B_m, C_1, \ldots, C_n \in inst(P)$, each B_i is positive, each C_i is negative, and $M \models C_1, \ldots, C_n$. If $M = T_{GL(M,P)} \uparrow \omega$, then M is a stable model of P.

If the program is definite, then a stable model of that program exists, it is unique, and it is the least fixed point of the T_P operator. That's nice since it gets rid of the problem for the Clark completion we noted in the preceding section. Recall that for the transitive closure program (p. 135), the Clark completion predicted neither $tp(a, b)$ nor $\neg tp(a, b)$. The stable model semantics, on the other hand, predicts $\neg tp(a, b)$, which is intuitively correct.

Some logic programs have inconsistent Clark completions. Since anything can be deduced from an inconsistent set of sentences, this is not very useful. Consider the example program below (from Wallace 1993).

$$p \leftarrow \neg p$$

$$q(a)$$

This program makes perfectly good operational sense for queries concerning $q/1$, it even has a unique least Herbrand model, despite its use of negation. Despite these facts, its Clark completion is inconsistent, and thus useless. The stable model semantics, on the other hand, provides a single stable model, namely $\{q(a)\}$.

There is not always a unique stable model. Consider the example below.

$$p \leftarrow \neg q$$

$$q \leftarrow \neg p$$

We get one stable model where p is true and q is false, and one where q is true and p is false. It would seem that similar, not quite so pathological examples might occur in practice. It is therefore interesting that the stable model semantics has something useful to say about those cases, when viewed in terms of satisfiability.

The basic idea of the stable model semantics is that anything that can't be proven in finite time is assumed to be false. It achieves this effect by requiring M to be the least fixed point of $GL(M, P)$. Notice that the stable model semantics is not constructive, and still suffers from the same drawbacks as other canonical model approaches. A candidate model needs to be a least fixed point for some operator, something that can not be checked mechanically. However, of all the canonical model approaches that have been proposed for handling negation in logic programming, it has the most intuitive appeal. Finally, Wallace (1993) defines a completion semantics that is equivalent to the stable model semantics for Herbrand interpretations. This shows that the stable model semantics can even be reified as a completion that works with the standard first order model theory. It is thus reasonable to take the stable model semantics as our point of departure. We have chosen to present the stable model semantics in terms of its original formulation, and not Wallace's completion, since our own approach looks much more like the original version. However, the reader is referred to (Wallace 1993) for his very elegant completion semantics.

6.6.2 Grammars with full negation

We now turn back to feature constraint grammars. In this section, we will lift the restriction on the use of negation imposed previously, and provide a model definition for the amended syntax. For the syntax, we only need to modify the definition of a relation term.

Definition 6.20 (relation terms)

- ϕ *if ϕ is a feature term*
- A *if A is an atom*
- $\neg\,\phi$ *if ϕ is a relation term*
- $\phi_1 \wedge \phi_2,\ \phi_1 \vee \phi_2$ *if ϕ_1 and ϕ_2 are relation terms*

We say that a clause is normal if the clause body is a conjunction of atoms and negated atoms. Although a general definition of the semantics could be given, we restrict our attention to normal clauses, as this considerably simplifies some technical details. This restriction clearly does not affect the expressive power of the system. A grammar is then as before, except for the amended definition of relation terms, and the restriction to normal clauses.

We begin our definition of the models of a grammar with a definition of an infinite sequence of sets, providing partial information about the model for a set of clauses.

Definition 6.21 (nt)
Let S be a set and $N \subset \mathbb{N}$ a finite set of natural numbers. Define $nt(S, N)$ to be the set of all n-tuples over S, for all $n \in N$, i.e., $nt(S, N) = \{\langle a_1, \ldots, a_n\rangle \mid a_1, \ldots, a_n \in S,\ n \in N\}$

We write $\mathrm{ran}(f)$ for the range of a function f.

