A DOMAIN AWARE GRAMMAR FOR PARSING REQUIREMENTS INTO TYPED FOL

by

Amin Sharifi
B.Eng., Shiraz University, Department of Computer Engineering, 2004

A Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
Summer 2009

APPROVAL

Name: Amin Sharifi

Degree: Master of Science

Title of Thesis: A Domain Aware Grammar for Parsing Requirements into Typed FOL

Examining Committee: Dr. Ramesh Krishnamurti
Chair

Dr. Veronica Dahl, Senior Supervisor

Dr. Robert F. Hadley, Supervisor

Mr. Reto Ferri, Supervisor,
Zurich University of Applied Sciences, Switzerland

Dr. Fred Popowich, Examiner

Date Approved: July 20th 2009

Declaration of Partial Copyright Licence

Abstract

To document software and business requirements, it is desirable to use natural language text which can be processed by computer (as is done e.g. by RavenFlow software). We argue that requirements and their application domain model (including ontology) are interdependent, and that by using the domain model we can parse and semantically analyze the requirements more effectively. To this end we develop a type system for application domain entities that includes multiple inheritance, as well as specializable and negated types. We integrate it into the HPSG grammar formalism and extend the Minimal Recursion Semantics by adding new predications and constraints. We show how, by informing the parser with criteria to reject sentences on the basis of type mismatch, our system can less ambiguously translate natural language sentences into a Domain-Typed First Order Logic format, through pruning some semantically incorrect options within the search space.

To my parents with love
and to my teachers with gratitude.

Acknowledgments

I offer my deepest appreciation to my senior supervisor Professor Veronica Dahl whose unlimited support and guidance made this research possible. I am absolutely grateful to know her. She is not only excellent in science and creativity but also excellent in character and a very caring mentor whose energy and happiness is truly inspiring.

I would like to thank Professor Robert F. Hadley whose support, patience and feedback played a major role in the completion of this dissertation. Discussions with Professor Hadley have always given me new insights on the topic.

Special thanks to Mr. Reto Ferri, who has been supporting me for many years, whom I have always appreciated for his attention to details and his intellectually overwhelming software design skills. I thank Mr. Ferri for accepting to travel to Canada from Europe to be present at my thesis defense.

I would like to thank Dr. Ann Copestake for answering my question about MRS by email, and Dr. Norbert E. Fuchs and Dr. Rolf Schwitter for answering my email and referring me to useful resources.

I thank Professor Fred Popowich who accepted to examine my thesis and provided useful feedback and Professor Ramesh Krishnamurti who accepted to chair the defense in short notice.

And last but not least, I thank my family whose endless care, support and patience helped me during the years that I have been far away.

Contents

List of Tables

List of Figures

List of Programs

Chapter 1

Introduction

In this thesis we develop a grammar, a parser and a program that extracts semantics from controlled natural language text, through parsing it with reference to an evolving ontology. By the adjective "evolving" we mean that the natural language text might add new knowledge to the ontology. The semantics of the text can be stored in a knowledge-base for future querying or reasoning. This has been of interest in the Natural Language Processing community for long. We blend it with the requirements sub-discipline of software engineering and business analysis [44] to create a desirable application in industry. We would like our program to parse requirement texts (system behavior and business process descriptions in terms of use cases) and extract their semantics in some suitable formal representation.[1]

Extracting semantics from natural language text has been a very challenging task in Natural Language Processing. Although it is very easy to write small sized parsers and semantic extractors for processing simple and toy sentences, it is really hard to write a program for parsing and analyzing sentences of a real world application. Applications of the system developed in this thesis must be undertaken with caution.

Disclaimer:
The grammar, parser, antecedent resolution and FOL conversion methods as described in this thesis with the provided implementation are research results and should not be directly used without expert supervision in a real application. Neither the student nor the university provide any form of warranty on the theory and implementation provided here.

[1] Although this is the desired application, for the development of the grammar we use very simple English sentences throughout the thesis, for ease of exposition of the grammar concepts.

1.1 Requirements

The development of medium to large sized software systems begins with a *requirement phase*, through which an understanding is sought by the software team, led by a business/system analyst[2], about what the client wants from the desired software. This phase typically needs several meetings in which the client explains to the software team what exactly she/he wants from the software[3], and which results in statements describing the desired behavior of the software system as agreed by both the client and the software team. These statements constitute what we call the *software requirements* of that specific project. The success of a project is defined in terms of the satisfaction of all of the requirements requested by the client.

In this thesis we are only interested in parsing *functional requirements*, and *business rules*. *Functional requirements* also known as *behavioral requirements* describe the behavior of a system, its capabilities and features. They comprise a significant portion of the requirements. They are so important that requirements are usually categorized as functional or non-functional.[4] Functional requirements do not state the details of "how" the system is going to perform a task, but rather state "what" tasks it must be able to perform. *Business rules*, according to [44]: describe "a policy, guideline, standard, or regulation upon which the business operates." "A business rule is a statement that defines or constrains some aspect of the business." Reference to business rules in the requirements of an upcoming software system is necessary if the software is going to be used by a business to achieve some of its business goals.

1.2 Use Cases: A Standard Way of Documenting Functional Requirements

An excellent way to document functional requirements is by *use cases* [16, 49]. A use case is a natural language text that describes a story about how the upcoming system can be used. It involves the interactions between users and the upcoming system (or the components of

[2] By *software team* we refer to business/system analysts, software architect, and software developers. See page 82 of Larman [49].

[3] See for example page 76 of Larman [49].

[4] Other requirements include usability requirements, reliability requirements, performance requirements, supportability requirements. For details please refer to [39].

the system in lower levels).

Users of the system that interact with it through a use case to accomplish some goals are called the *stakeholders* of that use case.

An *actor* in a use case is anyone or anything with behavior (with regards to the use case), i.e., someone or something that interacts with another entity in the use case.

A *scenario* in a use case is a sequence of steps described in natural language that specifies the order of the events and interactions that happen in that use case. There is a main scenario in which no unexpected thing occurs and all operations succeed. This scenario is called the *main success scenario*. *Alternate scenarios* are usually necessary to describe how the system responds when an error happens.

The conditions that must hold in order for a use case to be applicable are called the *preconditions* of that use case. And the conditions that must hold when the use case terminates are called *postconditions*.

There are different formats to write a use case depending on the level of detail needed, namely *brief*, *casual* and *fully dressed*. Without going into details of these formats[5], we emphasize that natural language is used to write use cases.

Example 1.1 Here is an example of a simplified use case[6] describing the main success scenario of using an ATM machine in casual format.

Preconditions:
The ATM machine shows the welcome message.
Main Success Scenario:

1. A customer enters her/his banking card into the card slot of the ATM machine.
2. The ATM machine asks the customer to enter her/his PIN number.
3. The customer enters the PIN number.
4. The ATM machine verifies the PIN number.
5. The ATM machine shows the customer's account balance on its screen.
6. The customer presses the OK button.
7. The ATM machine returns the banking card.
8. The customer pulls the card out of the slot.
9. The ATM machine shows the welcome screen.

[5] The interested reader can refer to [16, 49].

[6] We have omitted the details including alternate scenarios and the post condition.

Of course, a good choice is to store these documents electronically. So the use of editor software for writing these documents is trivial. This would ease the collaboration among the client and the members of the software team, as documents can be electronically shared[7]. We will use this as one component of an argument we provide in the following sections to justify the use of software that is also able to perform some syntactic and semantic analysis on the documents. An example that is already available in industry to manage use cases is the *RavenFlow*[8] software [64].

1.3 Business Process Modeling

Use cases can be used to document the operation and processes of an organization, where they are called *business use cases* [16]. A business use case describes the interactions between the people and departments in the organization. They could refer to the existing design of the business or its future design. The act of identifying and documenting the processes within an organization, by use cases for example, is called business process modeling. The resulting use cases, describe the functional requirements of an organization rather a software system.

The following shows a simplified business use case for the Citizenship and Immigration Canada describing the process of issuing a study permit for an international student.

Preconditions:
A university in Canada admitted an international_student.
The student received the admission_letter from the university.

Main Success Scenario:
1. The student obtains an IMM1294 application form and fills it.
2. She/he attaches the admission letter to the application form.
3. She/he submits the application form to a visa office.
4. The visa office verifies the admission letter.
5. It[9] records the application form.

[7] For example, by using a version control software over a network.

[8] See http://www.ravenflow.com

[9] The antecedent of this pronoun is grammatically ambiguous, however with semantic analysis we are able to determine that the correct antecedent is *the visa office*. However one should note that resolving pronoun references automatically is a fallible process, as a general rule, and human supervision is required on the

6. It issues a study permit for the student.
7. It sends the permit to the student.

If all of the business processes are modeled with use cases that are parsed and semantically analyzed, it would be easier to improve the processes, modify the organization and etc. Because we would be able to identify the dependent processes that need to be modified as a result of a modification in another part of the business. Also, we can check the consistency and the completeness of the business processes by semi-automated tools to ensure, to some extent[10], that the organization's design does not have flaws.

1.4 Requirements and Domain Models

As part of gathering requirements, it is necessary to model the domain of the system, which results in a *domain model*. Any important entity of the system or its usage is represented by a *domain object* or a *conceptual class*. The domain model contains these classes with their relations. A domain model can be thought of as containing an *ontology* that describes the domain.

From one direction we observe that the domain model is the product of analyzing and understanding the requirements in natural language. For example an early approach proposed by Abbott [1] identifies the noun phrases in the requirements and proposes them as the possible conceptual classes of the domain. A human being can then prune away rudimentary and unnecessary classes from the proposed candidates.

From the other direction, it can be thought that the requirements refer to and depend on an upcoming domain model, if the supported behavior of each entity is specified in the domain model. From this perspective, if a domain model with supported behaviors is available (even a partial one), then some sentences cannot be accepted as valid requirements. These sentences are the ones that expect an unsupported behavior from an entity. Like the sentence *the table talks* or *the university issues a study permit for the student.* We say these sentences contain *type mismatches*, that is an expression with an inappropriate domain type is used, such as *table* or *university* respectively for the mentioned sentences.

As another example, the following sentence for our ATM use case example is invalid

automatic results.

[10] This is due to the fact that analyzing natural language and extracting semantics is a very challenging task, so eventually supervision by human experts is imperative.

with reference to the domain of ATMs and customers. The reason is, in this domain, the customer is not able verify the authenticity of a banking card, i.e., the behavior is not supported by an entity of type customer.

10. The customer verifies the authenticity of the banking card.

So, the domain model with supported behavior of entities and the requirements are interdependent. We assume the domain model contains the supported behavior of the entities it describes in the rest of the thesis.[11]

1.5 Advocating the Use of Semi-automated Software to Maintain Requirements

While the maintenance of use cases by software is the default choice for ease of sharing and editing, it might be also promising to have a semi-automated tool (such as *RavenFlow* [64]) that is able to syntactically and to some extent semantically analyze the requirements. Using such a tool could reduce the semantic mistakes in requirements.

On the other hand since requirements must be precise and unambiguous, it could be desirable to have a tool translate the requirements into a formal representation which is exact and unambiguous. This formal representation can be automatically used to do consistency and completeness checks and to answer queries about the system [34, 68].

We speculate that this can be achieved to some extent by a parser, analyzer that translates a portion of English into a variant of the First Order Logic. Since a complete analysis of syntactic and semantic analysis of natural language is still an open problem, we have to restrict the language coverage to only a portion that is well understood. A *controlled language* is such a portion of the whole language for which precise grammar rules exist.

One of the most prominent work in controlled English that is also motivated by the potential application in parsing requirements is the work by Fuchs et al. [34, 35, 33], specially the Attempto project where they develop a parser from controlled English into Discourse Representation Structures [46]. In Attempto a large number of DCG rules are

[11] This might seem contrary to the Unified Process definition of the domain model [49], in which behaviors are not specified. The supported behavior of entities are implicit in that context. However, in this thesis for the purposes of semantic analysis we need to explicitly specify these behaviors. In the Unified Process framework, the behaviors are explicitly specified in the design model after the domain model is explored.

applied to model a fragment of English. This number as of 2008 is more than 220. This amount of rules makes the maintenance of the system a bit difficult and also makes it very dependent to the choice of language, in this case English. In this thesis we apply Head-driven Phrase Structure Grammar (HPSG) that dramatically reduces the number of rules and obtains a higher degree of language independence. So, in our system one can switch to another language by just using a different lexicon (with lexical rules).

Another limitation of Attempto is the restricted form of English that is used. For example the interpretation of a sentence with multiple quantifiers is fixed according to the order in which the quantifiers appear. The system that we develop in this thesis is more flexible by allowing all the interpretations be produced and the appropriate one be chosen by the user. Of course, later heuristics can be employed to prioritize interpretations.

Among other important controlled languages are the Semantics of Business Vocabulary and Business Rules (SBVR) Structured English [59, 10, 50], and I1CE-V1 [72] which is based on SBVR Structured English and the Discourse Representation Theory approach of Attempto.

All of these instances of controlled languages for the purpose of documenting business or software requirements show the desire to document requirements in a standard way that is processable by computers.

Doubtless, using a controlled language for requirements is indeed a limiting factor of the real world usage of a system that maintains and analyzes requirements. However there is evidence such as the use of controlled language in Boeing [76] that suggests the use of controlled language in industry is useful.

Since there is an interdependence between the domain ontology and the requirements, we propose to develop a a *type system*[12] for the domain entities and use it in the grammar and parser. Using such a type system can prevent the parsing of some of the natural language sentences with type mismatches. Also it will help resolve anaphora by choosing antecedents that are compatible with their referents with respect to the ontology and the supported behavior of the entities.

[12] A type system is a set of rules that assigns types to expressions, and avoids certain combinations of expressions according to their types. In programming language community the expressions are pieces of computer code but here they are natural language expressions.

1.6 Focus and Organization of the Thesis

The main contribution of this thesis is the development of a type system for application domain entities, as opposed to for just grammar entities as was the focus in most previous work. We formalize our type system as an extension of typed λ-calculus, to include multiple inheritance, specializable and negated types. We integrate it into a popular grammar formalism - HPSG - and show how, by informing the parser with criteria to reject sentences on the basis of type mismatch, it can translate natural language sentences into a Domain-Typed First Order Logic with less ambiguity, resulting from pruning of semantically incorrect options within the search space. We also provide a brief example, in chapter 8 of an interesting potential application: parsing requirements.

The thesis is broken down to 3 parts. In the first part we focus on the type system. In chapter 2 we study some of the existing type systems that are in use in grammar formalisms, all of which focus on the grammar entities rather than the application domain entities. We specify the desirable features of a type system for the application domain, including multiple inheritance, specializable and negated types. In chapter 3 we elaborate on the type system and present its characteristics by providing the relevant theorems, and we implement the type system.

In part 2 of the thesis, we set the goal of integrating the type system we developed in the previous chapters to the HPSG formalism. In chapter 4 we present a brief introduction to the HPSG formalism. In chapter 5 we study the syntactic properties of the grammar. In chapter 6 we study the semantic component, where we also add new contributions like additional capability for analyzing the discourse by *discourse predications*, and various variable binding conditions that facilitate conversion of the semantics to a variant of First Order Logic. In chapter 7 we specifically describe how the type system of part 1 can be integrated to the HPSG grammar that we developed in chapters 4, 5, and 6.

In part 3 of the thesis we discuss how we have implemented the grammar and the parser, and we demonstrate how the user interface of our system can be used. We finally show an example of a potential application of the system in parsing use cases.

Chapter 9 provides our conclusion, and highlights our contributions and discusses future work.

Note:

Although we intend to use the grammar for parsing requirements, for ease of exposition and more understandable examples, most of the sample sentences that we provide throughout the thesis are very simple and not from an actual use case. A solid use case example that is processable by our system is provided in chapter 8.

Part I:
The Type System

In this part of the thesis we discuss the importance of types and study how they have been used in grammar. We propose a new type system that can be used for the domain entities of a grammar's application domain. We argue how this type system can be used to reduce the number of readings of natural language sentences by eliminating those readings that lead to type mismatches. We extend the subtype relationship from simple types to complex, specializable, and negated types. We then implement this type system.

Chapter 2

Types

2.1 Introduction

Human beings have the capability of abstracting the surrounding world in terms of separable, and distinguishable forms called objects.[1] The normal human brain among its many known and unknown functions is a classifier of objects, i.e., it can identify and/or learn the common properties of objects and group them in classes, e.g., a class of substance called *water*, a class of objects called *birds*, a class of substances *food*, and etc.[2] Objects can be either physical entities or conceptual abstractions. Some examples of conceptual classes include a class of conceptual objects called *jokes*, a class consisting of all instances of *happiness*, and so on.

So classes are used to describe a set of objects that share some common behavior or attributes. For example *this thesis* is an object belonging to the class *book*.

Definition 2.2 Instance Relation
If an object x belongs to a class σ we say x is an instance of σ, and denote it by $x : \sigma$
For any object x and class σ,
$x : \sigma \iff x$ is an instance of σ

Furthermore there is an organization of classes in our knowledge of the world. A class

[1] We do not claim this ability is exclusive to human beings, for example, a pigeon might also see the world around it in form of objects, but that, we do not speculate on.

[2] We mentioned 'normal human brain' because there is evidence that damage to certain areas of brain can cause object classification and recognition problems in some accident survivors. Also there are a very few number of patients not involved in any accident who are unable to recognize objects. This condition is known as *agnosia* in human neuropsychology.

can be a *subclass* of another. For example *bird* is a subclass of *animal*, and *elephant* is a subclass of *mammal*, etc. Figure 2.1 exemplifies in a graphical way. Each edge represents a subclass relation between two classes. The class that the edge goes out from is a subclass of the class that the edge goes into. The subclass relationships in a domain are called the *taxonomy* or the *class hierarchy* or the *type hierarchy* of the domain.

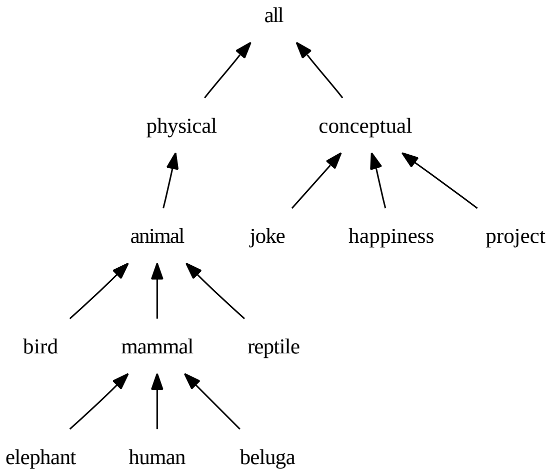


Figure 2.1: An example class hierarchy

In this thesis we use the terms *types* and *classes* interchangeably. So a subtype is the same as subclass in this thesis.[3]

Definition 2.3 Subclass (Subtype) Relation : $\sqsubseteq$
The subclass relation models the natural language *is-a* relation. That is for classes σ and τ we have σ is a subclass of τ if and only if any instance of σ *is an* instance of τ too.

Example 2.4 Here we show how to represent this class and instance information:
A beluga is a mammal. A human is a mammal. An elephant is a mammal. A mammal is an animal. Tiqa[4] is a beluga.

[3] We should note that in some Object Oriented Languages type is slightly different from class. A class is referred to by an *object type*, whereas a type can be a non-object type, i.e., a primitive type (e.g., *integer* type for integer numbers, *boolean* type for True, False values, or *character* type for character symbols). The distinction between primitive and object types however is for technical and historical reasons and here there is no need to retain it in our formulation as our level of representation is more abstract than any specific programming language.

[4] Tiqa is the name of a beluga whale that was born in Vancouver Aquarium on June 10, 2008

$beluga \sqsubseteq mammal$
$human \sqsubseteq mammal$
$elephant \sqsubseteq mammal$
$mammal \sqsubseteq animal$
$Tiqa : beluga$

The concepts of objects and classes are so natural and basic that they can be found in almost all natural language sentences. If a sentence is a combination of a subject and a predicate, then at least the subject of the sentence refers to some object or some class of objects.

In the early 60s, a few decades after the first computer programming languages emerged, the importance of the idea of objects and classes was recognized by researchers in the programming language design community and `SIMULA I`, the first Object Oriented Programming (OOP) language, was created by Ole-Johan Dahl and Kristen Nygaard. The main goals in the design of this language included system description (for system analysis) and system prescription (for implementing the system)[22, 21]. `SIMULA I` (together with C) was the basis of a very important OOP programming language: C++ [70], and C++ was the basis of the next generation of OOP languages such as Java and C#.

Business and system analysts also heavily use the concept of objects and classes (no wonder why `SIMULA I` had the facilitation of system analysis as one of its objectives). Since it is our goal to use our requirement analyzer system in business/system analysis, we should pay special attention to classes and objects in the parser of our requirement analyzer system. To this end we sometimes draw analogies or contrasts between natural language concepts and programming language concepts.

This chapter is organized as follows. In section 2.2 we give an introduction to how type theory and types have been used in linguistics and Natural Language Processing, and in particular, how a type system can help disambiguate a syntactically ambiguous sentence. Then in section 2.3 we introduce *specializable types* and show how they can be used in semantics. In section 2.4 we present the work of Dahl [23, 24] about *incomplete types* that could be thought of as an implementation of specializable types with Prolog. Then in section 2.5 we introduce the notion of *multiple inheritance* and show briefly how it can be handled in extensions of incomplete types [31, 51]. We discuss the limitations of these schemes, and set the stage ready for the next chapter, in which we introduce our type system that efficiently deals with both specializable types and multiple inheritance.

2.2 Types in Previous Theories of Grammar and Semantics

In what follows we present three approaches to types and type hierarchies that have been used in theories of grammar. Then we discuss how types can be used for disambiguation. The term ‘category’ sometimes is used for what we refer to as ‘type’.

2.2.1 General Organization of the Knowledge of Syntax

Grammarians have for a long time used the notion of syntactic categories, ones like *verbs*, *nouns*, *adjectives*, etc to organize their knowledge about the distribution and the role of the expressions that belong to each of these categories. Each category represents a group of *words* or *phrases* (which are valid combinations of words according to grammar rules), that have some common roles and distribution restrictions in grammatical sentences. If these categories are applied to words they are called *parts of speech* or *lexical categories*. On the other hand if they are applied to phrases they are called *phrasal categories*. A simple classification of syntactic categories is shown in figure 2.2.

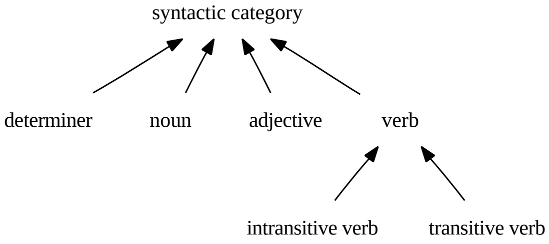


Figure 2.2: A simple classification of syntactic categories

This classification of expressions is used in a context free grammar (CFG) to form rules that combine words and phrases to build more complex phrases or eventually sentences. We shall use symbols D, A, N, IV, TV for determiner, adjective, noun, intransitive verb, and transitive verb categories respectively. NP is used for a noun phrase, VP for a verb phrase, and AP for an adjective phrase. Some CFG rules might look like:

S → NP VP
VP → IV

VP → TV NP
NP → D N
NP → D A N

These categories can be thought of as types, and diagrams like figure 2.2 are in fact type hierarchies for the domain of objects that are parts of speech in grammar.

Besides the use of type hierarchies in CFG, more recent theories of grammar, especially Head Phrase Structure Grammars (HPSG) heavily use an extensive type hierarchy for the organization of feature structures. An HPSG type hierarchy is similar, basically to a type hierarchy that organizes verbs, nouns, adjectives, determiners (as above). We shall see them in more detail in chapter 4.

2.2.2 Categorial Grammars and Typed Lambda Calculus

Categories (or types) in categorial grammars can be either syntactic or semantic. A semantic type is assigned to every syntactic category, to ensure that the syntactic category directly carries its meaning functionality [13] (We assume that there is a mapping from syntactic types to semantic types). In the lexicon, every lexical item is assigned to a syntactic category (type), and a meaning which is a λ-term of a type equal to the semantic type associated with the syntactic category of the lexical item. Phrases carry syntactic categories and meanings that are the result of the combining constituent syntactic categories and meanings. A phrase's meaning is a λ-term that is compatible with the semantic type associated with the syntactic category of that phrase. In what follows we briefly introduce these ideas.

Simply Typed λ-Calculus

Terms of a simply typed λ-calculus are called λ-terms. These terms are used for the semantic component of a categorial grammar. To this end they should enable us to denote relations between individuals (objects) as well as individuals. In particular, relations can hold or not hold among certain individuals, thus they have a truth value in $\{True, False\}$. Predicates and in general formulas of a first order logic language can be used to form λ-terms. **Ind** is defined to be the type of individuals and **Bool** is defined to be the type of propositions (that can be either $True$, or $False$)[5]. **Ind** and **Bool** are included in the set of *basic types*[6]. From basic types more complex types are built using the following definition. The fact that λ-terms in natural language semantics can refer in their simplest forms to individuals or truth values justifies the choice of **Ind** and **Bool** as the basic types. We denote the set of basic types by **BasTyp**.

Definition 2.5 λ-types

- Every basic type is a λ-type.
- If σ and τ are λ-types, then $\sigma \to \tau$ is also a λ-type, which is the type of a function that takes an argument of type σ and returns a value of type τ.

The set of types constructed this way is denoted by **Typ**.
Note that we define $\to$ to be right associative, i.e., $\sigma \to (\gamma \to \tau)$ can be abbreviated to $\sigma \to \gamma \to \tau$.

Definition 2.6 λ-Calculus Vocabulary[7]
A vocabulary of a λ-calculus language consists of

- a set of basic types **BasTyp**, by which a set of λ-types **Typ** is obtained using definition 2.5
- a collection of special symbols of *argument place holders* which we denote usually by symbols a_1, a_2, ..., each associated to a λ-type (we sometimes use the notation a_i^τ to explicitly show the type (in this case τ) of the argument place)
- an infinite set **Var** of variable symbols, such as $x, y, z, x_1, ...$ disjoint from argument place holders, where each variable is associated to a λ-type (we sometimes use the notation x^τ to explicitly show the type (in this case τ) of the variable)

[5] Here we use a notation similar to that of Carpenter [13]. Montague [55] uses e for individuals and t for truth values.

[6] Some authors like Gunter [40] use the term 'ground types' instead of 'basic types', and use 'higher types' to refer to types that are built from ground types using definition 2.5

[7] What we present in this definition is not used in Carpenter [13], but we believe the use of a λ-vocabulary helps us provide a more precise and formal definition of λ-terms and their relation to natural language semantics.

- for each natural number $n \geq 0$, a set of basic n-ary functions denoted by $\mathbf{BasFunc}_n$, of the form: $f^\tau(a_1^{\sigma_1}, ..., a_n^{\sigma_n})$ where f is a function symbol, each a_i is an argument place holder of type σ_i, and τ is the type of the return value of the function f.

Note: A functions of arity zero is called a constant.

Definition 2.7 λ-terms

- (i) every variable x^τ is a λ-term of type τ
- (ii) for every $f^\tau(a_1^{\sigma_1}, ..., a_n^{\sigma_n}) \in \mathbf{BasFunc}_n$,
  f is a λ-term of type $\sigma_1 \rightarrow ... \rightarrow \sigma_n \rightarrow \tau$.
- (iii) if α is a λ-term of type $\sigma \rightarrow \tau$ and β is a λ-term of type σ then $(\alpha(\beta))$ is a λ-term of type τ. (application rule)
- (iv) if α is a λ-term of type τ and if x is a variable of type σ then $\lambda x.\alpha$ is a λ-term of type $\sigma \rightarrow \tau$. (abstraction rule)

Remark 2.8 If a basic function $\mathtt{P}^{\mathbf{Bool}}(x_1^{\mathbf{Ind}}, ..., x_n^{\mathbf{Ind}})$ is a member of $\mathbf{BasFunc}_n$ of a λ-calculus, then $P(x_1, ..., x_n)$ can actually be thought of as a first order logic (FOL) n-ary predicate symbol. Note that formulas in FOL are in fact functions from individuals to truth values. We can use this remark to convert the usual FOL semantics of a natural language expression to its λ-calculus counterpart.

Definition 2.9 Free and Bound Variables
The set $Free(\alpha)$ of *free variables* occurring in the λ-term α is defined recursively by:

- $Free(x) = \{x\}$ if $x \in \mathbf{Var}$
- $Free(f) = \emptyset$ if $f \in \mathbf{BasFunc}_n$, for some $n \in \mathbb{N}$
- $Free(\alpha(\beta)) = Free(\alpha) \cup Free(\beta)$
- $Free(\lambda x.\alpha) = Free(\alpha) - \{x\}$

A variable is said to be *bound* in a λ-term if it is not free.

Definition 2.10 Simple Substitution
For any λ-terms α, y, and any variable x,
$\alpha.[x \mapsto y]$ is a λ-term that is resulted from replacing any occurrence of y with x in α.

Definition 2.11 Reduction Rules
λ-terms can be transformed to simpler but logically equivalent terms. A transformation of this kind is called a *reduction.* Reductions are possible through the following *reduction rules*:

- (α-reduction) : $\lambda x.\alpha \Rightarrow \lambda y.(\alpha.[x \mapsto y])$
  if y is not a free variable in α and none of the free variables of y become bound when y is substituted for x in α.
- (β-reduction) : $(\lambda x.\alpha)(\beta) \Rightarrow \alpha.[x \mapsto \beta]$
  if none of the free variables of β become bound when β is substituted for x in α.
- (η-reduction) : $\lambda x.(\alpha(x)) \Rightarrow \alpha$
  if x is not a free variable of α.

Example 2.12 Suppose our vocabulary's $\mathbf{BasFunc}_0$, $\mathbf{BasFunc}_2$ are respectively
$\{\texttt{john}^{\mathbf{Ind}}, \texttt{mary}^{\mathbf{Ind}}\}$
$\{\texttt{likes}^{\mathbf{Bool}}(a_1^{\mathbf{Ind}}, a_2^{\mathbf{Ind}})\}$.

The first argument of `likes` has the role of the person (or the object) being liked, and the second argument has the role of the person who likes what is referred to by the first argument.

By applications of part (ii) of definition 2.7 we obtain that the following are λ-terms:
`john`,
`mary`,
`likes`

respectively of the following types:
Ind
Ind
$\mathbf{Ind} \to \mathbf{Ind} \to \mathbf{Bool}$

By applications of (iii) in definition 2.7, we obtain that:

$(\texttt{likes}(\texttt{mary}))$

is a λ-term of type $\mathbf{Ind} \to \mathbf{Bool}$.

This could represent that `mary` is liked. However, who likes her is underspecified.

By application of the same rule we obtain that:

$((\texttt{likes}(\texttt{mary}))(\texttt{john}))$

is a λ-term of type **Bool**. This could be used to represent the semantics of the natural language sentence:

John likes Mary.

The Category System

Here by a category we are refer to a syntactic category, as we discussed about in section 2.2.1. Like λ-calculus we start from a set **BasCat** of basic categories. Basic categories are usually chosen NP (for noun phrase), N (for noun), S (for sentence).

Definition 2.13 Categories
The set **Cat** of categories is built from the set of basic categories using the following rules:

- (i) Every basic category A is a member of **Cat**
- (ii) If $A, B \in \mathbf{Cat}$, then $(A/B), (B\backslash A) \in \mathbf{Cat}$

A/B is the syntactical category of a natural language expression that can be combined from the right with another expression of syntactic category B to produce a phrase of syntactic category A. For example NP/N is the category of determiners, meaning that a determiner can be combined with a noun to its right to form a noun phrase. Similarly, $NP\backslash S$ is the category of intransitive verbs, as they could combine with a noun phrase from the left to produce a sentence.

As said before, we assume each syntactic category has a direct relation to its semantic functionality, specifically, the λ-type of the meanings its expressions can carry. So every category is associated to a λ-type. This association is the responsibility of what is called *type assignment function*. Here we define the type assignment in two stages. In the first stage, every basic category is associated to a proper type by the grammarian, and in the second stage, the types of complex categories (those that are built by the application of (ii) in definition 2.13) are defined.

Definition 2.14 Basic Type Assignment
The *basic type assignment* is a function bta : $\mathbf{BasCat} \mapsto \mathbf{Typ}$, that should be provided in the grammar.

One possible basic type assignment can be:

$$\begin{aligned} bta(NP) &= \mathbf{Ind} \\ bta(N) &= \mathbf{Ind} \rightarrow \mathbf{Bool} \\ bta(S) &= \mathbf{Bool} \end{aligned}$$

Definition 2.15 Type Assignment
The *type assignment* is a function ta : $\mathbf{Cat} \mapsto \mathbf{Typ}$, that is defined this way:

- $ta(X) = bta(X)$ if $X \in \mathbf{BasCat}$.
- $ta(A/B) = ta(B\backslash A) = ta(B) \rightarrow ta(A)$

The Categorial Lexicon

The *categorial lexicon* is the first place where the all this information about basic expressions of a language comes together:

- orthographic information (the way the basic expression is written)
- the syntactic category that the basic expression belongs to
- the meaning of the basic expression (a λ-term)

Basic expressions are the simplest expressions for which the above information is provided, and from which more complex expressions can be built using the rules that we will see later in this section.

Definition 2.16 Categorial Lexical Entries
A *lexical entry* of a categorial lexicon is a tuple $< e, A, \alpha >$, where e is a basic expression, A is the syntactic category that the expression belongs to, and α is the meaning that is associated to the expression. The meaning must be a λ-term of type $ta(A)$.
Here we use the following notation to express the fact that the above tuple is in the lexicon. This comes handy when we use the grammar to parse phrases, and to determine their meaning.

$Lexicon \Rightarrow e \equiv (\alpha, A)$

Generally we use $e \equiv (\alpha, A)^8$ to denote that e is an expression of syntactic category A with meaning α. In the next definitions we show how phrases (complex expressions) can be built from other expressions.

Now we state the two most important grammar rules of a categorial grammar, namely forward application and backward application, known as *application schemes*.

Definition 2.17 Application Schemes

Forward application : $e_1 \equiv (\alpha, A/B), e_2 \equiv (\beta, B) \Rightarrow e_1 \oplus e_2 \equiv (\alpha(\beta), A)$
Backward application : $e_2 \equiv (\beta, B), e_1 \equiv (\alpha, B\backslash A) \Rightarrow e_2 \oplus e_1 \equiv (\alpha(\beta), A)$

where $\oplus$ is the concatenation operation.

Definition 2.18 Phrases
If an expression of the grammar is not in the lexicon, but can be built from grammar rules, then it is called a *phrase*.

Using the information in the lexicon and the application schemes we can parse complex phrases in a bottom-up fashion. As a phrase is constructed using application schemes, the meanings of the constituents are combined to form the meaning of the phrase, this is referred to as *semantic compositionality*.

Example 2.19 Suppose the lexicon contains these items:

$< \text{“}\texttt{john}\text{”}, NP, \texttt{john} >$
$< \text{“}\texttt{mary}\text{”}, NP, \texttt{mary} >$
$< \text{“}\texttt{likes}\text{”}, (NP\backslash S)/NP, \lambda x \lambda y.\texttt{likes}(x, y) >$

then we will have:

$Lexicon \Rightarrow \text{“}\texttt{mary}\text{”} \equiv (\texttt{mary}, NP) \quad (*1)$
$Lexicon \Rightarrow \text{“}\texttt{likes}\text{”} \equiv (\lambda x \lambda y.\texttt{likes}(x, y), (NP\backslash S)/NP) \quad (*2)$
$forward\ application : (*1), (*2) \Rightarrow$
$\quad \text{“}\texttt{likes mary}\text{”} \equiv ((\lambda x \lambda y.\texttt{likes}(x, y)(\texttt{mary})), NP\backslash S) \quad (*3)$

[8] This is somewhat different from the notation that Carpenter [13] employs, in which the orthographical information of the expression, what we denoted by e, is omitted after the first application of a rule that produces a phrase. However, we should mention that Carpenter brings the expression e, with the meaning and syntactic category together in phrase structures. We use $\equiv$ instead of $:$ because we already use $:$ for instance relation.

$\textit{β-reduction} : (*3) \Rightarrow$ “$\texttt{likes mary}$” $\equiv (\lambda y.\texttt{likes}(\texttt{mary}, y), NP\backslash S) \quad (*4)$

$\textit{Lexicon} \Rightarrow$ “$\texttt{john}$” $\equiv (\texttt{john}, NP) \quad (*5)$
$\textit{backward application} : (*5), (*4) \Rightarrow$
“$\texttt{john likes mary}$” $\equiv ((\lambda y.\texttt{likes}(\texttt{mary}, y)(\texttt{john})), S) \quad (*6)$

$\textit{β-reduction} : (*6) \Rightarrow$ “$\texttt{john likes mary}$” $\equiv (\texttt{likes}(\texttt{mary}, \texttt{john}), S) \quad (*7)$

A Limitation of CG with Simply Typed λ-Calculus

Categorial grammar (CG) provides a subtle connection between syntax and semantics and provides a comprehensive type system for semantic and syntactic categories. However, the type system was not originally designed to deal with the type hierarchies that exist among individuals. In other words no elaborate type system is provided for **Ind**, the individuals.

It is claimed in [13][9] that the type system used in CG can be extended to a sophisticated type system employing inheritance-based polymorphism. Next we will very briefly study one possible such type system.

2.2.3 λ-Calculus with Subtyping

By *polymorphism* we mean that a λ-term can be of multiple types. *Inheritance-based polymorphism* is the polymorphism that results from type hierarchies in the type system. This happens if there is a type hierarchy below the type **Ind**.

Example 2.20 Back to the type hierarchy provided in figure 2.1, if for example $\texttt{tiqa}$ is a *beluga*, then at the same time she is a *mammal*, and an *animal*. The relevant information is:

$\textit{animal} \sqsubseteq \mathbf{Ind}$
$\textit{mammal} \sqsubseteq \textit{animal}$
$\textit{beluga} \sqsubseteq \textit{mammal}$
$\textit{Tiqa} : \textit{beluga}$

Now if the intransitive verb *breathes* is defined to be a basic function of the form:

$\texttt{breathes}^{\mathbf{Bool}}(a_1^{mammal})$

then $\texttt{breathes}$ will be a λ-term of type: $\textit{mammal} \rightarrow \mathbf{Bool}$. But this leaves us in a situation

[9] Although a specific type system is not suggested.

where the following two terms cannot combine together to form the phrase *"Tiqa breathes"*.

$$\begin{cases} \texttt{tiqa} : \mathit{beluga} \\ \lambda x.\texttt{breathes}(x) : \mathit{mammal} \rightarrow \mathbf{Bool} \end{cases}$$

The reason for this is that the application rule (item (iii) of definition 2.7) expects the same type for the expected argument of the function (*mammal*), and the actual argument (*beluga*).

To solve the problem shown above, together with other problems, some additional axioms are provided in extensions of λ-calculus [54, 43, 42, 58, 62, 26]. Some of the important ones are shown below.

Subtype Axioms

$$\begin{array}{ll} \tau \sqsubseteq \tau & : (\mathit{REFL}) \\ (\sigma \sqsubseteq \tau) \wedge (\tau \sqsubseteq \zeta) \Rightarrow \sigma \sqsubseteq \zeta & : (\mathit{TRANS}) \\ (\tau_1' \sqsubseteq \tau_1) \wedge (\tau_2 \sqsubseteq \tau_2') \Rightarrow \tau_1 \rightarrow \tau_2 \sqsubseteq \tau_1' \rightarrow \tau_2' & : (\mathit{ARROW}) \\ (x : \sigma) \wedge (\sigma \sqsubseteq \tau) \Rightarrow x : \tau & : (\mathit{SUB}) \end{array}$$

Applying the axiom (*SUB*) on example 2.20 allows us to infer that

$\texttt{tiqa} : \mathit{mammal}$

which then allows the application rule to derive:

$(\lambda x.\texttt{breathes}(x)(\texttt{tiqa}))$

that using β-reduction simplifies to:

$\texttt{breathes}(\texttt{tiqa})$

An application of (*SUB*) is called a *subtype coercion*, which is one case of the more general *type coercion* [63].

Definition 2.21 Type Coercion, and Type Shifting Operators

Pustejovsky [63] defines type coercion to be a semantic operation that converts an argument to the type which is expected by a function, where it would otherwise result in a type error. This conversion is not arbitrary and is according to an available set of *shifting operators*. A type shifting operator takes an instance $x : \tau_1$ and outputs $x : \tau_2$. Type shifting operators are all type axioms.[10]

In programming languages, axiom (SUB) is used when a function (or procedure) requiring an argument of some type τ is applied to an argument of a type σ lower in the type hierarchy but in the same branch as τ, to temporarily change its type to τ. In object oriented programming languages, an application of (SUB) to an object, is called *up-casting*. If (SUB) is applied to a variable of a simple type, for example to an integer number when a function expecting a real number is applied to it, it is called *coercion*.[11]

2.2.4 Semantic Selection Restrictions and Types

Katz and Fodor in their influential paper [47][12] about the characteristics of an acceptable semantic framework of natural language show how the constituents of a natural language phrase impose semantic requirements on each other. These requirements are called *semantic selection restrictions*. For some expressions to combine and form a semantically valid phrase, it is necessary that the restrictions that each expression impose on the rest be satisfied. As far as the semantics is concerned (and not pragmatics), it is the interaction of these restrictions that determine whether a semantic reading of a syntactically well-formed phrase is acceptable or not. A syntactically well-formed phrase with no plausible semantic reading is said to be 'odd' or 'peculiar', ... or as we call it here, 'semantically invalid'.

According to Katz and Postal [48], "... each reading in the dictionary entry for a lexical item must contain a *selection restriction*, i.e., a formally expressed necessary and sufficient condition for that reading to combine with others".[13]

Pustejovsky [63] elaborates on the notion of semantic selection restrictions by developing a semantic type system which employs type coercions, and types that are used in the argument structure of lexical items. His work goes further by applying event structures, qualia structures and lexical inheritance structures whose exploration is beyond the scope of this thesis.

[10] Later we will see there are some type axioms like (NEG) that do not only output $x : \tau_2$, but also output formulas like $\neg(x : \tau_3)$.

[11] Coercion is not specific to object oriented programming languages, it is used in the compilers of non-OOP languages like FORTRAN, Pascal, and C.

[12] For a critical analysis of the theory of Katz and Fodor the interested reader can refer to the second chapter of [73].

[13] For a more recent discussion on semantic selection restriction the interested reader can refer to section 4.2.6 of [27].

Although the notion of semantic selection restrictions is not new and has been applied for decades, a satisfactory type system suited for bridging the linguistics and the application domain model has not yet been proposed. On one side we have domain entity type hierarchies, with the subtype relation and the need of multiple inheritance as we will see in section 2.5, where every entity is associated with a domain type. And on the other side we have the noun phrases that should carry their semantics in the grammar, while the exact domain type of the entity that a pronoun is referring to is not known beforehand. The type system that we develop in this thesis will bridge the gap between the grammar and the application. In what follows we will provide some introductory examples and define type restrictions that can be used for detecting a type mismatch or for disambiguation.

Example 2.22 Consider the sentence:

(1) * The paint is silent.

Native English speakers immediately find the sentence odd. One can justify this peculiarity by saying that the adjective *silent* is normally used to modify noun phrases that represent objects in the world that are in some well-known way associable with sound. The association can be the capability of generating sound (like *the silent man*), or the capability of containing other objects generating sound (like *the silent room*), or the capability of co-occurring (or being synchronized) with sound (like *a silent film*, or *a silent night*). However, *paint* is not in any usual setting associable with sound, so the use of the adjective *silent* is inappropriate with the noun phrase *paint.*

On the other hand this sentence is acceptable by native English speakers:

(2) The paint is wet.

The reason is that adjective *wet* in common settings modifies a noun phrase that refers to a physical object, which is associable with wetness.

An *ambiguous* word or phrase[14] is one that has multiple semantic readings or *senses*. Ambiguity of phrases may root from the ambiguity in the words that build the phrase (there are other types of ambiguity). If a word has multiple senses, then each sense has its own semantic selection restrictions on other expressions in the containing phrase. Now

[14] As we later see in chapter 5, a sentence is a saturated verb phrase.

if such a word is used in a syntactically well-formed phrase, then the selection restrictions of each of the senses of that word on other expressions in the phrase and vice versa, helps the interpreter choose the proper sense(s) of that word in the phrase, and thus reduce the number of acceptable readings of the phrase. So, the semantic selection restrictions of the words may help us *disambiguate* the phrase.[15]

Example 2.23 The verb *support* has at least two senses. One sense, s_1, is related to the relative physical situation of some physical objects as used in (3a). Another sense, s_2, is related to some human objects who agree with the objectives of some project as used in (3b).[16]

(3) a. The blue block supports the pyramid.

b. A democrat supports the stem cell research.

Suppose the semantic selection restrictions of the senses s_1, and s_2 are given in (4), and (5) respectively.

(4) the subject and the object of the verb must be physical objects.

(5) the subject of the verb must be a human or a group of humans, and the object of the verb must be a project.

Since *stem cell research* is not a physical object, (4) is not satisfied and hence meaning s_1 cannot be chosen for sentence (3b). On the other hand, since *the blue block* is not a human or a group of humans (5) is not satisfied and meaning s_2 cannot be used for (3a).

Type Restrictions

Some of the semantic selection restrictions can be defined in terms of the expected type of noun phrases that participate in a phrase with reference to a type hierarchy. We call such semantic selection restrictions, *type restrictions*. Using the class hierarchy of figure 2.1 restriction (4) is obviously saying that the noun phrases in subject and object positions must refer to objects of type *physical*. And restriction (5) is saying that the noun phrase

[15] Sometimes this is not enough, as in : "I love Georgia", where "Georgia" could be either a place or a person.

[16] A very similar example exists in Winograd [75] about disambiguation.

occupying the subject position must refer to an object of type *human* and the noun phrase occupying the object position must refer to an object of type *project*.

Definition 2.24 Type Restrictions[17]

A type restriction of a sense s of a phrase can be represented by a set TR_s of ordered pairs $(role, type)$, where *role* is the grammatical role (such as *subject*, *object*, *complement*, *modified nominal*, ...) of an expression ξ in the phrase, and *type* is the expected type of the object that ξ refers to. Basically the type restriction of an expression restricts the type of the arguments of that expression.

Example 2.25 Type restrictions of the senses s_1 and s_2 in example 2.23 are:
$TR_{s_1} = \{(subj, physical), (obj, physical)\}$
$TR_{s_2} = \{(subj, human), (obj, project)\}$

Definition 2.26 Type Restriction Satisfaction (Version I)[18]

A type restriction TR of a sense for an expression is satisfied in a phrase if and only if for every pair $(role, type) \in TR$, the expression ξ that assumes the grammatical role *role* in the phrase, refers to an object of type τ where we have either:

- $\tau = type$
- there is a shifting operator that shifts $\xi : \tau$ to $\xi : type$

Using the knowledge of type hierarchies and selection restrictions in a phrase to disambiguate it is what Pustejovsky [63] calls *sortally constrained disambiguation*. However one should note that satisfying a type restriction might involve applying type shifting operators to the types of noun phrases to convert them to the types that the restriction expects. In the next section we will see that subtype coercion (SUB) is not the only semantic type shifting operator that we need in natural language.