Definition 6.22
Let $S = \langle \mathcal{T}, \preceq, \mathcal{F}, approp, \mathcal{R}, Ar\rangle$ be a signature, C a set of normal clauses, $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{P}\rangle$ an S-interpretation and α a variable assignment. We write $[\![r(\phi_1, \ldots, \phi_n)]\!]^{\mathcal{I}}_{\alpha} \in \mathcal{P}$ iff $\exists\langle u_1, \ldots, u_n\rangle \in \mathcal{P}(r).\ u_1 \in [\![\phi_1]\!]^{\mathcal{I}}_{\alpha}, \ldots, u_n \in [\![\phi_n]\!]^{\mathcal{I}}_{\alpha}$

Definition 6.23 ($\vec{\mathcal{P}}$)
Let $S = \langle \mathcal{T}, \preceq, \mathcal{F}, approp, \mathcal{R}, Ar\rangle$ be a signature, C a set of normal clauses and $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{P}\rangle$ an S-interpretation. We say the infinite sequence $\vec{\mathcal{P}} = \langle \mathcal{P}_0, \mathcal{P}_1, \ldots\rangle$ models C iff

- *for each* $n \in \mathbb{N}$, $\mathcal{P}_n : \mathcal{R} \rightarrow nt(\mathcal{U}, ran(Ar))$
- $\forall r \in \mathcal{R}.\ \mathcal{P}_0(r) = \emptyset$
- $\forall n > 0.\ \forall r \in \mathcal{R}.\ \langle u_1, \ldots, u_n \rangle \in \mathcal{P}_n(r)$ *iff*
  - $n = Ar(r)$
  - $\exists\, r(\phi_1, \ldots, \phi_n) \leftarrow B_1 \wedge \ldots \wedge B_i \wedge \neg C_1 \wedge \ldots \wedge \neg C_j \in C$ *and* $\exists\, \alpha$ *s.t.*
    - $u_1 \in [\![\phi_1]\!]^{\mathcal{I}}_{\alpha}, \ldots, u_n \in [\![\phi_n]\!]^{\mathcal{I}}_{\alpha}$
    - $[\![B_1]\!]^{\mathcal{I}}_{\alpha}, \ldots, [\![B_n]\!]^{\mathcal{I}}_{\alpha} \in \mathcal{P}_{n-1}$
    - $\forall k \in \mathbb{N}.\ [\![C_1]\!]^{\mathcal{I}}_{\alpha}, \ldots [\![C_j]\!]^{\mathcal{I}}_{\alpha} \notin \mathcal{P}_k$

Definition 6.24 (model)
Let $G = \langle S, \Theta, C \rangle$ *be a grammar, where* $S = \langle \mathcal{T}, \preceq, \mathcal{F}, approp, \mathcal{R}, Ar \rangle$, *and* $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{P} \rangle$ *an interpretation.* $\mathcal{I}$ *is a model of* G *iff*

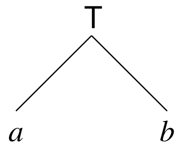
- *for each* $\theta \in \Theta.\ [\![\theta]\!]^{\mathcal{I}} = \mathcal{U}$
- $\vec{\mathcal{P}}$ *is defined*
- $\forall r \in \mathcal{R}.\ \forall \langle u_1, \ldots, u_n \rangle.\ \langle u_1, \ldots, u_n \rangle \in \mathcal{P}(r)$ *iff*
  for some $k \in \mathbb{N}.\ \langle u_1, \ldots, u_n \rangle \in \mathcal{P}_k(r)$

This ugly piece of mathematics defines a semantics for clauses that is similar to the least fixed point approach in logic programming. The intuition is very similar to the T_P function used to construct the least Herbrand model (Lloyd 1984). If we only had definite clauses, we would get exactly the least fixed point interpretation as the only possible model for a given domain. The situation is complicated by the presence of negative literals. Like the stable model semantics, our semantics does not say anything about negative literals locally, for a certain $\mathcal{P}_i$ in the derivation. In the stable model semantics, this is achieved by requiring that the final model should entail the negative literals in a clause, which is exactly what we do. Our equivalent of the T_P function used in the stable model semantics is the positive part of the $\vec{\mathcal{P}}$ construction. The treatment of negative literals in clause bodies is part of the definition of $\vec{\mathcal{P}}$ in our case, since that seems more intuitive, but the difference is clearly only notational.