2.3 The Need for a Specialization Shifting Operator

In section 2.3.1 we describe a problem with the types that we associate with pronouns in the lexicon and we see that the type axioms so far are not sufficient to address it. The incomplete

[17] It is noteworthy that a concept similar to what we called type restrictions is used in the SHADOW system [41].

[18] We will revise this definition in chapter 3

types proposed by Dahl [23] provides a solution to this problem using the unification facility of logic programming languages such as Prolog. Before we investigate incomplete types, we present in section 2.3.2 an abstract solution to the problem by introducing *specializable types* with a new type shifting operator ($SPECIAL$). This helps us fit incomplete types into the type theory we have developed so far. In section 2.3.3 we present two other possible applications of specializable types. Then in section 2.4 we investigate how unification can act as a specializing shifting operator that can handle ($SPECIAL$).

2.3.1 A Problem with Pronouns

Suppose we add a new type *dancer* to our type hierarchy under the type *human*:

(6) $dancer \sqsubseteq human$

Also suppose that we do not distinguish between female and male humans for simplicity. Now consider the pronoun *she*. We should assign a semantic type to this pronoun, and the only proper type in the type hierarchy is *human*:

(7) $she : human$

Suppose the verb *dance* has only one sense with the type restriction:

(8) $TR_{dance} = \{(subj, dancer)\}$

that is the subject of the verb *dance* must be a noun phrase that refers to an object of type *dancer*. So for sentence (9) to be semantically valid it is necessary that the type restriction of the verb *dance* in the sentence be satisfied.

(9) she dances.

Since *she* refers to an object of type *human*, the satisfaction of TR_{dance} requires that either $human = dancer$ which is not the case, or there should be a type shifting operator that outputs $she : dancer$. The only type shifting operator that we have so far is (SUB) that shifts the type of an object some levels higher in the type hierarchy. However, *dancer* is lower in our type hierarchy than *human*. So with (SUB) as our only type shifting operator, sentence (9) is not recognized as valid.

The problem is with assumption (7). The fact is that *she* not only can refer to an object of type *human* but also it can refer to an object of a type that is anywhere below *human* in the type hierarchy. I.e.,

(10) *she* : τ, where $\tau \sqsubseteq$ *human*

This will lead to multiple type declarations for *she*, including:

(11) a. *she* : *human*

b. *she* : *dancer*

and it will be possible for *dance* to choose type declaration (11b) to satisfy its type restriction.

However, this requires that we have multiple lexical items for *she* each with a different type. Note that the number of type declarations that we need would be the number of subtypes that *human* has in the type hierarchy because of (10). This is equal to the number of nodes that the subtree rooted by *human* has in the type hierarchy. This number can be large in a type hierarchy of a real natural language application. Also this problem is shared not only by *she* but also by any pronoun (and we will shortly see it is shared by some other nouns too). This means our lexicon will have a lot of entries for the same words with only different semantic types. This will pose a problem with the efficiency of the parser implementation.

Also this method does not obey modularity of the design principles, that is every time a new type is added to the type hierarchy below a node in the subtree rooted by *human*, we have to add a new lexical item for *she* and all pronouns referring to humans with their semantic type equal to the new type.

2.3.2 The Solution : Introducing Specializable Types

An alternative to declaring multiple entries for a word in the lexicon to handle the kind of problem described in the previous section, is to expand the set of basic types by introducing a type in form of a function *spec*(.)[19] that takes an input type and outputs a type that is a *specializable type* below the input type. A specializable type below a root type τ denoted by *spec*(τ) is a type that can be *specialized* or shifted to a subtype of τ when required by the type shifting operator ($SPECIAL$) given below:

[19] The formal definition of this function is provided in chapter 3, definition 3.15, where we define specializable types on top of natural types. There we provide another type axiom that accompanies specializable types.

$$\forall \tau \in \mathbf{BasTyp}\ \forall \sigma \in \mathbf{BasTyp}\ \forall x.\ x : spec(\tau)\ \wedge\ (\sigma \sqsubseteq \tau) \Rightarrow x : \sigma \qquad : (SPECIAL)$$

Using this, we can declare the type of an object that *she* refers to, to be a specializable type under *human*, that is:

(12) *she* : *spec*(*human*)

and the sentence *she dances* can now be recognized as semantically valid, because the type restriction of the verb *dance* can now be satisfied using a type coercion with operator $(SPECIAL)$ that shifts *human* to *dancer*.

In Object Oriented Programming languages applying $(SPECIAL)$ to an object of a higher type in the type hierarchy to cast it as an object of a lower type in the hierarchy is called *down-casting*. However the use of this operator is not automated in compilers of languages such as C++, Java, and C#, and down-casting must be strictly forced by the programmer. Often usage of down-casting in programming is not encouraged.

2.3.3 Other Possible Applications of Specializable Types

Besides their usefulness in dealing with pronouns, specializable types can be helpful in other situations too. Here we present two more cases, general classes, and proper nouns. These applications however might not be very intuitive, and care must be taken before using them.

General Classes

We define a *general class* to be a class whose instances are instances of one of its subclasses. For example, the type *animal* is a general class, because every *animal* instance should be a *mammal*, or a *bird*, or a *fish*, etc. In other words we don't have an object that is classified as an *animal* but does not belong to any subclass of *animal*.[20]

There is a connection between what we here call a general class and what in Object Oriented Programming languages is called an *abstract class*. In Object Oriented Programming languages, a class A is said to be an *abstract class* if any instance of type A is an instance of a subtype of A. That is A has no direct instance.

[20] Later in chapter 3 we see that an example of a general class is a type which is partitioned into its subtypes.

Despite this similarity, there is a difference. What we present below allows a general class to be arbitrarily shifted to a lower type in the type hierarchy. This is not allowed in modern OOP languages such as C++, Java, and C#, unless the programmer forces it by an explicit down-casting operator. The use of general classes arbitrarily might not be the best option if the grammar is going to be used in requirement analysis. However, it is needed for some cases. For example if we would like sentences like (13) to be acceptable. Or if we need queries like (14) to be analyzable.

(13) The animal barks.

(14) Does an animal bark?

Suppose our type hierarchy below *mammal* now contains the type *dog* as shown in figure 2.3. Also suppose that our lexicon now contains the verb *barks* with the type restriction:

(15) $TR_{barks} = \{(subj, dog)\}$

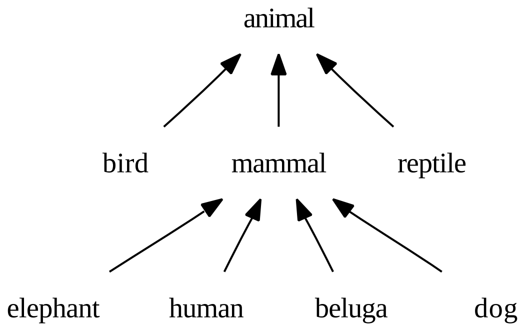


Figure 2.3: dog is now in the type hierarchy

Every noun phrase must be associated with a type in our grammar. *Animal* used as a word refers to any object that is an instance of type *animal*. One possible choice of type for *animal* could be:

(16) $animal : animal$

But with this declaration, sentence (13) cannot be recognized as semantically valid, because the type restriction of the verb *barks* requires the subject to refer to an object of type *dog.* One might argue that this sentence is legitimately ruled out, because not all animals bark, and the verb *bark* is not suitable for the subject *the animal.* Ruling out such cases can be desirable in software requirements, because sentences in software requirements should be as precise as possible, and a sentence like (13) could have the implicit interpretation that *barking* is a valid behavior of all animals.

However with the presence of *dog* in the type hierarchy, sentence (13) can have a possible semantic reading that *the animal* refers to a dog that barks. To allow for this semantic reading we can declare the word *animal* to refer to an object of a specializable type below *animal*, that is:

(17) $animal : spec(animal)$

Note that by definition any instance of a general class is an instance of one of its subclasses. So the definition above make sense, as it binds the object that *animal* refers to, to a subtype of *animal* in the type hierarchy.

This will satisfy the type restriction of the verb *bark.*

Proper Nouns with Incomplete Type Information

In requirement specification we should try to provide all details about a name (a proper noun) that is used in the requirement. However in natural language it is acceptable that the type information of an object that is referred to by a proper noun is gathered gradually in steps. For example sentence (18a) gives some visual and/or physical information about *Mary*, and in sentence (18b) more information about the type of that person is provided, that is *Mary* is a dancer.

(18) a. This is Mary.

b. Mary dances.

An important issue with proper nouns is how they should be defined in the lexicon, more specifically what type should be assigned to them. For some proper nouns like *Mary* we know that they refer to a human object, but exactly what subclass of *human Mary* belongs to could be unknown at the time we define the lexicon for some applications. If our knowledge about what exact object the proper noun refers to at the time of its definition in the lexicon is incomplete, then we can use specializable types. For example we can declare:

(19) $mary : spec(human)$

and with this declaration, the type restriction of the verb *dance* in sentence (18b) will be satisfied after an application of $(SPECIAL)$.

2.4 Incomplete Types : An Implementation of Specializable Types through Unification

Dahl [23] proposed the hierarchical representation of types using first order logic terms in Prolog, and unification between the hierarchical representations to check their agreement. Type agreement was used to check what in this chapter we have called the 'type restrictions' of expressions. The work in [23] addresses the encoding of a taxonomy that has a tree shaped hierarchy. This method was extended by Mellish in [51, 53] where it is possible to encode some more general taxonomies that are no longer tree shaped, but are in the form of graphs, or more specifically, directed acyclic graphs (DAGs). These DAG shaped hierarchies[21] allow types that have more than one parent in the hierarchy. This is called multiple inheritance, which we will discuss in subsection 2.5. We discuss a little about these and other extensions in subsection 2.5.1.

This section is organized as follows. First, in subsection 2.4.1 we show how FOL (and Prolog) terms can be used to form a hierarchical representation of types, and how unification can be used to check their agreement. In subsection 2.4.2 we mention how Prolog lists can be used for hierarchical type representations. Then in subsection 2.4.3 we show how incomplete types can be used to deal with $(SPECIAL)$ in satisfying type restrictions, and we discuss whether (SUB) can also be handled by incomplete types.

2.4.1 Hierarchical Representation of Types

Hierarchical representations of types are terms in a first order logic. The vocabulary of the first order logic consists of:

- *nil* as a constant symbol

[21] We still call these DAGs, type hierarchies, as done in [31] too, because the concept of being lower, or higher among types, still exists in DAGs. If a type is lower in rank than another, it is a descendant and a subtype of that type.

- every node label in the type hierarchy as a constant symbol
- & as a binary infix function symbol, which is left-associative
- an infinite number of variable symbols beginning with an uppercase letter, e.g., $X, Y, Z, V, \ldots$

For our type hierarchy shown in figure 2.4 the following constant symbols are used for each node:

$$all,\ physical,\ animal,\ bird,\ mammal,\ elephant, \ldots$$

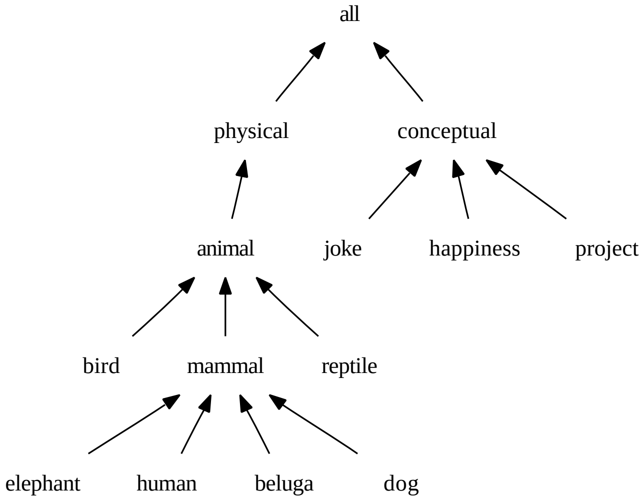


Figure 2.4: Our example type hierarchy

The infix left-associative function symbol & is used to connect the nodes in the path from the top to node τ to form an FOL expression that is the hierarchical representation of type τ. We call such expressions, *path expressions*[22]. For example in the type hierarchy of figure 2.4,

$animal \rightarrow physical \rightarrow all$

is a path, which corresponds to the path expression:

$((nil \ \& \ animal) \ \& \ physical) \ \& \ all.$

[22] This is what we call them in this section to define the hierarchical representation formally, the term 'path expressions' is not used in the original work.

In the following definition path expressions are defined more formally.

Definition 2.27 Path Expressions

- *nil* & n is a path expression for the trivial path from node n to n.
- V & n is a path expression of any path from node n downwards in the hierarchy to an underspecified node (which could be n itself)
- if P_n is the path expression of a path π from node n downwards in the hierarchy, and m is the parent of n in the hierarchy, then P_n & m is the path expression for the concatenation of the path from node m to node n and π.

Definition 2.28 Hierarchical Representation of Type τ
The hierarchical representation of type τ is the path expression of the path from the top node *all* downwards to the node τ.

For example the hierarchical representation for type *mammal* is:

$$(((nil \ \& \ mammal) \ \& \ animal) \ \& \ physical) \ \& \ all$$
$$=$$
$$nil \ \& \ mammal \ \& \ animal \ \& \ physical \ \& \ all$$

Definition 2.29 Hierarchical Representation of a Specializable Type below τ_1
The hierarchical representation of type *spec*(τ_1) is the path expression of a path from the top node *all* to an underspecified node in the hierarchy through the node τ_1, which is equal to:

$$(V \ \& \ \tau_1) \ \& \ \tau_2 \ \& \ ... \ \& \ \tau_n$$

where V is a variable not used anywhere else, and τ_{i+1} is the parent node of τ_i in the hierarchy, and $\tau_n = all$.

Type agreement is checked through unification. Informally, two terms unify if they are exactly equal, or if they contain variables that can be instantiated in such a way that the resulting terms are equal [9]. If the unification succeeds the variable instantiations will persist.

For example, the following two terms unify:

(20) a. $nil \,\&\, mammal \,\&\, animal \,\&\, physical \,\&\, all$

b. $V \,\&\, animal \,\&\, physical \,\&\, all$

because variable V can be instantiated to $nil \,\&\, mammal$ that makes both terms equal. But the following terms cannot unify:

(21) a. $V \,\&\, animal \,\&\, physical \,\&\, all$

b. $nil \,\&\, project \,\&\, conceptual \,\&\, all$

because no matter what value is assigned to variable V in (21a) the resulting term cannot be equal to (21b).

2.4.2 Using Prolog Lists to Represent Types

The notation used in Dahl [23] is easily convertible to Prolog lists. A list in Prolog is a term that represents a sequence of terms. A list can be decomposed into a head element, which is the first element in the sequence and a tail list, which is the remainder of the sequence after the first element is removed. A list in Prolog can be specified by listing the elements between square brackets. For example the list of the first three natural numbers is:

(22) `[0, 1, 2]`

In Prolog there is a special symbol '|' that can be used in the bracketed notation of the list to separate the head from the tail. $[H|T]$ is a list with head H and tail T. Function symbol '.' is actually the function & with inverted order of arguments and `[]` is *nil* in Dahl [23]. The bracketed notation is used in Dahl [24]. For example

(23) $V \,\&\, mammal \,\&\, animal \,\&\, physical \,\&\, all$

is equivalent to

(24) `[`$all,\ physical,\ animal,\ mammal \mid V$`]`

2.4.3 Incomplete Types and Satisfying Type Restrictions with $(SPECIAL)$

Now if we go back to the problem we posed in subsection 2.3.1, with the formula $dancer \sqsubseteq human$ incorporated into our type hierarchy as shown in figure 2.5, then using incomplete types the type restriction of the verb *dance*, and the lexicon declaration of the pronoun *she* can be given by:

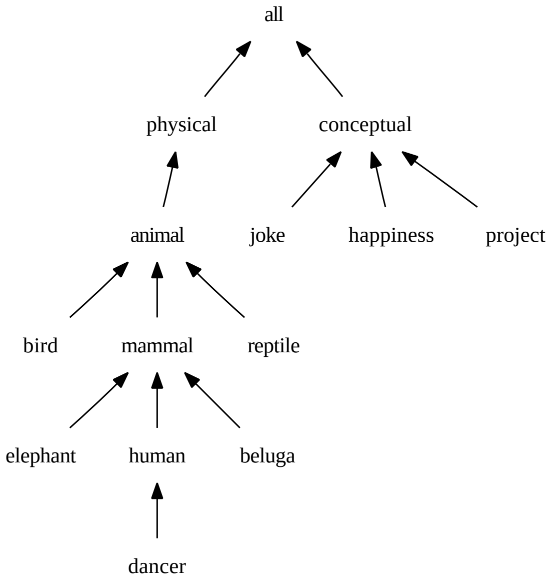


Figure 2.5: Using incomplete types with pronouns

(25) $TR_{dance} = \{(subj, [all,\ physical,\ animal,\ mammal,\ human,\ dancer \mid X])\}$

(26) $she : [all,\ physical,\ animal,\ mammal,\ human \mid V]$

Then the unification operator binds V to $[dancer \mid X]$, and the type restriction of *dance* will be satisfied with no further hassle. Actually unification acts as $(SPECIAL)$ silently and automatically.

On the other hand (SUB) still needs to be processed as the element of the incomplete list cannot be removed to make the corresponding type more general.

Revisiting example 2.20 but this time with incomplete types, we can write the type restriction of the verb *breathes* as:

(27) $TR_{breathes} = \{(subj, [all,\ physical,\ animal,\ mammal \mid X])\}$

If we declare *tiqa* in the lexicon by:

(28) $tiqa : [all,\ physical,\ animal,\ mammal,\ beluga \mid V]$

then the type restriction of the verb *breathes* is satisfied with the binding:

(29) $X = [beluga \mid V]$

and the sentence *tiqa breathes* can be recognized.

However, this method does not formally fit into the framework that we have been developing so far. In the next chapter we see how we can benefit from some features of incomplete types while completely being in a formal framework that can be directly applied to $\lambda-$calculus and HPSG-like grammars to allow domain specific type hierarchies.

2.5 Multiple Inheritance

Multiple inheritance is a term used for types that have multiple parents in the type hierarchy. A type that inherits from more than one super-type, is a *composite type* that combines the semantic behavior and characteristics of its super-types. *Single inheritance* on the other hand is a term that used for a type that only has one parent in the type hierarchy. In Object Oriented Programming languages, specially after a powerful OOP language, C++, came into existence with the possibility to define classes that could inherit from multiple base classes, there was a long debate till the mid 90's about the benefits and problems of multiple inheritance. Singh [69] outlines some of the discussions around multiple inheritance. As a result newer OOP languages like Java and C# make more careful use of multiple inheritance [28, 71]. They allow multiple inheritance only at the *interface level*. An interface is a type in newer OOP languages that only declares the supported behaviors without any implementation. In other words an interface is a *protocol* of how an object of that type can interact with other objects.[23].

The idea of multiple inheritance in some cases is so natural that some researchers have tried to model it in languages that do not easily allow it (like Java, and C#). An elegant example is the work presented in Mössenböck [56]. By using taxonomies that use multiple inheritance information can be represented in a more natural, efficient, and compact way.

One should note that if multiple inheritance is used, the type hierarchy diagrams will not be tree shaped any more. The type diagram will be a directed acyclic graph (DAG). We will still call the taxonomy a type hierarchy, because the notion of being lower or higher in rank still exists, because in DAGs there are no cycles and the rank ordering is maintained. If a type is lower in rank than another it is a subtype of that type. If from the context it

[23] For more information about interfaces see [28].

is not clear whether the hierarchy is tree shaped or DAG shaped, we will mention exactly which kind it is.

Example 2.30 An Example : The University Bookstore
Here we describe the need for multiple inheritance by using a hypothetical software project in which a software team is assigned with a task of modeling the business process of a university bookstore. The entities of interest in such a project are usually *customers*, *students*, *text books*, *amount of money*, and so on. One possible type hierarchy for these entities is shown in figure 2.6.

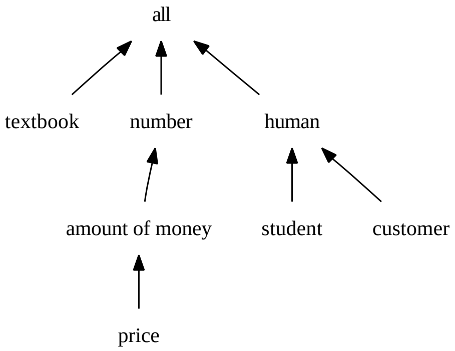


Figure 2.6: A typical type hierarchy for a bookstore

One business rule of this bookstore applied when the customer is paying for the items she/he purchased is:

(30) If the customer is a student, then the customer pays %90 of the total price.

This implicitly means that some students can be customers too. Such students are instances of a new type that is a subtype of both *student*, and *customer*. This type is denoted by *student*customer* and is shown in the type hierarchy of figure 2.7.

Notation 2.31 Asterisk Operator for Multiple Inheritance
We generally use the asterisk operator '*' for combining two types to create a type that is a subtype of both. We will later define this operator formally in definition 3.7.

Example 2.32 Another Example : Dancer and Writer
Suppose in the type hierarchy the node *human* has two children *dancer* and *writer* as shown in figure 2.8.

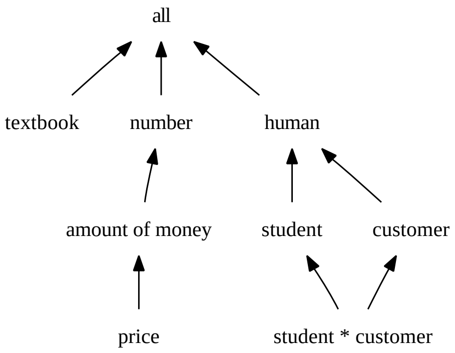


Figure 2.7: bookstore DAG shaped type hierarchy with multiple inheritance

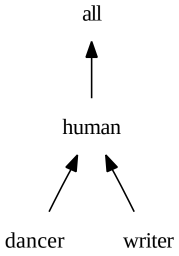


Figure 2.8: dancer and writer

Now if the following sentence is presented to the parser, then the type that is associated to *Jane* must be *dancer* * *writer*.

(31) Jane is a dancer and a writer.

The resulting DAG shaped type hierarchy is shown in figure 2.9.

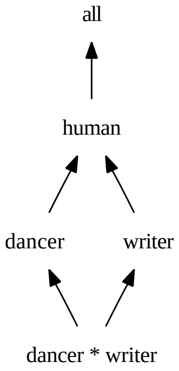


Figure 2.9: dancer and writer combined to form a new type that inherits from both

2.5.1 Multiple Inheritance and Logical Terms for Encoding Hierarchies

Although the method in [23] is only designed to work with tree shaped hierarchies, extensions such as [51] that are based on [23] can work with special cases of DAG shaped hierarchies. The characterization of those hierarchies for which some working encodings exist is provided in [53, 52]. Basically all taxonomies that are lower semi-lattices[24] can be encoded. Andrew Fall in his thesis [31] extensively studied the encodings of taxonomies including some extensions of [23] for efficient subtype checks, meet, and join operations[25]. A later work by Erbach [29], which is based on [51, 53, 5, 4, 7] applies another extension that allows the power of multiple inheritance in Prolog. The framework is called *multi-dimensional multiple inheritance*. This method is at the heart of ProFIT[26] (Prolog with

[24] Hierarchies with the properties of a lower semi-lattices are those for which the most general common subtype (see definition 3.42) is unique for every two types.

[25] A meet operation is finding the most general common subtype of a set of types (see definition 3.42), join is the operation of finding the most specific common super type. The idea of a join is antisymmetric to a meet.

[26] We use ProFIT for our grammar implementation as will be shown in the next chapters.

Features Inheritance and Templates) [30]. Other related works are Aït-Kaci's LOGIN [2], and LIFE [3].

In frameworks such as [51], each type t is associated with a first order logic term using a mapping $\tau(t)$ that is the type encoding. This encoding must have this characteristic that for two types $t_1,\ t_2$, $t_1 \sqsubseteq t_2$ if and only if $\tau(t_1),\ \tau(t_2)$ can be unified and the unification does not bind any variables in $\tau(t_1)$. This sort of unification is called *subsumption*. If such an encoding exists, then the encoding of type $t_1 * t_2$ can be calculated by unifying $t_1,\ t_2$.

For example, the DAG shaped type hierarchy of figure 2.9, can be encoded by the following mapping, where $\_$ denotes an arbitrary free variable:

$\tau(all) = f(\_)$

$\tau(human) = f(human(\_, \_))$

$\tau(dancer) = f(human(dancer, \_))$

$\tau(writer) = f(human(\_, writer))$

$\tau(dancer * writer) = f(human(dancer, writer))$

Limitations of Term Encoding

Although if such encodings exist they will help us efficiently deal with multiple inheritance and subtype checking, as Fall [31] mentions, finding these encodings might be a difficult task. Moreover, an update to the taxonomy might require recomputation of the entire, or a significant part of the encoding. The method applied in [29, 30], requires the complete knowledge of the disjoint types at the beginning to compute the encoding.

There is also a minor problem with specializable types if term encodings are used. Variables are an integral part of the encoding, and they cannot be exclusively used for specializable types. To allow specializable types, we need to inform the subsumption checker to allow instantiations of variables that are present in the encoding of a specializable type.

2.6 Conclusion

In this chapter we discussed types in grammars. We proposed some desirable features of a type system to be used for the application domain of a grammar. The most important contribution of this chapter was the idea of specializable types that was inspired by the incomplete types proposed by Dahl [23, 24].

In the next chapter we layout a type system that benefits the exclusive usage of variables for specializable types and handles multiple inheritance naturally, which does not need a computation of an encoding. This type system has the power to deal with some special cases of negated types as we will see in the next chapter. Negated combined with specializable types have not been explored in previous works. However for negated types to be manageable we need the taxonomy to conform to some natural conditions that we will see in the next chapter. The type system we develop in the next chapter will be used in our grammar through the rest of this thesis.

Chapter 3

Our Type System

In definition 2.2 we provided a definition of instances of a class (or type), here we define a set of all objects that are instances of a type:

Definition 3.1 Instance Set
The set of all instances of a type σ is denoted by I_σ and contains all objects x such that x is an instance of σ. In other words:
$I_\sigma = \{x \mid x : \sigma\}$

Using instance sets, the subtype relation that we defined in definition 2.3 can be expressed by:

Corollary 3.2 Subtype Relation and Instance Sets
For any two types σ, τ,
$\sigma \sqsubseteq \tau \iff I_\sigma \subseteq I_\tau$

Using definition 3.1, and corollary 3.2 the subtype axiom (SUB) can be easily proved, and hence need not be treated as an axiom anymore.

Definition 3.3 Type Equality
Two types are equal if they have the same instance sets. In other words for any two types σ and τ: $\sigma = \tau \iff I_\sigma = I_\tau$

Definition 3.4 The Bottom Type : $\bot$
The bottom type denoted by $\bot$ is a type with empty instance set, i.e., $I_\bot = \emptyset$

3.1 Basic Types in our Type System

In definition 2.6 we used a set of basic types **BasTyp**, but we did not discuss their nature in detail. In this section we break down the basic types into *simple*, *composite*, *specializable*, and *negated* types. The union of these will make up our well-formed basic types that can be used in the λ-calculus that we developed so far. The main purpose of elaborating on this type system is not for the λ-calculus terms to represent the meaning of an utterance like done in CG (although this can be done), but rather here we will use the type system to implement the type restrictions (given in definition 2.24) in our HPSG grammar of the next chapter.

Definition 3.5 Simple Types
Formally the set **SimTyp** is a finite set of types whose instance sets are neither empty nor expressible using set theory operators on other instance sets. We treat the top type *all* as a simple type.

Definition 3.6 Disjoint Types
Two types σ and τ are said to be disjoint if and only if $I_\sigma \cap I_\tau = \emptyset$ and we denote it by $\sigma \parallel \tau$. If τ, σ are not disjoint we write $\sigma \nparallel \tau$.

Definition 3.7 Composite Types
The purpose of composite types is to model multiple inheritance. Formally the set **ComTyp** of well-formed composite types is defined recursively by:

$$\sigma \in \mathbf{SimTyp} \ \wedge \ \tau \in \mathbf{SimTyp} \ \wedge \ \sigma \nparallel \tau \Rightarrow \sigma * \tau \in \mathbf{ComTyp}$$
$$\sigma \in \mathbf{SimTyp} \ \wedge \ \tau \in \mathbf{ComTyp} \ \wedge \ \sigma \nparallel \tau \Rightarrow \sigma * \tau \in \mathbf{ComTyp} \ \wedge \ \tau * \sigma \in \mathbf{ComTyp}$$
$$\sigma \in \mathbf{ComTyp} \ \wedge \ \tau \in \mathbf{ComTyp} \ \wedge \ \sigma \nparallel \tau \Rightarrow \sigma * \tau \in \mathbf{ComTyp}$$

A composite type $\sigma * \tau$ is a maximal type which is a subtype of both σ and τ, that is, it contains any object that belongs to both σ and τ, i.e.,

$$I_{\sigma * \tau} = I_\sigma \cap I_\tau$$

If σ and τ are disjoint then $\sigma * \tau$ is not defined.

Theorem 3.8 Subtype Composition
If $\sigma \sqsubseteq \tau$ then $\sigma * \tau = \sigma$.

PROOF: Since $\sigma \sqsubseteq \tau$ by corollary 3.2 we get:

$I_\sigma \subseteq I_\tau$

by applying rules of set theory we have:

$I_\sigma \cap I_\tau = I_\sigma$

By the definition above we have $I_{\sigma * \tau} = I_\sigma \cap I_\tau = I_\sigma$

so:

$I_{\sigma * \tau} = I_\sigma$

And by applying definition 3.3 we get:

$\sigma * \tau = \sigma \quad \blacksquare$

Theorem 3.9 Composition Subtype

$\sigma * \tau \sqsubseteq \sigma$ and $\sigma * \tau \sqsubseteq \tau$

PROOF: By definition 3.7 we have $I_{\sigma * \tau} = I_\sigma \cap I_\tau$ and by set theory results we know:

$I_\sigma \cap I_\tau \subseteq I_\sigma$ and $I_\sigma \cap I_\tau \subseteq I_\tau$

By applying corollary 3.2 we get:

$\sigma * \tau \sqsubseteq \sigma$ and $\sigma * \tau \sqsubseteq \tau \quad \blacksquare$

Notation 3.10 Natural Types

It will help us if we can refer to the set of simple types and composite types using one symbol, so we define the set of *natural types* to be:

$\mathbf{NatTyp} = \mathbf{SimTyp} \cup \mathbf{ComTyp}$

Note that the bottom type $\perp$ is not included in either **SimTyp** or **ComTyp** and hence is not a natural type.

Definition 3.11 Direct Subtype Relation $\sqsubset_d$

The direct subtype relation, denoted by $\sqsubset_d$, is an irreflexive relation that must be provided by the user of the grammar, that specifies for each natural type τ, what natural type is its immediate super type or subtype. This relation must not be transitive, and more strongly it must be an intransitive relation, that is:

$\forall \tau, \sigma, \omega \in \mathbf{NatTyp}\ .\tau \sqsubset_d \sigma \ \wedge\ \sigma \sqsubset_d \omega \Longrightarrow \tau \not\sqsubset_d \omega$

However, the transitive closure of $\sqsubset_d$ union the identity relation (=) must respect the inclusion of the natural instance sets, that is:

$\forall \tau, \sigma \in \mathbf{NatTyp}\ .\ I_\tau \subseteq I_\sigma \Longleftrightarrow \tau \sqsubseteq_d^* \sigma$

, where $\sqsubseteq_d$ is $\sqsubset_d \cup =$, and $\sqsubseteq_d^*$ is the transitive closure of $\sqsubseteq_d$

Combining the above statement with corollary 3.2 we get:

$\forall \tau, \sigma \in \textbf{NatTyp} \ . \ \tau \sqsubseteq \sigma \iff \tau \sqsubseteq_d^* \sigma$

which means subtype relation $\sqsubseteq$ over natural types is actually equal to the transitive closure of direct subtype relation $\sqsubseteq_d$.

Remark 3.12 Obvious Immediate Super-types of Composite Types
Every composite type $\tau * \sigma$, has at least two immediate super types, namely τ and σ. These two immediate super-types need not be explicitly given by the user because their presence is immediately obvious (by theorem 3.9).

Definition 3.13 Natural Type Hierarchy
A *natural type hierarchy diagram* is a directed acyclic graph (DAG) rooted by the top type *all*, in which edges are a graphical representation of the direct subtype relation ($\sqsubset_d$). In a *complete* type hierarchy diagram, every natural type is associated with a node.

In a real application the complete natural type hierarchy is huge, as every combination of non-disjoint types are composed by multiple inheritance. If there are n simple types, the number of nodes in a complete natural type hierarchy is in the order of $O(2^n)$, because in the worst case 2^n possible compositions of the simple types can exist. In the implementation of our type system, it is not needed that the complete type hierarchy be stored in memory. The space should be enough to store the non-obvious portion of the direct subtype relation $\sqsubset_d$, the list of simple types **SimTyp**, and the information about the disjoint types, and partitions (which will be defined in definition 3.35). Moreover, the information about disjoint types and partitions can be asked from the user when it is needed. Unlike the term encoding methods (such as [51, 52, 53, 29, 30]) this information is not needed at the beginning. Additional nodes and edges are added to the DAG type hierarchy in the system just when they are required. But the complete hierarchy is totally present in the theory that we are developing and every theorem that follows is aware of its presence.

In what follows we may use the term ‘type hierarchy’ instead of ‘natural type hierarchy’ for brevity.

Theorem 3.14 For any natural types τ, σ, τ_1:

$(\tau_1 \sqsubseteq \tau \ \wedge \ \tau_1 \sqsubseteq \sigma) \iff \tau * \sigma$ is a well-formed natural type and $\tau_1 \sqsubseteq \tau * \sigma$

Proof:

$\Leftarrow$) Assume $\tau_1 \sqsubseteq \tau * \sigma$

Using theorem 3.9 we have:

$\tau * \sigma \sqsubseteq \sigma$ and $\tau * \sigma \sqsubseteq \tau$

So we have:

$\tau_1 \sqsubseteq \tau * \sigma$ and $\tau * \sigma \sqsubseteq \tau$

By transitivity of $\sqsubseteq$ we get:

$\tau_1 \sqsubseteq \tau$

Similarly we get:

$\tau_1 \sqsubseteq \sigma \quad \square$

$\Rightarrow$) Assume: $\tau_1 \sqsubseteq \tau \ \wedge \ \tau_1 \ \sqsubseteq \ \sigma$

Using corollary 3.2 we get:

$I_{\tau_1} \subseteq I_\tau \ \wedge \ I_{\tau_1} \subseteq I_\sigma$

For an arbitrary object x if $x \in I_{\tau_1}$ then by the definition of subsets in set theory and the statement above we get:

$x \in I_\tau \ \wedge \ x \in I_\sigma$

By definition of intersection in set theory we get:

$x \in I_\tau \cap I_\sigma$

This means τ, and σ are not disjoint, and by definition of composite types (definition 3.7) $\tau * \sigma$ is a well-formed composite type and we also have:

$x \in I_{\tau * \sigma}$

So far we proved $\forall x \ . \ x \in I_{\tau_1} \Rightarrow x \in I_{\tau * \sigma}$

Using the definition of subsets we get:

$I_{\tau_1} \subseteq I_{\tau * \sigma}$

By corollary 3.2 we get:

$\tau_1 \sqsubseteq \tau * \sigma \quad \square$

$\blacksquare$

Definition 3.15 Specializable Types

If σ is a natural type, then for every $i \in \mathbb{N}$, $spec_i(\sigma)$ is a specializable type below σ whose instance set is non-empty with these axioms (the second one is redundant, and can in fact be derived):

$\forall \tau, \sigma \in \textbf{NatTyp} \ . \ \forall i \in \mathbb{N} \ . \ \forall x. \ x : spec_i(\tau) \ \wedge \ (\sigma \sqsubseteq \tau) \Rightarrow x : \sigma \quad : (SPECIAL)$

$\forall \tau \in \textbf{NatTyp} \ . \ \forall i \in \mathbb{N} \ . \ spec_i(\tau) \sqsubseteq \tau \quad : (SPECSUB)$

$spec_i(\tau)$ is an *underspecified type*. Although one can declare (or assume) that an object is an instance of a specializable type (for example the pronoun *she* can be declared in the lexicon to be an instance of $spec_i(human)$) the complete instance set of a specializable type cannot be determined. Another important note is that there may be an infinite number of specializable types under a single type due to the use of a natural number subscript in specializable types.[1] The set of all specializable types is denoted by **SpecTyp**.

Now we define an infix operator of types that can build expressions that can represent either fully-specified or underspecified traversals of the type hierarchy downwards. The expressions built from this function are like path expressions that we presented in definition 2.27 but in a reverse order and we allow jumps, that is, we do not have to list every node in the path. For this reason we call this operator the *jump operator*. Underspecified expressions make use of variables. We suppose all variables are chosen from the set $\{V_0, V_1, V_2, V_3, ...\}$. So every variable has a subscript. Even if the subscript is not shown in a formula, it implicitly exists.

Definition 3.16 Jump Operator : $\curvearrowright$, and Jump Expressions
$\curvearrowright$ is an infix right associative operator that is used to build *jump expressions* and is defined this way:

- if σ and τ are natural types such that $\tau \sqsubseteq \sigma$ then $\sigma \curvearrowright \tau$ is a well-formed jump expression headed by σ, and we have $\sigma \curvearrowright \tau = \tau$
- if σ is a natural type and V_i is a variable then $\sigma \curvearrowright V_i$ is a well formed jump expression headed by σ, and we have $\sigma \curvearrowright V_i = spec_i(\sigma)$
- if ξ is a well formed jump expression headed by τ and σ is a natural type such that $\tau \sqsubseteq \sigma$ then $\sigma \curvearrowright \xi$ is a well-formed jump expression headed by σ, and we have $\sigma \curvearrowright \xi = \xi$

The set of all well-formed jump expressions is denoted by **JumpExp**

Example 3.17 Using the type hierarchy of figure 2.4 the following are well-formed jump expressions:

[1] The axiom (SPECIAL) is in fact a shifting operator that shifts $spec_i(\tau)$ to σ. More rigorously this axiom should be only applicable if the type $spec_i(\tau)$ has not already been shifted. As soon as it is shifted, it will be equal to the type it is shifted to. From the technical point of view, any reference to an entry with a specializable type from the lexicon should be followed by updating the specializable type subscript to a new natural number that has not been used in the system before to avoid unwanted type equalities among different occurrences of the same pronoun.

expression	headed by
$\mathit{all} \curvearrowright \mathit{physical} \curvearrowright \mathit{animal} \curvearrowright \mathit{bird}$	*all*
$\mathit{all} \curvearrowright \mathit{animal} \curvearrowright \mathit{bird}$	*all*
$\mathit{animal} \curvearrowright \mathit{mammal} \curvearrowright \mathit{beluga}$	*animal*
$\mathit{animal} \curvearrowright \mathit{beluga}$	*animal*
$\mathit{conceptual} \curvearrowright \mathit{joke} \curvearrowright V$	*conceptual*
$\mathit{human} \curvearrowright V$	*human*

Remark 3.18 Note that headedness of jump expressions is a pure syntactic property. As a result there can be two jump expressions that represent the same type (and hence equal) while they have different heads. For example $\mathit{all} \curvearrowright \mathit{bird}$ and $\mathit{animal} \curvearrowright \mathit{bird}$ both represent the type *bird* and we have $\mathit{all} \curvearrowright \mathit{bird} = \mathit{animal} \curvearrowright \mathit{bird} = \mathit{bird}$, but the head of the first expression is *all* and the head of the second one is *animal*. It is like arithmetic expressions such as $1 + 2$ and 3 that are equal (represent the same number) but have different syntactic properties, that is, the first one is a function applied on two constants, and the second one is a constant.

Remark 3.19 Note that different variables can be chosen to end a jump expression. This is compatible with the fact that two specializable types under a specific type need not be equal, and that there could be infinite specializable types under a type. The subscripts used for specializable types are the variable indices used to form the corresponding jump expressions. If two jump expressions which represent specializable types under a specific type end with the same variable, they are equal.

Remark 3.20 Jump expressions are types themselves, because for every case in definition 3.16, the expression is equal to a natural type, or a specializable type or a shorter jump expression, which will eventually reduce to either a natural type or a specializable type. Equality among different jump expressions form an equivalence relation on **JumpExp**. And since every jump expression equals to either a natural type or a specializable type, we can claim that the union of all equivalence classes induced by every natural type together with every specializable type is exactly the set **JumpExp**. That is,

$$(1)\ \ \mathbf{JumpExp} = \bigcup_{a \in \mathbf{NatTyp}} [a] \quad \cup \quad \bigcup_{a \in \mathbf{SpecTyp}} [a]$$

Also we have:

$\mathbf{SpecTyp} \subseteq \mathbf{JumpExp}$

Definition 3.21 Semantics of Jump Expressions
If there is no variable in a jump expression then by definition 3.16 the expression is equal to the natural type at the rightmost position. But if the expression contains a variable then by definition 3.16 this variable must be at the rightmost position of the expression, and the expression will be equal to: $\tau \curvearrowright V$, where V is the last operand and τ is the second last operand in the jump expression written without parentheses.

$\tau \curvearrowright V$ is an expression that contains a free variable. The semantics of an expression with a free variable depends on the context where it appears.

If this expression is presented to a *satisfiability* checker then the free variable is implicitly bound by an existential quantifier, and the satisfiability checker must ensure the existence of values $y \in \mathbf{NatTyp} \cup \mathbf{JumpExp}$ for V such that the formula resulting from substituting y for V is well-formed and true. If $y \in \mathbf{NatTyp}$, y is called a *natural answer*. The satisfiability checker should output the answers if required. Sometimes it is possible for the satisfiability checker to find more general answers $y \in \mathbf{JumpExp}$. In that case y is called a *specializable answer*. Specializable answers are favored because the number of outputs is reduced while the same information is carried.

If the expression with the free variable is used in a standalone formula (other than a formula with satisfiability semantics) then the free variable is implicitly bound by a universal quantifier with the range of values such as y with the restriction that $y \in \mathbf{NatTyp} \cup \mathbf{JumpExp}$ and that the formula resulting from substituting y for V is well-formed.

Remark 3.22 Unless specified otherwise in the context, we simply use *answer* instead of *natural answer* for brevity.

As a special case of the above definition we present a definition of satisfiability of predicates with jump expressions.

Definition 3.23 Satisfiability of Predicates with Jump Expression Arguments
For any binary predicate symbol R, and for any jump expression $\tau \curvearrowright V$ and for any σ:

$$(\tau \curvearrowright V)\ R\ \sigma \text{ is satisfiable } \iff \exists \tau_0 \in \mathbf{NatTyp} \text{ such that } (\tau_0 \sqsubseteq \tau) \wedge (\tau_0\ R\ (\sigma.[\tau_0/V]))\ ,$$

where $\sigma.[\tau_0/V]$ is the result of substituting τ_0 for V wherever it occurs in σ.

Note that for $\tau_0 \; R \; (\sigma.[\tau_0/V])$ in the right hand side of the above equation to be true (or satisfiable if it contains free variables other than V) it is required that it be a well-formed formula, and as a result any sub-expressions should be well-formed too.

Also note that the order of the arguments of R is arbitrary, and it can be reversed in both side of the biconditional.

Theorem 3.24 For any two natural types τ, σ:

If $(\tau \curvearrowright V) \sqsubseteq \sigma$ is satisfiable then τ and σ are not disjoint.

Proof: Based on definition 3.23 there should be a natural type τ_0 such that $\tau_0 \sqsubseteq \tau \wedge \tau_0 \sqsubseteq \sigma$. Now because $\tau_0 \neq \bot$ we have $I_{\tau_0} \neq \emptyset$, so there is an object x such that $x \in \tau_0$. And by applying (SUB) we have:

$x : \tau_0 \wedge \tau_0 \sqsubseteq \tau \Rightarrow x : \tau$

$x : \tau_0 \wedge \tau_0 \sqsubseteq \sigma \Rightarrow x : \sigma$

That is, there is an object x that belongs to both types τ and σ, so $I_\tau \cap I_\sigma \neq \emptyset$ and hence σ and τ are not disjoint. ■

Notation 3.25 Unnegated Types:

It helps us later if we can refer to natural types and jump expressions together. Thus we define:

$\mathbf{UNegTyp} = \mathbf{NatTyp} \cup \mathbf{JumpExp}$

Definition 3.26 Negated Types

For any natural types σ and τ, where $\sigma \not\sqsubseteq \tau$, and any free variable V:

- $\sigma - \tau$ is a well-formed negated type, and it accompanies the following axiom:

  $$x : \sigma - \tau \Longleftrightarrow x : \sigma \;\wedge\; \neg(x : \tau) \qquad : (NEG)$$

  and we have:

  $I_{\sigma - \tau} = I_\sigma - I_\tau = I_\sigma \cap I'_\tau \neq \emptyset$, where I'_τ is the complement set of I_τ

- $\sigma \curvearrowright V - \tau$ is a well-formed negated type, whose semantics is governed by the semantics of jump expressions, and is accompanied by the following (redundant) axiom:

  $\forall \delta \in \mathbf{NatTyp} \;.\; \forall x.\; x : \sigma \curvearrowright V - \tau \;\wedge\; (\delta \sqsubseteq \sigma) \;\wedge\; (\delta \not\sqsubseteq \tau)$

  $\qquad \Rightarrow x : \delta - \tau \qquad : (SPECNEG)$

The set of all negated types is denoted by **NegTyp**.
Any negated type must be different from the bottom type $\bot$.

Note: Since the type axioms use the first order logic negation connective, for completeness we need to consider the contrapositive of every type axiom as another type axiom.

Theorem 3.27 For any natural types τ, σ such that $\tau \not\sqsubseteq \sigma$ we have:
$\tau - \sigma \sqsubseteq \tau$

Proof: First we observe that $\tau - \sigma$ is a well-formed negated type because $\tau \not\sqsubseteq \sigma$. From the definition we have:
$I_{\tau - \sigma} = I_\tau - I_\sigma$
But from set theory results the right hand side of the above equation is a subset of I_τ so we have:
$I_{\tau-\sigma} \subseteq I_\tau$
And from corollary 3.2 we obtain:
$\tau - \sigma \sqsubseteq \tau$
$\blacksquare$

Example 3.28 A legitimate use of negated types is for the pronouns that cannot refer to humans. For example the pronoun *it* can be defined in the lexicon by:
$it : all \curvearrowright V - human$
And if the lexicon contains the verbs *bark* and *talks* with the type restriction $TR_{barks} = \{(subj, dog)\}$ and $TR_{talks} = \{(subj, human)\}$ then sentence (2) should be recognized as semantically valid, while sentence (3) should be recognized as semantically invalid.

(2) It barks.

(3) * It talks.

However we do not yet have the necessary theoretical means to prove this, one main reason is that type restrictions (see definition 2.24) need to be updated to use axiom (NEG) that not only outputs a type statement like $x : \tau$ but also outputs a type statement like $\neg(x : \tau)$.