Notice that our definition is clearly *not* constructive, since the condition on negated atoms refers to the entire sequence. It is not even *functional*, in the sense that $\vec{\mathcal{P}}$ may not be unique. Since this is the same for the stable model semantics, anything else would have been a surprise.

6.6.3 Some examples

Let us reconsider the transitive closure grammar G from p. 135, which we repeat here for convenience. We had the the following type hierarchy



and the set of clauses below.

$$p(a, a)$$

$$tp(X, Y) := p(X, Y)$$

$$tp(X, Z) := p(X, Y) \wedge tp(Y, Z)$$

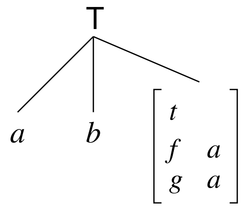
We saw that with the semantics from the last section, we predicted a model $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{P} \rangle$, where

- $\mathcal{U} = \{a', b'\}$,
- $\mathcal{S}(a') = a$, $\mathcal{S}(b') = b$,
- $\mathcal{P}(p) = \{\langle a', a' \rangle\}$ and $\mathcal{P}(tp) = \{\langle a', a' \rangle, \langle a', b' \rangle\}$.

This meant that G predicted $tp(a, b)$, which was not intended. With our new semantics, $\mathcal{I}$ is not a model of G anymore. To see this, consider $\vec{\mathcal{P}}$ with respect to the grammar and interpretation above. If $[\![tp(a, b)]\!]^{\mathcal{I}}_{\alpha} \in \mathcal{P}$ for some α, then for some $n \in \mathbb{N}$, $[\![tp(a, b)]\!]^{\mathcal{I}}_{\alpha} \in \mathcal{P}_n$. From the grammar, however, we can see that then, $[\![tp(a, b)]\!]^{\mathcal{I}}_{\alpha} \in \mathcal{P}_{n-1}$, and so on. Since we know that $[\![tp(a, b)]\!]^{\mathcal{I}}_{\alpha} \notin \mathcal{P}_0$, we have a contradiction. In fact, the *only* possible model of G with identical $\mathcal{U}$, $\mathcal{S}$ and $\mathcal{A}$ is $\mathcal{I}' = \langle \mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{P}' \rangle$, where

$$\mathcal{P}'(p) = \{\langle a', a' \rangle\} \text{ and } \mathcal{P}'(tp) = \{\langle a', a' \rangle\}.$$

For our next example, we will keep the same grammar, but extend our type hierarchy a little.



Consider the queries:

- $\neg\, p(a, a)$
- $f : (X \wedge a) \wedge g : (Y \wedge a) \wedge \neg\, p(X, Y)$

Neither of those queries should be predicted by G. However, the first one is in fact predicted by G, the second one isn't. This is so because there can be models that have no objects of type a in their domain, and where $\mathcal{P}(p)$ is thus empty. This is different in the second query, where we explicitly require objects of type a to exist. To paraphrase, the first query asks if there are models such that it is not the case that there are objects of type a that stand in the p relation. The second query, on the other hand, asks if there are models that contain objects of type a that do not stand in the p relation. One can therefore say that by mentioning the arguments *outside* of the negative literal, we have moved the existential quantifier in the above paraphrase out of the scope of the negation operator.

It is not possible to change this behavior of the logic, unless we take recourse to some canonical base interpretation (like the set of abstract, totally well-typed and sort-resolved feature structures, for example). We could then give a canonical model semantics that always includes all admissible objects from the base interpretation. Although this might be an interesting alternative, we chose not to pursue it here, since we prefer the close connection to first order logic our present architecture gives us.

6.7 Conclusion

In this chapter, we have examined the addition of a relational component to our feature logic grammars. We first noted that it is not possible to employ the constraint logic programming scheme of Höhfeld and Smolka (1988).

We then extended feature logic constraint grammars with a definite clause component. Relational atoms were not allowed to appear in the scope of negation anywhere (except, of course, implicitly in the clause head). Our semantics for relations was based on a constraint version of Clark's completion semantics (Clark 1978). We then went on to show that with this conservative relational extension, we were able to obtain the same soundness and completeness results as in ch. 5.

In the final section of this chapter, we investigated possibilities to relax the restriction on the use of negation. We first observed that we would lose our computability results if we did that. We then presented the stable model semantics (Gelfond and Lifschitz 1988) as a promising approach. Finally, we showed how a version of the stable model semantics could be implemented for the relational part of feature logic constraint grammars. We saw that it is possible to give a somewhat intuitive treatment of negation in this system, but that we lose both clarity and computational properties in the process.

Frank Richter and Manfred Sailer have investigated relations for HPSG in particular (see, e.g., Richter et al. 1999). The two approaches are not easily compared, since (Richter et al. 1999) not only introduce relations, but also full-fledged general quantification. They thus obviate the need for a special clausal notation for relations. Instead, relations can be defined any way one can think of. Of course, it is not easy to see how one would compute with a general system like that (but that is not the point of their proposal). As far as we're aware of, there has been no other proposal for integrating implicational and relational constraints with a declarative and computational semantics.

For our own proposal, we expect that a sound procedural semantics could be implemented using constructive negation (Chan 1988). However, this is beyond the scope of this dissertation.

Chapter 7

Conclusion

In this dissertation, we investigated the computational modeling of HPSG grammars. Our goal was to directly use a formalization of HPSG based on classical logic for computation. We used the work of King (1989, 1994) as our point of departure.

We first focused on the problem of satisfiability for feature terms. We showed how satisfiability problems can elegantly be solved using constraint solving methods. We discussed how constraint solving differs from unification-based approaches in making computation independent of semantic structures. This yields a clearer separation of declarative and computational semantics. Constraint-based methods are also more easily extended than unification-based ones.

We examined the notion of grammaticality for grammars. In a logic-based setting, this is naturally expressed as a relation between theories (sets of formulae) and queries (individual formulae). We called this relation prediction. Given these basic ingredients, there are two sorts of relationships we might employ: satisfiability and logical consequence. We established that for formalisms based on classical logic, there is a correspondence between constraint-based grammar formalisms and a formulation of prediction in terms of satisfiability. Conversely, a formalization of prediction as logical consequence is natural for rule-based grammars.

For our own grammar formalism, we gave a definition of prediction based on satisfiability. We established that prediction was undecidable, and not recursively enumerable. However, we found that the co-problem, non-prediction, is enumerable.

We further examined the relationship between satisfiability and logical consequence-based approaches by giving a translation of feature logic constraint grammars into constraint logic programs. As expected, the translation turned out to be complete only for a subset of the decidable grammars.

We then established a proof method for the prediction problem based on lazy type evaluation. This method turned out to be much simpler than the translation to constraint logic programs. We showed that our method was sound, and complete for the non-prediction case.

The addition of relations to our grammar formalism posed some interesting problems. We wanted a notion of relation close to logic programming. Our grammar formalism, however, supports a satisfiability-based notion to prediction, as opposed to logical consequence in logic programming. We resolved this issue by giving a semantics to definite clauses inspired by Clark's completion semantics, while overall retaining our satisfiability-based approach. We extended our proof method to include definite relations, and obtained the same soundness and completeness results as in the relation-free case.

Finally, we considered relations in the scope of negation. We employed an approach resembling the stable model semantics. It turned out that even under a satisfiability-based approach, this semantics delivers intuitive results in the presence of relations in the scope of negation.