Definition 3.29 Type Restriction Satisfaction (Version II)
A type restriction TR of a sense for an expression is satisfied in a phrase if and only if for

every pair $(role, type) \in TR$, the expression ξ that assumes the grammatical role *role* in the phrase, refers to an object of type τ where an instantiation $type'$ of *type* exists[2] such that we have either:

- $\tau = type'$
-
  - there is a type axiom that outputs $\xi : type'$, and
  - there is no type axiom that outputs $\neg(\xi : type')$

Using the above definition and applying it on sentence (2), we have:

(4) $it : all \curvearrowright V - human$

Since $dog \sqsubseteq all$ and $dog \not\sqsubseteq human$ we can apply $(SPECNEG)$ by setting $\delta = dog$ and derive:

(5) $it : dog - human$

and an application of axiom (NEG) outputs:

(6) a. $it : dog$, which is required by the verb *barks*, and

b. $\neg(it : human)$

And there is no type axiom that outputs $\neg(it : dog)$ so the type restriction of the verb *barks* is satisfied and the sentence is recognized as a semantically valid sentence.

On the other hand the pronoun *it* is also used in sentence (3), and in (6b) we obtained that $\neg(it : human)$ which contradicts the restriction of the verb *talks*. So sentence (3) is recognized as semantically invalid.

Definition 3.30 Basic Types in our Type System
The set of basic types denoted by **BasType** is the union of simple, composite, jump and negated types that is:

BasTyp = **SimTyp**∪**ComTyp**∪**JumpExp**∪**NegTyp**. This could result in an infinite number of basic types due to possibly an infinite number of specializable types in **JumpExp**.

[2] if *type* does not have any free variables $type' = type$.

3.2 Most General Specializable Answers

As the type checking algorithm that we will introduce in section 3.5 heavily relies on a satisfiability checker over subtype formulas[3] that is developed in section 3.4, we are interested in finding values for variables in jump expressions that satisfy the subtype relations. Such values are called *answers*. There might be multiple answers for a variable. Sometimes it is possible to find a specializable type that can be specialized to every answer and to no type that is not an answer. We call such answers the *most general specializable answers*. If they exist, they encompass all possible answers and reduce the number of outputs of our satisfiability checker. This will result in less computational effort by our parser, because each answer often associates with a possible reading of a phrase, and less readings for each phrase will result less possible combinations of readings for building larger phrases. First we clarify what we mean by the generality of types. Next we formally define most general specializable answers. Then in the next section we provide theorems that enable us to verify a subtype relation for almost all basic types.

Definition 3.31 Generality of Types
For two types σ and τ we say σ is more general than τ if and only if $\tau \sqsubset \sigma$. Likewise we say σ is less general than τ if and only if $\sigma \sqsubset \tau$.

Definition 3.32 The Most General Specializable Answer
For any binary relation R and any natural type τ and any type σ, the most general answer of V satisfying $(\tau \curvearrowright V)\ R\ \sigma$ is $\omega \curvearrowright V'$ for a fresh variable[4] V' if and only if:

(7) $\omega \in \mathbf{NatTyp}$

(8) $\forall \tau_0 \in \mathbf{NatTyp}\ .\ (\tau_0 \sqsubseteq \tau\ \wedge\ \tau_0\ R\ (\sigma.[\tau_0/V])) \Longleftrightarrow \tau_0 \sqsubseteq \omega$

where $\sigma.[\tau_0/V]$ is the result of substituting τ_0 for V if it occurs in σ.

[3] Hoang et al. in [43] call a similar problem the Satisfiability of Subtype Inequalities (SSI), where several subtype formulas are simultaneously being checked for satisfiability. Here, if there are several formulas to be satisfied, we check just one formula at a time, and unify the variables in the rest of the formulas to their corresponding answers in the formula just checked. The system provided in [43], however, lacks the $(SPECIAL)$ axiom because this axiom does not seem to be helpful for OOP programming languages which is their target. $(SPECIAL)$ corresponds to down-casting in OOP, which in popular languages is not automatic, and the programmer must force it when required.

[4] By a fresh variable we mean it has not been previously used in any other formula. This is needed to avoid variable clashes between independent formulas. Not using fresh variables in places where we require in this thesis can result in inconsistency and circular terms.

This means any answer to $(\tau \curvearrowright V)\ R\ \sigma$ will be equal to $\omega \curvearrowright V'$ if V' is instantiated to an appropriate natural type. This requires any answer to be less general than or equal to ω and any subtype of ω to be an answer.

Corollary 3.33 If $\omega \curvearrowright V'$ is the most general specializable answer of V satisfying $(\tau \curvearrowright V)\ R\ \sigma$ then:

$$(9) \qquad \omega \sqsubseteq \tau$$

$$(10) \qquad \omega\ R\ (\sigma.[\omega/V])$$

This is obtained by plugging the value ω for τ_0 in (8).

Remark 3.34 Note that the most general specializable answer might not exist for some cases. For example if *human* has some strict subtypes in the type hierarchy and the input of the satisfiability checker is the following subtype relation:
$human \sqsubseteq human \curvearrowright V$
then the only answer for V is *human* (which is non-specializable).

3.3 Checking $\sqsubseteq$ for All Unnegated Types and Special Cases of Negated Types

In applications where only a given type might be affected by negation (as the type *human* in this thesis) there is a condition whose truth enables us to find an efficient algorithm of subtype checking for negated types as well as unnegated types. For example it suffices in the HPSG grammar of this thesis, which we develop in the next chapters that the negations operate only with the type *human*. That is, every negated type, which we will use in the grammar is either $\tau - human$ or $\tau \curvearrowright V_i - human$, for some natural type τ. For expressing this condition we need some definitions first.

Definition 3.35 Partitions
For any natural type τ with subtypes $\tau_1, \tau_2, ..., \tau_n$ we say τ is partitioned into n types $\tau_1, \tau_2, ..., \tau_n$ if and only if,
$I_\tau = I_{\tau_1} \cup I_{\tau_2} \cup ... \cup I_{\tau_n}$, and
$\tau_i \parallel \tau_j$ for any $1 \leq i < j \leq n$
that is $\tau_1, \tau_2, ..., \tau_n$ are disjoint and their instances make up the all of the instances of τ.

Each one of $\tau_1, \tau_2, ..., \tau_n$ are called *partitions of* τ. We denote this partitioning by :
$\tau = (\tau_1 \mid \tau_2 \mid ... \mid \tau_n)$

Remark 3.36 Partitions and General Classes
If τ is partitioned into $(\tau_1 \mid \tau_2 \mid ... \mid \tau_n)$ then according to the definition above every instance of τ must be an instance of exactly one of its subtypes $\tau_1, ..., \tau_n$. This fits the description of the general classes that we provided in the previous chapter.

Definition 3.37 Triangular Types
A natural type σ is called a *triangular type* if and only if, for every other natural type τ one of the following 3 conditions is true:
$\tau \sqsubseteq \sigma$, that is, τ is a subtype of σ
$\sigma \sqsubset \tau$, that is, σ is a strict subtype of τ
$\tau \parallel \sigma$, that is, τ and σ are disjoint

Now we can express the *triangularity condition*.

Triangularity Condition:
Suppose the only type that appears on the right hand side of the negation operator $(-)$ is σ. The *triangularity condition* holds in a type hierarchy if and only if σ and all of its super types are triangular types.

So, it is important that in the type hierarchy of the application, *human* with all of its super types be triangular types. This is natural, because being triangular simply means that the types at that level and above are all in non-overlapping partitions. This is true for example in the type hierarchy of figure 3.1. We do not have any type that overlaps with and is not contained in nor contained by *physical*. That is, any type is either a subtype or super type of *physical* or it is disjoint with it. Same naturally holds for any parent of *human*.

Note that we do not pose any restriction on the types that are below *human*. For example the hierarchy shown in figure 3.1 can be extended below human, or below any type that is disjoint with human to suit the application's domain. Note that no multiple inheritance is allowed with parents among different partitions of a partitioned type. This would contradict that the partitions must be disjoint. This condition is similar to what is described as multi-dimensional inheritance by Erbach in [29].

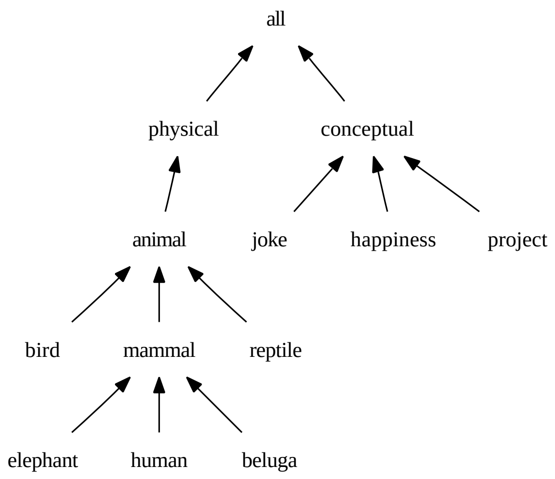


Figure 3.1: An example class hierarchy

With this restriction in mind, we introduce the subtype checking problem below, and in the following subsections we provide theorems about different cases associated with this problem.

3.3.1 Subtype Checking Problem

A *subtype check* is the problem of finding the truth value of a given input formula $A \sqsubseteq B$ if it does not contain any variables, or is the problem of satisfying $A \sqsubseteq B$ if it contains variables (specializable types) by finding the answers.

The solution to a subtype check depends on the nature of the types A, and B. A can be:

- a natural type
- a specializable type
- a negated but not specializable type
- a negated and specializable type

Same possibilities hold for B. There will be $4 \times 4 = 16$ possibilities. Moreover, there are additional cases if A, and B are both specializable, either negated or unnegated. There are

$2 \times 2 = 4$ cases where both A, and B are specializable. Each of these 4 cases breaks down into two further cases:

- i) Variables in the jump expressions associated with A, and B are equal
- ii) Variables in the jump expressions associated with A, and B are different

So the total number of possibilities becomes:

$16 - 4 + 4 \times 2 = 20$

Table 3.1: Theorems of Subtype Checking

	A	B	
Row	Left hand side of $\sqsubseteq$	Right hand side of $\sqsubseteq$	Theorem
1	Natural	Natural	none ($\sqsubseteq_d^*$)
2	$A' \curvearrowright V$	Natural	Theorem 3.38
3	Natural	$B' \curvearrowright V$	Theorem 3.40
4	$A' \curvearrowright V$	$B' \curvearrowright V$	Theorem 3.45
5	$A' \curvearrowright V$	$B' \curvearrowright X$	Theorem 3.47
6	$A' - \sigma$	Natural	Theorem 3.50
7	Natural	$B' - \sigma$	Theorem 3.51
8	$A' - \sigma$	$B' - \sigma$	Theorem 3.52
9	$A' \curvearrowright V - \sigma$	Natural	Theorem 3.53
10	$A' - \sigma$	$B' \curvearrowright V$	Theorem 3.54
11	$A' \curvearrowright V - \sigma$	$B' \curvearrowright V$	Theorem 3.55
12	$A' \curvearrowright V - \sigma$	$B' \curvearrowright X$	Theorem 3.58
13	Natural	$B' \curvearrowright V - \sigma$	Theorem 3.59
14	$A' \curvearrowright V$	$B' - \sigma$	Theorem 3.60
15	$A' \curvearrowright V$	$B' \curvearrowright V - \sigma$	Theorem 3.64
16	$A' \curvearrowright V$	$B' \curvearrowright X - \sigma$	Theorem 3.67
17	$A' \curvearrowright V - \sigma$	$B' - \sigma$	Theorem 3.68
18	$A' - \sigma$	$B' \curvearrowright V - \sigma$	Theorem 3.69
19	$A' \curvearrowright V - \sigma$	$B' \curvearrowright V - \sigma$	Theorem 3.70
20	$A' \curvearrowright V - \sigma$	$B' \curvearrowright X - \sigma$	Theorem 3.72

Each of these 20 cases needs a slightly different treatment. We provide a theorem that deals with the case for each one. The proofs are provided in appendix B. These theorems enable us to decide whether $A \sqsubseteq B$ holds or if it is satisfiable and if so what its answers are. These possibilities together with the theorems that deal with them are enumerated in table 3.1.

3.3.2 Subtype Checking Theorems, Unnegated Types

Here we provide the theorems that deal with unnegated types in a subtype checking problem, together with some useful remarks in between. The proofs are provided in appendix B.

Theorem 3.38 Specializable Types on the Left Hand Side of $\sqsubseteq$
For any two natural types τ, σ:

- (a) If $\tau \curvearrowright V \sqsubseteq \sigma$ is satisfiable then $\tau \nparallel \sigma$ and the most general specializable answer of V is $\tau * \sigma \curvearrowright V'$, for a fresh free variable V'.
- (b) If $\tau \nparallel \sigma$ then $\tau \curvearrowright V \sqsubseteq \sigma$ is satisfiable and the most general answer of V is $\tau * \sigma \curvearrowright V'$, for a fresh free variable V'.

Remark 3.39 We make this assumption that we do not need answers to the variables of specializable types on the right hand side of $\sqsubseteq$ as outputs of our subtype satisfiability checker. The satisfiability checker must just ensure that an answer for the variable on the right hand side of $\sqsubseteq$ exists, and the input subtype equation is satisfiable. However we need the answers to the variables used on the left hand side of $\sqsubseteq$ and the satisfiability checker must find them.

Theorem 3.40 Specializable Types on the Right Hand Side of $\sqsubseteq$
For any two natural types τ, σ:
$\tau \sqsubseteq \sigma \curvearrowright V$ is satisfiable $\Longleftrightarrow \tau \sqsubseteq \sigma$

Definition 3.41 Common Subtype
A common subtype of two natural types τ and ω is a natural type μ such that $\mu \sqsubseteq \tau$ and $\mu \sqsubseteq \omega$.

Definition 3.42 The Most General Common Subtype (greatest lower bound, meet)
A most general common subtype of two natural types τ and ω is μ, a common subtype of τ and ω such that no strict super-type of μ is also a common subtype of τ and ω. In literature this is also called a *greatest lower bound* of τ and ω. A set that contains all of the greatest lower bound of τ, ω is denoted by $\tau \sqcap \omega$.

Theorem 3.43 The most general common subtype of two natural types τ and σ is $\tau * \sigma$ if they are not disjoint, and does not exist if they are disjoint

PROOF:

- Case i) τ and σ are not disjoint.
  First note that by theorem 3.9 we have $\tau * \sigma \sqsubseteq \tau$ and $\tau * \sigma \sqsubseteq \sigma$, that is, $\tau * \sigma$ is a common subtype of τ and σ. Next by theorem 3.14 any common subtype of τ and σ is a subtype of $\tau * \sigma$. So $\tau * \sigma$ is the most general common subtype of τ and σ, because first, it is a common subtype of τ and σ, and second any other common subtype of τ and σ is as well a subtype of $\tau * \sigma$. □

- Case ii) τ and σ are disjoint.
  We prove the theorem in this case by contradiction. Suppose ω is a common subtype of τ and σ. Then by our definition of common subtype it is implied that ω is a natural type, and hence is not equal to the bottom type. So its instance set is non-empty. Suppose $x \in I_\omega$.
  Also by the corollary 3.2 we have $I_\omega \subseteq I_\tau$ and $I_\omega \subseteq I_\sigma$.
  Then we must have:
  $x \in I_\tau$ and $x \in I_\sigma$
  This means $x \in I_\tau \cap I_\sigma$ that is, τ and σ are not disjoint, which is a contradiction. □

■

Algorithm 3.44 The above theorem enables us to easily find the most general common subtype of two arbitrary natural types t_1, and t_2. We first ensure that these types are not disjoint. Then the result is simply $t_1 * t_2$. There are two special cases that we can simplify $t_1 * t_2$ to either t_1, or t_2. These special cases are respectively for cases when we have:

(11) a. $t_1 \sqsubseteq t_2$

b. $t_2 \sqsubseteq t_1$

A simple algorithm to do all of this is provided in the program 3.1.

Theorem 3.45 Specializable Types on Both Sides of $\sqsubseteq$ (case 1)

For any two natural types τ, ω and any free variable V:

```
function most_general_common_subtype($t_1$:natural, $t_2$:natural) : a natural type, or $nil$
/* any non-recursive call clears the visited types automatically */
begin
  /* we will define disjoint(a, b) later */
  if disjoint($t_1$, $t_2$) then return $nil$
  /* we will see later that $\sqsubseteq_d^*$ is computed by subtype_natural */
  else if $t_1 \sqsubseteq_d^* t_2$ then return $t_1$
  else if $t_2 \sqsubseteq_d^* t_1$ then return $t_2$
  else return $t_1 * t_2$
  end if
end
```

Program 3.1: Finding the most general common subtypes of t_1, t_2

- a) Any answer to V satisfying $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright V$ is a subtype of the most general common subtype of τ and ω.

- b) Any subtype of the most general common subtype of τ and ω is an answer to V satisfying $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright V$.

Remark 3.46 If the most general common subtype μ of τ, ω exists, then we are able to provide the most general specializable answer to V satisfying $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright V$. This answer is: $\mu \curvearrowright V_h$, for a fresh variable V_h, with an appropriate index h that was not used as a variable index, or as a subscript in a specializable type before.

Theorem 3.47 Specializable Types on Both Sides of $\sqsubseteq$ (case 2)
For any two natural types τ, ω and any two free variable V, X:

- a) For any answer $(v_1,\ x_1)$ to $(V,\ X)$ satisfying $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright X$, v_1 is an answer to V satisfying $\tau \curvearrowright V \sqsubseteq \omega$.

- b) Any answer v_1 to V satisfying $\tau \curvearrowright V \sqsubseteq \omega$ contributes to an answer $(v_1,\ \omega)$ to $(V,\ X)$ satisfying $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright X$.

3.3.3 Subtype Checking Theorems for Special Cases of Negated Types

In this section we provide some theorems that enable us to decide about the satisfiability of subtype relations that involve a negated type argument.

Theorem 3.48 Negated Non-Specializable Types on the Left Hand Side of $\sqsubseteq$ ($\Longrightarrow$ direction)
For any two natural types τ, ω, and any natural type σ such that any natural super type of σ is triangular we have:
$\tau - \sigma \sqsubseteq \omega \Longrightarrow \tau \not\sqsubseteq \sigma \ \wedge (\ \tau = (\omega \mid \sigma) \ \vee \ \tau \sqsubseteq \omega)$

Theorem 3.49 Negated Non-Specializable Types on the Left Hand Side of $\sqsubseteq$ ($\Longleftarrow$ direction)
For any two natural types τ, ω, and any natural type σ such that any natural super type of σ is triangular we have:
$\tau - \sigma \sqsubseteq \omega \Longleftarrow \tau \not\sqsubseteq \sigma \ \wedge (\ \tau = (\omega \mid \sigma) \ \vee \ \tau \sqsubseteq \omega)$

Theorem 3.50 Negated Non-Specializable Types on the Left Hand Side of $\sqsubseteq$ (both directions)
For any two natural types τ, ω, and any natural type σ such that any natural super type of σ is triangular we have:
$\tau - \sigma \sqsubseteq \omega \Longleftrightarrow \tau \not\sqsubseteq \sigma \ \wedge (\ \tau = (\omega \mid \sigma) \ \vee \ \tau \sqsubseteq \omega)$

Theorem 3.51 Negated Non-Specializable Types on the Right Hand Side of $\sqsubseteq$
For any natural types τ, ω and and any natural type σ such that any natural super type of σ is triangular we have:
$\tau \sqsubseteq \omega - \sigma \Longleftrightarrow \omega \not\sqsubseteq \sigma \ \wedge \ \tau \sqsubseteq \omega \ \wedge \ \tau \parallel \sigma$

Theorem 3.52 Negated Non-Specializable Types on Both Sides of $\sqsubseteq$
For any two natural types τ, ω, and any natural type σ whose any natural super is triangular we have:
$\tau - \sigma \sqsubseteq \omega - \sigma \Longleftrightarrow \tau \not\sqsubseteq \sigma \ \wedge \ \omega \not\sqsubseteq \sigma \ \wedge \ (\tau = (\omega \mid \sigma) \ \vee \ \tau \sqsubseteq \omega)$

In the following theorems we seek ways to reduce answers to negated specializable types in subtype satisfiability problem to simpler cases.

Theorem 3.53 Negated Specializable Types on the Left Hand Side of $\sqsubseteq$
For any two natural types τ, ω, and any natural type σ such that any natural super type of σ is triangular we have:
Any answer to $\tau \curvearrowright V - \sigma \sqsubseteq \omega$ is either
an answer to the equation $V = (\omega \mid \sigma)$ that is also a natural subtype of τ

or

an answer to $\tau \curvearrowright V \sqsubseteq \omega$ that is not a subtype of σ

and vice versa.

Theorem 3.54 Negated Non-Specializable Types on the Left Hand Side of $\sqsubseteq$ and Specializable Types on the Right Hand Side of $\sqsubseteq$
For any two natural types τ, ω, and any natural type σ such that any natural super type of σ is triangular we have:
$\tau - \sigma \sqsubseteq \omega \curvearrowright V$ is satisfiable if and only if $\tau - \sigma \sqsubseteq \omega$ holds.

Theorem 3.55 Negated Specializable Types on the Left Hand Side of $\sqsubseteq$ and Specializable Types on the Right Hand Side of $\sqsubseteq$ (case 1)
For any two natural types τ, ω, and any natural type σ such that any natural super type of σ is triangular we have:

- Any answer to V satisfying $\tau \curvearrowright V - \sigma \sqsubseteq \omega \curvearrowright V$ is a common subtype of τ and ω that is not a subtype of σ
- Any common subtype of τ and ω that is not a subtype of σ is an answer to V satisfying $\tau \curvearrowright V - \sigma \sqsubseteq \omega \curvearrowright V$

Algorithm 3.56 Here we present an algorithm to find the most general subtypes of t_1, t_2, (if possible specializable types) that are not a subtype of σ. t_1, t_1 and σ are natural, and additionally σ and any of its parents are triangular. This algorithm will be used in remark 3.57 and remark 3.71. We begin by finding the most general common subtypes of t_1, and t_2, and from there run a depth first search algorithm. This algorithm is provided in program 3.2. The assumption that σ is a triangular type ensures that exactly one of the three cases investigated by the `if` statement in `check_subtypes(.,.)` holds, and the algorithm covers all possible cases.

Remark 3.57 Using theorem 3.55 we are able to reduce the problem of finding the answers to V satisfying $\tau \curvearrowright V - \sigma \sqsubseteq \omega \curvearrowright V$, to the problem of finding the common subtypes of τ, and ω that are not a subtype of σ. And these values can be found by a call to:

`most_general_common_subtypes_not_contained_by(τ, ω, σ)`.

```
function most_general_common_subtypes_not_contained_by(
  $t_1$ :  natural, $t_2$ :  natural, $\sigma$ :  triangular)
    :  a set of natural types or jump expressions
begin
  $t \leftarrow$ most_general_common_subtype($t_1$, $t_2$)
  if $t = nil$ then return $\emptyset$
  return check_subtypes($t$, $\sigma$)
end

function check_subtypes($t$, $\sigma$) :  a set of natural types or jump expressions
begin
  /* If $t \sqsubseteq \sigma$ then every subtype of $t$ is also */
  /* a subtype of $\sigma$ and no answer from this sub-DAG of */
  /* the type hierarchy can be found */
  /* We will see later that $\sqsubseteq_d^*$ is computed by subtype_natural */
  if $t \sqsubseteq_d^* \sigma$ then return $\emptyset$
  /* If $t$ and $\sigma$ are disjoint then no subtype of $t$ */
  /* can be a subtype of $\sigma$, that is, every subtype of $t$ */
  /* is an answer, so we are able to output a specializable */
  /* answer that covers this sub-DAG of the hierarchy */
  /* (we will define disjoint(a, b) later) */
  if disjoint($t$, $\sigma$) then
    return $\{t \curvearrowright V_t\}$
  /* Otherwise $t$ is an answer and we continue by checking all */
  /* immediate subtypes of $t$ */
  /* The test in the following if always passes, because $\sigma$ is triangular */
  if $\sigma \sqsubset_d^* t$ then
    $A \leftarrow \{t\}$
    for each immediate subtype $t'$ of $t$ do
      $A \leftarrow A \cup$ check_subtypes($t'$, $\sigma$)
    end for
    return $A$
end
```

Program 3.2: Enumerating common subtypes of t_1, t_2 that are not a subtype of σ

Theorem 3.58 Negated Specializable Types on the Left Hand Side of $\sqsubseteq$ and Specializable Types on the Right Hand Side of $\sqsubseteq$ (case 2)
For any two natural types $\tau,\ \omega$, and any natural type σ such that any natural super type of σ is triangular we have:

- i) For any answer $(v_1,\ \omega_1)$ to $\tau \curvearrowright V - \sigma \sqsubseteq \omega \curvearrowright X$, v_1 is an answer to $\tau \curvearrowright V - \sigma \sqsubseteq \omega$
- ii) For any answer v_1 to $\tau \curvearrowright V - \sigma \sqsubseteq \omega$, (v_1, ω) is an answer to $\tau \curvearrowright V - \sigma \sqsubseteq \omega \curvearrowright X$

Theorem 3.59 Negated Specializable Types on the Right Hand Side of $\sqsubseteq$
For any two natural types $\tau,\ \omega$, and any natural type σ such that any natural super type of σ is triangular we have:

- i) If $\tau \sqsubseteq \omega \curvearrowright V - \sigma$ is satisfiable then $\tau \parallel \sigma$ and any answer to $\tau \sqsubseteq \omega \curvearrowright V - \sigma$ is also an answer to $\tau \sqsubseteq \omega \curvearrowright V$
- ii) If $\tau \parallel \sigma$, then any answer to $\tau \sqsubseteq \omega \curvearrowright V$ is also an answer to $\tau \sqsubseteq \omega \curvearrowright V - \sigma$

Theorem 3.60 Negated Non-Specializable Types on the Right Hand Side of $\sqsubseteq$ and Specializable Types on the Left Hand Side of $\sqsubseteq$
For any two natural types $\tau,\ \omega$, and any natural type σ such that any natural super type of σ is triangular we have:

- i) If $\tau \curvearrowright V \sqsubseteq \omega - \sigma$ is satisfiable then any answer to $\tau \curvearrowright V \sqsubseteq \omega - \sigma$ is an answer to $\tau \curvearrowright V \sqsubseteq \omega$ that is disjoint with σ
- ii) Any answer to $\tau \curvearrowright V \sqsubseteq \omega$ that is disjoint with σ is also an answer to $\tau \curvearrowright V \sqsubseteq \omega - \sigma$

Theorem 3.61 If A type μ is disjoint with a type σ, then any subtype of μ is also disjoint with σ.

Algorithm 3.62 We present an algorithm to find all of the most general natural subtypes μ_i of a natural type μ that are disjoint with another natural type σ This algorithm will be used in remark 3.63. We can use depth first search from μ and for each node visited check if the type associated with that node is disjoint with σ or not, if so, the depth first search at that branch terminates. The algorithm is shown in program 3.3.

```
function most_general_disjoint_subtypes($\mu$:natural, $\sigma$:natural) : set of natural types
/* any non-recursive call clears the visited types automatically */
begin
  if $\mu$ is already visited then return $\emptyset$ else mark $\mu$ as visited end if
  /* we will define disjoint(a, b) later */
  if disjoint($\mu$, $\sigma$) then return $\{\mu\}$
  $A \leftarrow \emptyset$
  /* for all children $c$ of $\mu$ in the type hierarchy do */
  for all $c$ such that $c \sqsubset_d \mu$ do
    $A \leftarrow A \cup$ most_general_disjoint_subtypes($c$, $\sigma$)
  end for
  return $A$
end
```

Program 3.3: Finding all of the most general subtypes of μ that are disjoint with σ.

Remark 3.63 Using theorem 3.60 we can reduce the problem of finding the answers of $\tau \curvearrowright V \sqsubseteq \omega - \sigma$, to the problem of finding the answers to $\tau \curvearrowright V \sqsubseteq \omega$ that are disjoint with σ. According to theorem 3.38, $\tau \curvearrowright V \sqsubseteq \omega$ has a most general specializable answer that we here refer to as $\mu \curvearrowright X$. There is no guarantee that μ is disjoint with σ. But we can find all n most general natural subtypes μ_i of μ that are disjoint with σ, for $1 \leq i \leq n$, using algorithm 3.62.

According to theorem 3.61 for all $1 \leq i \leq n$ any subtype μ' of μ_i is also disjoint with σ, and by theorem 3.60 μ' is an answer to V satisfying $\tau \curvearrowright V \sqsubseteq \omega - \sigma$.

On the other hand for any answer v to V satisfying $\tau \curvearrowright V \sqsubseteq \omega - \sigma$, theorem 3.60 ensures that v is disjoint with σ, and an answer to V satisfying $\tau \curvearrowright V \sqsubseteq \omega$, which by theorem 3.38 has a most general specializable answer $\mu \curvearrowright X$. According to the definition of most general specializable answers (definition 3.32), this needs v to be a subtype of μ. Now we run the algorithm shown in program 3.4.

When the loop ends, t is a most general natural subtype of μ that is disjoint with σ, which is also a super-type (not necessarily a strict one) of v. This means t must be equal to some μ_i, and because t is a super-type of v we can claim that $v \sqsubseteq \mu_i$.

So any answer to V satisfying $\tau \curvearrowright V \sqsubseteq \omega - \sigma$ is a subtype of some μ_i, for $1 \leq i \leq n$, and vice versa. This enables us to provide n most general specializable answers to V satisfying $\tau \curvearrowright V \sqsubseteq \omega - \sigma$. These answers are of the form for each $1 \leq i \leq n$:

$\mu_i \curvearrowright V_{h+i}$, for a fresh variable V_{h+i}, with an appropriate index h that is chosen such that

```
$t \leftarrow v$
while $t \neq \mu$ do
  /* since $t$ is a natural subtype of $\mu$, a parent of $t$ in the type hierarchy, */
  /* which we name $p$, must be a subtype of $\mu$ */
  /* (not necessarily a strict subtype) */
  $p \leftarrow$ a parent of $t$ that is a subtype of $\mu$
  if $p$ is disjoint with $\sigma$ then
    $t \leftarrow p$
  end if
end while
```

Program 3.4: Reaching a most general natural subtype of μ that is disjoint with σ

$h + i$ was not used as a variable index, or as a subscript in a specializable type before.

Theorem 3.64 Negated Specializable Types on the Right Hand Side of $\sqsubseteq$ and Specializable Types on the Left Hand Side of $\sqsubseteq$ (case 1)
For any two natural types τ, ω, and any natural type σ such that any natural super type of σ is triangular we have:

- i) Any answer to $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright V - \sigma$ is a common subtype of τ, and ω that is disjoint with σ.

- ii) Any common subtype of τ, and ω that is disjoint with σ is an answer to $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright V - \sigma$

Remark 3.65 The above theorem reduces the problem of finding the answers to $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright V - \sigma$ to the problem of finding the common subtypes of τ, ω that are disjoint with σ. By theorem 3.43, we know that any common subtype of τ, ω is a subtype of the most general common subtype. So we first check if the most general common subtype of τ, ω exists, if not, no answer to $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright V - \sigma$ can be found. Otherwise the most general common subtype of τ, ω is $\mu = \tau * \omega$, which is calculated by algorithm 3.44.

Next we find all n most general natural subtypes $\mu_i \sqsubseteq \mu$ that are disjoint with σ (by using algorithm 3.62) for $1 \leq i \leq n$. Then according to theorem 3.64 any subtype v of any of μ_i is an answer to $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright V - \sigma$.

On the other hand there is no answer v to $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright V - \sigma$ that is not a subtype of some μ_i, because theorem 3.64 requires v to be a common subtype of τ, ω that is disjoint with σ, and by running the algorithm presented in remark 3.63 we are able to prove that v is a subtype of a μ_i.

This enables us to find n specializable answers to $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright V - \sigma$ that cover any possible answer. These answers are of the form:

$\mu_i \curvearrowright V_{h+i}$, for a fresh variable V_{h+i}, with an appropriate index h that is chosen such that $h + i$ was not used as a variable index, or as a subscript in a specializable type before.

Lemma 3.66 For any two natural types ω, σ, and any natural subtype $\omega_1 \sqsubseteq \omega$ such that $\omega_1 \not\sqsubseteq \sigma$ we have:

$\omega_1 - \sigma \sqsubseteq \omega - \sigma$

Theorem 3.67 Negated Specializable Types on the Right Hand Side of $\sqsubseteq$ and Specializable Types on the Left Hand Side of $\sqsubseteq$ (case 2)

For any two natural types τ, ω, and any natural type σ such that any natural super type of σ is triangular we have:

- i) For any answer $(v_1,\ \omega_1)$ to $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright X - \sigma$, v_1 is an answer to $\tau \curvearrowright V \sqsubseteq \omega - \sigma$
- ii) Any answer v_1 to $\tau \curvearrowright V \sqsubseteq \omega - \sigma$, contributes to an answer (v_1, ω) to $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright X - \sigma$

Theorem 3.68 Negated Specializable Types on the Left Hand Side of $\sqsubseteq$ and Negated Non-Specializable Types on the Right Hand Side of $\sqsubseteq$

For any two natural types τ, ω, and any natural type σ such that any natural super type of σ is triangular, and that $\omega \not\sqsubseteq \sigma$ we have:

Any answer to $\tau \curvearrowright V - \sigma \sqsubseteq \omega - \sigma$ is an answer to $\tau \curvearrowright V - \sigma \sqsubseteq \omega$ and vice versa.

Theorem 3.69 Negated Non-Specializable Types on the Left Hand Side of $\sqsubseteq$ and Negated Specializable Types on the Right Hand Side of $\sqsubseteq$

For any two natural types τ, ω, and any natural type σ such that any natural super type of σ is triangular, and that $\tau \not\sqsubseteq \sigma$ and $\omega \not\sqsubseteq \sigma$ we have:

$\tau - \sigma \sqsubseteq \omega \curvearrowright V - \sigma$ is satisfiable if and only if $\tau - \sigma \sqsubseteq \omega - \sigma$ holds.

Theorem 3.70 Negated Specializable Types on the Both Sides of $\sqsubseteq$ (case 1)

For any two natural types τ, ω, and any natural type σ such that any natural super type of σ is triangular, and a variable V we have:

- a) Any answer to V satisfying $\tau \curvearrowright V - \sigma \sqsubseteq \omega \curvearrowright V - \sigma$ is a common subtype of τ, and ω that is not a subtype of σ.

- b) Any common subtype of τ, and ω that is not a subtype of σ is an answer to V satisfying $\tau \curvearrowright V - \sigma \sqsubseteq \omega \curvearrowright V - \sigma$.

Remark 3.71 Using theorem 3.70 we are able to reduce the problem of finding the answers to V satisfying $\tau \curvearrowright V - \sigma \sqsubseteq \omega \curvearrowright V - \sigma$, to finding the common subtypes of τ, and ω that are not a subtype of σ. And these values can be found by a call to `most_general_common_subtypes_not_contained_by(`τ`, `ω`, `σ`)`.

Theorem 3.72 Negated Specializable Types on the Both Sides of $\sqsubseteq$ (case 2)

For any two natural types τ, ω, and any natural type σ such that $\tau \not\sqsubseteq \sigma$, $\omega \not\sqsubseteq \sigma$, and any natural super type of σ is triangular, and two variables V, X we have:

- a) For any answer $(v_1,\ x_1)$ to $(V,\ X)$ satisfying $\tau \curvearrowright V - \sigma \sqsubseteq \omega \curvearrowright X - \sigma$ v_1 is an answer to V satisfying $\tau \curvearrowright V - \sigma \sqsubseteq \omega - \sigma$

- b) Any answer v_1 to V satisfying $\tau \curvearrowright V - \sigma \sqsubseteq \omega - \sigma$ contributes to an answer $(v_1,\ \omega)$ to $(V,\ X)$ satisfying $\tau \curvearrowright V - \sigma \sqsubseteq \omega \curvearrowright X - \sigma$

3.4 Extended Subtype Checking Algorithm

Now with the theory we developed about types, we are able to provide an algorithm that checks whether or not a subtype formula holds or is satisfiable. The information the algorithm needs is the following:

- set of simple types **SimTyp**

- set of well-formed natural types **NatTyp**

- direct subtype relation $\sqsubset_d$

- the information about the partitions that exist among the natural types, represented by a set P that contains all the partitions. That is:

$$P = \{(\tau,\ \{\tau_1, ..., \tau_n\}) \mid n \in \mathbb{N}\ \wedge$$
$$\tau \in \textbf{NatTyp}\ \wedge\ \forall 1 \le i \le n . \tau_i, \in \textbf{NatTyp}\ \wedge$$
$$\tau = (\tau_1 \mid ... \mid \tau_n)\}$$

The domain and the range of $\sqsubset_d$ are equal to **NatTyp**. And one can find whether or not two natural types are disjoint by checking if their composition is in **NatTyp**. However, as noted in the definition of natural type hierarchies (definition 3.13), the complete $\sqsubset_d$ relation can be huge, because it takes a space that is in $O(2^n)$, where n is the number of simple types.

An alternative is to keep the partial but necessary information of $\sqsubset_d$, by removing the obvious subtypes or super-types of composite types (as shown by theorem 3.9), and removing all the composite types that do not have a non-obvious immediate super-type or subtype, and accompany this partial information of $\sqsubset_d$ with the the necessary information of **NatTyp**. We use $\sqsubset_{d!}$ to denote the non-obvious direct subtype relationship.

An important note is that the information of **NatTyp** and P can be provided to the system gradually, and only if they are required, by making the system an interactive one. That is, the exact sets are not needed at the very beginning. Whenever a query to these sets is issued by the type checking algorithm, the system can see if it has enough information to respond to the query, and if the information it has is not enough, a question to the user can be issued to get the necessary information, which is then integrated to data base of the system, so that same questions are not issued repeatedly.

We begin by providing an interactive algorithm to check whether a given type is natural. We make this observation that if $t_1 * t_2 * ... * t_n$ is natural, then any composition of a subset of $t_1, ..., t_n$ is also a natural type (which is the super type of the original one). An efficient way of representing this is to keep the maximal set of simple types whose composition makes a natural type, and in a dual fashion keep the minimal set of simple types whose compositions is the bottom type. These maximal (minimal) sets are members of the larger set Nat^+ (Nat^-).

A naive algorithm using this idea is provided in program 3.5. The set of positive example of natural types is initially equal to:

$$\bigcup_{s \in \mathbf{SimTyp}} \{\{s\}\}.$$

In this algorithm we use a special subtype operator $\sqsubseteq'$ for sets. This operator checks that each member of its right hand side operand is a natural super-type of a member in its left hand side operand, that is:

$A \sqsubseteq' B \Leftrightarrow \forall b \in B;\ \exists a \in A;\ a = b \vee a \sqsubset_d b$

Using this operator in the algorithm avoids the unnecessary user interrogation when a

```
function is_natural($t$ : type) : boolean
  side effects include updates to $Nat^+$, $Nat^-$.
begin
  if $t = t_1 - t_2$ then return false
  if $t = t_1 \curvearrowright V$ and $V$ is a free variable then return false
  if $t = t_1 * t_2 * ... * t_n$ (with $t_i \in$ SimTyp) then
    $S \leftarrow \{t_1,\ t_2, ...,\ t_n\}$
    remove any $t_i$ that is a natural super-type of some $t_j$ in $S$, where $i \neq j$
    for every $Maxi \in Nat^+$ do
      if $S \subseteq Maxi$ then return true
      if $Maxi \sqsubseteq' S$ then return true
    end for
    for every $Mini \in Nat^-$ do
      if $Mini \subseteq S$ then return false
      if $S \sqsubseteq' Mini$ then return false
    end for
    /* The information stored in the system is not enough to answer this query */
    /* User is asked to answer whether $t$ is a natural type */
    print "is " + $t$ + " a natural type?"
    answer $\leftarrow$ get user input
    if answer is yes then
      $Nat^+ \leftarrow Nat^+ \cup \{S\}$
      for every $Maxi \in Nat^+$ do
        if $Maxi \subset S$ then
          remove $Maxi$ from $Nat^+$
        end if
      end for
      return true
    else
      $Nat^- \leftarrow Nat^- \cup \{S\}$
      for every $Mini \in Nat^-$ do
        if $S \subset Mini$ then
          remove $Mini$ from $Nat^-$
        end if
      end for
      return false
    end if
  end if
  return false
end
```

Program 3.5: Interactive natural type detection algorithm portion of S is a set whose elements are all subtypes of a minimal negative example in Nat^-. In other words, if the composition of the types in $Mini$ is not a natural type, then the composition of a set of types containing subtypes of every type in $Mini$ is not a natural type either.

Also in a dual fashion, if every element of S is a natural super-type of some member of a maximal positive example $Maxi$ then the composition of the elements in S will result a natural type.

This algorithm can be improved by dealing with the special cases of triangular types, specifically *human*. That is, if $human \in S$, then the composition of elements of S is a natural type if and only of every element of S is a natural super-type or subtype of *human*. This will result in less unnecessary interactions with the user.

This algorithm requires subtype checking among natural types which will be discusses in the next subsection.

Now we can provide an algorithm that checks whether two types are disjoint or not. The algorithm is shown in program 3.6.

```
function disjoint($t_1$ : natural, $t_2$ : natural) : boolean
begin
  if is_natural($t_1 * t_2$) then return false
  return true
end
```

Program 3.6: Simple algorithm to check if two natural types are disjoint.

The set P can be broken down in a similar way to P^+, and P^-. The information in Nat^+, and Nat^- can also help answer a query about partitions, because for some types to form a partition it is necessary that they are disjoint. The interactive algorithm about partitions is shown in program 3.7.

3.4.1 Checking Subtype Formulas Among Natural Types

Note that natural types are present in a complete type hierarchy diagram, and according to the definition of direct subtypes (definition 3.11) we know that the subset relation $\sqsubseteq$ over natural types is equal to the transitive closure of $\sqsubseteq_d$. The complete relation $\sqsubseteq_d$ can be calculated by adding the obvious subtype relation to $\sqsubset_{d!}$. The obvious portion is the following relation:

```
function is_partition($t$ : natural or a variable, $t_1, ..., t_n$ : natural) : boolean
  side effects include updates to $t$, $P^+$, $P^-$, $Nat^-$.
begin
  /* The following call instantiates $t$ upon success if it is a variable */
  if $(t,\ \{t_1, ...\ t_n\}) \in P^+$ return true
  if $(t,\ \{t_1, ...\ t_n\}) \in P^-$ return false
  for all $N \in Nat^+$ do
    if $|\{t_1, ...\ t_n\} \cap N| > 1$ then
        $P^- \leftarrow P^- \cup \{(t,\ \{t_1, ...\ t_n\})\}$
        return false
  end for
  /* The information stored in the system is not enough to answer this query */
  /* User is asked to answer whether $t, t_i$s make a partition or not */
  print "is it true that " + $t = (t_1|...|t_n)$ + "?"
  answer $\leftarrow$ get user input
  if answer is yes then
    if $t$ is a variable then
      print "enter the type for $t$", $t \leftarrow$ get user input
    end if
    $P^+ \leftarrow P^+ \cup \{(t,\ \{t_1, ...\ t_n\})\}$
    for every $1 \leq i < j \leq n$ do
      $Nat^- \leftarrow Nat^- \cup \{\{t_i, t_j\}\}$
      for every $Mini \in Nat^-$ do
        if $\{t_i, t_j\} \subset Mini$ then
          remove $Mini$ from $Nat^-$
        end if
      end for
    end for
    return true
  if answer is no then
    $P^- \leftarrow P^- \cup \{(t,\ \{t_1, ...\ t_n\})\}$
    return false
end
```

Program 3.7: Interactive partition algorithm

```
function subtype_natural($\tau$, $\sigma$) :  boolean
/* any non-recursive call clears the visited types automatically */
begin
  if $\tau$ is already visited then return false else mark $\tau$ as visited end if
  if $\tau \sqsubseteq_d \sigma$ then return true
  for all $\tau' \neq \tau$ such that $\tau \sqsubseteq_d \tau'$ do
    if subtype_natural($\tau'$, $\sigma$) = true then
      return true
    end if
  end for
  return false
end
```

Program 3.8: Subtype checking over natural types, version I

$$\mathit{Obvious} = \{(x_1 * ... * x_n, y_1 * ... * y_m) \mid x_1 * ... * x_n \in \textbf{NatTyp} \wedge \{y_1, ..., y_m\} \subseteq \{x_1, ..., x_n\}\}$$

So:

$$\sqsubseteq_d \ = \ \mathit{Obvious} \cup \sqsubset_{d!}$$

A naive algorithm that follows the transitive edges of the direct subtype relation is presented in program 3.8. Note that this algorithm basically performs a DFS on the complete type hierarchy, and its time complexity is in $O(2^{2n})$, where n is the number of simple types. This complexity is pretty bad, and with an observation that we see in what follows we are able to improve it from exponential to polynomial in terms of the size of the inputs. The complexity of the better algorithm will be $O(n^2 + m)$, where $m = |\sqsubset_{d!}|$ is the number of edges of the non-obvious portion of the natural type hierarchy, which is the size of the subtype information that is needed to be provided by the user.

An Observation on the Immediate Super-types of a Composite Type

Theorem 3.73 A composite type $x_1 * x_2 * ... * x_n$ cannot have any immediate super-type other than $x_1, x_2, ..., x_n$.

Proof: By contradiction suppose that $x_1 * x_2 * ... * x_n$ also has an immediate super-type y which is not equal to any x_i. Then by theorem 3.14 $x_1 * y$ is a well-formed natural type and we have:

$$x_1 * x_2 * ... * x_n \sqsubseteq x_1 * y$$

Since by theorem 3.9 we have $x_1 * y \sqsubseteq y$, $x_1 * y$ is a less general super-type of $x_1 * x_2 * ... * x_n$ than y. And this contradicts the assumption that y is an immediate super-type of $x_1 * x_2 * ... * x_n$. ■

Example 3.74 An example in C++ and its corresponding type hierarchy in our formalism is shown in figure 3.2. The composite type itself as we use it in our formulation is not an explicit type in OOP languages.

```
class X1 {
  /* some stuff */
}
class X2 {
  /* some other stuff */
}
class X : public X1, public X2 {
  /* some more stuff */
}
```

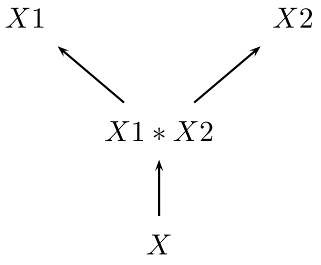


Figure 3.2: A simple C++ example for multiple inheritance, with its type hierarchy in our formalism.

By theorem 3.73, we know there is no composite type $x_1 * ... * x_n$ participating at the left hand side of $\sqsubset_{d!}$, because the only direct super types are $x_1, ..., x_n$ which are obvious super-types that are not included in $\sqsubset_{d!}$. And the pairs in *Obvious* have as their second component only simple types.

With this observation, we can provide a more efficient algorithm for `subtype_natural` that operates on $\sqsubset_{d!}$ instead of $\sqsubseteq_d$. This algorithm[5] is shown in program 3.9.