7.1 Implementation

As mentioned in the introduction, large parts of what has been described in this thesis has actually been implemented in the ConTroll[1] system. ConTroll is based on its predecessor, Troll (Gerdemann et al. 1995), from which it inherits the type system. The following list provides an overview of the ConTroll features relevant to this thesis.

- Signatures, terms and their interpretation are implemented along the lines discussed in ch. 2. The only restriction is that variables can not occur in the scope of negation. That is, path inequations are not implemented.
- Grammars are implemented as described in ch. 6. That is, definite

[1]See `http://www.sfs.nphil.uni-tuebingen.de/controll/`, as well as (Götz and Meurers 1997a) and (Götz et al. 1997).

relations are fully supported. Grammar normalization as described towards the end of ch. 3 is also supported. That means that the user can write implicational constraints with complex antecedents, which will then get compiled into the simpler form described in ch. 3.

- Lazy evaluation is implemented as described in ch. 5. The user must therefore ensure that grammars are type consistent.

Of course, there is much more to ConTroll than just an implementation of the logic. There are tools to ease the development of relatively large grammars such as a GUI front-end, debugging tools, off-line grammar optimization (Meurers and Minnen 1999) etc. See (Götz and Meurers 1997a) for an overview.

At a much more basic level, the proof systems developed in ch. 5 and 6 tell us nothing about an efficient evaluation strategy for goals. From a practical point of view, this is a much more difficult problem than the correct implementation of the logic. The execution strategy of ConTroll is based on the concurrent Andorra principle (Haridi and Janson 1990), which basically says, do as much deterministic computation as you can before you set a choice point. In addition, ConTroll provides a number of concurrent control primitives that allow the users to fine-tune the execution of their grammars. The control primitives for ConTroll are described in (Götz et al. 1997).

The ConTroll system has been used successfully to implement a large fragment of German. See (Hinrichs et al. 1997) for a description of the fragment and its implementation.

Bibliography

Aït-Kaci, H., A. Podelski, and S. C. Goldstein (1993). Order-sorted theory unification. Technical Report 32, Digital Equipment Corporation.

Aldag, B. (1997). A proof theoretic investigation of prediction in HPSG. Master's thesis, Seminar für Sprachwissenschaft, Universität Tübingen.

Baader, F. and J. Siekmann (1994). Unification theory. In D. M. Gabbay, C. Hogger, and J. A. Robinson (Eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming*, Volume 2: Deduction Methodologies, pp. 41–125. Clarendon Press.

Boolos, G. S. and R. C. Jeffrey (1974). *Computability and Logic* (third ed.). Cambridge University Press.

Bouma, G., E. Hinrichs, G.-J. Kruijff, and R. Oehrle (Eds.) (1999). *Constraints and Resources in Natural Language Syntax and Semantics.* Cambridge University Press.

Carpenter, B. (1992). *The logic of typed feature structures*, Volume 32 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.

Carpenter, B., C. Pollard, and A. Franz (1991). The specification and implementation of constraint-based unification grammars. In *Proceedings of the 2nd International Workshop on Parsing Technology*, pp. 143 – 153.

Chan, D. (1988). Constructive negation based on the completed database. In R. A. Kowalski and K. A. Bowen (Eds.), *Proceedings of the 5th International Conference and Symposium on Logic Programming*, pp. 111–125. MIT Press.

Chomsky, N. (1981). *Lectures on Government and Binding*, Volume 9 of *Studies in Generative Grammar*. Foris Publications.

Clark, K. L. (1978). Negation as failure. In H. Gallaire and J. Minker (Eds.), *Logic and Databases*, pp. 293–322. New York: Plenum Press.

Dörre, J. (1994). Feature-Logik und Semiunifikation. Arbeitspapiere des SFB 340 Nr. 48, Universität Stuttgart.