The time complexity of the new algorithm is in $O(n^2 + m)$ where $m = |\sqsubset_{d!}|$ is the size of the direct subtype information that should be provided by the user, and n is the number of simple types. The argument is as follows. First note that the algorithm is recursive only in terms of its first argument. We say the algorithm visits z if a call to the algorithm is made with z as the first argument. Also note that the algorithm is a DFS on a graph including the edges in $\sqsubset_{d!}$ plus the of edges in the graph of $Obvious'$, where $Obvious' \subseteq Obvious$, and the pairs in $Obvious'$ are restricted to have as their first component the composite types

[5] In this algorithm we suppose no two x_i, and x_j can be found such that $x_i \sqsubseteq x_j$. The same condition applies for y_i's. To remove this limitation several more recursive calls must be added that for simplicity we do not discuss. Our implementation, however, works generally for all cases with no such limitation.

```
function subtype_natural($\tau$, $\sigma$) : boolean
/* any non-recursive call clears the visited types automatically */
begin
  if $\tau = \sigma$ then return true
  if $\tau$ is already visited then return false else mark $\tau$ as visited end if
  /* see footnote 5 */
  let $x_1 * ... * x_p \leftarrow \tau$
  let $y_1 * ... * y_m \leftarrow \sigma$
  if $\{y_1, ..., y_m\} \subseteq \{x_1, ..., x_p\}$ then return true
  if $p > 1$ then
    for all $1 \leq i \leq p$ do
      if subtype_natural($x_i$, $\sigma$) = true then
        return true
    end for
  end if
  for all $\tau'$ such that $\tau \sqsubset_{d!} \tau'$ do
    if subtype_natural($\tau'$, $\sigma$) = true then
      return true
  end for
  return false
end
```

Program 3.9: Subtype checking over natural types, version II

that have a simple type child, plus a pair for the composite type τ for the first call to the algorithm (if it is a composite type). The reason for this choice of $Obvious'$ is that the algorithm will not visit any other composite type along its search. All the composite types visited are the first composite type presented to the algorithm as the first argument plus the composite types that are parents of some simple type. The number of composite types used as a parent of a simple type in the type hierarchy cannot be more than the number of simple types itself. So the graph of $Obvious'$ cannot contain more than $2 \times n + 1$ nodes, which is in $O(n)$. The number of edges of this graph is in $O(n^2)$. So the total number of edges that are traversed by the algorithm is in $O(n^2) + O(|\sqsubset_{d!}|)$, which will be the complexity of the DFS algorithm.

3.4.2 The General Case

Based on theorems 3.38 - 3.72 for different cases of a subtype checking problem we can define a recursive algorithm to determine the truth value or satisfiability of a subtype formula

$A \sqsubseteq B$.

The algorithm is provided in appendix A. It is a function with the signature:

```
function subtype(A : BasTyp, B : BasTyp) :  set of answers
```

We assume all negated types use triangular types whose parents are also triangular. The return value of the function is the *nil* set if the formula does not hold or is not satisfiable. If the formula holds or is satisfiable and no variables occur in A, then the return value is an empty set. If a variable occurs in A, and the formula is satisfiable, then the return value is the set of answers (specializable or not).

3.5 Type Restriction Satisfaction through Subtype Checking

Now that we have an algorithm to determine whether $A \sqsubseteq B$ for all basic types A, B (with the restrictions of triangular types in negated types), we can redefine the satisfaction of type restrictions by the following definition.

Definition 3.75 Type Restriction Satisfaction (Version III)
A type restriction TR of a sense for an expression is satisfied in a phrase if and only if for every pair $(role, type) \in TR$, the expression ξ that assumes the grammatical role *role* in the phrase, refers to an object of type τ, where the following formula holds or is satisfiable from a consistent set of suppositions:

(12) $\tau \sqsubseteq type$

Next we prove that versions II, and III of type restrictions are equivalent.

Theorem 3.76 Equivalence of Type Restriction Versions II, III

Proof:

- Suppose that type restriction satisfaction version II holds, that is, we are able to infer $\xi : type'$ from $\xi : \tau$, where $type'$ is an instantiation of *type*. Since the type axioms do not rely on the internal structure of ξ but only on its type, ξ can be replaced by an arbitrary symbol x, and we can derive $x : type'$ using the same type axioms and the same procedure used to derive $\xi : type'$. By the definition of instance sets we get:

  $I_\tau \subseteq I_{type'}$

  which is equivalent to:

$\tau \sqsubseteq type'$

which means $\tau \sqsubseteq type$ holds (if it contains no variables) or is satisfiable (if it has a variable), because $type'$ is just an instantiation of $type$. On the other hand version II requires that $\neg(\xi : type')$ not be derivable. This means there is no contradiction in place, or otherwise everything could be derived. So type restriction satisfaction version III holds. □

- Suppose that type restriction satisfaction version III holds, that is, $\tau \sqsubseteq type$ holds or is satisfiable, with no contradictions in place.

  This means for an instantiation $type'$ of $type$ we have:

  $\tau \sqsubseteq type'$

  And by (SUB) from $\xi : \tau$ we can derive $\xi : type'$

  This means $\neg(\xi : type')$ cannot be derived, or otherwise we would have a contradiction in the system which is contrary to the supposition. So type restriction satisfaction version II holds. □

■

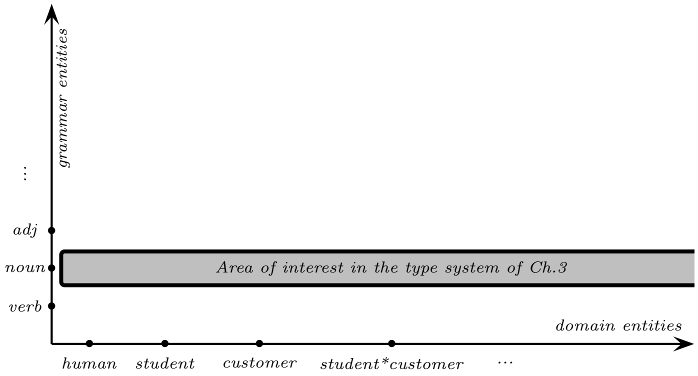
3.6 Conclusion

Our contribution in this chapter is a comprehensive type system that includes multiple inheritance, negated types and specializable types. A detailed theory was laid out and we studied how the notion of type restrictions as defined in the previous chapter can be defined in terms of the type system we developed.

In the next part of the thesis we introduce the HPSG formalism and will eventually combine this type system with it.

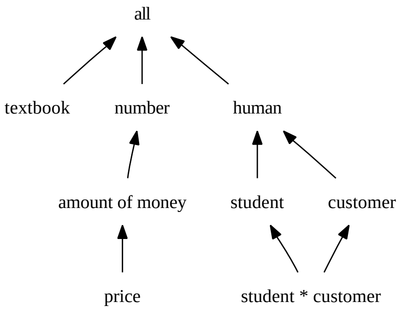
Part II: The Grammar

In this part of the thesis we combine the type theory of chapter 3 with the HPSG formalism. In doing so, however, we keep in mind the two distinct type systems in play. One is the type system for the grammatical categories that are well-developed in terms of the logic of typed feature structures (see for example Carpenter [12]). This type system provides a type theory for *grammar entities*, such as *sentences*, *verbs*, *nouns*, *adjectives*, and etc. Another is the type theory that describes the semantic domain of the grammar application. The types in this system refer to the types of the *domain entities*.



The two dimensional type system for the integrated grammar.

As a result we see the type system of an integrated grammar in two non-conflicting dimensions, as depicted in the figure above. The first dimension is the *grammar entity* dimension, the other is the *domain entity dimension.* An example of the domain type hierarchy is shown on the next page. This hierarchy is taken from the second chapter. For an example of the grammar entity type hierarchy see figure 4.1 in section 4.6.



The bookstore domain entity type hierarchy

These dimensions can be observed in categorial grammar in a smaller scale. λ-types are the types that govern the semantics, including the domain entity types which have been represented solely by **Ind** in simple existing theories. What we did in chapter 3 can be thought of as extending the type hierarchy below **Ind** by providing the necessary theoretical means. The grammar entity dimension in the categorial grammar is the category system, that included categories like S, N, NP/N, NP\S, and etc.

The organization of the chapters in this part is as follows. First in chapter 4 we present an introduction to the important concepts of HPSG with some background. We present the grammar entity type hierarchy for our grammar, with very brief discussion of what our contributions and modifications are. Then in chapter 5 we go through the syntactic features of our grammar with the grammar rules and principles that govern syntax, together with some examples. Next in chapter 6 we go through the semantic features of our grammar with some modifications to the grammar rules and some additional grammar principles to incorporate the semantics. Finally in chapter 7 we discuss how type restrictions can be encoded in our grammar.

Chapter 4

Introduction to HPSG

4.1 Background

The grammar that we develop in this and the next chapters is based on Sag et al. "Syntactic Theory : a formal introduction" [67] which describes a theory of grammar that is most closely related to the Head-driven Phrase Structure Grammar (HPSG) formalism. HPSG [61, 60] mainly evolved from Generalized Phrase Structure Grammar (GPSG) [37, 38] and is also influenced by ideas from categorial grammar (CG), which we looked into very briefly in chapter 2, arc pair grammar (APG) [45], lexical-functional grammar (LFG) [25], semantics, and also computer science (ideas from data type theory, knowledge representation, unification-based formalisms).

GPSG is an extension of CFG that overcomes some shortcomings of CFG by at least providing *complex categories* and *meta rules* [38]. In GPSG, grammatical categories are not taken to be simple monadic labels (such as V, N, VP, NP), but have internal structures in the form of *features*. Metarules capture some generalizations among grammar rules. A metarule can be thought of as a rule that operates on other grammar rules instead of grammar categories. Features are heavily used in HPSG, however metarules are not. In fact some GPSG metarules can be encoded via HPSG lexical rules (for an example see page 174 of [61]) that we see later. Another idea used in GPSG is the immediate dominance (ID) rules and linear precedence (LP) rules. ID rules roughly specify which phrases can appear as daughters of the mother phrase that is licensed by that rule. LP rules specify constraints about the order of daughters. These ideas are preserved in HPSG but in this thesis we do not use them, because our grammar is based on [67] and ID/LP rules are not used there either. We only use phrase structure rules that specify daughters with their order simultaneously, as in CFG rules.

4.1.1 Features

Features in GPSG are pairs of attribute names and attribute values that are usually represented by Attribute Value Matrices (AVMs). An AVM for a structure with n features looks like a $n \times 2$ matrix as shown in (1), where each row corresponds to a feature. The first column is for the feature name, and the second column is used for the feature value. Rows and columns could be missing in case the value for the corresponding features is not specified.

(1) $$\begin{bmatrix} FEATURE_1 & value_1 \\ FEATURE_2 & value_2 \\ \multicolumn{2}{c}{...} \\ FEATURE_n & value_n \end{bmatrix}$$

A structure like above that comprises a set of features is called a *feature structure*. Features in GPSG have a range of permissible values. These values however are atomic, in the sense that they are not feature structures themselves (feature structures are not nested). A complex scheme that allowed nested feature structures was proposed in [38], however it was dismissed in later GPSG work [37]. Nested feature structures are heavily used in HPSG. We should mention that the use of complex categories in linguistics is not specific to GPSG, however, the importance of complex categories was not fully recognized in some linguistic circles until the emergence of GPSG [11]. Although, complex categories in the form of logic grammar symbols have been used in logic programming since the late 70's [17].

4.1.2 How Features Can Help

An example [67, 18] of how features can be helpful is the *subject-verb agreement* phenomenon in English combined with the transitivity of the verbs. A very simple CFG grammar of English is provided in (2a) - (2d). This grammar combines a subject with a verb phrase to build a sentence. A verb phrase can be built from an *intransitive* (IV), *transitive* (TV) or *di-transitive* verbs (DTV) with different number of objects.

(2) a. S → NP VP

b. VP → IV

c. VP → TV NP

d. VP → DTV NP NP

In present tense English sentences with third-person subjects, the verb must agree with its subject in *number* (plurality, or singularity)[1]. For example the sentence (3a) is not acceptable, whereas (3b) is acceptable. At the same time (3c) is acceptable but (3d) is not.

(3) a. * John write.

b. John writes.

c. The students write.

d. * The students writes.

This cannot be handled by the simple CFG rules of (2a) - (2d) alone, because they license all of the above sentences. In technical terms, these rules *over-generate*. To solve this problem, one can break down the categories VP, and NP to smaller categories in which the number is specified. These categories can be named VP-SG for singular verb phrases, VP-PL for plural verb phrases, NP-SG for singular noun phrases, and NP-PL for plural noun phrases. However, this requires the categories IV, TV, and DTV to be refined too. The new categories will be IV-SG, IV-PL, TV-SG, TV-PL, DTV-SG, and DTV-PL. Using these new categories the grammar can be revised to (4a) - (4h).

(4) a. S → NP-SG VP-SG

b. S → NP-PL VP-PL

c. VP-SG → IV-SG

d. VP-PL → IV-PL

e. VP-SG → TV-SG NP

[1] This agreement is not specific to present tense as we will see later. However for the sake of this example it suffices to focus on the present tense.

f. VP-PL → TV-PL NP

g. VP-SG → DTV-SG NP NP

h. VP-PL → DTV-PL NP NP

This effectively doubles the number of rules in the grammar. The problem is that a generalization is being missed, which is the number of the verb and its subject must agree without the need of actually specifying the number. The situation gets more tedious when new rules are added and new linguistic phenomena are considered. To avoid this problem, we can think of VP, and NP as complex categories with a feature NUMBER (NUM) that can take values from {SG, PL}. Then the grammar would become:

(5) a. S → NP[NUM *num*] VP[NUM *num*]

b. VP[NUM *num*] → IV[NUM *num*]

c. VP[NUM *num*] → TV[NUM *num*] NP

d. VP[NUM *num*] → DTV[NUM *num*] NP NP

As we see above, the value of NUM feature (*num*) is repeated for each rule. As we mentioned earlier, HPSG allows nested features, and the repetition of nested features in rules and structures can become cumbersome. To avoid the repetition of the shared values in AVMs *tags* are used. A tag is a numbered box that appears in place of the shared value (as in (6a) - (6c) below), or right before the shared value (with no spaces) in at least one occurrence of the value if the value must be shown (as in (6d) below). Tags with equal numbers represent the same values (this corresponds to same-name variables in a clause of logic programs).

(6) a. $\text{S} \rightarrow \text{NP}\begin{bmatrix}\text{NUM} & \boxed{1}\end{bmatrix} \text{VP}\begin{bmatrix}\text{NUM} & \boxed{1}\end{bmatrix}$

b. $\text{VP}\begin{bmatrix}\text{NUM} & \boxed{1}\end{bmatrix} \rightarrow \text{IV}\begin{bmatrix}\text{NUM} & \boxed{1}\end{bmatrix}$

c. $\text{VP}\begin{bmatrix}\text{NUM} & \boxed{1}\end{bmatrix} \rightarrow \text{TV}\begin{bmatrix}\text{NUM} & \boxed{1}\end{bmatrix} \text{NP}$

d. $\text{VP}\begin{bmatrix}\text{NUM} & \boxed{1}\mathit{num}\end{bmatrix} \rightarrow \text{DTV}\begin{bmatrix}\text{NUM} & \boxed{1}\end{bmatrix} \text{NP NP}$

Later we see that with more advanced features, and more general grammar principles the number of rules for intransitive, transitive and di-transitive verbs can be further reduced.

4.2 Key Characteristics of HPSG

4.2.1 Use of Typed Feature Structures

Every feature structure in HPSG is labeled with a type. This label is shown at the top of the AVM representing it, as shown in (7).

(7)
$$\begin{bmatrix} \mathit{type} & \\ FEATURE_1 & value_1 \\ FEATURE_2 & value_2 \\ \multicolumn{2}{c}{...} \\ FEATURE_n & value_n \end{bmatrix}$$

Similar to the notion of appropriateness in Carpenter's logic of typed feature structures [12], each feature structure type must declare which features are introduced by it. Also each feature needs to be associated with a type, such that the values appropriate for that feature can be restricted if necessary. These types form a type hierarchy. Every feature defined by a type τ is inherited by all subtypes of τ in the type hierarchy. Each type may introduce a set of constraints on the values that some of its features can take. These constraints are inherited by all subtypes of the original type introducing them. A type in the hierarchy can be associated with atomic values (atomic feature structures), e.g., sg, pl, or $\mathit{1}, \mathit{2}, \mathit{3}$.

Distinction of Feature Structure Descriptions and Feature Structures

Unless an AVM of type τ specifies all features introduced or inherited by the leaf type τ in the type hierarchy, it is a *feature structure description* rather than a feature structure. Feature structure descriptions are different from feature structures as formulated by Sag et al. in [67]: "A description may be partial in not specifying values for every feature, in specifying only part of the (complex) value of a feature, in failing to specify a type, or in specifying nothing at all". However a feature structure is complete, that is, it specifies values for every feature appropriate for its type. From this aspect, it is a total function from features appropriate for its type to appropriate values.

A rigorous definition of feature structure description and feature structures together with feature structure description satisfaction by a feature structure is given in [67]. Roughly, a feature structure $\mathcal{FS}$ satisfies a feature structure description d of type τ, iff:

- ∘ $\mathcal{FS}$ is of type τ', where τ' is a most specific subtype of τ, i.e., τ' is a leaf in the type hierarchy that lies in the sub-DAG rooted by τ.
- ∘ $\mathcal{FS}$ provides a value V for each feature F of type σ introduced or inherited by τ' (and no other feature), such that V is a feature structure or an atomic value of type σ, and compatible with the value of F in d (if specified).
- ∘ $\mathcal{FS}$ obeys any constraint that the grammar associates with τ' or any of its super-types.

A feature structure description describes a set of feature structures, more specifically those feature structures that satisfy that description. Finally a feature structure description that describes the empty set is invalid [18]. An invalid feature structure description is denoted by $\perp$ (same as the bottom type).

Definition 4.1 Conjunction (Unification) of Two Feature Structure Descriptions
Conjunction or unification of two feature structure descriptions d_1, d_2 results in a feature structure description d that describes a set of feature structures that is the intersection of the set of feature structures described by d_1, and d_2 [18]. Conjunction of d_1, and d_2 is denoted by d_1d_2 (no spaces in between) or d_1 & d_2. [2]

Example 4.2 Suppose X, and Y below are feature structure descriptions.

$$\mathrm{X} = \begin{bmatrix}\mathrm{NUM}\ \mathit{sg}\end{bmatrix}$$
$$\mathrm{Y} = \begin{bmatrix}\mathrm{PER}\ \mathit{3}\end{bmatrix}$$

Then the unification of X, and Y will result in:

$$\mathrm{XY} = \mathrm{X}\ \&\ \mathrm{Y} = \begin{bmatrix}\mathrm{PER}\ \mathit{3}\\ \mathrm{NUM}\ \mathit{sg}\end{bmatrix}$$

But the following unification fails, as the result is the invalid feature structure description, because the values of the feature PER are inconsistent in the descriptions.

$$\mathrm{Y}\begin{bmatrix}\mathrm{PER}\ \mathit{1}\\ \mathrm{NUM}\ \mathit{sg}\end{bmatrix} = \perp$$

[2] The conjunction (unification) operator is denoted by $\sqcup$ by Carpenter in [12], $\sqcap$ is used by Copestake in [18], and & is used by Sag et al. in [67], and by Erbach in the ProFIT language.

4.2.2 Constraint Based

HPSG is a constraint based formalism, in the sense that well-formed phrases are licensed on the basis of satisfaction of a set of constraints on the constituents of the phrase. These constraints are expressed by a set of *grammar rules*, a set of *grammar principles* and the constraints posed by feature structure descriptions of the phrasal constituents. Note that feature structure descriptions are actually constraints on the features and feature values in feature structures that satisfy them. We will introduce rules and principles in the following sections. Logical term *unification* has been a typical method of solving and satisfying these constraints. For this reason HPSG is also called a unification-based approach in some texts, but Sag et al. [67] mention this could be misleading, as unification is just a method of satisfying the constraints, and it is the constraints themselves that are important.

4.2.3 Sign Based

HPSG is a system of *signs*. A sign in HPSG is considered to be a structured complex of phonological, syntactic, and semantic information [67, 61]. In this thesis we use lists of English orthographies for phonological forms for the sake of simplicity, as done also in [61, 67, 18]. A sign can be represented by the following feature structure description:

$$(8)\quad \begin{bmatrix} \textit{sign} \\ \text{ORTH} \\ \text{SYN} \\ \text{SEM} \end{bmatrix}$$

Values of the features ORTH, SYN, and SEM correspond to the orthography, syntax and semantics of the sign respectively.

With this definition of a sign, *words*, *phrases* and *lexical entries* can all be thought of as signs.[3] Words and phrases can be thought of as feature structures of types *word*, *phrase* respectively. However the level of generalization can go further. There are a family of words that are closely related to each other in phonology, syntax and semantics, such as the group of words shown in (9):

[3] Note that our treatment of lexical entries in categorial grammar as defined in definition 2.15 matches the notion of signs in HPSG.

(9) *love, loving, loves, loved, ...*

In linguistics the basic information about these words is gathered in structures called *lexemes*. *Lexical rules* transform lexemes into words or other *intermediate lexemes*, with some modification to the phonology, syntax and the semantics. In HPSG lexemes are also signs, and lexical entries can be thought of as lexemes. A lexeme is a feature structure description of type *lexeme*. By using lexemes and lexical rules in HPSG (and in any other theory of natural language) a great amount of generalization is achieved and the size of the lexicon is effectively reduced.

4.2.4 Importance of the Notion of Heads

The grammar rules and principles of HPSG often refer to the notion of heads. By using heads in rules and principles we are able to achieve significant generalizations. Here is an informal definition of a head in a phrase.

Definition 4.3 Heads
The *head* of a phrase is its obligatory constituent. For example, in a *noun phrase*, *noun* is the head, in a *verb phrase*, *verb* is the head. From a semantic point of view, a headed phrase[4] describes a kind of what its head describes [6]. For example the noun phrase *sharp pencil* describes a kind of *pencil*, the verb phrase *walking to school* describes a kind of *walking*, and so on.

Heads determine some syntactic and semantic properties of their mother phrases, and some of their sisters, as we see later in this chapter. Factoring what the mother phrase and its head daughter have in common enables us to reduce the number of grammar rules by a large amount. The idea of heads is not specific to HPSG. GPSG used it [37], however in HPSG this idea is more emphasized.

4.2.5 Surface Oriented

Like CFG, the HPSG grammar provides structures that are in a simple correspondence with the string of words in a sentence (or phrase). The decision whether or not a sentence is grammatically valid is easily derivable from the feature structures of each word comprising it.

[4] Not all phrases are headed. As an example, a coordinated noun phrase is headless.

4.2.6 Strongly Lexicalist

HPSG is said to be strongly lexicalist since it needs a lot of syntactical and semantical information to be encoded in lexical entries. This makes a sizable lexicon but dramatically reduces the number of grammar rules needed.

4.3 General Format of Phrase Structure Rules in this Thesis

A phrase structure rule, is a rule that specifies how a phrase can be built from smaller constituents. CFG rules are examples of phrase structure rules. In HPSG however, instead of monadic category labels, we use complex categories that are feature structure descriptions. These extended rules can look like 6a - 6d. But note that categories like S, NP, VP are feature structure descriptions for the complex grammar categories that we investigate in detail in the following sections. As a convention, if X is a category, and $[\mathcal{FD}]$ is a feature structure description, then X$[\mathcal{FD}]$ is a complex category and the feature structure description that results from conjoining (unifying) the feature structure description of X, and $[\mathcal{FD}]$ (see definition 4.1).

With these notes and conventions the format of an extended phrase structure (PS) rule is one of the following, depending on whether it is a headed rule or not:

(10) Phrase → Daughter_1 Daughter_2 ... Daughter_n

(11) Phrase → Daughter_1 ... **H**Daughter_h ... Daughter_n

where Phrase is the feature structure description and the complex grammar category of the mother phrase, and Daughter_i is the feature structure description and the complex grammar category of the i'th daughter constituent of the phrase. This rule as it is specifies the order of daughters in the mother phrase to be the same as the order they appear on the right hand side of →.[5]

If the rule is not headed (that is the mother phrase is not a headed phrase), format (10) is used, but if it is headed format (11) is used. A bold **H** is placed right before the head

[5] If the daughters are separated by commas, the rule will become an ID (immediate dominance) rule, which puts no constraint whatsoever on the order of daughters. To complement ID rules, LP (linear precedence) rules are needed that put constraints on the daughters of *any* phrase, but we do not use ID and LP rules in this thesis.

daughter. Note that there is only one head in a headed phrase and headed rule. As we will see later, the mother phrase shares some important features with its head daughter. Also the head daughter can specify some feature values of its sisters.

An HPSG rule can have a variable number of daughters on the right hand side of →. Sometimes it is the head daughter which determines the number of its sisters.

4.4 Satisfaction of an Extended Phrase Structure Rule

In this subsection we abstractly provide the formal definitions of word structure and phrase structure licensing. Examples are given in the next sections using real grammar rules.

Definition 4.4 Well-formed Phrase Structure Licensing
A grammar rule ρ = Phrase $\rightarrow$ Daughter$_1$ ... Daughter$_n$, headed or not headed, is said to *license* a well-formed *phrase structure* Φ_0, if and only if there are feature structures $\Phi_0, \Phi_1, ..., \Phi_n$ such that:

- for each $i > 0$, Φ_i is a well-formed phrase structure that is licensed by a grammar rule or is a well-formed word structure that is licensed by a sequence of lexical rules.
- for every $i > 0$, Φ_i satisfies Daughter$_i$ and every constraint that is associated with its feature structure type.
- Φ_0 satisfies Phrase and every constraint that is associated with its feature structure type.
- Φ_0, Φ_1, ... Φ_n respect all grammar principles applied on ρ.

In this case it is also said that Φ_0, Φ_1, ... Φ_n satisfy ρ (and all grammar principles). This defines a well-formed phrase structure recursively, however the base case of the recursion which relies on the notion of lexical rules and well-formed word structures is still not presented. This is defined next.

Definition 4.5 Well-formed Word Structure Licensing
A sequence of lexical rules $\rho_0, ..., \rho_m$ *license* a well-formed *word structure* Φ if there is a lexical entry *lex-entry* = lex_0 in the lexicon, and a list of lexemes lex_1, ..., lex_m, and a feature structure description W of type *word* such that:
ρ_0 derives lex_1 from lex_0
ρ_1 derives lex_2 from lex_1

...

ρ_m derives W from lex_m

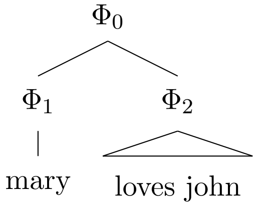
and

Φ is a feature structure that satisfies W.

4.5 Phrase Structure Trees

For any given rule ρ = Phrase $\rightarrow$ Daughter$_1$ ... Daughter$_n$, headed or not headed, if Φ_0 is a phrase structure licensed by ρ, and each Φ_i is either a phrase structure or a word structure licensed by the grammar, then the tree with the root Φ_0, and the daughters Φ_1, ..., Φ_n is called a *phrase structure tree* licensed by the rule $\rho =$. For every $1 \leq i \leq n$, the ith daughter can itself be a phrase structure tree rooted by Φ_i.

However, for any subtree in the phrase structure tree, the bottom level must be the string of words that correspond to the parent word structure or phrase structures, like the following phrase structure tree:



4.6 The Type Hierarchy of Grammar Entities

Before we start looking at the grammar constraints, rules, and principles, in this section we present the type hierarchy that we use for our grammar, and list the features and their appropriate types for each grammar entity. The type hierarchy is shown in figures 4.1, and 4.2. Features introduced by each type are shown in AVMs below the type. In appendix C we have provided a table that lists each feature structure type with its features and the permissible values for each feature.

Our contributions or modifications are underlined. Our most important contributions are:

- TYPE semantic feature in *sem-cat*, which provides a type from **BasTyp** for the object that the expression refers to (which is **Bool** for verb phrases, and a domain entity type

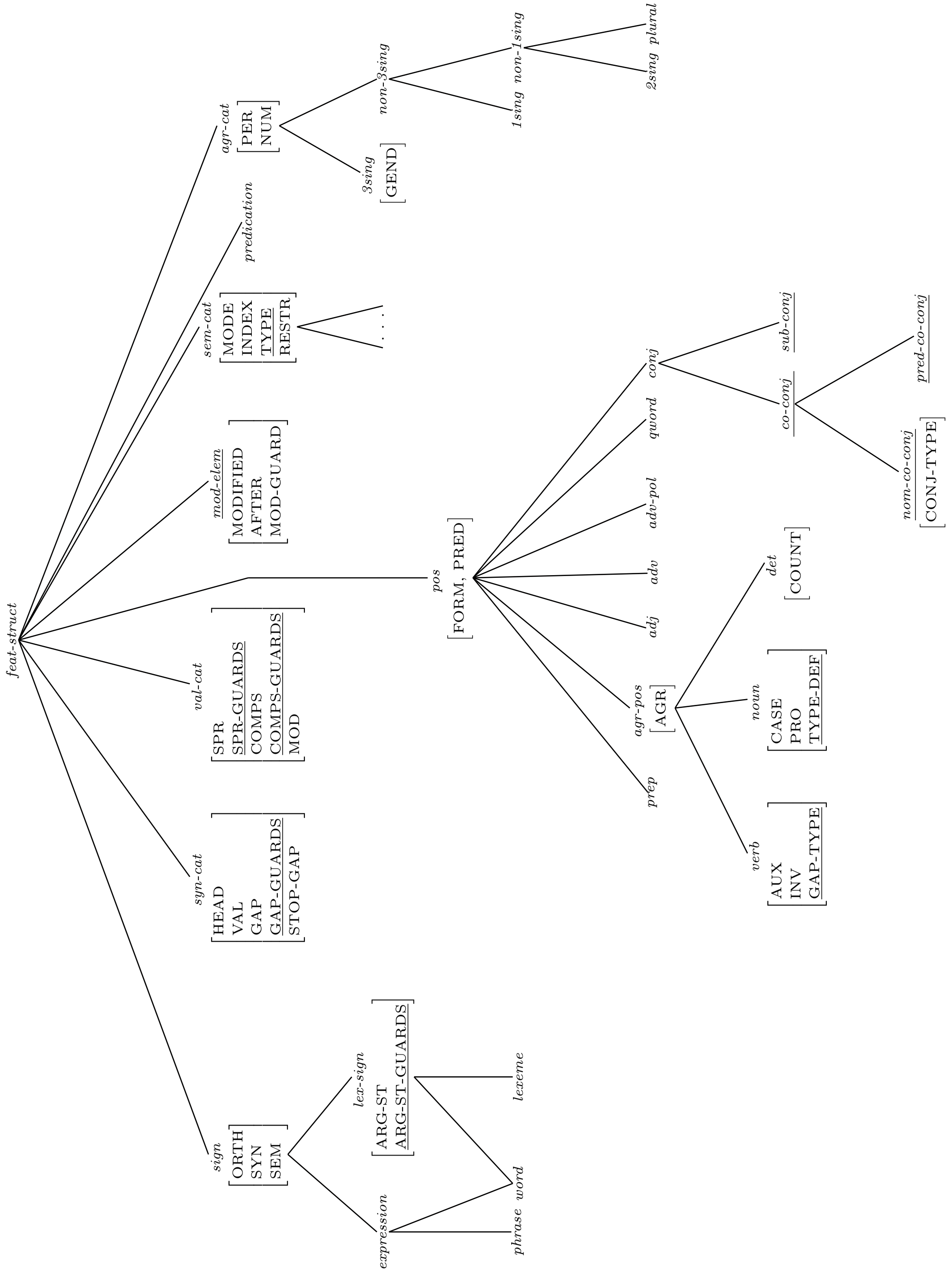


Figure 4.1: Our grammar's feature structure type hierarchy

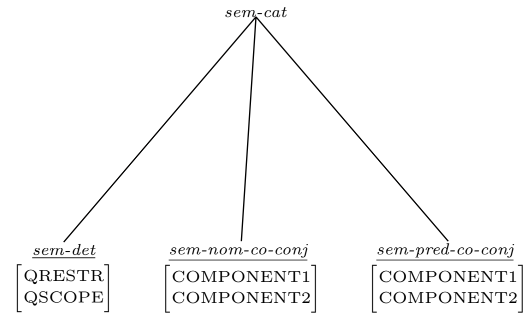


Figure 4.2: Our grammar's semantic feature structure type hierarchy

for noun phrases[6]).

- SPR-GUARDS, COMPS-GUARDS of *val-cat*, and GAP-GUARDS of *syn-cat*, and MOD-GUARD of *mod-elem* which provide a way to encode type restrictions (see definition 3.76) of senses of expressions.
- TYPE-DEF boolean feature of *noun* that specifies if a noun phrase is introducing a new type (a domain entity type) for the domain model.

Non-major additions to the grammar of [67] are:

- ◦ *qword*, which is a grammar entity type we associate to question words such as *who, what* that pose a question about the missing subject or a complement of the sentence that follows them.
- ◦ *sub-conj*, and *co-conj* subcategories of *conj* for subordinate conjunctions, and coordinate conjunctions respectively.
- ◦ And breaking down the coordinate conjunctions into two categories: nominal coordinate conjunctions (represented by *nom-co-conj* type) and predicative coordinate conjunctions (represented by *pred-co-conj* type). Later in chapter 6 we see that nominal coordinate conjunctions can be used for specifying multiple inheritance in natural language.

The type hierarchy of lexemes, and constructions that are present in [67] are not considered in this thesis. Also for the purpose of conciseness the analysis of passive constructs,

[6] For other kinds of phrases this feature is not used in our analysis, although this can be extended in future works.

dummies and idioms, infinitival complements are not covered in this thesis. Thus the relevant features needed for these analyses are omitted from the feature structures that we study here.

Here are some notation conventions for showing feature structure descriptions, useful for the next chapters.

Notation 4.6 We use $+$, and $-$ for the boolean values `True`, and `False` respectively.

Notation 4.7 If all features appropriate for a feature structure type τ are unspecified in a given feature structure description $\mathcal{FD}$ of that type then instead of using an AVM containing only τ to represent the feature structure description $\mathcal{FD}$ we can use τ without the brackets. For example, we can use *word* instead of $\begin{bmatrix}\textit{word}\end{bmatrix}$.

Notation 4.8 If the type of a feature structure description is clear from the context, or if the feature structure description is used as a feature value whose appropriate type does not have any subtypes in the type hierarchy (i.e., it is a leaf type), then it is not necessary to mention the type as the first row of the AVM representing that feature structure description.

Notation 4.9 If we want to refer to a feature that is deeply embedded in a feature structure description, and there is no confusion in the naming of the features, it is not necessary to draw AVMs for each outer feature structure description that contains the specific feature we are looking for. For example the feature structure description:

$$\begin{bmatrix}\text{SYN} & \begin{bmatrix}\text{HEAD} & \textit{verb}\end{bmatrix}\end{bmatrix}$$

can be abbreviated to the following feature structure description.

$$\begin{bmatrix}\text{HEAD} & \textit{verb}\end{bmatrix}$$

4.7 Conclusion

In this chapter we provided a brief introduction to the HPSG formalism, and iterated its most important characteristics. We presented the grammar entity type hierarchy and the feature structure types of the grammar that we develop in this thesis. In the next chapter we present the syntax for this grammar.

Chapter 5

Syntactic Features and their Rules and Principles

We begin our journey into the features, rules and principles by exploring syntactic features with the rules and principles that govern them. Syntactic features are embedded in the SYN feature value of the *sign* type. For now we concentrate on the *expression* subtype of the *sign*. An *expression* is either a *word* or a *phrase*. As shown in table C.1 the value of SYN is of type *syn-cat*. A very important feature of this type is HEAD. We start by studying this feature in some detail.

5.1 The Head Feature

The value of the head feature feature tells us about the part of speech that the expression, which is embedding it corresponds to. To represent the part of speech, we use a typed feature structure (description) of type *part-of-speech* or *pos* for short. *verb*, *noun*, *determiner* (*det*), conjunction (conj), *adjective* (*adj*), *adverb* (*adv*), and *preposition* (*prep*) are all subtypes of *pos*. Here we refer to nouns, noun phrases, and pronouns by the *nominal* subtype of *part-of-speech*. Later we see there are features that specify which one is exactly the case for a given *sign*.

Example 5.10 If the value of the HEAD feature embedded in the SYN feature of an expression is of type noun, then that expression is a noun or a noun phrase, like the following feature structure description that describes the noun *Mary*.

(12) $$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{mary}] \\ \text{SYN} & \begin{bmatrix}\text{HEAD} & \textit{noun}\end{bmatrix} \end{bmatrix}$$

The following is the feature structure description of the verb *dances*.

(13) $$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{dances}] \\ \text{SYN} & \begin{bmatrix}\text{HEAD} & \textit{verb}\end{bmatrix} \end{bmatrix}$$

Besides a simple part-of-speech information, the HEAD feature provides other useful information using embedded features. The additional information could be relevant to only certain parts of speech. For example, recall that we use *noun* for both nouns (together with noun phrases) and pronouns. So we need a means of distinguishing between the two. This is achieved by the boolean PRO feature that is introduced by *noun.* If the value is + then the sign refers to a pronoun, and otherwise it refers to a noun or a noun phrase. This feature is only appropriate for *nouns.* As another example note that *agreement* in English is only relevant to verbs, nouns, and determiners. For this reason *pos* has a special subtype *agr-pos* that carries the agreement information in the AGR feature. And *verb*, *noun*, and *det* are all subtypes of *agr-pos*, so that all inherit the AGR feature. Using subtyping in the type hierarchy enables us to provide certain features to only those grammar entities for which those features are meaningful. So for example, the feature structure (description) of a *conjunction* does not have the AGR feature, neither the PRO feature. This not only makes our grammar more concise but also makes it more precise and avoids some mistakes and misuses of some features that are not appropriate in some contexts.

5.1.1 The Agreement Feature

The value of this feature must be of type *agr-cat* according to table C.1. This value must provide the information about the person (PER), and number (NUM) of the grammar entity. *agr-cat* is subtyped appropriately according to the agreement in English language. The type hierarchy rooted by *agr-cat* is shown in figure 5.1.

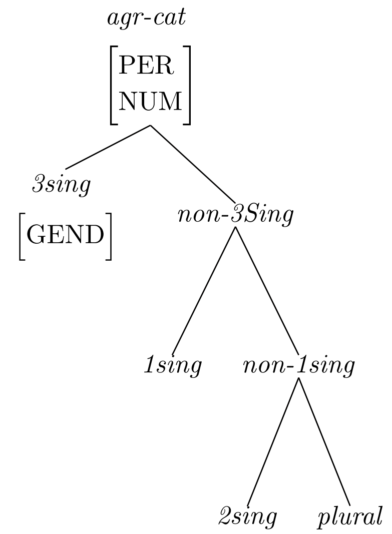


Figure 5.1: Agreement type hierarchy

For third person singular, the gender is also important in English. That is why GENDER (or GEND for short) is introduced by *3sing*. The possible values of this feature are feminine (fem), masculine (masc), and neutral (neut). However, GENDER is not relevant to any other combination of person and number. Again, by using subtyping we are able to avoid providing irrelevant information in our grammar entities.

Example 5.11 The following shows the feature structure description of *Mary*, with agreement information.

$$\text{(14)}\quad \left[\begin{array}{ll} \textit{word} & \\ \text{ORTH} & [\text{mary}] \\ \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \left[\begin{array}{ll} \textit{noun} & \\ \text{AGR} & \left[\begin{array}{ll} \textit{3sing} & \\ \text{GEND} & \textit{fem} \end{array}\right] \\ \text{PRO} & - \end{array}\right] \end{array}\right] \end{array}\right]$$

For *verbs*, the agreement information is in fact the expected agreement information of its subject. Later we see that the subject of a verb is treated as its *specifier*. We will define specifiers in section 5.2.1. There we pose a grammar principle that unifies the agreement feature of the head expression with its specifier. For now, it suffices to say that the agreement feature of a verb must be unified with the agreement feature of its subject.

Example 5.12 Here is the feature structure description of *dances*, with agreement information.

$$(15)\quad \begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{dances}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{AGR} & \textit{3sing} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Note that the GENDER feature for the verb *dance* is unspecified, because it does not matter for this verb whether the subject is a female, a male or a (grammatically) neutral entity.

5.1.2 The FORM Feature

The interpretation of the FORM feature depends on which part of speech it is used in. Its main usage is for verbs and prepositions. For all other parts of speech, the only value that FORM can take, is chosen as an atomic value, which is used only for that part of speech. This value is nform for *noun*, aform for *adj*, avform for *adv*, cform for *conj*, and so on.

For prepositions, the value of FORM is simply an atomic value that is chosen to be the orthography of that proposition. Using this, we will be able to refer to prepositional phrases that begin with a certain preposition, and not others. Such a situation is useful, for example, if we want to restrict the form of a prepositional phrase that can follow a prepositional transitive verb like *borrow* to only begin with *from*. So sentence (16a) is acceptable whereas (16b) is not.

(16) a. Can I borrow some money from you?

b. * I borrowed a book to John

Since the FORM feature of a prepositional phrase is used quite often in HPSG grammars, it is more convenient to abbreviate PP[FORM *xyz*] to PP[*xyz*].

For verbs, the value of the FORM feature represents the verb form. The permissible values of this feature with their corresponding verb form and an example are given in

Table 5.1: Permissible feature values of FORM for verbs

FORM	verb form	example
base	bare uninflected form (bare infinitive)	*You may **go**.*
fin	finite : simple present or past tenses	*He **listens** to music.*
prp	present participle	*Chris is **climbing** the mountain.*
psp	past participle	*Have you **seen** Chris?*
pass	passive	*This email was **written** by him.*

table 5.1. Note that we do not study passive constructs in this thesis, the interested reader can refer to [67].

Later we see how the FORM feature can be used in the coordination rule to force the coordinating constituents to be of the same part of speech.

5.1.3 The PRED Feature

The PRED feature is accessible for all parts of speech. This feature specifies whether the corresponding part of speech can follow the auxiliary verb *be* to form a predicate that can in turn combine with a subject to build a sentence. In the following we show some examples of how some subtypes of *pos* can be in predicative use, which is specified by PRED + in the HEAD feature.

In (17) we have shown an *adj* in predicative use. *unsatisfied with John* is an adjective phrase that can follow *be* to convey the meaning that the subject (*Mary*) is not happy with the entity that is referred to by the complement of the adjective phrase (*John*). The feature structure description of unsatisfied with John in shown in (18). Note that the type of the feature structure description is *phrase* instead of *word*. Also there is no agreement information, that is the feature AGR is absent, because *adj* is not a subtype of *agr-pos*.

(17) Mary is unsatisfied with John.

(18)

$$\left[\begin{array}{ll} \textit{phrase} & \\ \text{ORTH} & [\text{unsatisfied, with, john}] \\ \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \left[\begin{array}{ll} \textit{adj} & \\ \text{PRED} & + \end{array}\right] \end{array}\right] \end{array}\right]$$

Not all adjectives are PRED +. An example is *lone* that cannot follow *be.* The only acceptable use of *lone* is before a *noun*, as shown in (19b).

(19) a. * Mary is lone.

b. Mary is the lone survivor of the accident.

A prepositional phrase can also be in predicative mode, as shown in (20). The feature structure description of *on the roof* is shown in (21). Note that as we discussed in the previous subsection, the value of the FORM feature is the same as the orthography of the head preposition, which is *on.*

(20) The cat is on the roof.

(21)
$$\begin{bmatrix} \textit{phrase} & \\ \text{ORTH} & [\text{on, the, roof}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{prep} & \\ \text{FORM} & \text{on} \\ \text{PRED} & + \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Not all prepositions are in predicative mode. In fact the same preposition can have both predicative and non-predicative modes. The non-predicative mode of a preposition is *argument marking*. The argument marking mode of a preposition has the implication that the preposition does not add any meaning to the sentence, rather it provides a syntactic indicator about where the objects (or the complements) of a constituent (like a verb) are located. Two examples of argument marking prepositions are given in (22a, 23a).

In (22a), *Joe* is the object of the prepositional intransitive verb *rely. Joe* is marked by the argument-marking preposition *on.* Note that in (20) *on* was used as a predicative preposition. The feature structure description of *on Joe* is shown in (22b).

(22) a. We relied on Joe.

b. $$\begin{bmatrix} \textit{phrase} & \\ \text{ORTH} & [\text{on, joe}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{prep} & \\ \text{FORM} & \text{on} \\ \text{PRED} & - \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(23) a. You talked to Jack.

b. $$\begin{bmatrix} \textit{phrase} & \\ \text{ORTH} & [\text{to, jack}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{prep} & \\ \text{FORM} & \text{to} \\ \text{PRED} & - \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

The preposition *to*, however, is hard to use as a predicative one. The only reasonable occurrence of *to* after the verb *be* is when combined with a base form of a verb to form an infinitive, as in (24). However, for the purposes of conciseness we do not study the use of *to* in such cases.

(24) This letter is to inform you of the recent changes in the regulations.

5.1.4 Verb Specific Features

Since *verb* is a subtype of *agr-pos*, and *pos*, it inherits all the features declared by *agr-pos* and *pos*, which are FORM, PRED, AGR. Besides these, *verb* introduces some new features including the boolean feature AUX, which is + for auxiliary verbs, and − for all other verbs, and the boolean feature INV, which is + for inverted auxiliary verbs, for questions, and − for normal verbs. An inverted verb requires its subject to follow it rather than precede it. We will have more to say about INV in section 5.5, where we discuss auxiliary verbs and questions.

5.1.5 Nominal Specific Features

We have already discussed the boolean PRO feature, which distinguishes between a pronoun and a noun. The new feature is CASE which ranges over *nom* (for nominative), and *acc* (for accusative). If a nominal is CASE nom it means it is the subject of a sentence. If the value of the CASE feature is acc, it means the nominal is not the subject of any sentence, and that it is an object or a complement of a verb or some other constituent. This feature is needed in English because some pronoun forms are acceptable in certain positions only. For example in (25a), the personal pronoun *He* is used as a subject, with a nominative CASE. But *he* cannot be used as in (25b), where it is an object, because in English *he* cannot accept the object role. The feature structure description of *he* is shown in (26).

(25) a. He loves Mary.

b. * Mary loves he.

(26) $$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{he}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{noun} & \\ \text{AGR} & \begin{bmatrix} \textit{3sing} & \\ \text{GEND} & \textit{masc} \end{bmatrix} \\ \text{CASE} & \textit{nom} \\ \text{PRO} & + \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

5.1.6 Determiner Specific Feature : COUNT

The only syntactic feature that *det* introduces is COUNT, which indicates whether that determiner is appropriate to be used for a countable noun or not. For example *many* is only appropriate for countable nouns, as used with *books* in (27a). It cannot be used with a mass noun like *water* as in (27b).

(27) a. I have not read many books.

b. * I do not drink many water.

The feature structure description of *many* is presented in (28). First, note that the agreement feature value is *plural*, which means *many* must be used only with plural nouns. Second, the value of COUNT is +, which indicates this determiner is only usable with countable nouns. It is the responsibility of a *noun* to specify which kind of *determiner* it requires. This is done through the SPECIFIER (SPR for short) feature, which we will study in section 5.2.

(28) $$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{many}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{det} & \\ \text{AGR} & \textit{plural} \\ \text{COUNT} & + \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

5.1.7 The Head Feature Principle

In this section we are going to introduce the first principle of our grammar, namely, Head Feature Principle or HFP for short. As mentioned in definition 4.3, a headed phrase describes a kind of what its head describes. This implies that the mother phrase and its head daughter share their part of speech (*pos*). In fact, the head features are chosen carefully so that the mother phrase HEAD feature is the same as its head daughter HEAD feature. In a phrase structure tree of headed phrases, the HEAD feature value is percolated up from the head daughters to their mothers. This is known as the Head Feature Principle:

(29) Head Feature Principle (HFP):
In any headed phrase, the value of the HEAD feature of the mother phrase and the value of the HEAD feature of its head daughter are the same.