Dörre, J. and M. Dorna (1993). CUF - a formalism for linguistic knowledge representation. In J. Dörre (Ed.), *Computational aspects of constraint based linguistic descriptions I*, pp. 1–22. Universität Stuttgart: DYANA-2 Deliverable R1.2.A.

Dörre, J. and A. Eisele (1991, January). A comprehensive unification based formalism. DYANA Deliverable R3.1.B, Universität Stuttgart.

Gazdar, G., E. Klein, G. Pullum, and I. Sag (1985). *Generalized Phrase Structure Grammar*. Harvard University Press.

Gelfond, M. and V. Lifschitz (1988). The stable model semantics for logic programming. In R. A. Kowalski and K. A. Bowen (Eds.), *Proceedings of the 5th Conference and Symposium on Logic Programming*, pp. 1070–1080. MIT Press.

Gerdemann, D., T. Götz, J. Griffith, S. Kepser, and F. Morawietz (1995). *Troll manual*. Seminar für Sprachwissenschaft, Universität Tübingen.

Gerdemann, D. and P. J. King (1993). Typed feature structures for expressing and computationally implementing feature cooccurence restrictions. In *Proceedings of 4. Fachtagung der Sektion Computerlinguistik der Deutschen Gesellschaft für Sprachwissenschaft*, pp. 33–39.

Götz, T. (1994). A normal form for typed feature structures. Arbeitspapiere des SFB 340 Nr. 42, Universität Tübingen.

Götz, T. (1995). Compiling HPSG constraint grammars into logic programs. In *Proceedings of the joint ELSNET–COMPULOG-NET–EAGLES workshop on computational logic for natural language processing*, Edinburgh.

Götz, T., D. Meurers, and D. Gerdemann (1997). *The ConTroll manual and user's guide*. Seminar für Sprachwissenschaft, University of Tübingen.

Götz, T. and W. D. Meurers (1995). Compiling HPSG type constraints into definite clause programs. In *Proceedings of the Thrirty-Third Annual Meeting of the ACL*, Boston. Association for Computational Linguistics.

Götz, T. and W. D. Meurers (1997a). The ConTroll system as large grammar development platform. In *Proceedings of the ACL/EACL post-conference workshop on Computational Environments for Grammar Development and Linguistic Engineering*, Madrid, Spain.

Götz, T. and W. D. Meurers (1997b). Interleaving universal principles and relational constraints over typed feature logic. In *Proceedings of the 35th Annual Meeting of the ACL and the 8th Conference of the EACL*, Madrid, Spain.

Götz, T. and W. D. Meurers (1999). The importance of being lazy. In G. Webelhuth, J.-P. Koenig, and A. Kathol (Eds.), *Lexical and Constructional Aspects of Linguistic Explanation*, pp. 249–264. CSLI Publications.

Haridi, S. and S. Janson (1990). Kernel Andorra Prolog and its computation model. In D. H. D. Warren and P. Szeredi (Eds.), *Proceedings of the seventh international conference on logic programming*, pp. 31–46. MIT Press.

Hinrichs, E., D. Meurers, F. Richter, M. Sailer, and H. Winhart (1997). Ein HPSG-Fragment des Deutschen, Teil 1: Theorie. Arbeitspapiere des SFB 340 Nr. 95, Universität Tübingen.

Höhfeld, M. and G. Smolka (1988). Definite relations over constraint languages. Technical Report 53, LILOG, IBM Deutschland.

Johnson, M. (1988). *Attribute-Value Logic and the Theory of Grammar.* CSLI lecture notes.

Johnson, M. (1991). Features and formulae. *Computational Linguistics 17*(2), 131–151.

Johnson, M. (1994). Two ways of formalizing grammar. *Linguistics & Philosophy 17*, 221–248.

Kaplan, R. and J. Bresnan (1982). Lexical-functional grammar: A formal system for grammatical representation. In J. Bresnan (Ed.), *The Mental Representation of Grammatical Relations*, pp. 173–281. MIT Press.