As a result of this principle, for example, the following feature structure descriptions of *orange*, and *the juicy orange* share the same HEAD feature.

(30) a. $$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{orange}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{noun} & \\ \text{AGR} & \begin{bmatrix} \textit{3sing} & \\ \text{GEND} & \text{neut} \end{bmatrix} \\ \text{PRO} & - \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

b. $$\left[\begin{array}{ll} \textit{phrase} & \\ \text{ORTH} & [\text{the, juicy, orange}] \\ \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \left[\begin{array}{ll} \textit{noun} & \\ \text{AGR} & \left[\begin{array}{ll} \textit{3sing} & \\ \text{GEND} & \text{neut} \end{array}\right] \\ \text{PRO} & - \end{array}\right] \end{array}\right] \end{array}\right]$$

Although the two feature structure description share a lot of similarity, more specified versions reveal differences in syntactic features other than HEAD. These features specify if the corresponding expression can combine with other expressions, and if so how. This is the topic of the next section.

5.2 Valence Features : How Expressions can Combine Together

The second feature of *syn-cat* that we are going to study is VAL (for valence[1]). The value of this feature must be of type *val-cat*, which has three standard fields: SPR (for specifier), COMPS (for complements), and MOD (for modifier).

5.2.1 Specifiers

In a headed phrase, a specifier is an expression that precedes the head. In other terms, the specifier is a daughter of a headed phrase that comes before the head daughter. In the formulation of [67] there cannot be more than one specifier in a headed phrase, as it is not needed. However a headed phrase might have no specifier. The SPR feature of an expression is the list of specifiers of a phrase headed by that expression. The number of elements of this list is either 0, or 1. If there is no element in SPR list, no specifier is necessary. If the list contains one expression, then that expression will serve as the sole specifier.

Specifiers generally fall into two main categories: determiners of common nouns, and subjects of verbs. There is a third category that is needed for the analysis of possessive

[1] This name is borrowed from chemistry where valence is the capacity of an atom to combine with other atoms in a molecule.

's as done in [61], which we will describe in this section. In short, *'s* has a noun specifier that has the owner role in the possessive construct. We will discuss each category after we present our first HPSG grammar rule, the Head Specifier Rule that enables us to combine a specifier and a head.

With the notion of specifiers being defined, we are now able to present the Head Specifier Rule. We will later modify this rule in chapter 7) to allow for the type restrictions to be processed before a phrase can be formed by using it.

(31) Head Specifier Rule (HSR), Version I:[2]

$$\begin{bmatrix} \textit{phrase} & \\ \text{VAL} & \begin{bmatrix}\text{SPR} & \langle\ \rangle\end{bmatrix}\end{bmatrix} \rightarrow \boxed{1} \quad \mathbf{H}\begin{bmatrix}\text{HEAD} & \begin{bmatrix}\text{PRED} & -\end{bmatrix}\\ \text{VAL} & \begin{bmatrix}\text{SPR} & \langle \boxed{1} \rangle\end{bmatrix}\end{bmatrix}$$

To avoid licensing of some invalid verb phrases, like *he eating*, we posit a constraint that if the head is verbal, then its form must be fin (for finite).

Notice the use of tags in the rule. The internal structure of the expression tagged by 1 is totally unspecified. We only know that this expression, when considered in the application of this rule, both comes before the head, and appears as the only element of the SPR list of the head. If the feature structure description of the element in the SPR list of a candidate head expression and the candidate specifier expression are not exactly the same, then these two feature structure descriptions must be unified before HSR can be applied.

Determiner and Noun Co-occurrence Restrictions

As mentioned in section 5.1.6, it is the responsibility of a *noun* to specify what kind of *determiner* it requires, if any. The SPR feature of *noun* can accomplish this, by having a partially specified feature structure description of a *determiner* that is appropriate for that *noun*. For example, a feature structure description of *books* is shown in (32).

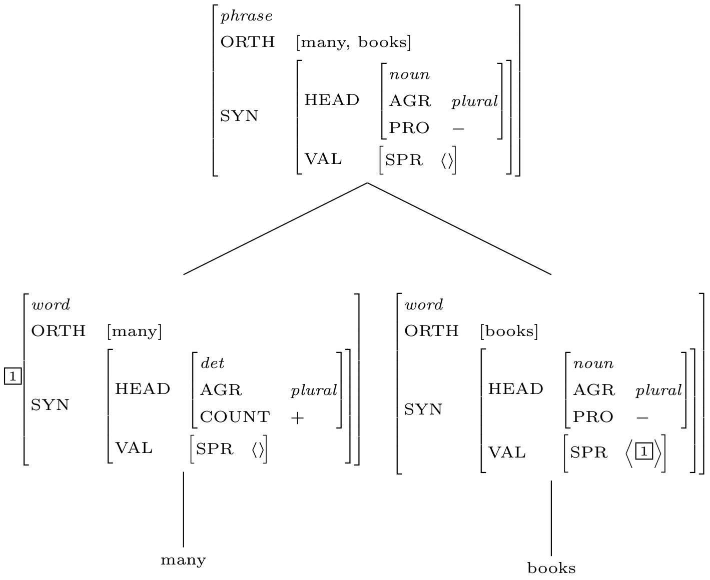
[2] This rule is a bit different from HSR that is presented in [67] in that we deliberately do not require the COMPS list to be empty, and that we require the head to be PRED −. The reason for this additional constraint will be studied in section 5.5. COMPS feature is introduced in section 5.2.2.

(32)

$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{books}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{noun} & \\ \text{AGR} & \textit{plural} \\ \text{PRO} & - \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \left\langle \begin{bmatrix} \textit{word} & \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{det} & \\ \text{AGR} & \textit{plural} \\ \text{COUNT} & + \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

The only element of the SPR list matches with the feature structure description of *many* that we provided in (28). The combination of *many* and *books* is achieved by the Head Specifier Rule. By using this rule, the feature structure descriptions (28), and (32) can be combined to form the phrase *many books.* This is shown by the phrase structure tree in (33).

(33)



Count and Mass Nouns

One difference between the feature structure description of count and mass nouns in the lexicon is the description of the specifier in their SPR list. The feature structure description of a mass noun can have a *det* specifier, whose COUNT feature must be −. On the other hand, the feature structure description of a count noun has a *det* specifier with the value of COUNT being only +.

For example, the feature structure description of a mass noun, *water*, is shown in (34). We have placed the specifier element in parentheses to denote that the determiner is optional for *water*, as in (35a). In (35b), *water* is used with a determiner.

(34)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{water}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{noun} & \\ \text{AGR} & \textit{3sing} \\ \text{PRO} & - \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \left\langle \left(\begin{bmatrix} \textit{word} & \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{det} & \\ \text{AGR} & \textit{3sing} \\ \text{COUNT} & - \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix} \right) \right\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(35) a. Water boils at 100 degrees Celsius.

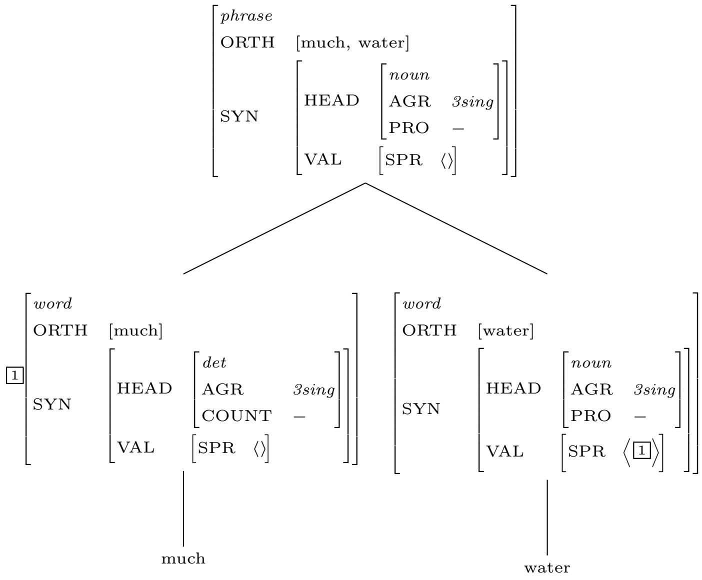
b. I do not drink much water before going to the movies.

Note that the determiner *many* cannot combine with the mass noun *water*, because the feature structure description of the specifier element in *water* is inconsistent with the feature structure description of *many.* The reason for this inconsistency is the mismatching COUNT values.

However the determiner *much* with the feature structure description shown in (36) can be combined with *water* as shown in the phrase structure tree of (37).

(36)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{much}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{det} & \\ \text{AGR} & \textit{3sing} \\ \text{COUNT} & - \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(37)



So from a syntactic point of view the major differences between a count noun and a mass noun are:

- If the SPR list of a mass noun is non-empty it should contain a *det* with COUNT $-$, whereas the SPR list of a count noun should contain a *det* with COUNT $+$.
- No *non-3sing* form of a mass noun should be present in the lexicon. Later we see this means there should be no lexical rule that changes the agreement feature of a mass noun from *3sing* to any leaf subtype of *non-3sing*.

Specifier Head Agreement Constraint (SHAC)

Note that in all phrase structure trees above that are licensed by HSR, the AGR feature of the determiner and the head noun match. The same restriction applies for subjects and verbs. We can generalize this co-occurrence restriction by a constraint that is called Specifier Head Agreement Constraint (SHAC) that is associated with any feature structure description of *nouns* and *verbs*:

(38) Specifier Head Agreement Constraint (SHAC)

Any feature structure description of type *verb* or *noun* with a non-empty SPR list, (implicitly) bears the following constraint:

$$\begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \text{AGR } \boxed{1} \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} \left\langle \begin{bmatrix} \text{AGR } \boxed{1} \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Subject and Verb Co-occurrence Restrictions

The SPR feature of a *verb* must contain a single expression, a *noun* (possibly a noun phrase), that serves as the subject of that verb. For example, (39) shows the feature structure description of the verb *breathes.* The SPR list contains an NP[CASE nom]. NP is an abbreviation we use for the most general feature structure description of a noun phrase. The internal structure of NP is shown in (40).[3] By NP[CASE nom] we are referring to a NP feature structure description whose CASE feature (embedded in HEAD which in turn is inside SYN) is nom (for nominative).

(39)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & \text{[breathes]} \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{FORM} & \text{fin} \\ \text{PRED} & - \\ \text{AGR} & \textit{3sing} \\ \text{AUX} & - \\ \text{INV} & - \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \left\langle \text{NP}\begin{bmatrix} \text{CASE} & \text{nom} \end{bmatrix} \right\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

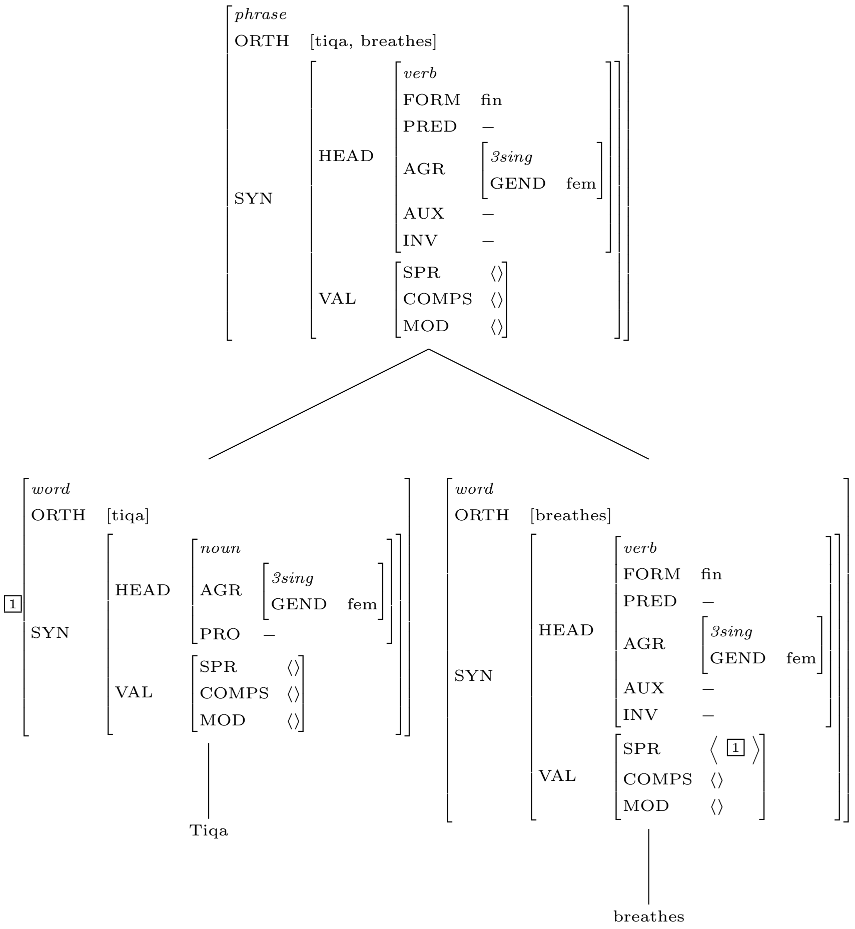
(40)
$$\text{NP} = \begin{bmatrix} \textit{expression} & \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \textit{noun} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

In (41) we have shown a feature structure description of the word *Tiqa.* This feature structure description can be unified with the NP in the SPR list of *breathes.* Note that the AGR features also match. So HSR can license the phrase *Tiqa breathes*, as shown by the phrase structure tree in (42).

[3] Features COMPS and MOD are studied later in this chapter.

(41)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{tiqa}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{noun} & \\ \text{AGR} & \begin{bmatrix}\textit{3sing} & \\ \text{GEND} & \text{fem}\end{bmatrix} \\ \text{PRO} & - \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix}\end{bmatrix}\end{bmatrix}$$

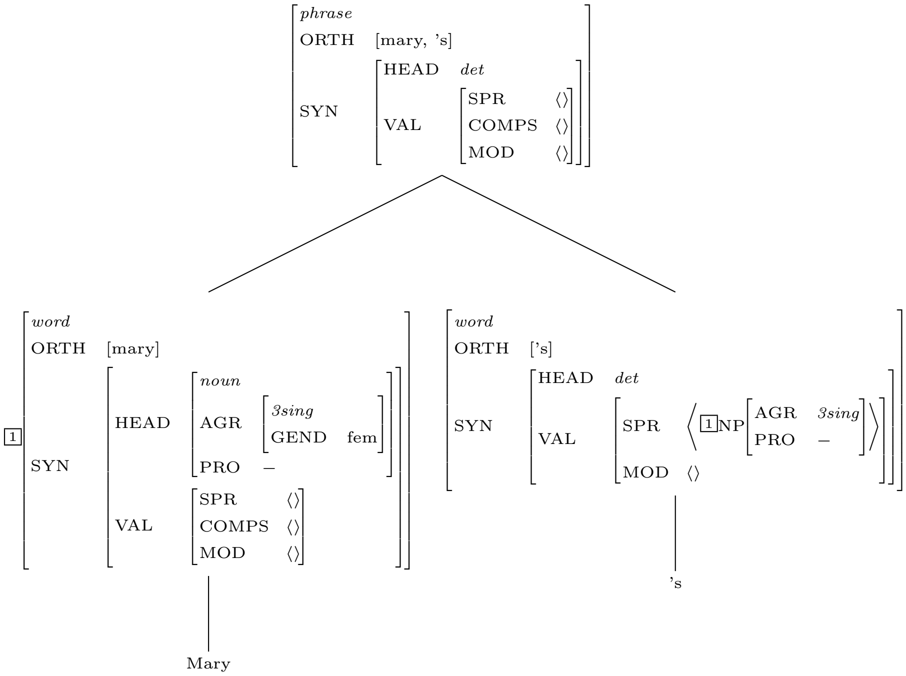
(42)



Syntactic Analysis of *'s* Possessives as Heads

The analysis we provide here is due to Sag, et al. in [61]. Here we focus on the syntax and in the next chapter we complete the analysis by providing the semantic counterpart. In their analysis, a noun phrase like *Mary's book* can be thought of as a combination of a *determiner phrase* and a *noun*. The determiner phrase is *Mary's* that can act in place of a determiner in any noun phrase. However the treatment of *Mary's* as a determiner phrase requires to have a special rule for combining a *noun phrase* with the possessive marker *'s*, or it requires *Mary's* to be a headed phrase. In their approach, they have chosen the second choice, and they treat *'s* as a head with a non-empty SPR list. The SPR list contains the *3sing* NP that plays the role of the owner. This noun phrase cannot be a pronoun, that is why the PRO feature has the value −. In case of the phrase *Mary's*, *Mary* is the owner, and the specifier of the head *'s*. In this approach no extra grammar rule is required, and *Mary's* will be licensed by HSR, as shown in (43).

(43)



5.2.2 Complements

A complement in a headed phrase is a constituent that follows the head. For example *a book* is the complement of the verb phrase *reads a book* in (44).

(44) He reads a book.

A headed phrase can have multiple complements, as in (45), where *him* is the first complement, and *a joke* is the second complement.

(45) I told him a joke.

The feature structure descriptions of the complements in a headed phrase in the order they should appear, are provided in the COMPS feature of *val-cat*. As an example, (46) shows a feature structure description of the verb *reads*. By NP[CASE acc] we mean that the value of the CASE feature embedded in the NP must be acc (for accusative). In English there is no difference between the nominative form and the accusative form of common nouns or proper nouns, however for pronouns the forms differ.

(46)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{reads}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{FORM} & \text{fin} \\ \text{PRED} & - \\ \text{AGR} & \textit{3sing} \\ \text{AUX} & - \\ \text{INV} & - \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \left\langle \text{NP}\begin{bmatrix}\text{CASE} & \text{nom}\end{bmatrix}\right\rangle \\ \text{COMPS} & \left\langle \text{NP}\begin{bmatrix}\text{CASE} & \text{acc}\end{bmatrix}\right\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

In (47) a feature structure description of the verb *told* is shown. Note that the COMPS list contains 2 NPs, the first one is the direct object and the second one is the indirect object.

(47)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{told}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{FORM} & \text{fin} \\ \text{PRED} & - \\ \text{AUX} & - \\ \text{INV} & - \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \left\langle \text{NP}\begin{bmatrix}\text{CASE} & \text{nom}\end{bmatrix}\right\rangle \\ \text{COMPS} & \left\langle \text{NP}\begin{bmatrix}\text{CASE} & \text{acc}\end{bmatrix}, \text{NP}\begin{bmatrix}\text{CASE} & \text{acc}\end{bmatrix}\right\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Now we need a rule that combines a head with its complements. This rule is the Head Complement Rule (or HCR for short). HCR is an example of an HPSG rule that has a variable number of daughters. The number of non-head daughters is identical to the number of elements in the COMPS feature of the head daughter. We will modify this rule later in (6) to incorporate type restrictions.

(48) Head Complement Rule (HCR), Version I:

$$\begin{bmatrix} \textit{phrase} & \\ \text{VAL} & \begin{bmatrix}\text{COMPS} & \left\langle\ \right\rangle\end{bmatrix}\end{bmatrix} \rightarrow \mathbf{H}\begin{bmatrix}\text{VAL} & \begin{bmatrix}\text{COMPS} & \left\langle \boxed{1}, ..., \boxed{n} \right\rangle\end{bmatrix}\end{bmatrix}\ \boxed{1}\ ...\ \boxed{n}$$

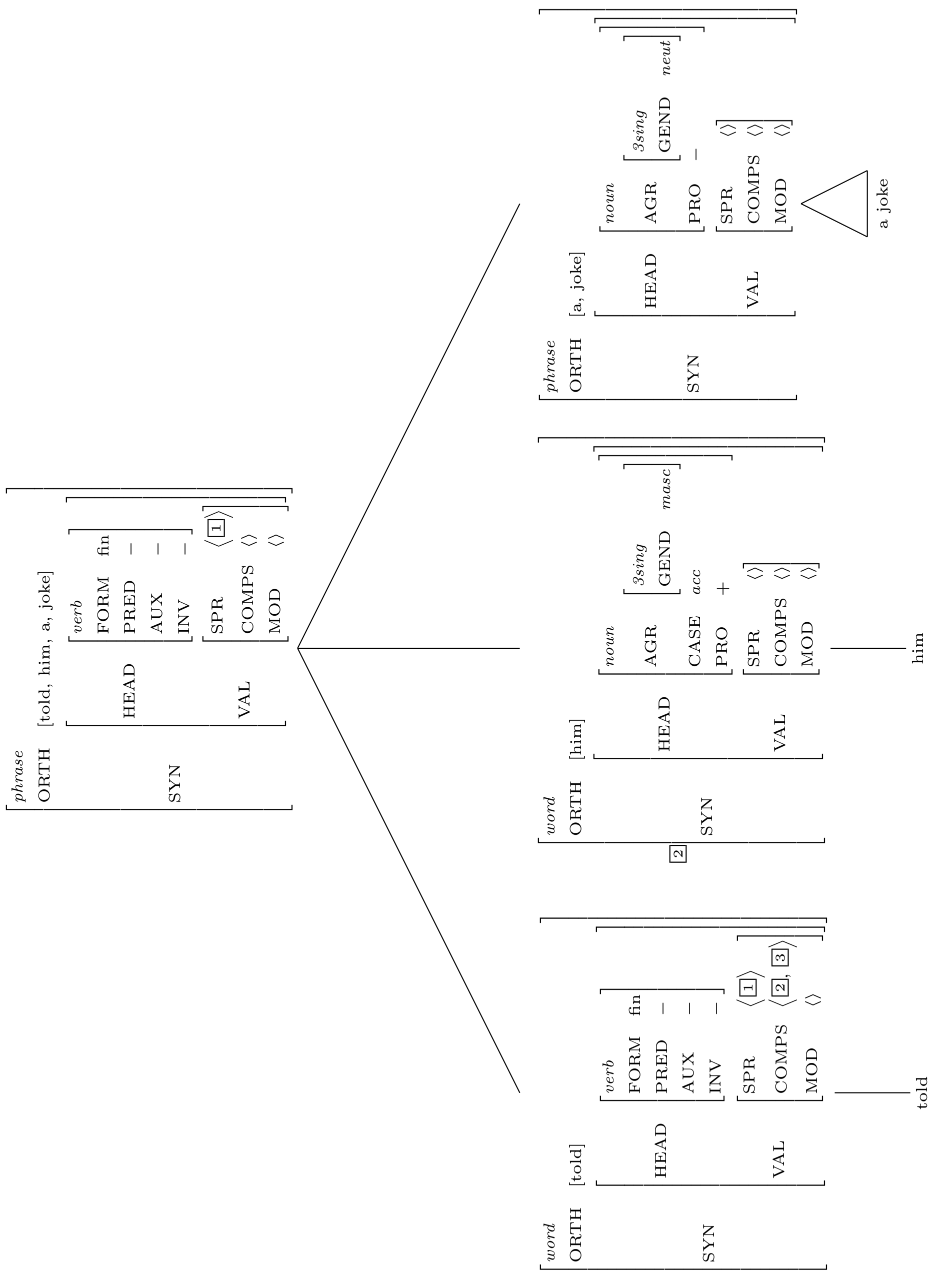
Suppose feature structure descriptions of *he*, *him*, and *a joke* are shown in (49.a), (49.b), and (50) respectively. Then by using HCR the verb phrase *told him a joke* can be licensed, as shown in (51).

(49) a.
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{he}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{noun} & \\ \text{AGR} & \begin{bmatrix} \textit{3sing} & \\ \text{GEND} & \textit{masc} \end{bmatrix} \\ \text{CASE} & \textit{nom} \\ \text{PRO} & + \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

b.
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{him}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{noun} & \\ \text{AGR} & \begin{bmatrix} \textit{3sing} & \\ \text{GEND} & \textit{masc} \end{bmatrix} \\ \text{CASE} & \textit{acc} \\ \text{PRO} & + \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(50)
$$\begin{bmatrix} \textit{phrase} & \\ \text{ORTH} & [\text{a, joke}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{noun} & \\ \text{AGR} & \begin{bmatrix} \textit{3sing} & \\ \text{GEND} & \textit{neut} \end{bmatrix} \\ \text{PRO} & - \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(51)



Yet, note that the invalid verb phrase shown in (52) cannot be licensed by HCR, because the CASE feature of *he* is nom, which is inconsistent with what is required by the complements of *told*.

(52) * told he a joke.

5.2.3 The Valence Principle

In (51) the SPR element of the mother verb phrase is identical to the SPR element of the head verb. This is due to another HPSG principle known as the Valence Principle.

(53) The Valence Principle:
In any headed phrase, the valence features of the mother are identical to the valence features of the head daughter, unless the rule that licenses the phrase specifies otherwise.

5.2.4 The Sentence

In HPSG a sentence is just a verb phrase whose valence features SPR, and COMPS all empty lists. It is said that a sentence is a *saturated* verb phrase, meaning that it has no more capacity of combining with the required constituents. For example the feature structure description of *Tiqa breathes* in (42) shows that both the SPR and COMPS features of the mother verb phrase are empty. Thus it is recognized as a sentence. The sentence is sometimes called as the *initial symbol* of the grammar.

We abbreviate sentence by S, whose internal structure is shown in (54). The abbreviation of a COMPS saturated verb phrase comes handy sometimes, and we use VP for such a verb phrase. The internal structure of VP is shown in (55).

(54)

$$\mathrm{S} = \begin{bmatrix} \textit{expression} & \\ \mathrm{SYN} & \begin{bmatrix} \mathrm{HEAD} & \textit{verb} \\ \mathrm{VAL} & \begin{bmatrix} \mathrm{SPR} & \langle\rangle \\ \mathrm{COMPS} & \langle\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(55)

$$\mathrm{VP} = \begin{bmatrix} \textit{expression} & \\ \mathrm{SYN} & \begin{bmatrix} \mathrm{HEAD} & \textit{verb} \\ \mathrm{VAL} & \begin{bmatrix} \mathrm{SPR} & \left\langle \mathrm{X} \right\rangle \\ \mathrm{COMPS} & \langle\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

A sentence can sometimes be missing some constituents known as gaps (the topic of section 5.6). To show that the sentence is complete we need to specify the GAP feature as the empty list too. This is denoted by:

S[GAP $\langle\rangle$]

5.2.5 Modification

A modifier is an expression that modifies another expression. Modifiers in this thesis fall into the following categories:

- *adjectives* that modify a nominal (defined below)
- *adverbs* that modify a verb phrase
- prepositional modifiers, like *on the street* that modify a noun phrase or a verb phrase
- *subordinate conjunctions*, like *if*, *that*, or *which* that modify a verb phrase by providing a condition, or a noun phrase by providing additional information

All of these expressions share one characteristic, that is, they have a non-empty list as the value of their MOD feature (which is embedded in valence features). Any expression that is not a modifier must have an empty MOD list.

For a modifier, the MOD list has only one element, which is of type *mod-elem*. This type has three features, the first one, MODIFIED, is the feature structure description of the expression that it modifies. The second is a boolean feature AFTER, which specifies whether the modifier comes after or before the expression it modifies. The last feature is used to encode type restrictions that we study in the next chapter.

A modifier that comes after the modified expression is called a *post-head modifier*, like *unsatisfied with her student* in (56).

(56) The professor unsatisfied with her student left the office.

A modifier that comes before the expression it modifies is called a *pre-head modifier*, like *red* in (57).

(57) John gave the red rose to Mary.

The Head Modifier Rule (HMR)

The Head Modifier Rule is the HPSG grammar rule which combines a head with a modifier that modifies it. We have two Head Modifier Rules, one for pre-head modifiers, and the other for post-head modifiers. We will later modify these rules in (17), and (18) of chapter 7 to allow for the type restrictions to be processed before a phrase is licensed by them.

(58) Head Modifier Rule (HMR), Pre-Head, Version I:

$$\begin{bmatrix} \textit{phrase} \end{bmatrix} \rightarrow \begin{bmatrix} \text{VAL} & \begin{bmatrix} \text{COMPS} & \langle\rangle \\ \text{MOD} & \left\langle \begin{bmatrix} \textit{mod-elem} & \\ \text{MODIFIED} & \boxed{1} \\ \text{AFTER} & - \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix} \ \mathbf{H}\boxed{1}\begin{bmatrix} \text{VAL} \begin{bmatrix} \text{COMPS} & \langle\rangle \end{bmatrix} \end{bmatrix}$$

(59) Head Modifier Rule (HMR), Post-Head, Version I:

$$\begin{bmatrix} \textit{phrase} \end{bmatrix} \rightarrow \ \mathbf{H}\boxed{1}\begin{bmatrix} \text{VAL} \begin{bmatrix} \text{COMPS} & \langle\rangle \end{bmatrix} \end{bmatrix} \begin{bmatrix} \text{VAL} & \begin{bmatrix} \text{COMPS} & \langle\rangle \\ \text{MOD} & \left\langle \begin{bmatrix} \textit{mod-elem} & \\ \text{MODIFIED} & \boxed{1} \\ \text{AFTER} & + \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix}$$

Adjectives

The feature structure description of an adjective has a HEAD value of type *adj*, with a non-empty MOD list that contains a nominal which we abbreviate by NOM. A nominal is a noun or a noun phrase that has a non-empty SPR list. The feature structure description of NOM is shown in (60). X represents an unknown expression.

(60)

$$\text{NOM} = \begin{bmatrix} \textit{expression} & \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \textit{noun} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \left\langle \text{X} \right\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

As an example of a pre-head adjective, consider the adjective *red* with the feature structure description shown in (61). The reason why the SPR list of *red* contains an NP is revealed in our semantic analysis of the copular and auxiliary verbs, where we discuss *subject sharing*.

Note that the PRED feature is +, which means *red* can follow the verb *be*. Although the SPR list of *red* is non-empty, HSR cannot directly combine *red* with a noun phrase described by the sole element of SPR list. The reason is that HSR requires the head to be non-predicative, i.e., its PRED feature must be −.

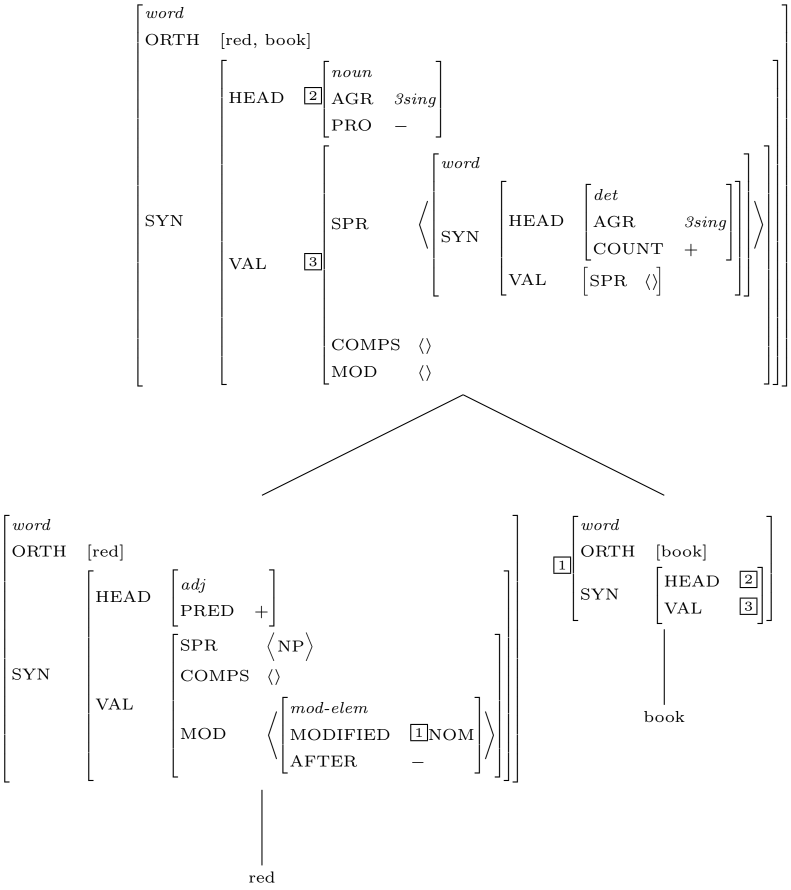
(61)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{red}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{adj} & \\ \text{PRED} & + \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle \text{NP} \rangle \\ \text{COMPS} & \langle \rangle \\ \text{MOD} & \left\langle \begin{bmatrix} \textit{mod-elem} & \\ \text{MODIFIED} & \text{NOM} \\ \text{AFTER} & - \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

The feature structure description of *book* shown in (62) matches with the description of NOM we provided in (60). So by applying the pre-head Head Modifier Rule the nominal phrase *red book* can be licensed as shown by the phrase structure tree in (63).

(62)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{book}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{noun} & \\ \text{AGR} & \textit{3sing} \\ \text{PRO} & - \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \left\langle \begin{bmatrix} \textit{word} & \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{det} & \\ \text{AGR} & \textit{3sing} \\ \text{COUNT} & + \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle \rangle \end{bmatrix} \end{bmatrix} \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Note that in (63), the value of the HEAD feature of the mother phrase, *red book*, and its head daughter, *book*, are identical as a result of the Head Feature Principle. The value of their valence features (VAL) of both expressions are also the same, as a result of the Valence Principle.

(63)



Adverbs

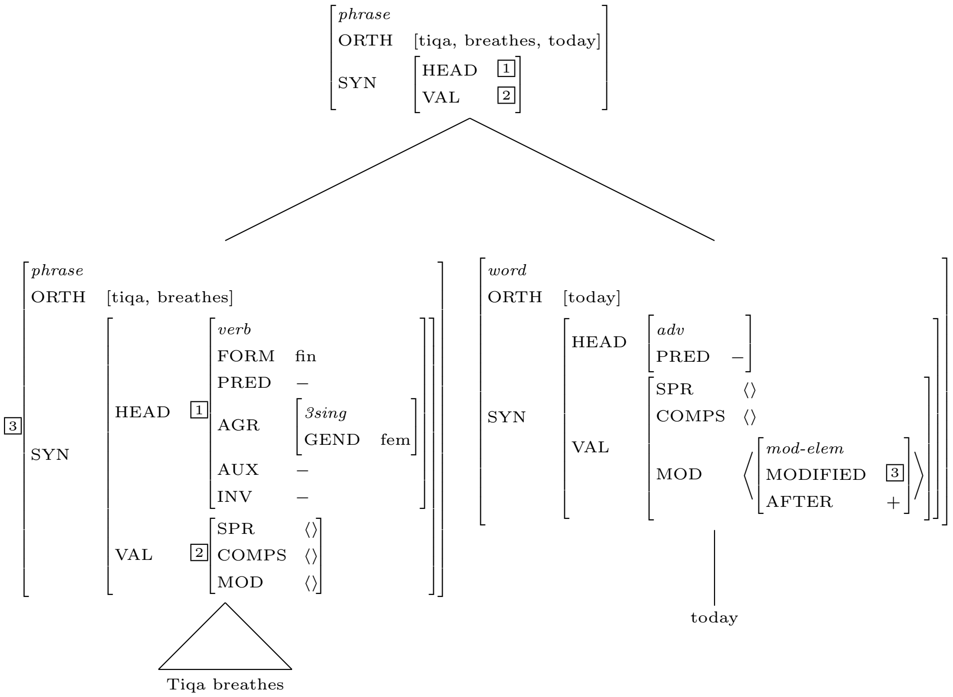
An adverb modifies the verb of a sentence. In the analysis we present here, an adverb can come before or after the sentence.[4] The feature structure description of an adverb has a HEAD of type *adv*, and it has a non-empty MOD list with only one element of type *mod-elem*, whose value of MODIFIED feature is an S. For example the feature structure description of *today* is shown in (64).

[4] In an alternative analysis an adverb modifies a verb phrase, and can come before or after the verb phrase. However in both analyses it depends on the adverb itself if it can come before the verb or after it or both. For example the sentence *Tiqa today breathes* might seem odd for native English speakers.

(64) $$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{today}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix}\textit{adv} & \\ \text{PRED} & -\end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \left\langle \begin{bmatrix}\textit{mod-elem} & \\ \text{MODIFIED} & \text{S}\end{bmatrix}\right\rangle \end{bmatrix}\end{bmatrix}\end{bmatrix}$$

The sentence *Tiqa breathes*, which is shown as the mother phrase in (42), can be combined with *today* using HMR to form the bigger sentence, *Tiqa breathes today.* This is shown by the phrase structure tree in (65).

(65)



Prepositional Modifiers

Prepositions in their predicative mode can modify a nominal (NOM) or a sentence (S). They must come after the expression they modify, and they have a NP complement. For an example, consider the sentence (66). Back in section 5.1.3 we mentioned that some prepositions can serve in a predicative mode. Here we discuss how they can serve as modifiers.

(66) The cat walks on the roof.

The feature structure description of the predicative *on* is shown in (67). Note that the MOD list has a *mod-elem* with the MODIFIED feature being a sentence (S), or a nominal (NOM).[5] Again like the adjective *red*, it has a non-empty SPR list, which we will discuss in section 5.5, and HSR cannot apply with *on* as a head, because the predicative *on* has its PRED feature set to +.

(67)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{on}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{prep} & \\ \text{FORM} & \text{on} \\ \text{PRED} & + \end{bmatrix} \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle \text{NP} \rangle \\ \text{COMPS} & \langle \text{NP} \rangle \\ \text{MOD} & \left\langle \begin{bmatrix} \textit{mod-elem} & \\ \text{MODIFIED} & \text{S} \mid \text{NOM} \\ \text{AFTER} & + \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix}$$

This preposition can combine with a NP to the right, by an application of the Head Complement Rule, and the resulting phrase structure can follow the sentence *The cat walks* to modify it by an application of the Head Modifier Rule (Post-head).

As expressed in the MODIFIED feature, this prepositional modifier can modify a nominal as well, like the sentence (68), where *on the roof* modifies the nominal *cat*.

(68) The cat on the roof is staring at you.

Subordinate Conjunctions

What we present in this section is enough to analyze the syntactic characteristics of subordinate conjunctions (represented by *sconj* subtype of *pos*) like, *when*, *after*, *before*, and *if*. Later in the next chapter we see how the their semantic properties can be handled. The analysis of all of the subordinate conjunctions are done in the same fashion. For the importance of conditional sentences in software requirement texts, we focus on the subordinate conjunction *if*.

The structure of the conditional sentence that we analyze is shown in (69). Note that *then*, can be simply replaced by a comma to analyze another similar structure.

(69) $\underbrace{\textit{if} \ < \text{condition} > \ \textit{then}}_{\text{modifier}} \quad \underbrace{< \text{statement} >}_{\text{modified}}$

[5] We have used | to denote disjunction.

In our analysis, we treat *if* both as a head and a modifier in a single feature structure description:

- A head that takes two complements, the first is a sentence (that plays the role of the condition in the conditional statement), and the second one is the preposition *then.*
- A modifier that comes before the sentence it modifies.

The feature structure description of *if* suitable for this analysis is shown in (70).

(70)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{if}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \textit{sconj} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \left\langle \text{S, PP[then]} \right\rangle \\ \text{MOD} & \left\langle \begin{bmatrix} \textit{mod-elem} & \\ \text{MODIFIED} & \text{S} \\ \text{AFTER} & - \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Since the valence features of *if*, specify both a modifier role, and a head role with some complements, it is the collaboration of two grammar rules, namely, HCR, and HMR that licenses a conditional statement like (71).

(71) If you go to school then you can learn a new language.

However, an invalid sentence like (72) cannot be licensed by the unwanted collaboration of HCR, and HMR. The reason is that any application of HCR precedes the application of HMR, because HMR requires the COMPS list of the modifier to be the empty list.

(72) * If you go to school you can learn a new language then.

A complementary analysis of *if* treats it as a post head modifier that modifies a sentence:

(73) $\underbrace{<\text{statement}>}_{\text{modified}}\quad \underbrace{\textit{if}\ <\text{condition}>}_{\text{modifier}}$

The feature structure description of *if* with this analysis is presented in (74).

(74)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{if}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \textit{sconj} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \left\langle \text{S} \right\rangle \\ \text{MOD} & \left\langle \begin{bmatrix} \textit{mod-elem} & \\ \text{MODIFIED} & \text{S} \\ \text{AFTER} & + \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

We will have more to say about conditional statements from a semantic viewpoint, in the next chapter.

5.3 Coordination

A coordinative conjunction (represented by the *cconj* subtype of *pos*) is a conjunction that combines two or more expressions that belong to the same part-of-speech, and have identical valence features[6].

To deal with coordination, we have to posit new grammar rules. These rule are headless, as all constituents of a coordinated phrase have parallel roles and importance.

In this thesis we only study coordinated sentences and a special form of coordinated noun phrases.

5.3.1 Coordination of Sentences

As mentioned in section 5.2.4 a sentence is a saturated verb phrase, i.e., a verb phrase with empty SPR and COMPS lists. Sentences can be coordinated by conjunctions like *and*, and *or*.

We should prevent the combination of an inverted (INV +) sentence (for questions) with a non-inverted (INV −) sentence, so in the rule below the value of the INV feature is required to be identical for the coordinated sentences. The GAP feature is the topic of section 5.6, but using it here simply means that the sentences must not miss any of their subconstituents.[7]

(75) Coordination Rule for Sentences Version I:

$$\text{S}\begin{bmatrix}\text{INV} & \boxed{1}\\ \text{GAP} & \langle\rangle\end{bmatrix} \rightarrow \text{S}\begin{bmatrix}\text{INV} & \boxed{1}\\ \text{GAP} & \langle\rangle\end{bmatrix} \begin{bmatrix}\text{HEAD} & \textit{cconj}\end{bmatrix} \text{S}\begin{bmatrix}\text{INV} & \boxed{1}\\ \text{GAP} & \langle\rangle\end{bmatrix}$$

5.3.2 Coordination of Countable Noun Phrases

In this subsection we analyze a kind of coordination using the conjunction *and* between countable nouns N_1, and N_2 that semantically forms another countable noun that is a kind

[6] This excludes the guards as we will see in chapter 7

[7] In this thesis we do not study the coordination of expressions that are incomplete or *gappy*.

of N_1, and N_2, which is specially useful for multiple inheritance.[8]

(76) Coordination Rule for Noun Phrases (com3sg) Version I:

$$\left[\begin{array}{ll} \textit{phrase} & \\ \text{SYN} & \left[\begin{array}{ll} \text{HEAD } \boxed{0} & \left[\begin{array}{ll} \textit{noun} & \\ \text{AGR} & \textit{3sing} \\ \text{PRO} & - \\ \text{TYPE\_DEF} & + \end{array}\right] \\ \text{VAL} & \left[\begin{array}{ll} \text{SPR} & \left\langle \boxed{1} \right\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{array}\right] \\ \text{GAP} & \langle\rangle \end{array}\right] \end{array}\right] \rightarrow$$

$$\left[\begin{array}{ll} \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \boxed{0} \\ \text{VAL} & \left[\begin{array}{ll} \text{SPR} & \left\langle \boxed{1} \right\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{array}\right] \\ \text{GAP} & \langle\rangle \end{array}\right] \end{array}\right] \left[\begin{array}{ll} \text{ORTH} & [\text{and}] \\ \text{HEAD} & \left[\begin{array}{ll} \textit{nom-co-conj} & \\ \text{CONJ\_TYPE} & \text{com3sg} \end{array}\right] \end{array}\right] \left[\begin{array}{ll} \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \boxed{0} \\ \text{VAL} & \left[\begin{array}{ll} \text{SPR} & \left\langle \boxed{1} \right\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{array}\right] \\ \text{GAP} & \langle\rangle \end{array}\right] \end{array}\right]$$

To form a coordinated noun phrase of this kind, we need each constituent to be a third person singular and a countable noun. The feature TYPE_DEF is used here with the value +, to insist that the constituents must represent a type, and their coordination forms a noun that carries a type that is resulted from multiple inheritance. We will revise this rule in chapter 6 to provide semantics.

5.4 Argument Structure : ARG-ST Feature

Now that we have studied the standard valence features, and shown how they can be used to analyze several syntactic structures, we are ready to discuss the ARG-ST feature. This feature is introduced by the *lex-sign* type, from which *word*, and *lexeme* inherit. This feature provides a bridge between the syntax and the semantics of a *word* or a *lexeme*. Also using it makes the presentation of some feature structure descriptions more concise.

ARG-ST is a feature whose value is a list resulted from the concatenation of the SPR list with the COMPS list. Elements of this list are called the *arguments* of the *word* or *lexeme*. Specially, we later see that for verb phrases and predicative expressions, the semantic objects referred to by the elements of ARG-ST are actually the arguments of the semantic

[8] There is another form of coordination between noun phrases that semantically builds a collection. However we do not analyze the collective and plural nouns in this thesis, for conciseness and unnecessary complications.

predication introduced by that verb phrase or predicative phrase. By a predication we mean simple and basic FOL like predicates that are used to describe the semantics of expressions.

Formally for any feature structure description of a *word* or a *lexeme* we have:

(77) Argument Realization Principle (ARP):
ARG-ST = SPR $\oplus$ COMPS $\oplus$ GAP

where $\oplus$ denotes the concatenation operator. GAP which is a list of expressions is the topic of section 5.6. For simple phrases that do not miss any of their constituents it is the empty list. In this thesis we suppose that the lexical entries, have their GAP feature empty.[9]

5.5 Copulas and Auxiliary Verbs

Copulas or *copular* or *linking verbs* are those verbs like *be*, *seem*, *feel*, that link a subject to an adjective, a noun phrase or another predicative expression. The PRED feature in *pos* specifies if the corresponding expression can serve as a predicate after a copular verb.

Auxiliary verbs fall into two categories:

- *modals*, like *can*, *could*, *will*, *would*, *may*, *might*, *shall*, *should*, ... that add a meaning of possibility, willingness, permission, or obligation to the verb they attach to.
- *helping verbs*, like *be*, *have*, and *do* that add an extra meaning of continuation, completion, and emphasis to the verb they attach to.

In this section we analyze all these verbs under the same umbrella, as they have significant similarities. The difference is mostly in the semantics, where different meanings are added. These analyses use the notion of *subject raising* [66]. Note that all of the verbs in the categories above can be thought of as main verbs of the clause that have a VP or predicative complement. Subject raising basically has the effect that the subject of the copula or the auxiliary verb is syntactically and semantically the same as the subject of the embedded VP or the predicative complement. This effect is called *subject sharing* in [67].

The only copular verb that we study here is *be*. The analysis of *be* provided in [67] covers both its copular and auxiliary case (for progressive verb forms) simultaneously. So

[9] However, as we see in section 5.6, there are lexical rules that create *gappy* words, which have non-empty GAP features.

for simplicity we call this verb an auxiliary verb too (even in its copular mode), and as in [67] we will set its AUX feature to +.