Kay, M. (1979). Functional grammar. In *Proceedings fo the Fifth Annual Meeting of the Berkeley Linguistics Society*, pp. 17–19.

Kepser, S. (1994). A satisfiability algorithm for a typed feature logic. Arbeitspapiere des SFB 340 Nr. 60, Universität Tübingen.

King, P. J. (1989). *A logical formalism for head-driven phrase structure grammar*. Ph. D. thesis, University of Manchester.

King, P. J. (1994). An expanded logical formalism for head-driven phrase structure grammar. Arbeitspapiere des SFB 340 Nr. 59, Universität Tübingen.

King, P. J. (1995). From unification to constraint. Unpublished lecture notes, University of Tübingen.

King, P. J. and T. W. Götz (1993). Eliminating the feature introduction condition by modifying type inference. Arbeitspapiere des SFB 340 Nr. 31, Universität Tübingen.

King, P. J., K. I. Simov, and B. Aldag (1999). The complexity of modellability in finite and computable signatures of a constraint logic for head-driven phrase structure grammar. *The Journal of Logic, Language and Information 8*(1), 83–110.

Kunen, K. (1987). Negation in logic programming. *Journal of Logic Programming 4*(4), 289–308.

Lewis, H. R. and C. H. Papadimitriou (1981). *Elements of the Theory of Computation*. Prentice Hall.

Lloyd, J. W. (1984). *Foundations of Logic Programming*. Berlin, Germany: Springer-Verlag.

Manandhar, S. (1995). Deterministic consistency checking of LP constraints. In *proceedings of the 7th conference of the EACL*, Dublin, Ireland.

Meurers, W. D. (1994). On implementing an HPSG theory – aspects of the logical architecture, the formalization, and the implementation of head-driven phrase structure grammars. In: Erhard W. Hinrichs, W. Detmar Meurers, and Tsuneko Nakazawa: *Partial-VP and Split-NP Topicalization in German – An HPSG Analysis and its Implementation*. Arbeitspapiere des SFB 340 Nr. 58, Universität Tübingen.

Meurers, W. D. and G. Minnen (1999). Off-line constraint propagation. In G. Webelhuth, J.-P. Koenig, and A. Kathol (Eds.), *Lexical and Constructional Aspects of Linguistic Explanation*, pp. 299–314. CSLI Publications.

Minnen, G. (1998). *Off-line Compilation for Efficient Processing with Constraint-logic Grammars*. Ph. D. thesis, Univesität Tübingen.

Pollard, C. and I. A. Sag (1987). *Information-Based Syntax and Semantics*. CSLI lecture notes.

Pollard, C. and I. A. Sag (1994). *Head-Driven Phrase Structure Grammar*. Chicago: University of Chicago Press.

Richter, F., M. Sailer, and G. Penn (1999). A formal interpretation of relations and quantification in HPSG. In (Bouma et al. 1999).

Rogers, J. (1999). *A Descriptive Approach to Language-Theoretic Complexity*. Cambridge University Press.

Rounds, W. C. and R. T. Kasper (1986). A complete logical calculus for record structures representing linguistic information. In *1st IEEE Symposium on Logic in Computer Science*, pp. 38 – 43.

Shepherdson, C. J. (1987). Negation in logic programming. In J. Minker (Ed.), *Foundations of Deductive Databases and Logic Programming*, Chapter 1, pp. 19–88. Los Altos, CA: Morgan Kaufman.

Shieber, S. (1989). *Parsing and Type Inference for Natural and Computer Languages*. Ph. D. thesis, Stanford University. Published as SRI International, Technical Note 460.

Smolka, G. (1988). A feature logic with subsorts. LILOG Report 33, IWBS, IBM Deutschland.

Smolka, G. (1992). Feature-constraint logics for unification grammars. *Journal of Logic Programming 12*, 51–87.

Wallace, M. (1993). Tight, consistent, and computable completions for unrestricted logic programs. *Journal of Logic Programming 15*, 243–273.