The auxiliary verbs are subject to Negation, Inversion, Contraction and Ellipsis (NICE). In this thesis we cover only negation and inversion. Sag et al. in [67] provide hierarchies of lexemes working with lexical rules to deal with these phenomena. However, here for the purpose of conciseness we try to avoid referring to the hierarchies of lexemes and lexical rules. So we provide constraints that must be respected for the auxiliary verbs in each of these modes[10]:

- Non-Negated and Non-Inverted (NN-NI) Version I:

  The feature structure description of a word that is associated with a non-negated and non-inverted auxiliary verb must be unifiable with:

  (78)
  $$\begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{AUX} & + \\ \text{INV} & - \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle \boxed{1} \rangle \end{bmatrix} \end{bmatrix} \\ \text{ARG-ST} & \left\langle \boxed{1}\text{NP} , \begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{VAL} & \begin{bmatrix} \text{SPR} & \langle \boxed{1} \rangle \\ \text{COMPS} & \langle \rangle \end{bmatrix} \end{bmatrix} \end{bmatrix} \right\rangle \end{bmatrix}$$

- Non-Negated and Inverted (NN-I) Version I:

  The feature structure description of a word that is associated with a non-negated but inverted auxiliary verb must be unifiable with:

  (79)
  $$\begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{AUX} & + \\ \text{INV} & + \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle \rangle \end{bmatrix} \end{bmatrix} \\ \text{ARG-ST} & \left\langle \boxed{1}\text{NP} , \begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{VAL} & \begin{bmatrix} \text{SPR} & \langle \boxed{1} \rangle \\ \text{COMPS} & \langle \rangle \end{bmatrix} \end{bmatrix} \end{bmatrix} \right\rangle \end{bmatrix}$$

- Negated and Non-Inverted (N-NI) Version I:

  The feature structure description of a word that is associated with a negated but non-inverted auxiliary verb must be unifiable with:

[10] We will revise these constraints in section 6.10.1 of chapter 6 to incorporate the semantics of negation, and once again in chapter 7 to enable type restrictions.

(80) $$\begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{AUX} & + \\ \text{INV} & - \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle \boxed{1} \rangle \end{bmatrix} \end{bmatrix} \\ \text{ARG-ST} & \left\langle \boxed{1}\text{NP} , \begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{not}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \textit{adv-pol} \end{bmatrix} \end{bmatrix} , \begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{VAL} & \begin{bmatrix} \text{SPR} & \langle \boxed{1} \rangle \\ \text{COMPS} & \langle\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix} \right\rangle \end{bmatrix}$$

- Negated and Inverted (N-I) Version I:

  The feature structure description of a word that is associated with a negated and inverted auxiliary verb must be unifiable with:

  (81) $$\begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{AUX} & + \\ \text{INV} & + \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \end{bmatrix} \end{bmatrix} \\ \text{ARG-ST} & \left\langle \boxed{1}\text{NP} , \begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{not}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \textit{adv-pol} \end{bmatrix} \end{bmatrix} , \begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{VAL} & \begin{bmatrix} \text{SPR} & \langle \boxed{1} \rangle \\ \text{COMPS} & \langle\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix} \right\rangle \end{bmatrix}$$

Now we are able to provide the lexical entry for the verb *be* as shown in (82).

(82) $$\begin{bmatrix} \textit{lexeme} & \\ \text{ORTH} & [\text{be}] \\ \text{ARG-ST} & \left\langle \text{NP}, \begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \text{PRED} & + \end{bmatrix} \end{bmatrix} \end{bmatrix} \right\rangle \end{bmatrix}$$

There must be enough lexical rules to transform lexemes such as the one above to fully inflected verb forms that contain the inflected phonology (orthography here) with the agreement information, and the appropriate FORM feature. The end result is a *word* that must respect all the above constraints related to auxiliary verbs.

For example for first person plural the following feature structure description must be derivable from the lexeme shown in (82).

(83) $$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{are}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{FORM} & \text{fin} \\ \text{PRED} & - \\ \text{AGR} & \textit{plural} \\ \text{AUX} & + \\ \text{INV} & - \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle \boxed{1} \rangle \end{bmatrix} \end{bmatrix} \\ \text{ARG-ST} & \left\langle \boxed{1}\text{NP} , \begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \text{PRED} & + \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle \boxed{1} \rangle \\ \text{COMPS} & \langle\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix} \right\rangle \end{bmatrix}$$

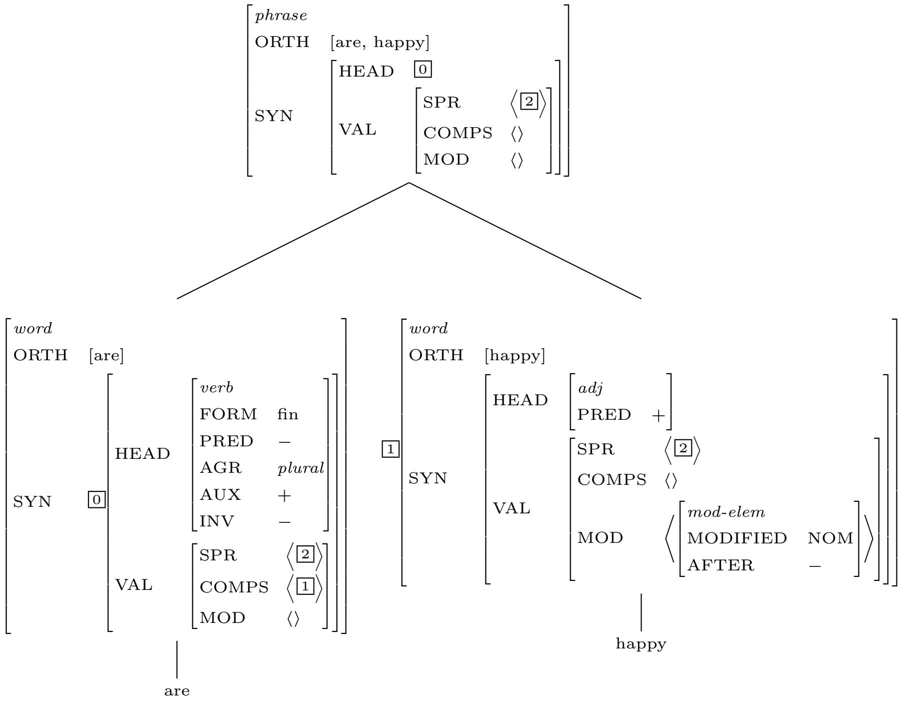
Suppose that the feature structure descriptions of *happy*, and *singing* are provided in (84) and (85) respectively.

(84)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{happy}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{adj} & \\ \text{PRED} & + \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle \text{NP} \rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \left\langle \begin{bmatrix} \textit{mod-elem} & \\ \text{MODIFIED} & \text{NOM} \\ \text{AFTER} & - \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

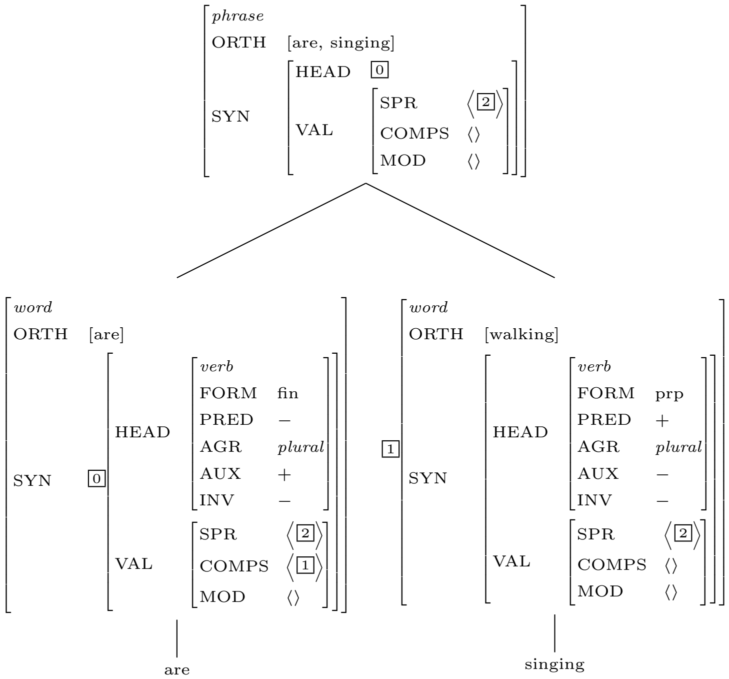
(85)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{singing}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{FORM} & \text{prp} \\ \text{PRED} & + \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle \text{NP} \rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Now using the Head Complement Rule, the verb *are* can be combined with each of the two words *happy*, and *singing*, as shown by the phrase structure trees of (86) and (87)

(86)



(87)



Then these two verb phrases can combine with *we* by an application of the Head Specifier Rule to form the sentences (88a) and (88b). This shows how the analyses of copular and helping verb uses of *be* can be simultaneously achieved.

(88) a. We are happy.

b. We are singing.

We have not yet discussed why the adjectives must have a non-empty SPR list. Note that this specifier is unified with the specifier of the auxiliary verb in (86). It is said that the adjective *happy* shares its subject with the copular verb *be*. The reason for this is discussed in our semantic analysis of copular and auxiliary verbs in chapter 6, section 6.10.

For negation, as seen in N-NI (80) and N-I (81) constraints, we have introduced an *adv-pol* part of speech, that sits between the subject and the complement of the auxiliary verb. This will enable HCR to license negated verb phrases such as:

(89) a. am not tired.

b. Am I not happy?

Note that (89b) does not require an application of HSR, because the SPR list is empty. However (89a) needs to be combined with a subject by HSR to form a sentence like (90).

(90) I am not tired.

The inversion constraints NN-I (79), and N-I (81) force the subject to come after the auxiliary verb by making the SPR list empty. However, since the ARG-ST is intact, the subject will have to appear as the first element of COMPS to keep the constraint (77) satisfied. Then applications of HCR will license sentences like (89b) or (91).

(91) Am I happy?

The analyses of other auxiliary verbs are very similar, the only difference lies in their lexical entries. For example the lexical entry of *can*, and *have* are provided in (92) and (93), without the semantic features.

(92)
$$\left[\begin{array}{ll} \textit{lexeme} & \\ \text{ORTH} & [\text{can}] \\ \text{SYN} & \left[\begin{array}{ll}\text{HEAD} & \left[\begin{array}{ll}\text{FORM} & \text{fin}\end{array}\right]\end{array}\right] \\ \text{ARG-ST} & \left\langle \text{NP}, \left[\begin{array}{ll}\text{SYN} & \left[\begin{array}{ll}\text{HEAD} & \left[\begin{array}{ll}\textit{verb} & \\ \text{FORM} & \text{base}\end{array}\right]\end{array}\right]\end{array}\right]\right\rangle \end{array}\right]$$

(93)
$$\left[\begin{array}{ll} \textit{lexeme} & \\ \text{ORTH} & [\text{have}] \\ \text{ARG-ST} & \left\langle \text{NP}, \left[\begin{array}{ll}\text{SYN} & \left[\begin{array}{ll}\text{HEAD} & \left[\begin{array}{ll}\textit{verb} & \\ \text{FORM} & \text{psp}\end{array}\right]\end{array}\right]\end{array}\right]\right\rangle \end{array}\right]$$

5.6 Gaps : The Missing Phrasal Constituents

Words that we have studied thus far have specified specifiers and complements, and the saturated (i.e., complete) phrases that can be built from them had to contain all elements that were initially in SPR and COMPS lists. However, in some cases such as relative clauses or *wh*-questions it is possible that one argument is missing from a phrase and that some of its syntactic and semantic characteristics are passed to another constituent that comes before the phrase, as in sentences (94a), (94b). To show the missing constituent we have used ___.

(94) a. I liked the book which you gave me ___.

b. Whom did you sell my ticket to ___?

In sentence (94a), *you gave me___* is a verb phrase that is missing its indirect object by itself. This verb phrase is attached to a main clause, *I liked the book*, by the *relativizer which*. *book* in the main clause plays the syntactic and semantic role of the missing indirect object in the relative clause.

In (94b), *did you sell my ticket to* ___ is a question that is again missing an indirect object. *Whom* that came before the question completes the question by providing the missing object.

A local subtree in a phrase structure tree is defined to be a mother node with its daughters. The rules and principles that we have studied thus far only constrained the feature structure descriptions of the nodes in a local subtree. In other words, co-occurrence restrictions operated on local subtrees. However, in a sentence like (94a) there are two local subtrees, and the syntactic and semantic restrictions that are posed on the missing expression by the phrase which misses it should be met by the expression in the other phrase that contains the element that takes the role of the missing expression. This means co-occurrence restrictions are not local for these kind of sentences.

Since multiple local subtrees are involved, and the restrictions are needed to be posed on expressions that could appear far from each other in a sentence, the analysis of such co-occurrence restrictions is called *long distance dependencies*.

The existence of phrases like *you gave me___* in natural language requires that we allow their licensing on the condition that the missing expression be filled in later. This is where features GAP and STOP-GAP introduced by *syn-cat* come handy.

The GAP feature represents the missing constituents, or *gaps* as we call them. As we mentioned in section 5.4, the lexical entries must specify their GAP as empty. But we assume there are lexical rules that can fill this feature while respecting the Argument Realization Principle at the same time, by removing some elements of SPR or COMPS and adding them to GAP. In this thesis we have introduced a feature GAP-TYPE (or GAP-T for short) for the *verb* to specify whether the corresponding *word* after the introduction of gaps by the mentioned lexical rules is missing a subject argument or not. Permissible values of this feature are gsubj, gnosubj, where gsubj indicates that the *word* has an element in its GAP feature that was removed from SPR (and is hence a subject).

The STOP-GAP feature is empty for all expressions that cannot act as heads. For an expression that can act as a head with a gappy argument, it can contain the feature structure description of an expression that fills in the gap of one of its sisters.

Assumption 5.13
In this thesis we suppose that a word cannot have more than one gap.

The following feature structures can be derived from the lexicon for the verb *gave*:

(95)
$$\left[\begin{array}{ll} \textit{word} & \\ \text{ORTH} & [\text{gave}] \\ \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \left[\begin{array}{ll} \textit{verb} & \\ \text{FORM} & \text{fin} \\ \text{PRED} & - \\ \text{AUX} & - \\ \text{INV} & - \\ \text{GAP-T} & \text{gnosubj} \end{array}\right] \\ \text{VAL} & \left[\begin{array}{ll} \text{SPR} & \left\langle \text{NP[CASE nom]} \right\rangle \\ \text{COMPS} & \left\langle \text{NP}_1\text{[CASE acc], NP}_2\text{[CASE acc]} \right\rangle \\ \text{MOD} & \langle\rangle \end{array}\right] \\ \text{GAP} & \langle\rangle \\ \text{STOP-GAP} & \langle\rangle \end{array}\right] \end{array}\right]$$

(96)
$$\left[\begin{array}{ll} \textit{word} & \\ \text{ORTH} & [\text{gave}] \\ \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \left[\begin{array}{ll} \textit{verb} & \\ \text{FORM} & \text{fin} \\ \text{PRED} & - \\ \text{AUX} & - \\ \text{INV} & - \\ \text{GAP-T} & \text{gsubj} \end{array}\right] \\ \text{VAL} & \left[\begin{array}{ll} \text{SPR} & \langle\rangle \\ \text{COMPS} & \left\langle \text{NP}_1\text{[CASE acc], NP}_2\text{[CASE acc]} \right\rangle \\ \text{MOD} & \langle\rangle \end{array}\right] \\ \text{GAP} & \left\langle \text{NP[CASE nom]} \right\rangle \\ \text{STOP-GAP} & \langle\rangle \end{array}\right] \end{array}\right]$$

(97)
$$\left[\begin{array}{ll} \textit{word} & \\ \text{ORTH} & [\text{gave}] \\ \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \left[\begin{array}{ll} \textit{verb} & \\ \text{FORM} & \text{fin} \\ \text{PRED} & - \\ \text{AUX} & - \\ \text{INV} & - \\ \text{GAP-T} & \text{gnosubj} \end{array}\right] \\ \text{VAL} & \left[\begin{array}{ll} \text{SPR} & \left\langle \text{NP[CASE nom]} \right\rangle \\ \text{COMPS} & \left\langle \text{NP}_2\text{[CASE acc]} \right\rangle \\ \text{MOD} & \langle\rangle \end{array}\right] \\ \text{GAP} & \left\langle \text{NP}_1\text{[CASE acc]} \right\rangle \\ \text{STOP-GAP} & \langle\rangle \end{array}\right] \end{array}\right]$$

(98)
$$\left[\begin{array}{ll} \textit{word} & \\ \text{ORTH} & [\text{gave}] \\ \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \left[\begin{array}{ll} \textit{verb} & \\ \text{FORM} & \text{fin} \\ \text{PRED} & - \\ \text{AUX} & - \\ \text{INV} & - \\ \text{GAP-T} & \text{gnosubj} \end{array}\right] \\ \text{VAL} & \left[\begin{array}{ll} \text{SPR} & \left\langle \text{NP[CASE nom]} \right\rangle \\ \text{COMPS} & \left\langle \text{NP}_1\text{[CASE acc]} \right\rangle \\ \text{MOD} & \langle\rangle \end{array}\right] \\ \text{GAP} & \left\langle \text{NP}_2\text{[CASE acc]} \right\rangle \\ \text{STOP-GAP} & \langle\rangle \end{array}\right] \end{array}\right]$$

The phrase *you gave me* ___ in (94a) is licensed by applications of HCR and HSR on the word structure of (98).

So we have virtually dealt with the values of GAP and STOP-GAP for words. STOP-GAP is the empty list for phrases, because STOP-GAP already served its purpose when the phrase was formed. For the value of GAP inside phrases we need to introduce a new grammar principle.

(99) The GAP Principle

In any phrase, the value of the GAP feature of the mother phrase is equal to the following, where n is the number of daughters, and G_i is the value of the GAP feature of the i*th* daughter, and SG is the value of the STOP-GAP feature of the head daughter if the phrase is headed, or the empty list if the phrase is headless.

$G_1 \oplus ... \oplus G_n \ominus SG$

The $\ominus$ operation is the list subtraction. The result of this operation is the same as A if the elements of B do not occur in A. The result of A $\ominus$ B may be not unique, for example [NP_1, NP_2] $\ominus$ [NP] can be any of the following depending on the internal structure of NP, NP_1, and NP_2.

- [NP_1, NP_2]
- [NP_2]
- [NP_1]

In this thesis we only analyze two cases of long distance dependencies, namely, relative clauses, and *wh*-questions. Next we see how the gappy phrase *you gave me* ___ can be used in complete sentences as a relative clause or in *wh*-questions.[11]

5.6.1 Relative Clauses

A relative clause is the one that comes after nominals (NOMs) to identify them (defining relative clauses) or after nouns to give more information about them (non-defining). In this thesis we only study the defining relative clauses. A relative clause starts by a *relativizer*, which is either a *wh*-word or *that*, and followed by a gappy sentence. The whole relative clause acts as a modifier for the nominal that precedes it.

A relativizer is a kind of subordinate conjunction, which we talked briefly about at the end of section 5.2.5. So, in our analysis we treat relativizers as both heads (with gappy sentence complements) and as modifiers[12].

(100) $\underbrace{\text{NOM}}_{\text{modified}} \quad \underbrace{< \text{relativizer} > \quad < \text{gappy sentence} >}_{\text{modifier}}$

As an example, the feature structure description of the relativizer *which*[13] if presented in (101).

(101) $$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{which}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \textit{sconj} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \left\langle \begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{INV} & - \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \langle\rangle \end{bmatrix} \\ \text{GAP} & \left\langle \boxed{1}\text{NP}\begin{bmatrix}\text{AGR } \boxed{2}\end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix} \right\rangle \\ \text{MOD} & \left\langle \begin{bmatrix} \textit{mod-elem} & \\ \text{MODIFIED} & \text{NOM}\begin{bmatrix}\text{AGR } \boxed{2}\end{bmatrix} \\ \text{AFTER} & + \end{bmatrix} \right\rangle \end{bmatrix} \\ \text{GAP} & \langle\rangle \\ \text{STOP-GAP} & \langle \boxed{1} \rangle \end{bmatrix} \end{bmatrix}$$

[11] The full analysis of long distance dependencies requires at least one additional rule, the Head Filler Rule, but in this thesis we do not need it.

[12] Recall that *if* is a head and a modifier at the same time.

[13] Note that *which* can be used as a question word too, but that requires a different analysis, and an additional lexical entry.

Now we discuss how *which* can combine with the gappy sentence *you gave me* ___ to form a relative clause. The phrase structure of *you gave me* ___ that is obtained from (98) after an application of HCR followed by an application of HSR is shown in (102).

(102)

$$\left[\begin{array}{ll} \textit{word} & \\ \text{ORTH} & [\text{you, gave, me}] \\ \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \left[\begin{array}{ll} \textit{verb} & \\ \text{FORM} & \text{fin} \\ \text{PRED} & - \\ \text{AUX} & - \\ \text{INV} & - \\ \text{GAP-T} & \text{gnosubj} \end{array}\right] \\ \text{VAL} & \left[\begin{array}{ll} \text{SPR} & \langle\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{array}\right] \\ \text{GAP} & \left\langle \text{NP}_2[\text{CASE acc}] \right\rangle \\ \text{STOP-GAP} & \langle\rangle \end{array}\right] \end{array}\right]$$

The above feature structure description can be unified with the only COMPS element of *which*, so HCR licenses the phrase structure tree shown in (106). Note that the GAP feature of the mother phrase is empty as a result of the GAP principle, and the presence of an element in the head's STOP-GAP list which matches the complement's GAP element.

Since there are no more GAPs in the mother phrase, and that it is a subordinate conjunction with a non-empty MOD list, it acts as a normal modifier, which can be combined with a NOM like *book* by the Head Modifier Rule to form the nominal phrase shown in (103).

(103) book which you gave me

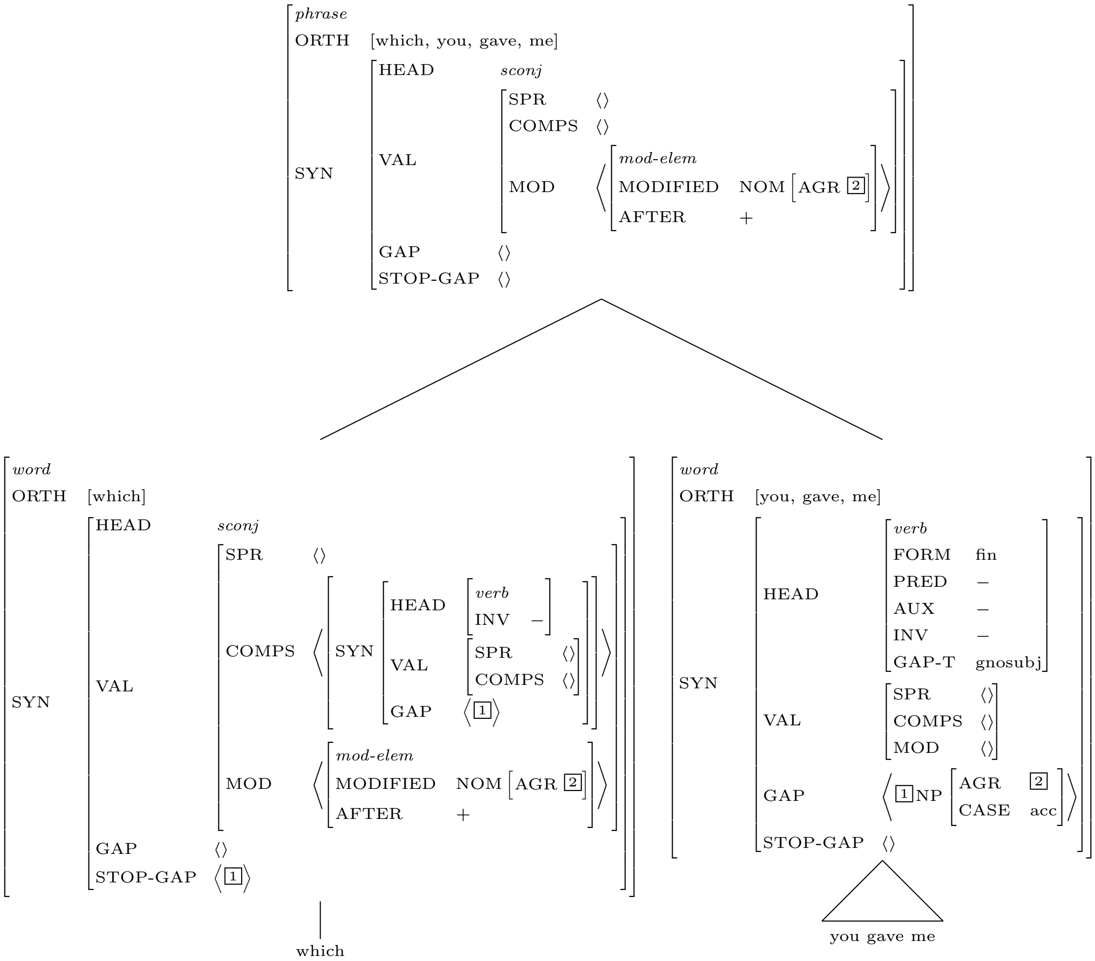
Later this nominal can be combined with the determiner *the* by the Head Specifier Rule to form the following noun phrase.

(104) the book which you gave me

And this can in turn be combined with *i like* by the Head Complement Rule to form the complete sentence below.

(105) I like the book which you gave me.

(106)


phrase
ORTH [which, you, gave, me]
SYN
HEAD sconj
VAL
SPR ⟨⟩
COMPS ⟨⟩
MOD ⟨ mod-elem MODIFIED NOM [AGR 2] AFTER + ⟩
GAP ⟨⟩
STOP-GAP ⟨⟩
word
ORTH [which]
SYN
HEAD sconj
VAL
SPR ⟨⟩
COMPS ⟨ SYN HEAD verb INV − VAL SPR ⟨⟩ COMPS ⟨⟩ GAP ⟨1⟩ ⟩
MOD ⟨ mod-elem MODIFIED NOM [AGR 2] AFTER + ⟩
GAP ⟨⟩
STOP-GAP ⟨1⟩
which
word
ORTH [you, gave, me]
SYN
HEAD verb FORM fin PRED − AUX − INV − GAP-T gnosubj
VAL SPR ⟨⟩ COMPS ⟨⟩ MOD ⟨⟩
GAP ⟨ 1NP [AGR 2 CASE acc] ⟩
STOP-GAP ⟨⟩
you gave me

5.6.2 *Wh*-Questions

For some semantic reasons, we would like to treat *wh* question words like *What*, *Who*, *Whom* as heads. These reasons include that the presence of these words at the beginning of a sentence turns the whole sentence into a question, which is a significant change in the semantic mode of the utterance. For syntactic purposes this also helps formulate a simple analysis of the *wh* questions.

We introduce a new grammar category *qword* which is a subtype of *part-of-speech* (*pos*) in our grammar type hierarchy. Any *wh*-question word needs a gappy sentence as its complement. A question word like *Who* that questions about the subject, requires a gappy sentence whose GAP-TYPE is gsubj, meaning that the sentence must be missing its subject. A question word like *Whom* that questions about an object, requires its complement to have gnosubj as its GAP-TYPE value. If *Who* is questioning the object, then its complement must be similar to that of *Whom.* Words like *Who* and, *What* that can pose a question on either subject or object of a sentence need different lexical entries for each sense.

As an example, we have shown the feature structure description of the question word *Who* in (107), where it is questioning the subject. Note the restriction on the complement sentence that requires it not to be inverted (INV $-$).

(107)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{who}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \textit{qword} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \left\langle \text{S} \begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \text{INV} & - \\ \text{GAP-TYPE} & \text{gsubj} \end{bmatrix} \\ \text{GAP} & \left\langle \boxed{1}\text{NP} \right\rangle \end{bmatrix} \end{bmatrix} \right\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \\ \text{GAP} & \langle\rangle \\ \text{STOP-GAP} & \left\langle \boxed{1} \right\rangle \end{bmatrix} \end{bmatrix}$$

As noted above in English it is acceptable that *Who* be used to question an object. In that case a new lexical entry for *Who* is needed with its GAP-TYPE feature set to gnosubj. Then it will be necessary for the complement sentence to be inverted. A feature structure description of *Who* in this sense is provided in (108).

(108)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{who}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \textit{qword} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \left\langle \text{S} \begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \text{INV} & + \\ \text{GAP-TYPE} & \text{gnosubj} \end{bmatrix} \\ \text{GAP} & \left\langle \boxed{1}\text{NP} \right\rangle \end{bmatrix} \end{bmatrix} \right\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \\ \text{GAP} & \langle\rangle \\ \text{STOP-GAP} & \left\langle \boxed{1} \right\rangle \end{bmatrix} \end{bmatrix}$$

Since the only means to produce inverted verb phrases is through the auxiliary verbs with their constraints, NN-I, N-I provided in (79), (81), it is necessary that the phrase contains an auxiliary verb too.

(109)
$$\begin{bmatrix} \textit{lexeme} & \\ \text{ORTH} & [\text{do}] \\ \text{SYN} & \begin{bmatrix}\text{HEAD} & \begin{bmatrix}\text{FORM} & \text{fin}\end{bmatrix}\end{bmatrix} \\ \text{ARG-ST} & \left\langle \text{NP}, \begin{bmatrix}\text{SYN} & \begin{bmatrix}\text{HEAD} & \begin{bmatrix}\textit{verb} & \\ \text{FORM} & \text{base}\\ \text{AUX} & -\end{bmatrix}\end{bmatrix}\end{bmatrix}\right\rangle \end{bmatrix}$$

Consider the feature structure description of the auxiliary verb *do* in (109). Then by applying the NN-I constraint we obtain the following feature structure description of *do*.

(110)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{do}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix}\textit{verb} & \\ \text{FORM} & \text{fin} \\ \text{AUX} & + \\ \text{INV} & + \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \left\langle \boxed{1}\text{NP}, \begin{bmatrix}\text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix}\textit{verb} & \\ \text{FORM} & \text{base} \\ \text{AUX} & - \end{bmatrix} \\ \text{VAL} & \begin{bmatrix}\text{SPR} & \left\langle \boxed{1} \right\rangle \\ \text{COMPS} & \langle\rangle \end{bmatrix} \end{bmatrix}\end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Then *you* can serve as the first complement of the inverted *do*, and *love* ___ can serve as its second complement. By an application of the Head Complement Rule the following gappy verb phrase can be licensed.

(111) do you love ___.

The gap in (111) is not a subject gap, and the value of GAP-T (short for GAP-TYPE) of this phrase is gnosubj. Note that we do not have a mechanism to pass the gap type of the non-head daughters to the mother. Although this can be easily done by pairing the elements of GAP with a gsubj or gnosubj value, it is not needed. The reason is that *wh*-questions with auxiliary verbs are followed by sentences that are missing some complement, whereas the subject is present. In other words, if an auxiliary verb has a verb phrase complement with a gap, that gap cannot be in the subject position, or otherwise the auxiliary verb would not be needed, and in fact should not be present at all. So the GAP-T of the auxiliary must be gnosubj, which is the default value for this feature.

(112)
$$\begin{bmatrix} \textit{phrase} & \\ \text{ORTH} & [\text{do, you, love}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{FORM} & \text{fin} \\ \text{PRED} & - \\ \text{AGR} & \textit{2sing} \\ \text{AUX} & + \\ \text{INV} & + \\ \text{GAP-T} & \text{nosubj} \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \\ \text{GAP} & \left\langle \boxed{1}\text{NP}\begin{bmatrix}\text{CASE acc}\end{bmatrix} \right\rangle \\ \text{STOP-GAP} & \langle\rangle \end{bmatrix} \end{bmatrix}$$

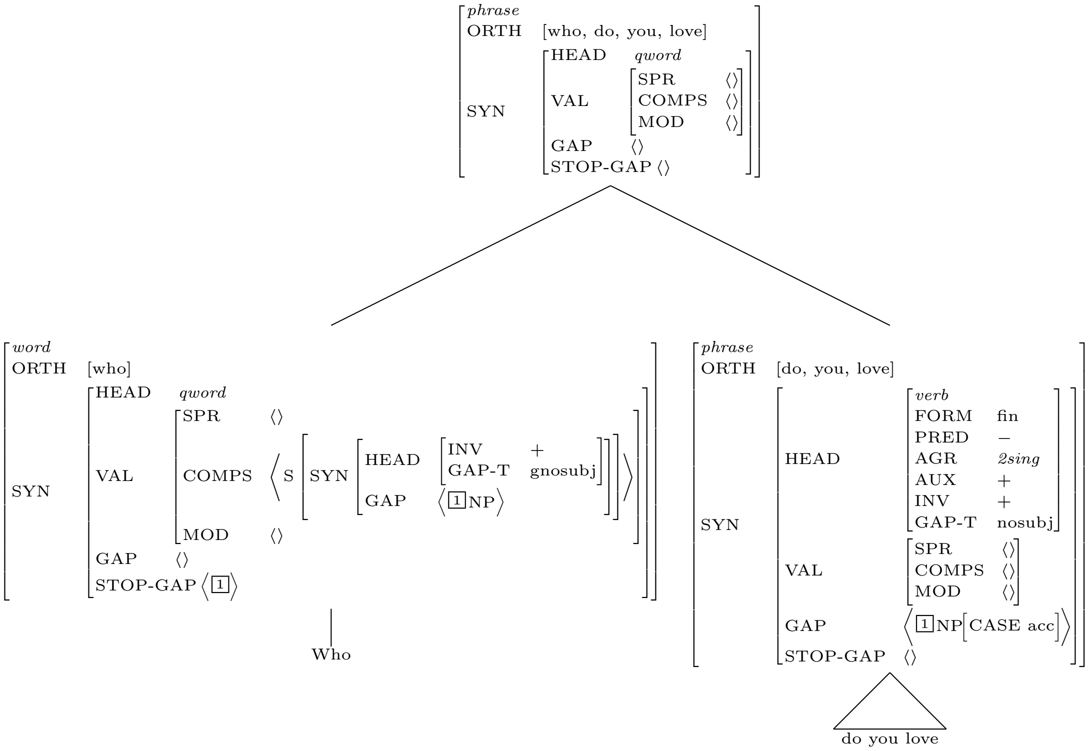
The phrase structure of the gappy inverted sentence (111) is shown in (112). This can serve as the complement of *Who* shown in (108). And the Head Complement Rule can license the following question with the phrase structure tree shown in (114).

(113) Who do you love ___?

5.7 Conclusion

In this chapter we explored the syntactic features of our grammar, and presented the grammar rules and principles that operate on syntax. The natural language structures we demonstrated in this chapter serve to specify the syntactic coverage of our grammar. In the next chapter we will provide a semantic framework for our grammar that will enable us to do some semantic analysis.

(114)


phrase
ORTH [who, do, you, love]
SYN
HEAD qword
VAL
SPR ⟨⟩
COMPS ⟨⟩
MOD ⟨⟩
GAP ⟨⟩
STOP-GAP ⟨⟩
word
ORTH [who]
SYN
HEAD qword
VAL
SPR ⟨⟩
COMPS ⟨S [SYN [HEAD [INV + GAP-T gnosubj] GAP ⟨1 NP⟩]]⟩
MOD ⟨⟩
GAP ⟨⟩
STOP-GAP ⟨1⟩
Who
phrase
ORTH [do, you, love]
SYN
HEAD
verb
FORM fin
PRED −
AGR 2sing
AUX +
INV +
GAP-T nosubj
VAL
SPR ⟨⟩
COMPS ⟨⟩
MOD ⟨⟩
GAP ⟨1 NP[CASE acc]⟩
STOP-GAP ⟨⟩
do you love

Chapter 6

Semantic Features and their Rules and Principles

The meaning, or *semantics*, of a linguistic utterance has a literal component which plays a significant role towards the total meaning, and a pragmatic component which often affects it also, involving the context and the implicit or explicit goals of those involved in the conversation.

In all that follows, we equate the semantics of an utterance with its literal component. This is in accordance with our objective: in a "formal" specification language, we must avoid non-relevant context, pragmatic issues, stylistic resources such as metaphor, etc., so that the meaning obtained from a formal specification (i.e., the result of its semantic analysis) is the literal meaning of the sentences that comprise it. This representation can later be used for human-assisted disambiguation and for reasoning and consistency checks for requirement specification and verification purposes.

We also adopt the compositionality hypothesis: we assume that a sentence's literal meaning can be constructed from the meaning of its parts. To construct it, we develop a framework based on Sag et al. [67], and we show how to incorporate semantics into the HPSG grammar presented in the previous chapter. For conciseness, we omit a discussion of the semantics of plural nouns, however an analysis very similar to chapter 8 of Carpenter [13] can be easily incorporated to our grammar.

Although pragmatics could be exempt from the "formal" specification aspect of software development, it should be considered in the general tasks of software design, development and documentation. Stylistic resources such as metaphor, analogy and examples could also help in documentations for human readers, by providing some preparation and assistance for understanding the formal specifications. However, we do not consider pragmatics in this thesis.[1]

6.1 Simple Semantic Features

HPSG like Categorial Grammar treats semantics in parallel with syntax. For this reason, every *sign* has a feature SEM of type *sem-cat* as shown in (1). This feature contains the semantics of the sign. As can be seen in the hierarchy presented in figure 4.1, *sem-cat* has 4 features, 3 of which are standard HPSG features (MODE, INDEX, RESTR), and a TYPE feature that we introduced to the grammar to represent the domain entity type of a sign. Next we study the standard features. We will study the TYPE feature in section 6.3.

$$\text{(1) a.}\begin{bmatrix} \textit{sign} & \\ \text{ORTH} & \\ \text{SYN} & \\ \text{SEM} & \textit{sem-cat} \end{bmatrix} \quad \text{b.}\begin{bmatrix} \textit{sem-cat} & \\ \text{MODE} & \{\text{prop, ques, dir, ref, ana, none}\} \\ \text{INDEX} & \textit{index} \\ \text{RESTR} & \textit{list of predication} \\ \underline{\text{TYPE}} & \underline{\textbf{BasTyp}} \end{bmatrix}$$

6.1.1 Semantic Mode

MODE	SHORT FOR	MEANING OF THE SIGN WITH THIS SEMANTIC MODE
prop	proposition	a proposition, for non-inverted (partial or complete) sentences
ques	question	a question, for inverted (partial or complete) sentences
dir	directive	a directive, for imperative (partial or complete) sentences
ref	referential	referring to a domain entity individual, for noun phrases
ana	anaphoric	referring to a domain entity individual, for reflexive pronouns
none	—	additional meaning for another constituent or no meaning

Table 6.1: Semantic Modes

The first semantic feature is MODE, whose values range within {prop, ques, dir, ref, ana, none}. Table 6.1 provides a short description of each. In this thesis we do not cover

[1] See chapter 6 of Bjørner [8] for a discussion on pragmatics.

imperative sentences and reflexive pronouns, so feature values 'dir', and 'ana' will not be used in our grammar.

[MODE prop], and [MODE ques] are usually used for verbs, verb phrases and sentences. In this thesis we use [MODE prop] also for predicative coordinate conjunctions that join two verb phrases or sentences, because the end result of the conjunction is a complex proposition.

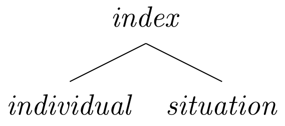
[MODE ref] is always used with pronouns, nouns and noun phrases, indicating that the expression's meaning is a reference to an individual from the application domain.

[MODE none] is used with all modifiers, and the grammar categories with no significant meaning contribution (such as argument marking prepositions). The meaning embedded in modifiers is copied to the phrase containing them by the *semantic compositionality principle* that we study later in this chapter.

6.1.2 Semantic Index

An index is a label of an *individual*, or a *situation*. We use natural numbers or lower case letters for indices. An *individual* is simply an entity in the domain of grammar application. It could be a person, an object or a group of people or objects, or a variable referring to them in certain constructs (like quantification). A *situation* is simply an event about some state holding or some action happening in the universe.

In feature structure descriptions, we use *individual* as a type of index that is used to refer to individuals. We also use *situation* as a type of index that is used to refer to an event. So an *index*, is either an *individual* or a *situation*, as shown in the type hierarchy below.



Following [67], if we use lower case letters as indices, then we use $i, j, k, ...$ for individuals and $s, u, w, s_1, s_2, ...$ for situations.

As we see later in section 6.5, it must be possible for indices to be unified if necessary. In Prolog variables can be used to represent indices, and this automatically allows for the unification of indices.

6.2 Semantic Restrictions

Now that we are able to refer to individuals or situations by *index* values, we need to specify the restrictions between them in the meanings that are carried by words or phrases. Along with the lines of Minimal Recursion Semantics (MRS) [19, 20][2] these restrictions are specified in terms of *elementary predications* or *EPs*. We will introduce the important concepts that are closely related to MRS through the rest of this chapter, as we give some semantic principles and examples along the way.

6.2.1 Elementary Predications

A very important assumption in MRS is that the linguistic semantics of an expression is expressible in terms of some atomic building blocks of meaning. These building blocks are called *elementary predications* or *EPs*. Each elementary predication conveys a very basic (atomic) meaning, and it can be underspecified by pointing to situations or individuals that are not accessible from the word or phrase that carries the predication as part of its meaning. These predications can combine to form complex meanings by the *semantic compositionality principle* that we see later in this chapter.

A predication[3] is basically a relation between some arguments. The feature structure description of a predication has at least 3 features. The first feature is RELN (short for RELATION) whose value is the relation name, or the predicate symbol. The second feature is SIT (short for SITUATION) whose value is a *situation* index. It labels its containing predication, so that it can be referred from other predications or from the semantics of a sign (by using the INDEX feature). We call the value of this feature the *label* of the predication that contains it.

We should mention that it is possible for several predications to have the same label. In such a case, all of those predications are referred to at once whenever their SIT value is used. In fact, we see later that the value of SIT is the scope in which the predicate that is labeled with it resides in. The third feature is the first argument of the relation. More features are added if the relation requires more arguments. The value of the argument features are usually *indices*. The argument features are given mnemonic names suitable for the role that

[2] The notation used in [19, 20] are a little different from [67]. Here we follow the notation used in [67].

[3] We use *predication* instead of *elementary predication* when it is clear that we are referring to *elementary predications*.

the argument plays in the relation.

For example, (2) shows a predication for the relation $\texttt{likes}(s,\ i,\ j)$.[4] The situation argument, s, basically acts as a label for the predication, so that it can be referred from other predications or directly from the INDEX feature of a sign. This predication has the meaning that s is a situation where the individual i likes the individual j. This meaning is underspecified in that we do not have enough information about the individuals i, and j.

(2) $$\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{like} \\ \text{SIT} & s \\ \text{LIKER} & i \\ \text{LIKED} & j \end{bmatrix}$$

Different Kinds of Elementary Predications

In this thesis we break down the predications into four categories:

- Ordinary predications, such as **like**, used for verbs and modifiers. They convey the meaning that some status holds among individuals or situations (events).

- Tense predications such as **time_present**, **time_past**, **time_future**, etc. that are used to add the tense information for verbs. We have chosen to use one argument, EVENT for these predications. They convey that the situation which is specified by their EVENT argument has happened in the past, is happening now, or is going to happen in the future, etc..

- Discourse predications, which access or modify the information that is collected so far in the discourse. This will be covered in sections 6.4, 6.6.

- Quantification predications, of which we only consider quantification predications over singular noun phrases.[5] This will be the topic of section 6.6.1.

- Outscope predications, these are used to express structural constraints on the semantics, useful for translating the predications to a Predicate Calculus (PC) representation

[4] This has an extra situation argument in comparison with the λ-term $\lambda j \lambda i.\texttt{likes}(i,\ j)$.

[5] Quantification over sets requires a representation language that goes beyond First Order Logic, as a set can be thought of as a unary relation, and quantification over relations is impossible in First Order Logic.

such as Typed First Order Logic (TFOL). We will see examples of these predications in section 6.8.2.[6]

6.2.2 The Value of RESTR Feature

(3)

$$\left[\begin{array}{ll} \textit{word} & \\ \text{ORTH} & [\text{likes}] \\ \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \left[\begin{array}{ll} \textit{verb} & \\ \text{FORM} & \text{fin} \\ \text{PRED} & - \\ \text{AGR} & \left[\textit{3sing}\right] \\ \text{AUX} & - \\ \text{INV} & - \end{array}\right] \\ \text{VAL} & \left[\begin{array}{ll} \text{SPR} & \left\langle \text{NP}_i[\text{CASE nom}] \right\rangle \\ \text{COMPS} & \left\langle \text{NP}_j[\text{CASE acc}] \right\rangle \\ \text{MOD} & \langle\rangle \end{array}\right] \\ \text{GAP} & \langle\rangle \\ \text{STOP-GAP} & \langle\rangle \end{array}\right] \\ \text{SEM} & \left[\begin{array}{ll} \text{MODE} & \text{prop} \\ \text{INDEX} & s \\ \text{RESTR} & \left\langle \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{like} \\ \text{SIT} & s_1 \\ \text{LIKER} & i \\ \text{LIKED} & j \end{array}\right], \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{time\_present} \\ \text{SIT} & s \\ \text{EVENT} & s_1 \end{array}\right] \right\rangle \end{array}\right] \end{array}\right]$$

A list of predications is used as a value of the RESTR (RESTRICTION) semantic feature of *sem-cat* type to complete the meaning of an expression or a lexeme (besides the semantic features MODE, and INDEX). This list of elementary predications describes the semantics (the literal meaning) of the corresponding sign in a flat representation.

For example the predication shown above in (2), together with a tense predication can be used to specify the meaning of the verb *likes* as in (3). We will describe how this conveys the meaning of the verb in what follows.

The reader has probably noticed that we have used *index* subscripts for the SPR and COMPS list elements in (3). This notation is explained below.

Notation 6.1 If a subscript i is used for a grammar category corresponding to an expression, then i is treated as the semantic index, i.e., the value of the INDEX feature of the feature structure description of that expression. In general we have:

[6] In the MRS literature such as [20, 19] these constraints are also called *handle constraints* and are maintained in a feature different from RESTR, that is HCONS. In this thesis we keep all of these predications in the RESTR feature.

(4) $\mathrm{X}_i = \mathrm{X}\left[\mathrm{SEM}\left[\mathrm{INDEX} \quad i\right]\right]$

For example:

(5) $\mathrm{NP}_i = \mathrm{NP}\left[\mathrm{SEM}\left[\mathrm{INDEX} \quad i\right]\right]$

The semantic index of a nominal expression (pronoun, noun, noun phrase) refers to the individual that the expression represents. So by using NP_i as the specifier of the verb (the subject of the verb) *likes*, and by using i again as the value of the argument LIKER in the semantic restrictions, we imply that the specifier of the verb plays the role of its subject or in this case the role LIKER.

Similarly we used NP_j as the only complement of the verb, and used j again as the value of the feature LIKED in the semantic restriction. This implies that the complement of the verb has the object role, or in this case, the individual that is being LIKED.

The meaning of the verb *likes* is conveyed through the two predications with relation names **like**, and **time_present**. The semantic index (value of INDEX) of the feature structure description of *likes* has the value s, which is the value of the SIT argument of the tense predication. The tense predication has an EVENT argument that has the value s_1, which is the value of the SIT argument of the **like** predication. The meaning comprises of two situations. One is the state s_1 of *liking* from i towards j, and the other is the situation s that asserts the state s_1 is happening at the present time. Simply, it means that i likes j at the present time.

The meaning of these predications can be given in a *predicate calculus* (PC for short) representation. The PC representation of:

(6)
$$\left\langle \begin{bmatrix} \textit{predication} & \\ \mathrm{RELN} & \textbf{like} \\ \mathrm{SIT} & s_1 \\ \mathrm{LIKER} & i \\ \mathrm{LIKED} & j \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \mathrm{RELN} & \textbf{time\_present} \\ \mathrm{SIT} & s \\ \mathrm{EVENT} & s_1 \end{bmatrix} \right\rangle$$

can be given by the formula:

(7) $\texttt{like}(s_1, i, j) \wedge \texttt{time\_present}(s,\, s_1)$

(8) Tense Predication Constraint:
In this thesis the tense predication always wraps a tense-less verb predication. We pose a constraint on the EVENT argument of the tense predication so that it cannot be shared by any predication other than the tense predication that embeds it. We call this the *tense predication constraint*.

We can rewrite the above formula in a different notation, where the value of the EVENT argument of the tense predication is replaced by the PC representation of the predication it refers to.

(9) $\texttt{time\_present}(s, \texttt{like}(s_1, i, j))$

Later in section 6.12 we see how a list of predications that correspond to a list of well-formed natural language sentences can be converted to a predicate calculus representation very similar to FOL.

Finally the semantic mode (value of MODE) is ‘prop’ meaning that the verb *likes* is going to be used in a sentence that makes a simple assertion or proposition. This is compatible with the arrangement of SPR and COMPS, and the INV feature value, that is, the verb is not inverted, the subject in the SPR comes before the verb, which is followed by the object in the COMPS list. So the order of the constituents is compatible with a sentence in propositional usage.

6.3 The TYPE Feature

This is a new semantic feature that we have introduced to the *sem-cat* type. The value of the TYPE feature for a nominal expression ξ is the domain entity type that the individual that is referred by ξ belongs to. The value of this feature is application specific, and must be defined in the lexicon for the nominal lexical entries for the specific application[7], although it might be possible to extract a basic lexicon which is common for several applications. To incorporate as much knowledge as possible we wish this type to be the most specific one that we are aware of at the time of defining the lexicon.

For example if *Mary* in an application domain is declared to be a *student*, where *student* is a subtype of *human*, then the feature structure description of the lexical entry of the

[7] For verbal lexical entries the value is **Bool**, and for all other non-nominal entries the value is the top element of the type hierarchy, i.e, *all*.

proper noun *Mary* for that application must specify the value of TYPE as *student* as shown below.

(10)
$$\begin{bmatrix} \text{ORTH} & [\text{mary}] \\ \text{SYN} & ... \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{ref} \\ \text{INDEX} & i \\ \text{TYPE} & \textit{student} \\ \text{RESTR} & ... \end{bmatrix} \end{bmatrix}$$

As another example we consider the personal pronoun *she*. The feature structure description of its lexical entry must specify the value of TYPE as ‘female-human$\curvearrowright$X’ as shown in (11), following our argument of using specializable types for pronouns in section 2.3.2.

(11)
$$\begin{bmatrix} \text{ORTH} & [\text{she}] \\ \text{SYN} & ... \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{ref} \\ \text{INDEX} & i \\ \text{TYPE} & \textit{female\_human}\curvearrowright\text{X} \\ \text{RESTR} & ... \end{bmatrix} \end{bmatrix}$$

In the next chapter we will see how we can restrict the applicability of our grammar rules, namely, HSR, HCR, HMR, and Coordination rules with respect to the TYPE feature of the constituents.

6.4 Semantics of Simple Singular Nominals

In our study of simple singular nominals in this section we are going to introduce the first three discourse predications we are using in this thesis. By simple singular nominals we mean one word singular noun phrases that do not need a determiner. These include proper nouns like *Mary*, and singular pronouns like *she*. The semantics of nominals in general adds a new individual to the *discourse context*, or it refers to an individual already in the discourse context.

Definition 6.2 Discourse Context[8]
The discourse context or DC for short in the set of structures that contain some information about the individuals in the context. For any individual in the context, this information must at least contain:

- the *index* used to refer to the individual
- the grammatical *agreement* information of the individual
- the *semantic type* of the individual

An elementary discourse predication with the relation name **instantiate** can encapsulate the above information for an individual. The feature structure description of this predication is shown in (12).

(12)
$$\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{instantiate} \\ \text{SIT} & \textit{situation} \\ \text{SYN\_AGR} & \textit{agr-cat} \\ \text{SEM\_TYPE} & \textbf{BasTyp} \\ \text{INST} & \textit{individual} \end{bmatrix}$$

The elementary discourse predication **dc_obj_add** adds a new individual to the discourse context. It should be coupled with a supplementary **instantiate** predication that provides the information about this new individual. The feature structure description of this predication is shown in (13).

(13)
$$\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{dc\_obj\_add} \\ \text{SIT} & \textit{situation} \\ \text{OBJ} & \textit{individual} \\ \text{INSTPRED} & \textit{situation} \end{bmatrix}$$

[8] The discourse context accompanied by the list of predications that refer to it to some extent resembles to the Discourse Representation Structure (DRS) [46] of that expression. However the discussion of discourse representation theory is beyond the scope of this thesis.

The value of the INSTPRED feature is the situation label of the **instantiate** predication that provides the discourse information about the new individual. This value must not be shared with any other predication except for the quantification predications as the value of QRESTR as we see later in this chapter. The value of OBJ feature is the index value of the individual being added to the context. This value must be identical to the INST feature value of the matching **instantiate** predication. We will see an example of **dc_obj_add** and **instantiate** shortly in section 6.4.1

Another elementary discourse predication that we use in our analysis of the semantics of singular nouns is **dc_obj_get**. Like **dc_obj_add** it should be coupled with an **instantiate** predication. It has the effect of searching the discourse context for an individual whose information matches the information provided by the supplementary **instantiate** predication. The feature structure description of this predication is shown in (14).

$$(14)\quad \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{dc\_obj\_get} \\ \text{SIT} & \textit{situation} \\ \text{OBJ} & \textit{individual} \\ \text{INSTPRED} & \textit{situation} \end{array}\right]$$

The value of the INSTPRED feature is the situation label of the coupled *instantiate* predication. The value of the OBJ feature is the index that is used to refer to the individual that is summoned from the context.

Next we study how these predications can be used to analyze the semantics of simple nominal expressions.

6.4.1 Semantics of Proper Nouns

A proper noun is usually a single word that refers to a named individual. Some examples of proper nouns are *Mary*, *John*, *King Edward*. The semantics of proper nouns comprises three predications. The first two are coupled discourse predications that add the individual to the discourse context, and the second one is the **name** predication that specifies the name of the individual. For example the feature structure description of the proper noun Mary is given in (15). We have omitted the syntactic features for keeping it relevant to semantics.

(15)

$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & \text{[mary]} \\ \text{SYN} & \ldots \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{ref} \\ \text{TYPE} & \textit{student} \\ \text{INDEX} & i \\ \text{RESTR} & \left\langle \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{dc\_obj\_add} \\ \text{SIT} & s \\ \text{OBJ} & i \\ \text{INSTPRED} & s_1 \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{instantiate} \\ \text{SIT} & s_1 \\ \text{SYN\_AGR} & \begin{bmatrix} \textit{3sing} & \\ \text{GEND} & \textit{fem} \end{bmatrix} \\ \text{SEM\_TYPE} & \textit{student} \\ \text{INST} & i \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{name} \\ \text{SIT} & s \\ \text{NAME} & \text{mary} \\ \text{NAMED} & i \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix}$$

In the above feature structure description, i, s, and s_1 are all different indices. And a new occurrence of *Mary* will use a potentially different set of indices. It depends on the application to decide whether two occurrences of *Mary* actually refer to the same individual or not. Here we assume they do refer to the same individual. For enforcing this assumption we will run a post parse procedure that unifies the indices of the individuals with the same name. This post process can be easily eliminated to suit the application.

6.4.2 Semantics of Singular Pronouns

Unlike proper nouns that introduce new individuals to the discourse context, pronouns refer to individuals already in the context. We call an individual that is in the discourse context, a context individual. For a context individual to serve as an antecedent of a pronoun, it is necessary that its relevant information (agreement and type) matches the corresponding information of the individual that the pronoun refers to. A pronoun needs to declare the requirements of a context individual, and this is done by a **dc_obj_get** predication that is coupled with an **instantiate** predication that contains the necessary information. For example, the semantics of the pronoun *it* is shown in (16).

(16)
$$\left[\begin{array}{ll} \textit{word} & \\ \text{ORTH} & \text{[it]} \\ \text{SYN} & \ldots \\ \text{SEM} & \left[\begin{array}{ll} \text{MODE} & \text{ref} \\ \text{TYPE} & \text{all}\frown\text{X-human} \\ \text{INDEX} & i \\ \text{RESTR} & \left\langle \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{dc\_obj\_get} \\ \text{SIT} & s \\ \text{OBJ} & i \\ \text{INSTPRED} & s_1 \end{array}\right], \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{instantiate} \\ \text{SIT} & s_1 \\ \text{SYN\_AGR} & \left[\begin{array}{ll} \textit{3sing} & \\ \text{GEND} & \textit{neut} \end{array}\right] \\ \text{SEM\_TYPE} & \text{all}\frown\text{X-human} \\ \text{INST} & i \end{array}\right] \right\rangle \end{array}\right] \end{array}\right]$$

The value of the INDEX feature is the same as the value of OBJ and INST features of **dc_obj_get** and **instantiate** respectively. This ensures that the pronoun refers to the same individual that is found in the context that matches the criteria provided by the **instantiate** predicate.

As another example, the feature structure description of the pronoun *he* is given in (17). Note that the RESTR feature contains an additional semantic restriction that the individual that the pronoun refers to must not be in conversation with the speaker at the time of utterance.

(17)
$$\left[\begin{array}{ll} \textit{word} & \\ \text{ORTH} & \text{[he]} \\ \text{SYN} & \ldots \\ \text{SEM} & \left[\begin{array}{ll} \text{MODE} & \text{ref} \\ \text{TYPE} & \text{male\_human}\frown\text{X} \\ \text{INDEX} & i \\ \text{RESTR} & \left\langle \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{dc\_obj\_get} \\ \text{SIT} & s \\ \text{OBJ} & i \\ \text{INSTPRED} & s_1 \end{array}\right], \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{instantiate} \\ \text{SIT} & s_1 \\ \text{SYN\_AGR} & \left[\begin{array}{ll} \textit{3sing} & \\ \text{GEND} & \textit{masc} \end{array}\right] \\ \text{SEM\_TYPE} & \text{male\_human}\frown\text{X} \\ \text{INST} & i \end{array}\right] \right\rangle \end{array}\right] \end{array}\right]$$

After the sentence is parsed, we should resolve the antecedents of pronouns. The semantic features we chose for the grammar enables to do the antecedent resolution by only referring to the semantic features of the parsed phrases. The necessary syntactic information are encoded as the SYN_AGR feature of the instantiate predication.

We can think of *antecedent resolution* as the resolution of **dc_obj_get** predications, or in other terms finding the context individuals that match their required criteria.

If multiple context individuals matching the required criteria can be found, then the corresponding sentence is ambiguous[9], and a context individual should be chosen as the antecedent of the pronoun manually by the user or with the help of a heuristic or probabilistic algorithm[10].

Some implicit requirements are in play in the process of antecedent resolution or generally in semantic analysis. One such requirement is that the semantics of a sentence (or a phrase) after antecedent resolution must be consistent with the relevant axioms of the application. Checking this consistency might be a hard task, and we might need some model builders or theorem provers to prove consistency or discover the inconsistency of the semantic analysis of a phrase respectively.

6.5 Semantic Principles and Constraints

Up to this point we have only discussed the semantics of single words. To be able to discuss the semantics of phrases we need to present two semantic principles that specify how the semantics of a phrase can be built from the semantics of its constituents. In the formulation of [67] there are two semantic principles, namely, semantic compositionality principle and semantic inheritance principle, given below.

(18) Semantic Compositionality Principle:
The value of the RESTR feature of any phrase structure $\mathcal{PS}$ is equal to the concatenation of the values of the RESTR features of the daughters of $\mathcal{PS}$.

(19) Semantic Inheritance Principle:
The values of MODE, INDEX, TYPE and any other semantic feature except RESTR for any phrase structure $\mathcal{PS}$ that is licensed by a headed rule are equal to the values of the corresponding features of the head daughter of $\mathcal{PS}$.

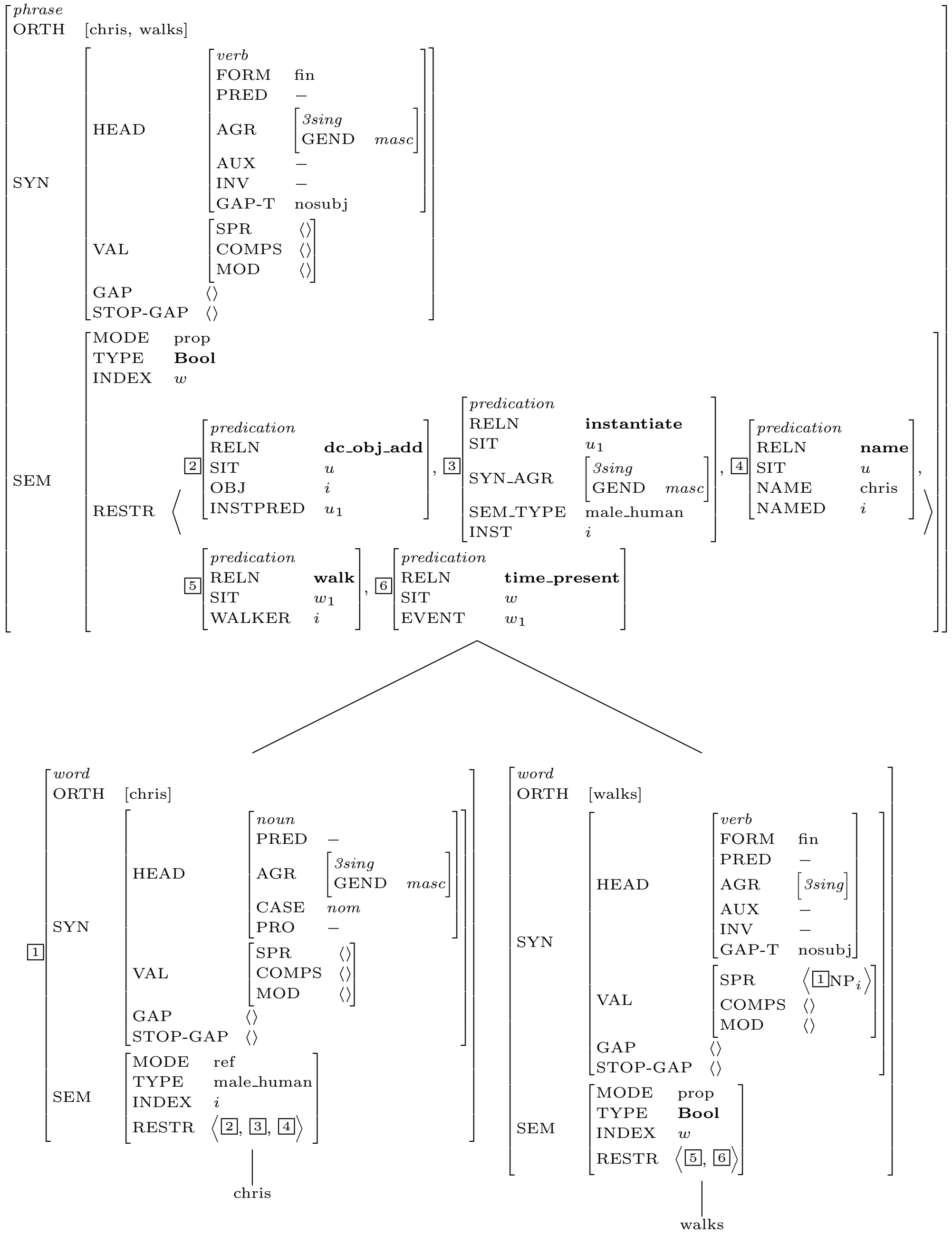
As an example of how these principles work we analyze the semantics of the simple sentence *Chris walks.* The phrase structure tree of this sentence is shown in (20).

As can be seen, the value of the MODE, INDEX, and TYPE features of the mother phrase are equal to those of the head daughter which in case is the word structure of the verb *walks*.

[9] Sentences can be ambiguous for other reasons too, but that is not the topic of this section.

[10] We do not study heuristic and probabilistic algorithms for anaphora resolution in this thesis.

(20)


phrase
ORTH [chris, walks]
SYN
HEAD
verb
FORM fin
PRED –
AGR 3sing GEND masc
AUX –
INV –
GAP-T nosubj
VAL
SPR ⟨⟩
COMPS ⟨⟩
MOD ⟨⟩
GAP ⟨⟩
STOP-GAP ⟨⟩
SEM
MODE prop
TYPE Bool
INDEX w
RESTR
2 predication RELN dc_obj_add SIT u OBJ i INSTPRED u1
3 predication RELN instantiate SIT u1 SYN_AGR 3sing GEND masc SEM_TYPE male_human INST i
4 predication RELN name SIT u NAME chris NAMED i
5 predication RELN walk SIT w1 WALKER i
6 predication RELN time_present SIT w EVENT w1
1
word
ORTH [chris]
SYN
HEAD
noun
PRED –
AGR 3sing GEND masc
CASE nom
PRO –
VAL
SPR ⟨⟩
COMPS ⟨⟩
MOD ⟨⟩
GAP ⟨⟩
STOP-GAP ⟨⟩
SEM
MODE ref
TYPE male_human
INDEX i
RESTR ⟨2, 3, 4⟩
chris
word
ORTH [walks]
SYN
HEAD
verb
FORM fin
PRED –
AGR 3sing
AUX –
INV –
GAP-T nosubj
VAL
SPR ⟨1 NPi⟩
COMPS ⟨⟩
MOD ⟨⟩
GAP ⟨⟩
STOP-GAP ⟨⟩
SEM
MODE prop
TYPE Bool
INDEX w
RESTR ⟨5, 6⟩
walks

Moreover, the value of the RESTR feature of the mother phrase is equal to the concatenation of daughters' RESTR value, as a result of the semantic compositionality principle. Thus, the RESTR value of the sentence *Chris walks* has two parts. The first part introduces the individual i who is named *Chris* in the situation u, and the second part asserts that the individual i walks at the present time in the situation w. Situations u, and w are not unified by the rules and principles we studied so far. However, we would like to somehow express that the two situations are the same. This can be done by some constraints that we present below.

6.5.1 Minimal Recursion Semantics General Constraints

The basis of the constraints we present in this section is [20], and [67]. Discourse analysis is discussed in neither [20] nor [67], so we use some constraints that are tailored to fit the presence of discourse predications.

As we have already seen, discourse predications introduce individuals with their respective *index* into the context of the discourse. To present the semantics of an utterance in a predicate calculus representation such as FOL, we need to covert the flat representation of the RESTR feature to a tree representation. This tree will correspond to a Predicate Calculus (PC)[11] formula. For this formula to be well-formed we need certain restrictions.

One important restriction is that an individual that is introduced by means of discourse predications will not be referred from a *scope* (formula subtree) where it is not defined. Conditions like this are called *variable binding conditions*. In this subsection we introduce the General Variable Binding Condition, and define the notion of a *scope resolved* list of predications. In our formulation, there are specific variable binding conditions for quantifiers and modifier predications that we introduce in sections 6.6.1 and 6.7.2 and 6.8.2.

The list of predications in the RESTR feature of expressions can be thought of as basic sub-formulas that will participate in the final PC formula representation of the expression's semantics. The situation values can be thought of as the labels of the sub-trees of the PC formula. In this respect, SIT values refer to scopes. All predications with the same SIT value are inside the same scope.

Definition 6.3 Scopal Features
A *scopal feature* is a feature other than SIT whose value is of type *situation*.

[11] Such as typed first order logic.

Definition 6.4 Immediate Outscope Relation:
A scope s immediately outscopes another scope u, denoted by $s <_i u$ if and only if u is the value of a scopal feature in a predication labeled by s. In this case we say s is the parent of u.

Definition 6.5 Outscope Relation:
A scope s outscopes a scope u, denoted by $s \leq u$, if and only if $s = u$ or there is a sequence $s_1, ..., s_n$, possibly empty such that we have:

$s <_i s_1 <_i ... <_i s_n <_i u$

We use the notation $s < u$ to indicate $s \leq u$ but $s \neq u$. If $s < u$ then s is an outer scope of u, and u is an inner scope of s.

(21) General Variable Binding Condition:[12]

Any index of an individual that is used in a predication with label s (and hence immediately inside scope s) must be bound by all discourse predications **dc_obj_add**, or **dc_obj_get** labeled t that introduce the individual i to the discourse context, such that $t \leq s$. In other words, the variable must be introduced by discourse predications immediately in the same scope or in an outer scope.

The reason that we have stated *all* discourse predications that introduce the individual i is that a **dc_obj_get** predication can retrieve an individual from the discourse context that was previously introduced to the discourse by another discourse predication.

Definition 6.6 Graph Representation of the Immediate Outscope Relation:
In the graph representation of the relation $<_i$ every scope is associated with a node. For all scopes s, u if $s <_i u$ there is a corresponding directed edge from s to u in the graph.

Definition 6.7 Scope-resolved Predications:
A list of predications is said to be scope-resolved if the values of SIT and scopal features of the predications are equated in such a way that every scopal feature is set to the label of some predication, and the graph representation of the immediate outscope relation $<_i$ forms a tree, with exactly one root, and all scopal constraints including variable binding conditions and the constraints introduced by outscope predications[13] are satisfied.

[12] We will introduce specific variable binding conditions in sections 6.6.1, 6.7.2, and 6.8.2.

[13] We see examples of these predications in section 6.7.2.

Example 6.8 Consider the predication list representation of the sentence *Chris walks* in (20), which is repeated in (22).

(22)

$$\left\langle \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{dc\_obj\_add} \\ \text{SIT} & u \\ \text{OBJ} & i \\ \text{INSTPRED} & u_1 \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{instantiate} \\ \text{SIT} & u_1 \\ \text{SYN\_AGR} & \begin{bmatrix} \textit{3sing} & \\ \text{GEND} & \textit{masc} \end{bmatrix} \\ \text{SEM\_TYPE} & \text{male\_human} \\ \text{INST} & i \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{name} \\ \text{SIT} & u \\ \text{NAME} & \text{chris} \\ \text{NAMED} & i \end{bmatrix}, \right.$$

$$\left. \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{walk} \\ \text{SIT} & w_1 \\ \text{WALKER} & i \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{time\_present} \\ \text{SIT} & w \\ \text{EVENT} & w_1 \end{bmatrix} \right\rangle$$

Since u_1 is used as the value of a scopal argument of the predication with label u then we have:

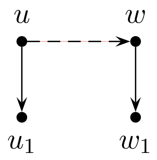
(23) $u <_i u_1$

Also, since i is the index of an individual used in scope w, but introduced by the discourse predication with label u by the variable binding condition we must have:

(24) $u \leq w$

The graph representation of the immediate outscope relation with this information is shown in (25). The solid edges are the immediate outscope pairs. The outscope constraint $u \leq w$ is shown by a dashed edge. The dashed edge is not part of the immediate outscope relation. In this status, the immediate outscope graph is not a single rooted tree.

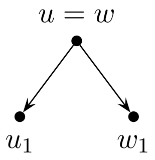
(25)



To make this graph a single rooted tree we must equate w to either u or u_1. However, in our definition of the discourse predication **dc_obj_add** we mentioned that the label of the **instantiate** predicate (u_1 in this case) cannot be shared by any predication (other than the quantifier predications). So the only option to make this graph complete is to equate u and w. This will yield a single rooted tree for scopes shown in (26) that satisfy all scopal constraints.

The resulting scope resolved list of predications is shown in (27).

(26)



(27)

$$\left\langle \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{dc\_obj\_add} \\ \text{SIT} & w \\ \text{OBJ} & i \\ \text{INSTPRED} & u_1 \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{instantiate} \\ \text{SIT} & u_1 \\ \text{SYN\_AGR} & \begin{bmatrix} \textit{3sing} & \\ \text{GEND} & \textit{masc} \end{bmatrix} \\ \text{SEM\_TYPE} & \text{male\_human} \\ \text{INST} & i \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{name} \\ \text{SIT} & w \\ \text{NAME} & \text{chris} \\ \text{NAMED} & i \end{bmatrix}, \right.$$
$$\left. \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{walk} \\ \text{SIT} & w_1 \\ \text{WALKER} & i \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{time\_present} \\ \text{SIT} & w \\ \text{EVENT} & w_1 \end{bmatrix} \right\rangle$$

As for a Predicate Calculus (PC) representation of the semantics, we should mention that since we use types, we need a version of Typed First Order Logic (TFOL) rather than First Order Logic. In this thesis, we use a version of TFOL that requires every individual to be associated with a type.

The **instantiate** predication translates to an *instance relation* formula, where the individual used as the value of INST feature is asserted to be of the type that is used as the value of SEM_TYPE feature. For example, the **instantiate** predication of (27) is translated to formula (28). Note that we used a subscript for ':' to represent the SIT label of the predication. We need the scopal information later to convert larger list of predications to TFOL.

(28) $i :_{u_1} \textit{male\_human}$

Other discourse predications do not make their way directly into the resulting FOL formula, however they have some side effects. **dc_obj_add** treats the index value of its OBJ feature as a unique identifier that is introduced by this discourse predication. On the other hand, **dc_obj_get** retrieves an identifier that was previously created for an individual matching the **instantiate** requirements that is coupled with it, and the *index* value of its OBJ feature is unified with the identifier just retrieved. Note that several identifiers matching the coupled **instantiate** requirement of **dc_obj_get** can be found, then the corresponding phrase is ambiguous. In such a case we need user's assistance to choose the right individual from the available individuals in the context, or we require her/him to paraphrase the sentence to avoid ambiguity.

Note that this does not mean that our theory is only capable of handling unambiguous sentences. The semantic representation for ambiguous sentences is perfectly fine. It is for the application of documenting and processing software and business requirements that we would like the user to be aware of the ambiguity and to choose the right antecedent or to rephrase the sentence to make the semantic representation and in turn the software or business requirements unambiguous.

Other predications will translate to simple atomic predicates that share their symbol with the predication's relation name (the value of RELN feature), and have a label argument that holds the SIT value of the predication and a list of other arguments that correspond to the arguments of the predication.

The predicate calculus representation of the predications that share their SIT value is equal to the conjunction of the PC representation of each of the predications. So the TFOL representation of the flat predications of this example becomes:

(29) $i :_{u_1} \mathit{male\_human}$, $\mathtt{name}(w, i, \mathit{chris}) \wedge \mathtt{time\_present}(w, \mathtt{walk}(w_1, i))$

Since we have two scope labels in the list of predications, we have ended up with two subformulas. One with the root label u_1, and the other with root label w. To resolve issues like this, we can unify scope labels α that are not used in any sub-formula as a non-label argument to their parents $\mathit{parent}(\alpha)$ in the immediate outscope relation tree. We perform the unification from top to bottom of the outscope tree. For the tree shown in (26) it means we should unify u_1 and w. So now we can conjoin the two formulas in (29) to derive the following single formula:

(30) $i :_{w} \mathit{male\_human} \wedge \mathtt{name}(w, i, \mathit{chris}) \wedge \mathtt{time\_present}(w, \mathtt{walk}(w_1, i))$

The meaning is that w is the situation where an individual i whose name is *chris* walks at the present time.

6.6 Semantics of Singular Countable Noun Phrases

A singular noun phrase with a count noun constituent is a combination of a *determiner* and a *count noun* in its singular form.[14] The general purpose of determiners is to help identify or quantify the individual that they refer to. In this respect they have a strong relation to the discourse context.

Quantifiers like *some, every, any, all* provide a means of quantification which is directly expressible in FOL. They also make the individual that they precede known to the context. They do so by adding the individual to the discourse context.

Articles *a/an* have a very similar semantics to quantifiers. In fact in this thesis we treat *a/an* as existential quantifiers. The noun that comes after these articles was previously unknown to the hearer, and these articles make them known. The article *the* on the other hand refers to an individual that is known to the hearer and hence must be already in the discourse context.

Possessives like *his, her, its, my* both retrieve an individual from the context (the owner) and add an individual (the object owned) to the context.

In all of the above cases of determiners we see that there is an intimate connection between determiners and the discourse predications **dc_obj_add**, and **dc_obj_get**. At least one of these predications will participate in the semantics of any singular countable noun phrase. For this similarity of determiners and other reasons we treat all determiners in the same category that we call *generalized quantifiers*. All generalized quantifiers act upon an individual that satisfies some restriction, and a scope that contains the semantics of the phrase that refers to the quantified individual.[15]

For this reason we have a subcategory of *sem-cat* for dealing specifically with the semantics of determiners (or generalized quantifiers). This category is *sem-det* and has two scopal features QRESTR and QSCOPE[16], as shown in figure 4.2, which we repeat in (31).

[14] In this thesis we refrain ourselves from discussing the semantics of mass nouns as this is a very vast topic which is beyond the scope of this thesis. For count nouns we focus on the semantics of singular nouns, which is the topic of this section.

[15] Some examples of *generalized quantifiers* in HPSG are *a/an*, *some*, *the*, *much*, *most*, *many*, ... Since the target of our grammar in this thesis is parsing software requirements, we avoid using intrinsically ambiguous determiners such as *much*, *most*, *many*. Later in this section we will study the analysis of *some*, and *the* as generalized quantifiers.

[16] The quantifier predications have two features with the same names that carry the same values.

(31) $\begin{bmatrix} \textit{sem-det} & \\ \text{QRESTR} & \textit{situation} \\ \text{QSCOPE} & \textit{situation} \end{bmatrix}$

6.6.1 Singular Quantification and Count Nouns

Simple Logical Quantifiers

We know from the ordinary First Order Logics (FOL) that there are two quantifiers, namely the existential quantifier and the universal quantifier. These are the first two generalized quantifiers that we study in this thesis.

In FOL, quantified formulas have one of the following forms:

(32) $\forall x\ P(x)$

(33) $\exists x\ P(x)$

where $P(x)$ is another FOL formula that refers to x. With this notation the quantified variable x is unrestricted unless the restrictions are applied within the formula $P(x)$. If the FOL language is expressive enough to allow the use of sets, variable x can be restricted at the beginning of the quantified formula using the following forms.

(34) $\forall x \in S;\ P(x)$

(35) $\exists x \in S;\ P(x)$

where S is the set that x ranges over. The above formulas can be rewritten in longer but equivalent forms shown below.

(36) $\forall x\ (x \in S \Rightarrow P(x))$

(37) $\exists x\ (x \in S \wedge P(x))$

In natural language the quantified individual is always restricted by the noun that follows it. The quantifier *every*, for example, is always followed by a count noun that serves as the restriction of the individual that is quantified, such as *person* in *every person*, or *book* in *every book*.

If we consider the quantified noun phrase *every book*, and if we suppose that the set of all books is denoted by *Books*, then the semantics of this noun phrase can be presented by the partial formula shown in (38). The box in the formula represents the scope of the quantifier, or simply the scope of the variable x.

(38) $\forall x \in Book; \underbrace{\boxed{\quad\quad\quad ... \quad\quad\quad}}_{scope \text{ of } x}$

A deeper contemplation reveals that every count noun in fact refers to a type. For example, the count noun *book* refers to the type *book*. Then, quantifiers can be thought to introduce an individual within the range of the instance set of some type. This type is carried by the semantics of the noun that follows the quantifier. For example the noun phrase *every book* can be converted to a partial semantic representation presented in (39), together with a **dc_obj_add** discourse predication that adds the individual to the discourse context. The **dc_obj_add** predication must be coupled with an **instantiate** predication to serve as the restriction of the quantifier. The **instantiate** predication must be carried by the semantics of the count noun that follows the quantifier.

(39) $\forall x \in I_{book}; \underbrace{\boxed{\quad\quad\quad ... \quad\quad\quad}}_{scope \text{ of } x}$

This formula is equivalent to:

(40) $\forall x : book; \underbrace{\boxed{\quad\quad\quad ... \quad\quad\quad}}_{scope \text{ of } x}$

With this analysis, the feature structure description of the natural language quantifier *every* is presented in (41)[17]. We have used a special predication **all** that contains the information about the quantified individual (which is the value of the BOUND_VAR feature), and the quantifier restriction (which is the scopal value of the QRESTR feature), and the quantifier's scope (which is the scopal value of the QSCOPE feature).

[17] In this thesis we do not use *handle constraints* or *outscope constraints* for the QRESTR feature of quantifier predications. These are needed to analyze quantified nouns with COMPS feature values containing another noun phrase. An example where these constraints are needed the analysis of the sentence *Every nephew of some famous politician runs.* A thorough discussion of these constraints is provided in [20].

(41)
$$\left[\begin{array}{ll}
\textit{word} & \\
\text{ORTH} & [\text{every}] \\
\text{SYN} & \left[\begin{array}{ll}
\text{HEAD} & \left[\begin{array}{ll} \textit{det} & \\ \text{AGR} & \left[\textit{3sing}\right] \\ \text{COUNT} & + \end{array}\right] \\
\text{VAL} & \left[\begin{array}{ll} \text{SPR} & \langle\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{array}\right] \\
\text{GAP} & \langle\rangle \\
\text{STOP-GAP} & \langle\rangle
\end{array}\right] \\
\text{SEM} & \left[\begin{array}{ll}
\text{MODE} & \text{none} \\
\text{INDEX} & i \\
\text{QRESTR} & r \\
\text{QSCOPE} & \textit{scope} \\
\text{RESTR} & \left\langle \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{all} \\ \text{SIT} & s \\ \text{BOUND\_VAR} & i \\ \text{QRESTR} & r \\ \text{QSCOPE} & \textit{scope} \end{array}\right], \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{dc\_obj\_add} \\ \text{SIT} & s \\ \text{OBJ} & i \\ \text{INSTPRED} & r \end{array}\right] \right\rangle
\end{array}\right]
\end{array}\right]$$

The TFOL equivalent sub-formula of the **all** predication has four arguments:

(42) `all`(s, i, r, scope)

This is equivalent to:

(43) $\forall_s\ i :_r \underbrace{\boxed{}}_{\textit{some type}} ; \underbrace{\boxed{}}_{\textit{scope}}$

The type that is missing in (43) should be provided by an **instantiate** predication that is part of the semantic restrictions of the count noun that follows the quantifier. We will shortly see how this combination is done in this section.

In the scope resolution we present in this thesis, there are specific constraints for the scopal features of a quantifier predication:

(44) Quantifier Restriction Constraint:
The SIT label of the **instantiate** predication that is used as the value of the QRESTR feature of a quantifier predication cannot be equated to any other *situation* value in the scope resolution process.

(45) Quantifier Scope Constraint:
The value of the QSCOPE feature of any quantifier predication cannot be identical to the value of any other scopal feature.

There is a special variable binding condition that is only related to quantifiers:

(46) Quantifier Variable Binding Condition:
If i is the index of a quantified individual with the quantifier predication whose QSCOPE feature value equals to s, then for any predication that references i with label t we must have $s \leq t$.

The analysis of existential quantifiers such as *a/an*, *some* are very similar. As an example, the feature structure description of the natural language quantifier *some* is provided in (47).

(47)
$$\left[\begin{array}{ll}
\textit{word} & \\
\text{ORTH} & [\text{some}] \\
\text{SYN} & \left[\begin{array}{ll}
\text{HEAD} & \left[\begin{array}{ll} \textit{det} & \\ \text{AGR} & \left[\textit{3sing}\right] \\ \text{COUNT} & + \end{array}\right] \\
\text{VAL} & \left[\begin{array}{ll} \text{SPR} & \langle\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{array}\right] \\
\text{GAP} & \langle\rangle \\
\text{STOP-GAP} & \langle\rangle
\end{array}\right] \\
\text{SEM} & \left[\begin{array}{ll}
\text{MODE} & \text{none} \\
\text{INDEX} & i \\
\text{QRESTR} & r \\
\text{QSCOPE} & \textit{scope} \\
\text{RESTR} & \left\langle \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{exists} \\ \text{SIT} & s \\ \text{BOUND\_VAR} & i \\ \text{QRESTR} & r \\ \text{QSCOPE} & \textit{scope} \end{array}\right], \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{dc\_obj\_add} \\ \text{SIT} & s \\ \text{OBJ} & i \\ \text{INSTPRED} & r \end{array}\right] \right\rangle
\end{array}\right]
\end{array}\right]$$

Semantics of *the*

The determiner *the* is treated as a generalized quantifier, which refers to an object from the discourse context. Unlike logical quantifiers *every*, and *exists* that add an object to the discourse context, the quantifier *the* has a semantics that retrieves an object from the discourse context. The feature structure of *the* is shown in (48).

(48)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & \text{[the]} \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \textit{det} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \\ \text{GAP} & \langle\rangle \\ \text{STOP-GAP} & \langle\rangle \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{none} \\ \text{TYPE} & \text{all} \\ \text{INDEX} & j_0 \\ \text{QRESTR} & r \\ \text{QSCOPE} & \textit{scope} \\ \text{RESTR} & \left\langle \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{dc\_obj\_get} \\ \text{SIT} & s_0 \\ \text{OBJ} & j_0 \\ \text{INSTPRED} & s_1 \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix}$$

When this quantifier is combined with a noun by HSR, an object compatible with the noun is retrieved from the discourse context by the **dc_obj_get** predication. The information needed to determine the compatibility is provided by the **instantiate** predication that must be included in the semantics of the noun. Next we will study the semantics of count nouns, and later we will see how this works with the quantifier *the*.

Semantics of Count Nouns

In this thesis we assume that every count noun serves as the restriction of the general quantifier that precedes it, by providing the type that the quantified individual ranges over its instance set. We do not study nouns that have a non-empty COMPS list.[18] With this assumption and restriction, the feature structure description of *book* is given in (49).

[18] The analysis of such nouns requires the use of *handle constraints*, or *outscope constraints*, where the QRESTR value of the determiner in SPR list ($restr$) is not identical to the SIT of the **instantiate** predication ($restr'$). Rather a constraint is placed that $restr \leq restr'$. A complete discussion is provided in [20].

(49)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{book}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{noun} & \\ \text{AGR} & \begin{bmatrix}\textit{3sing} & \\ \text{GEND} & \textit{neut}\end{bmatrix} \\ \text{PRO} & - \\ \text{TYPE\_DEF} & + \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \left\langle \begin{bmatrix} \textit{expression} & \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{det} & \\ \text{COUNT} & + \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix}\end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \textit{sem-det} & \\ \text{INDEX} & j \\ \text{QRESTR} & \textit{restr}\end{bmatrix}\end{bmatrix}\right\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \\ \text{GAP} & \langle\rangle \\ \text{STOP-GAP} & \langle\rangle \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{ref} \\ \text{TYPE} & \text{book} \\ \text{INDEX} & j \\ \text{RESTR} & \left\langle \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{instantiate} \\ \text{SIT} & \textit{restr} \\ \text{SYN\_AGR} & \textit{3sing} \\ \text{SEM\_TYPE} & \text{book} \\ \text{INST} & j \end{bmatrix}\right\rangle \end{bmatrix} \end{bmatrix}$$

This is the pattern we use for every count noun. In other words, the feature structure description of a count noun derived from the lexicon must be unifiable with the feature structure description we presented in (50).

(50)
$$\left[\begin{array}{ll}
\multicolumn{2}{l}{\textit{word}} \\
\text{SYN} & \left[\begin{array}{ll}
\text{HEAD} & \left[\begin{array}{ll} \multicolumn{2}{l}{\textit{noun}} \\ \text{AGR} & \textit{3sing} \\ \text{PRO} & - \\ \text{TYPE\_DEF} & + \end{array}\right] \\
\text{VAL} & \left[\begin{array}{ll}
\text{SPR} & \left\langle \left[\begin{array}{ll}
\multicolumn{2}{l}{\textit{expression}} \\
\text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \left[\begin{array}{ll} \multicolumn{2}{l}{\textit{det}} \\ \text{COUNT} & + \end{array}\right] \\ \text{VAL} & \left[\begin{array}{ll} \text{SPR} & \langle\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{array}\right] \end{array}\right] \\
\text{SEM} & \left[\begin{array}{ll} \multicolumn{2}{l}{\textit{sem-det}} \\ \text{INDEX} & j \\ \text{QRESTR} & \textit{restr} \end{array}\right]
\end{array}\right] \right\rangle \\
\text{COMPS} & \langle\rangle \\
\text{MOD} & \langle\rangle
\end{array}\right] \\
\text{GAP} & \langle\rangle \\
\text{STOP-GAP} & \langle\rangle
\end{array}\right] \\
\text{SEM} & \left[\begin{array}{ll}
\text{MODE} & \text{ref} \\
\text{TYPE} & \sigma \\
\text{INDEX} & j \\
\text{RESTR} & \left\langle \left[\begin{array}{ll} \multicolumn{2}{l}{\textit{predication}} \\ \text{RELN} & \textbf{instantiate} \\ \text{SIT} & \textit{restr} \\ \text{SYN\_AGR} & \textit{3sing} \\ \text{SEM\_TYPE} & \sigma \\ \text{INST} & j \end{array}\right] \right\rangle
\end{array}\right]
\end{array}\right]$$

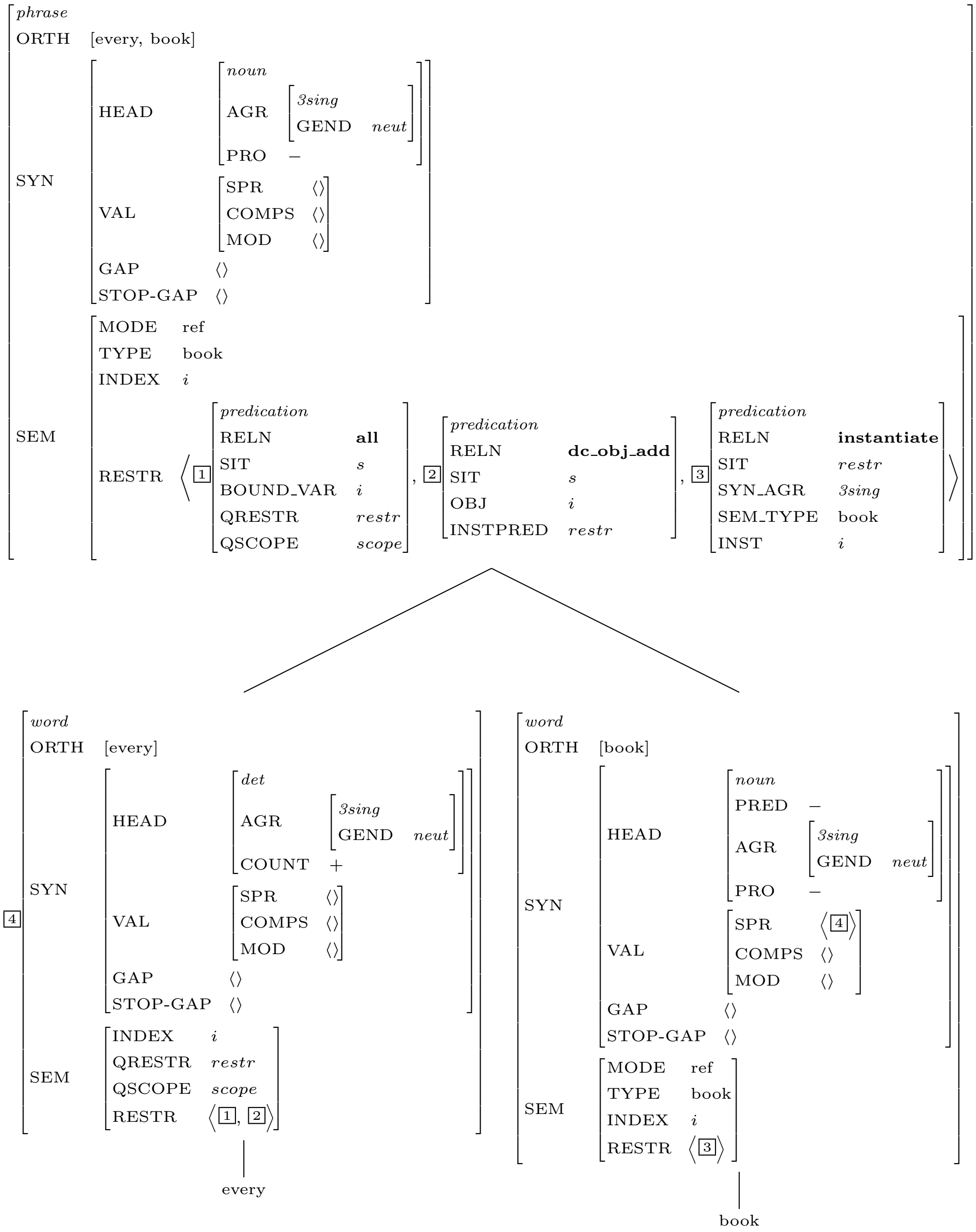
The first thing to discuss about is the TYPE_DEF feature that has the value +. This means that the count noun is actually carrying (or defining) a type. We use this feature in section 6.9.2 to form coordinated countable nouns that carry (or define) a composite type.

Note that the semantic INDEX of the determiner that is required in the SPR list is deliberately equated to the semantic INDEX of the count noun. Also the value of the QRESTR feature of the determiner is identical to the value of the SIT feature of the **instantiate** predicate used in the semantic restrictions of the count noun. These deliberate equated features enable the grammar to parse the noun phrase *every book* as shown in (51).

By the Semantic Compositionality Principle the value of RESTR of the mother phrase in (51) is the concatenation of the RESTR values of its daughters. Also by the Semantic Inheritance Principle the values of MODE, INDEX and TYPE of the mother phrase are identical to those of the head daughter (which is *book* in this case).

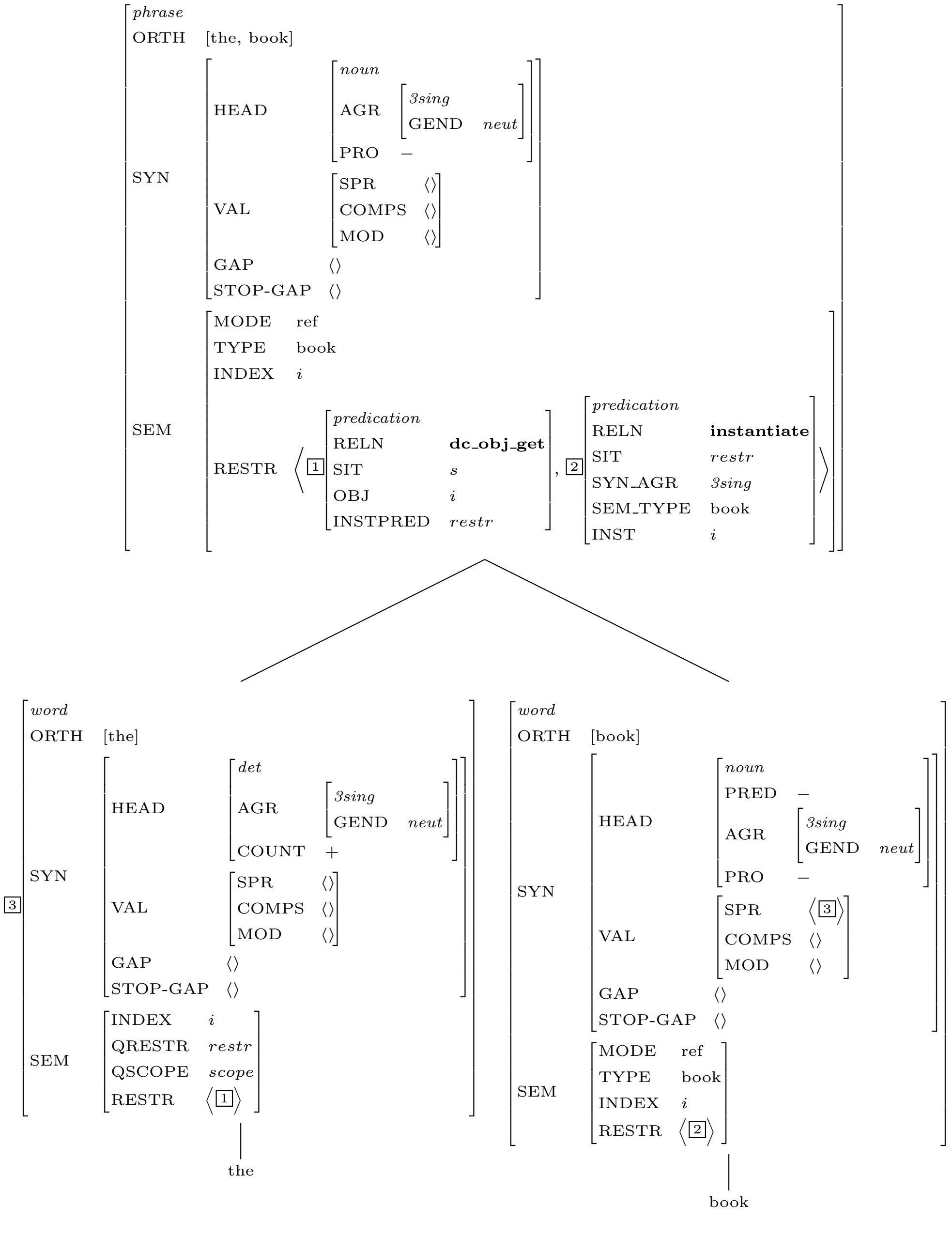
Note that the **instantiate** simultaneously plays two roles. In the first role it is used as the restriction of the quantifier in the noun phrase (only SEM_TYPE and INST features are used). In the second role it is coupled with the discourse predication **dc_obj_add** and provides information about the new individual that is added to the discourse context.

(51)



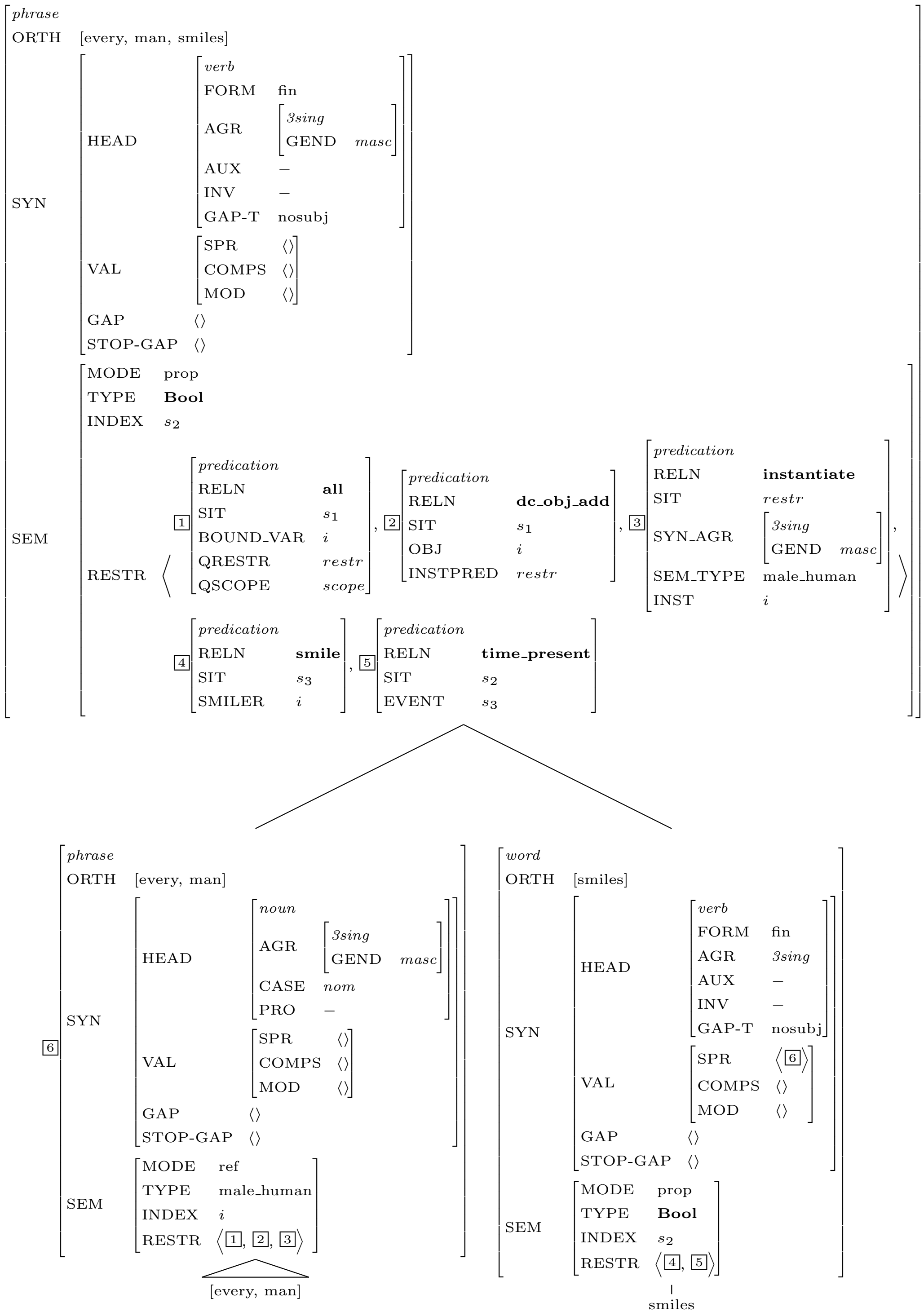
As another example we will consider the phrase *the book*, which is provided in the phrase structure tree (52). Note that the **dc_obj_get** predication from the semantics of *the* is automatically paired with the **instantiate** predication of the count noun. The **instantiate** predication contains the syntactic agreement and the semantic type of the object that should be retrieved from the discourse context by the **dc_obj_get** predication.

(52)



(53) Every man smiles.

(54)



As an example of a complete sentence that uses quantification consider the sentence (53) with the corresponding phrase structure tree shown in (54). The phrase is licensed by the Head Specifier Rule, and as can bee seen by the Semantic Compositionality Principle mother's RESTR list is identical to the concatenation of each daughter's RESTR list.

The scope of the quantifier, which is the value of QSCOPE feature of the **all** predication is left underspecified. This is deliberately always the case for any quantifier in the MRS representation. The reason is that if multiple quantifiers are present in a sentence, quantifiers can be introduced in the semantic representation in different orders, as pointed out by Montague [55] in the example *a woman loves every man*. However, the Quantifier Variable Binding Condition applies on the value of this feature.

With the scope of the quantifiers underspecified, it is left for the *scope resolution* algorithm to resolve the scopes of the quantifiers (and other scopes). When there are more than one quantifier there will be more than one scope-resolved MRS representation of the sentence.

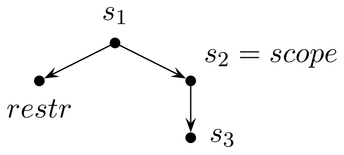
The scope resolution algorithm must equate scopes such that the immediate outscope relation graph becomes a tree with only one root (see definition 6.7).

In the semantic restriction list of mother phrase in (54) corresponding to the sentence (53), the only scopes without parents (see definition 6.4) are s_1, and s_2. Moreover *scope* must be equated to the label of some predication.

Candidates for the value of *scope* are s_1, s_2, *restr*, and s_3. According to the Tense Predication Constraint s_3 cannot be shared by other predications. If *scope* is set to s_1 then s_1 immediately outscopes itself and the graph representation will contain a loop, and hence it cannot be a tree. On the other hand, *resrt* is the value of the QRESTR feature of the quantification predication **all** predication, and by the Quantifier Restriction Constraint, it can only be shared by the SIT value of the instantiate predication combining with the quantifier.

Thus, the only possible value of *scope* is s_2. With the identity *scope* $= s_2$, the graph representation of the immediate outscope relation for this example is given in (55).

(55)



The individual i introduced by the **dc_obj_add** predication with scope s_1 is used as an argument of the **smile** predication with scope s_2. So by the variable binding condition we must have:

(56) $s_1 \leq s_2$

Also, since i is the index of a quantified individual with a quantifier with QSCOPE value being *scope*, and i is referenced from a predication with label s_2 then by the Quantifier Variable Binding Condition we must have:

(57) $scope \leq s_2$

Both of these conditions are satisfied for the tree shown in (55). The TFOL translation of the scope-resolved representation above is:

(58) $\forall_{s_1} i :_{restr} \ male\_human \ ; \ \texttt{time\_present}(s_2, \texttt{smile}(s_3, i))$

Note that we have used s_1 as the subscript for the quantifier to label its situation. We used *restr* as a subscript of the instance relation to label its scope too.

The meaning of this formula is that s_1 is the situation where all men participated in the event s_2 by smiling and that the event s_2 is at the present time.[19]

It is however unusual to have situations in a TFOL formula. It might be desirable to convert the above formula to:

(59) $\forall i : \ male\_human \ ; \ \texttt{time\_present}(\texttt{smile}(i))$

or even a simpler and more familiar TFOL formula if we do not need the tense information:

(60) $\forall i : male\_human; \texttt{smile}(i)$

In section 6.12 we describe a general procedure to convert a list of scope-resolved predications to TFOL formulas with options to remove the situation arguments and remove the tense predicates.

[19] As mentioned at the beginning of this chapter situations can simply be thought of as being references to events.

6.6.2 Semantic Analysis of Singular Possessives

Possessives like *his, her, its*, ... in English are treated as determiners. As we discussed in chapter 5 the possessive *'s* can also be treated as a determiner (in fact a determiner phrase). In this section we study the semantics of these possessives. However for simplicity we assume that the owner or the owned object is a singular noun or noun phrase. Also for brevity we do not discuss the semantics of possessive pronouns like *mine*, *yours*, ... Although these analyses are very similar and very simple indeed.

(61)
$$\left[\begin{array}{ll}
\multicolumn{2}{l}{\textit{word}} \\
\text{ORTH} & \text{[his]} \\
\text{SYN} & \left[\begin{array}{ll}
\text{HEAD} & \left[\begin{array}{ll} \multicolumn{2}{l}{\textit{det}} \\ \text{AGR} & \textit{3sing} \\ \text{COUNT} & + \end{array}\right] \\
\text{VAL} & \left[\begin{array}{ll} \text{SPR} & \langle\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{array}\right] \\
\text{GAP} & \langle\rangle \\
\text{STOP-GAP} & \langle\rangle
\end{array}\right] \\
\text{SEM} & \left[\begin{array}{ll}
\text{MODE} & \text{none} \\
\text{TYPE} & \textit{all} \\
\text{INDEX} & i \\
\text{QRESTR} & \textit{restr} \\
\text{QSCOPE} & \textit{scope} \\
\text{RESTR} & \left\langle \begin{array}{l}
\left[\begin{array}{ll} \multicolumn{2}{l}{\textit{predication}} \\ \text{RELN} & \textbf{dc\_obj\_get} \\ \text{SIT} & s_1 \\ \text{OBJ} & \textit{owner} \\ \text{INSTPRED} & \textit{rowner} \end{array}\right],
\left[\begin{array}{ll} \multicolumn{2}{l}{\textit{predication}} \\ \text{RELN} & \textbf{instantiate} \\ \text{SIT} & \textit{rowner} \\ \text{SYN\_AGR} & \left[\begin{array}{ll} \multicolumn{2}{l}{\textit{3sing}} \\ \text{GEND} & \textit{masc} \end{array}\right] \\ \text{SEM\_TYPE} & \text{male\_human} \\ \text{INST} & \textit{owner} \end{array}\right],
\left[\begin{array}{ll} \multicolumn{2}{l}{\textit{predication}} \\ \text{RELN} & \textbf{exists} \\ \text{SIT} & s_1 \\ \text{BOUND\_VAR} & i \\ \text{QRESTR} & \textit{restr} \\ \text{QSCOPE} & \textit{scope} \end{array}\right], \\
\left[\begin{array}{ll} \multicolumn{2}{l}{\textit{predication}} \\ \text{RELN} & \textbf{dc\_obj\_add} \\ \text{SIT} & s_1 \\ \text{OBJ} & i \\ \text{INSTPRED} & \textit{restr} \end{array}\right],
\left[\begin{array}{ll} \multicolumn{2}{l}{\textit{predication}} \\ \text{RELN} & \textbf{poss} \\ \text{SIT} & \textit{scope} \\ \text{POSSESSOR} & \textit{owner} \\ \text{POSSESSED} & i \end{array}\right]
\end{array} \right\rangle
\end{array}\right]
\end{array}\right]$$

Singular Possessive Pronouns

In what follows we analyze the semantics of the possessive pronoun *his*. Analyses of other possessive pronouns like *her*, *its* are very similar.

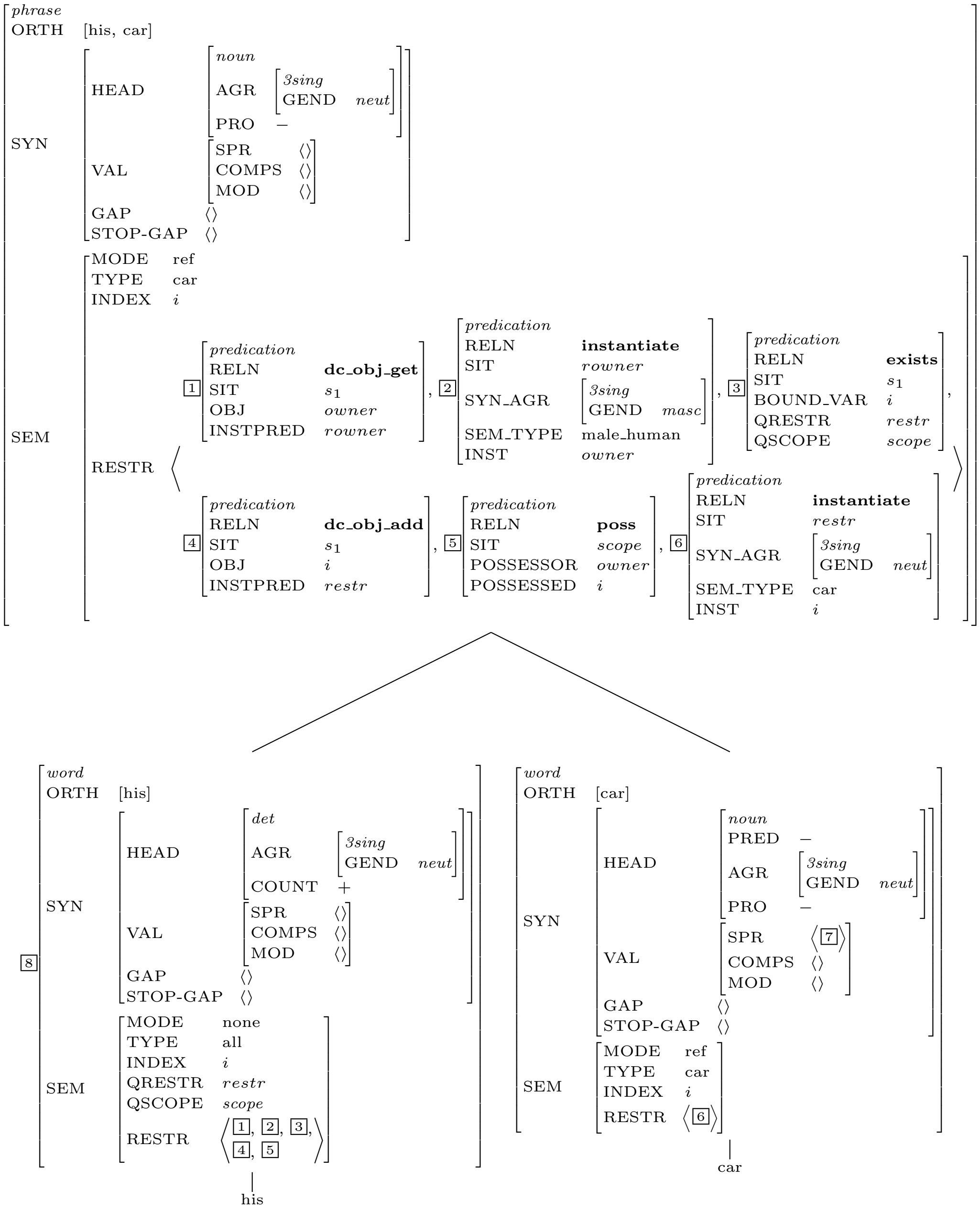
his acts as a determiner that refers to two individuals. The first individual has the owner role and should be a male individual present in the discourse context. The second individual is the owned object which is introduced by this determiner and is existentially quantified and added to the discourse context. The feature structure description of *his* is shown in (61).

As can be seen the semantic TYPE of the possessive is set to *all*, since we assumed for all categories other than verbs and nouns the value of this feature is *all*, and the semantic types of such expressions are not used in the thesis (other than trivially as we see later in our explanation of guards).

The first predication is a **dc_obj_get** that is used to retrieve the male human with the owner role from the discourse context. It is paired with the second predication, which is an **instantiate** predication that provides the agreement and semantic type information of the owner. The third predication is an existential quantification over the owned object. The fourth predication in the semantics of *his* is another discourse predication that adds the individual that is owned to the discourse context. Note that the *restr* value of this discourse predication is the same as the restriction value of the quantifier, and is also identical to the QRESTR feature of the determiner. The value of this feature will be unified with the SIT value of the **instantiate** predication of the count noun that follows the possessive pronoun. This is achieved as a result of the common feature structure description of count nouns provided in (50). The fifth predication is the **poss** predication which expresses that something belongs to someone or another thing. It has two non-scopal arguments, first the the POSSESSOR, which is the owner, and second is the POSSESSED which is the object that is owned. The individual owned is referred by index i that is identical to the semantic index of the determiner. When combined by HSR with a noun, this index will be unified with the semantic INDEX of the noun. The reason for this is the way we define count nouns as shown in (50). So the noun will play the role of the individual being owned. This predication is within the scope of the existential quantifier, because its SIT value is identical to the QSCOPE value of the quantifier.

This possessive pronoun can be combined with a count noun such as *car* by the Head Specifier Rule, which produces the phrase structure tree shown in (62). The TFOL translation of the semantics of this phrase is given in (63).

(62)


phrase
ORTH [his, car]
SYN
HEAD noun
AGR 3sing GEND neut
PRO −
VAL SPR ⟨⟩ COMPS ⟨⟩ MOD ⟨⟩
GAP ⟨⟩
STOP-GAP ⟨⟩
SEM
MODE ref
TYPE car
INDEX i
RESTR
1 predication RELN dc_obj_get SIT s1 OBJ owner INSTPRED rowner
2 predication RELN instantiate SIT rowner SYN_AGR 3sing GEND masc SEM_TYPE male_human INST owner
3 predication RELN exists SIT s1 BOUND_VAR i QRESTR restr QSCOPE scope
4 predication RELN dc_obj_add SIT s1 OBJ i INSTPRED restr
5 predication RELN poss SIT scope POSSESSOR owner POSSESSED i
6 predication RELN instantiate SIT restr SYN_AGR 3sing GEND neut SEM_TYPE car INST i
8 word
ORTH [his]
SYN
HEAD det
AGR 3sing GEND neut
COUNT +
VAL SPR ⟨⟩ COMPS ⟨⟩ MOD ⟨⟩
GAP ⟨⟩
STOP-GAP ⟨⟩
SEM
MODE none
TYPE all
INDEX i
QRESTR restr
QSCOPE scope
RESTR ⟨1, 2, 3, 4, 5⟩
his
word
ORTH [car]
SYN
HEAD noun
PRED −
AGR 3sing GEND neut
PRO −
VAL SPR ⟨7⟩ COMPS ⟨⟩ MOD ⟨⟩
GAP ⟨⟩
STOP-GAP ⟨⟩
SEM
MODE ref
TYPE car
INDEX i
RESTR ⟨6⟩
car

(63) $owner :_{s_1} male\_human \ \wedge \ \exists_{s_1} \ i :_{restr} car \ ; \ \texttt{poss}(scope, owner, i)$

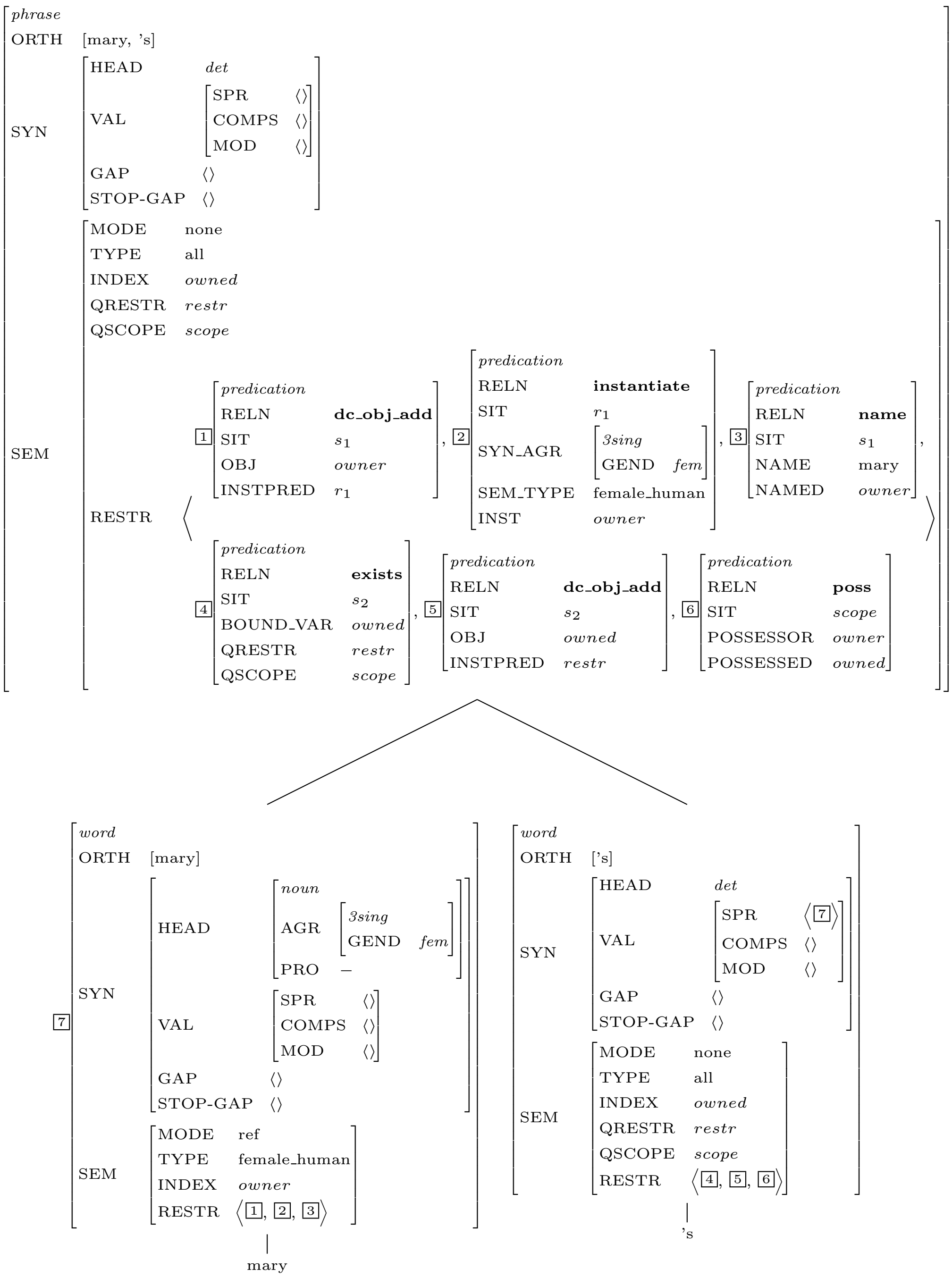
The Possessive *'s*

The syntactic analysis of the possessive *'s* was presented in section 5.2.1. The analysis is very similar to the analysis of the possessive pronouns except that the individual that takes the owner role is underspecified. The feature structure description of *'s* is shown in (64). Note that the NP in the SPR list has an *owner* subscript. This means that the semantic INDEX of this NP is *owner*. This index is used again in the **poss** predication as the value of the feature POSSESSOR. This is an example of semantic *role assignment*.

(64)
$$\left[\begin{array}{ll}
\textit{word} & \\
\text{ORTH} & \text{['s]} \\
\text{SYN} & \left[\begin{array}{ll}
\text{HEAD} & \textit{det} \\
\text{VAL} & \left[\begin{array}{ll}
\text{SPR} & \left\langle \text{NP}_{owner}\left[\begin{array}{ll}\text{AGR} & \textit{3sing} \\ \text{PRO} & -\end{array}\right]\right\rangle \\
\text{COMPS} & \langle\rangle \\
\text{MOD} & \langle\rangle
\end{array}\right] \\
\text{GAP} & \langle\rangle \\
\text{STOP-GAP} & \langle\rangle
\end{array}\right] \\
\text{SEM} & \left[\begin{array}{ll}
\text{MODE} & \text{none} \\
\text{TYPE} & \text{all} \\
\text{INDEX} & \textit{owned} \\
\text{QRESTR} & \textit{restr} \\
\text{QSCOPE} & \textit{scope} \\
\text{RESTR} & \left\langle \left[\begin{array}{ll}
\textit{predication} & \\
\text{RELN} & \textbf{exists} \\
\text{SIT} & \textit{s} \\
\text{BOUND\_VAR} & \textit{owned} \\
\text{QRESTR} & \textit{restr} \\
\text{QSCOPE} & \textit{scope}
\end{array}\right], \left[\begin{array}{ll}
\textit{predication} & \\
\text{RELN} & \textbf{dc\_obj\_add} \\
\text{SIT} & \textit{s} \\
\text{OBJ} & \textit{owned} \\
\text{INSTPRED} & \textit{restr}
\end{array}\right], \left[\begin{array}{ll}
\textit{predication} & \\
\text{RELN} & \textbf{poss} \\
\text{SIT} & \textit{scope} \\
\text{POSSESSOR} & \textit{owner} \\
\text{POSSESSED} & \textit{owned}
\end{array}\right] \right\rangle
\end{array}\right]
\end{array}\right]$$

The possessive *'s* can be combined with a noun phrase such as *Mary* by the Head Specifier Rule. The phrase structure tree of the phrase *Mary's* is shown in (65). Note that by the Semantic Inheritance Principle the values of the features MODE, TYPE, INDEX, QRESTR, and QSCOPE of the mother phrase are identical to those of the head daughter. This ensures that the determiner phrase formed by this rule also has the determiner specific features QRESTR, and QSCOPE with appropriate values.

(65)



After combination with the owner NP, this determiner phrase acts very similar to a usual possessive determiner like *his* that we already discussed. The semantic INDEX of the owned individual is set to the semantic INDEX of the determiner, and by the way we defined count nouns in (50) this INDEX is unified with the semantic INDEX of the noun.

This way the individual that is referred by the noun, which follows the possessive, takes the role of the owned individual.

6.7 Semantic Analysis of Modification

Modifiers basically add an extra predication to the semantics of the phrase they participate in. This predication has at least an argument that takes the index of the *individual* or the *situation* that is being modified. The relation name of the predication specifies what the modification is. Extra arguments might be present that relate the modified individual or situation to other individuals or situations. We call this predication the *modifier predication*.

As seen in our first encounter with modification in chapter 5, section 5.2.5, modifiers fall into these groups: adjectives, adverbs, prepositional modifiers and subordinate conjunctions. In the subsections that follow we study each of the first three briefly. Subordinate conjunctions are discussed separately in the next section.

6.7.1 Adjectives

An adjective syntactically modifies a nominal, what we denoted by NOM in chapter 5, and semantically modifies an *individual*. As an example let's consider the adjective *red*. The feature structure description of this adjective was presented in chapter 5 in (61). We show the feature structure description of *red* containing the semantic features in (66).

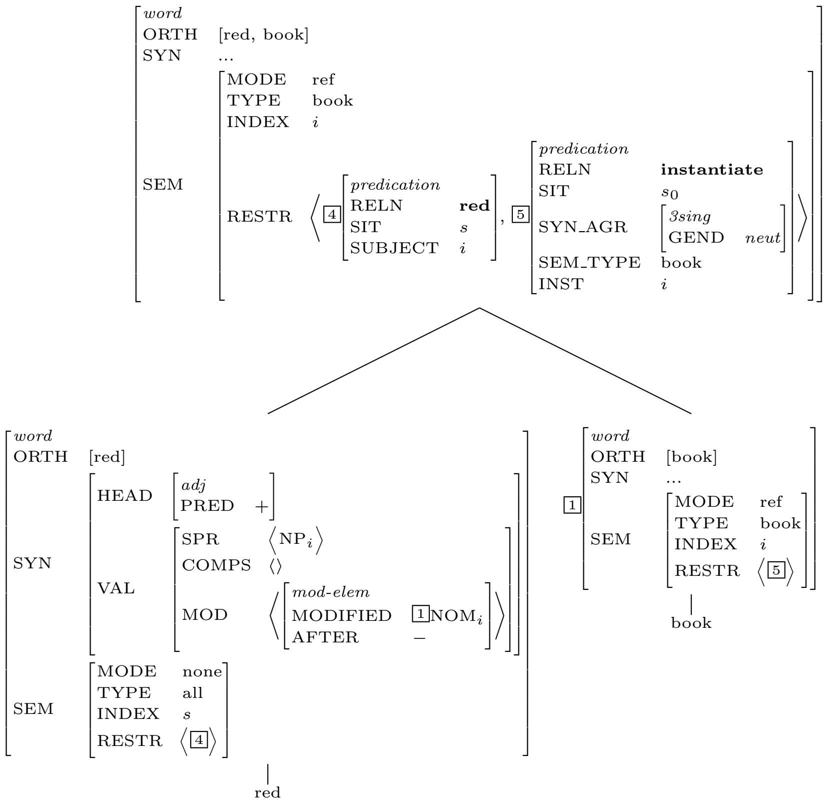
Note that the NOM in the MOD list is subscripted with i, and i is used again in the predication **red** as the value of the SUBJECT feature. By this, we have assigned the role of the modified individual to the individual that the NOM in the MOD list refers to. Again this is an example of *semantic role assignment*. We have done the same co-indexation for the NP in the SPR list. The reason will be discussed in section 6.10.

This adjective can be combined with a noun like *book* by the Head Modifier Rule. The phrase structure tree of the resulting phrase is shown in (67).

The semantics of this phrase is still incomplete, as the **instantiate** predication is not yet paired with a discourse predication. For this pairing to happen, this nominal phrase needs to be combined with a determiner. Without the determiner, the semantic restrictions of the phrase *red book* does not have enough information to be translated to TFOL. However the partial formulas that will participate in the complete TFOL formula are:

(66)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{red}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix}\textit{adj} & \\ \text{PRED} & +\end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \left\langle \text{NP}_i \right\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \left\langle \begin{bmatrix} \textit{mod-elem} & \\ \text{MODIFIED} & \text{NOM}_i \\ \text{AFTER} & - \end{bmatrix}\right\rangle \end{bmatrix}\end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{none} \\ \text{TYPE} & \textit{all} \\ \text{INDEX} & s \\ \text{RESTR} & \left\langle \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{red} \\ \text{SIT} & s \\ \text{SUBJECT} & i\end{bmatrix}\right\rangle \end{bmatrix}\end{bmatrix}$$

(67)



(68) a. $i :_{s_0} \textit{book}$

b. $\texttt{red}(s, i)$

We do not discuss in detail how this can be combined with a determiner, because the analysis is the same as the analysis of singular count nouns and quantification that we already discussed in section 6.6. The only difference is the presence of the additional modifier predication **red**.

We will finish the semantic analysis of adjectives in section 6.10, where we discuss how the meaning of copular verb phrases followed by adjectives is analyzed.

6.7.2 Adverbs

Adverbs in our formulation have two predications. The first one is the modification predication that takes a scopal argument, that specifies the scope that the adverb modifies. The second predication is a special **outscope** predication used to provide a structural constrain on the scopes.[20].

(69)
$$
\left[\begin{array}{ll}
\textit{word} & \\
\text{ORTH} & [\text{slowly}] \\
\text{SYN} & \left[\begin{array}{ll}
\text{HEAD} & \left[\begin{array}{ll}\textit{adv} & \\ \text{PRED} & - \end{array}\right] \\
\text{VAL} & \left[\begin{array}{ll}
\text{SPR} & \langle\rangle \\
\text{COMPS} & \langle\rangle \\
\text{MOD} & \left\langle \left[\begin{array}{ll}\textit{mod-elem} & \\ \text{MODIFIED} & \text{S}_{t_0}\end{array}\right]\right\rangle
\end{array}\right]
\end{array}\right] \\
\text{SEM} & \left[\begin{array}{ll}
\text{MODE} & \text{none} \\
\text{TYPE} & \text{all} \\
\text{INDEX} & s \\
\text{RESTR} & \left\langle \left[\begin{array}{ll}\textit{predication} & \\ \text{RELN} & \textbf{slowly} \\ \text{SIT} & s \\ \text{EVENT} & t\end{array}\right], \left[\begin{array}{ll}\textit{predication} & \\ \text{RELN} & \textbf{outscope} \\ \text{OUTER} & t \\ \text{INNER} & t_0\end{array}\right], \right\rangle
\end{array}\right]
\end{array}\right]
$$

outscope predications do not have a SIT value, as they do not participate directly in the final semantic representation of the phrase. They just provide structural constraints. These structural constraints need to be satisfied for any scope-resolved list of predications.

[20] In [20, 19] *qeq* constraints are used instead of outscope constraints, although it is hypothesized in [20] that in a grammar which perfectly obeyed the constraints on composition, it should be unnecessary to use *qeq* conditions rather than simple outscopes. Also in [19, 20] these constraints are maintained in a different HCONS feature. Here we treat these constraints as special kind of predications that are stored in the RESTR feature together with the rest of predications.

The feature structure description of the adverb *slowly*, as an example, with semantic features is given in (69). The modifier predication **slowly** takes a scopal argument for its EVENT feature.

As can be seen, this value is co-indexed with the value of the OUTER feature of the **outscope** predication, and the value of the INNER feature is co-indexed with the semantic INDEX of the sentence in the MOD list. So the modification predication asserts that the situation that is expressed by the sentence has happened or is happening the way that the modifier predication implies, which in this case is *slowly*.[21]

Definition 6.9 Scopal Predications:
A *scopal predication* is a predication that takes arguments of type *situation* that are or must be equated to the label of other predications. Basically a scopal predication is a predication with scopal features.

In this thesis we introduce a special group of scopal predications, narrow scopal predications, which we use for adverbs and prepositional modifiers that modify a verb. The modification predication of an adverb is a narrow scopal predication that is bound to the following variable binding condition:

(70) Narrow Scope Variable Binding Condition:
If an individual index i is referred from a scope s which is the label of a narrow scopal predication , then i must be bound by a discourse predication **dc_obj_add**, or **dc_obj_get** labeled w, such that $w \leq s$.

The above definition uses the notion of the *reference* of an individual from a predication. Although this can be understood by intuition we provide a formal definition below.

Definition 6.10 Reference to Individuals from Scopes:
For an individual index i:

- If i is used as the feature value of a predication $\mathcal{P}$ with label s then i is referred from the scope s.

[21] The reason that an **outscope** is used rather than just unifying t with t_0 is that there can be several adverbs in a sentence, and by the HFP, the sentence licensed by the HMR has the same semantic INDEX as its verb, and without the **outscope** predication we would have two modification predications that have the same EVENT feature values, which avoids the outscope graph to be resolved to a rooted tree.

- If i is referred from the scope s, which is the value of a scopal feature of a narrow scopal predication $\mathcal{P}$ with label t, then i is referred from the scope t.
- If i is referred from the s, which is the value of the INNER feature of an **outscope** predication with OUTER feature value t, then i is referred from the scope t.
- If the OBJ value i of a **dc_obj_get** predication is resolved to an individual in the discourse context with index j, then $i = j$ effectively, and a scope s refers to i if and only if S refers to j.

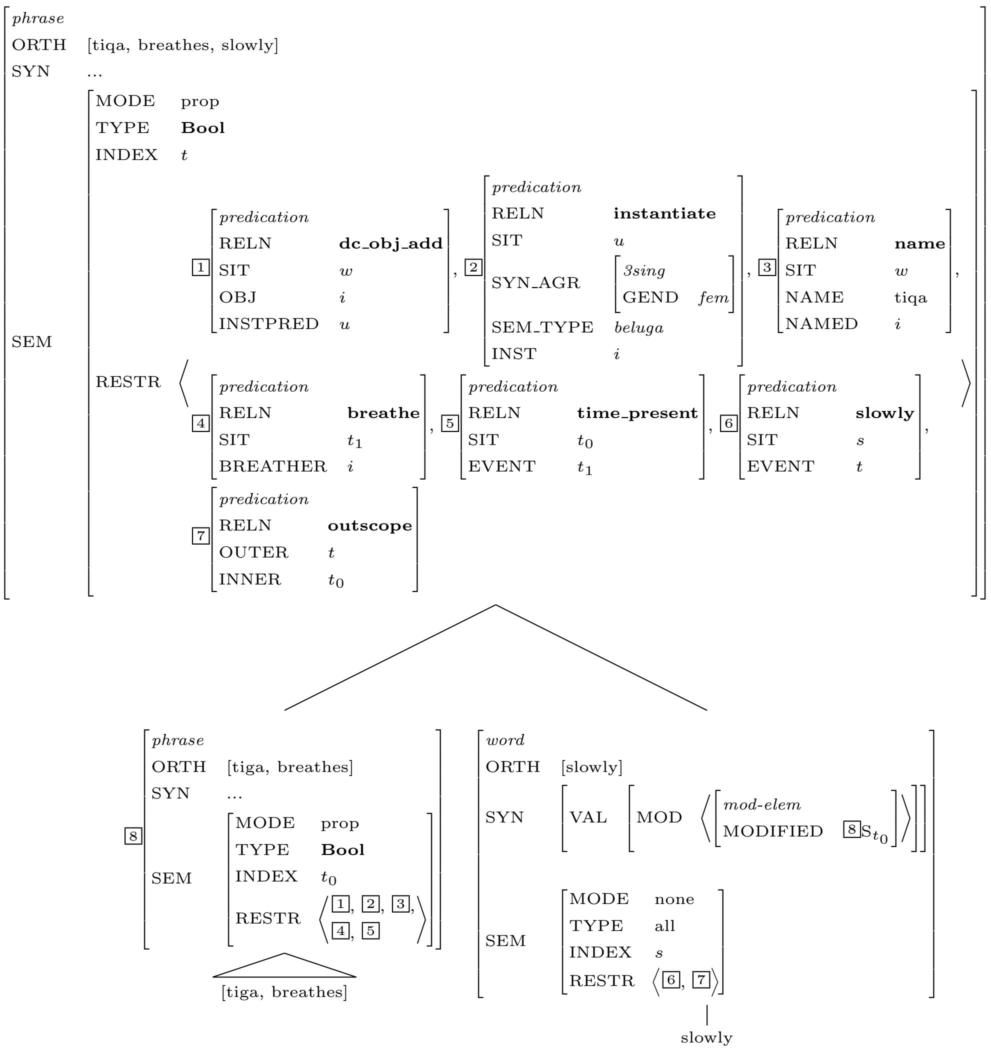
The last item above indicates that for calculating individual references it is first necessary to resolve the references made by **dc_obj_get** predication to the discourse context. The reason is simply that we have used **dc_obj_get** for pronouns, and the article *the* that acts like a pronoun after being combined with a noun. Both of these structures have an *antecedent*, i.e., another individual in the context that is semantically the same as the individuals that these structures represent. So any reference made to an individual that a pronoun or a noun phrase starting with *the* represents is as well a reference to an individual that its antecedent represents.

We will now show why the new Narrow Scope Variable Binding Condition is necessary to analyze a sentence such as:

(71) Tiqa breathes slowly

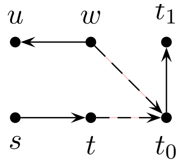
This sentence is licensed by the post Head Modifier Rule in the phrase structure tree shown in (72).

(72)


phrase
ORTH [tiqa, breathes, slowly]
SYN ...
SEM
MODE prop
TYPE Bool
INDEX t
RESTR
1 predication RELN dc_obj_add SIT w OBJ i INSTPRED u
2 predication RELN instantiate SIT u SYN_AGR 3sing GEND fem SEM_TYPE beluga INST i
3 predication RELN name SIT w NAME tiqa NAMED i
4 predication RELN breathe SIT t1 BREATHER i
5 predication RELN time_present SIT t0 EVENT t1
6 predication RELN slowly SIT s EVENT t
7 predication RELN outscope OUTER t INNER t0
8
phrase
ORTH [tiga, breathes]
SYN ...
SEM
MODE prop
TYPE Bool
INDEX t0
RESTR 1, 2, 3, 4, 5
[tiga, breathes]
word
ORTH [slowly]
SYN VAL MOD mod-elem MODIFIED 8 S t0
SEM
MODE none
TYPE all
INDEX s
RESTR 6, 7
slowly

Without the narrow scope variable binding condition the graph representation of the outscope relation is shown in (73). The solid edges represent immediate outscope relations, whereas dotted edges represent outscope relations (not necessarily immediate).

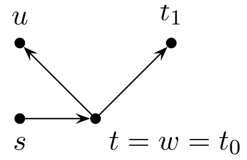
(73)



To satisfy a constraint expressed by a dotted edge, we need to equate scopes in such a way that the dotted edge falls on a path (possibly a trivial path of length zero) of solid edges with the same direction. In other words, there must be a path of solid edges starting from the start node of the dotted edge and ending with the end node of the dotted edge.

With these constraints only, a possible scope resolution could result by equating t, and w, resulting in the single rooted tree of (74).

(74)



Then the resulting TFOL sub-formulas will be[22]:

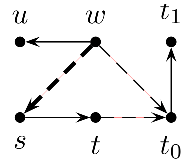
(75) $i :_u$ *beluga* , $\texttt{name}(t, i, tiqa) \ \wedge\ \texttt{time\_present}(t, \texttt{breathe}(t_1, i)) \ \wedge\ \texttt{slowly}(s, t)$

But the meaning of this formula is that not only the event of breathing at the current time, but also the event of naming happens slowly. Widening the scope of the **slowly** predication to include the **name** predication is not meaningful and not desirable.

This is where the Narrow Scope Variable Binding Condition comes into play. Note that i is an index that is referred from the narrow scopal predication **slowly**. So by the Narrow Scope Variable Binding Condition i must be introduced by a discourse predication (in this case **dc_obj_add**) with label w such that $w \leq s$.

We denote this new constraint by the thicker dashed edge in the graph representation of the outscope relation, shown in (76).

(76)

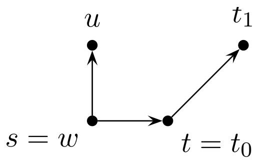


With this new constraint it is impossible to equate w and t, while keeping the graph a tree. So the unwanted reading of (75) cannot be generated.

The only possible scope-resolved tree is resulted by equating s, and w, which we have shown in (77).

[22] The two sub-formulas can be combined by the procedure described at the end of example 6.8. We present the general algorithm in section 6.12.

(77)



This will result in the TFOL sub-formulas shown below.

(78) $i :_u \mathit{beluga}$, $\texttt{name}(s, i, \mathit{tiqa}) \ \wedge\ \texttt{time\_present}(t, \texttt{breathe}(t_1, i)) \ \wedge\ \texttt{slowly}(s, t)$

The meaning is correctly captured in this formula that informs about a situation s that asserts an individual named *tiqa* is involved in a situation t of breathing at the current time, and t is occurring slowly in the situation s.

6.7.3 Prepositional Modifiers

Prepositional modifiers are prepositional phrases with *predicative prepositions*. By predicative preposition we mean a preposition that contributes to the semantics of the phrase in contrast to *argument marking prepositions*, which only mark an argument of another constituent without any contributions to the semantics.

An example of an argument marking preposition is *to* in the following sentence. *to* is used to mark the second object of the di-transitive verb *gave.*

(79) I gave the red rose to Mary.

An example of a predicative preposition is *on* in the following sentence. The semantic contribution is expressed by a narrow scopal modifier predication that relates the situation of *walking* to the physical situation of *being on the roof.*

(80) The cat walks on the roof.

Prepositional modifiers can modify nominals too, as in sentence (81).

(81) The black cat on the roof is jumping on you.

As the name implies, predicative prepositions must have their PRED feature set to +, which means they can appear after the verb *be*, like the sentence below.

(82) The cat is on the roof.

When used after the verb *be*, or when a nominal is modified, the feature structure description of the prepositional modifier looks like an adjective. So like adjectives, the SPR list is non-empty and contains a NP with the semantic INDEX equal to the individual that is being modified.

The feature structure description of the predicative preposition *on* as a nominal modifier is presented in (83). For the sentence modifier case, we need to add an additional **outscope** predication like adverbs. The modifier predication will count as a narrow scopal predication.

(83)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{on}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{prep} & \\ \text{FORM} & \text{on} \\ \text{PRED} & + \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \left\langle \text{NP}_i \right\rangle \\ \text{COMPS} & \left\langle \text{NP}_j\left[\text{CASE } \textit{acc}\right] \right\rangle \\ \text{MOD} & \left\langle \begin{bmatrix} \textit{mod-elem} & \\ \text{MODIFIED} & \text{NOM}_i \\ \text{AFTER} & + \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{none} \\ \text{INDEX} & s \\ \text{TYPE} & \textit{all} \\ \text{RESTR} & \left\langle \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{on} \\ \text{SIT} & s \\ \text{TOP} & i \\ \text{BOTTOM} & j \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix}$$

The non-empty SPR list is only used when the prepositional phrase is used as a predicate of the copular verb *be* by *subject sharing*. This is explained in section 6.10.

6.8 Semantics of Subordinate Conjunctions

In chapter 5 we studied the syntax of two subordinate structures, namely the conditional statement with *if*, and the relative clauses. We discuss the semantics of these structures in what follows.

6.8.1 Relative Clauses

As we mentioned in section 5.6.1, a defining relative clause is a gappy sentence introduced by a *wh*-word or *that* that modifies a nominal that precedes it.

(84) $$\underbrace{\text{NOM}}_{\text{modified}} \quad \underbrace{<\text{relativizer}> \quad <\text{gappy sentence}>}_{\text{modifier}}$$

In the syntactic analysis of section 5.6.1 we treated the relativizer as the head that is combined with a gappy sentence specified in its complement list COMPS. The saturated phrase with a relativizer head then acts as a modifier that modifies a nominal specified in the relativizer's MOD list. This was captured by the feature structure description of (101) we presented in chapter 5.

A simple semantic analysis of the relative clause is that the gap in the relative clause refers to exactly the same individual as the nominal that precedes it. This co-indexation is done in the feature structure description shown in (85).

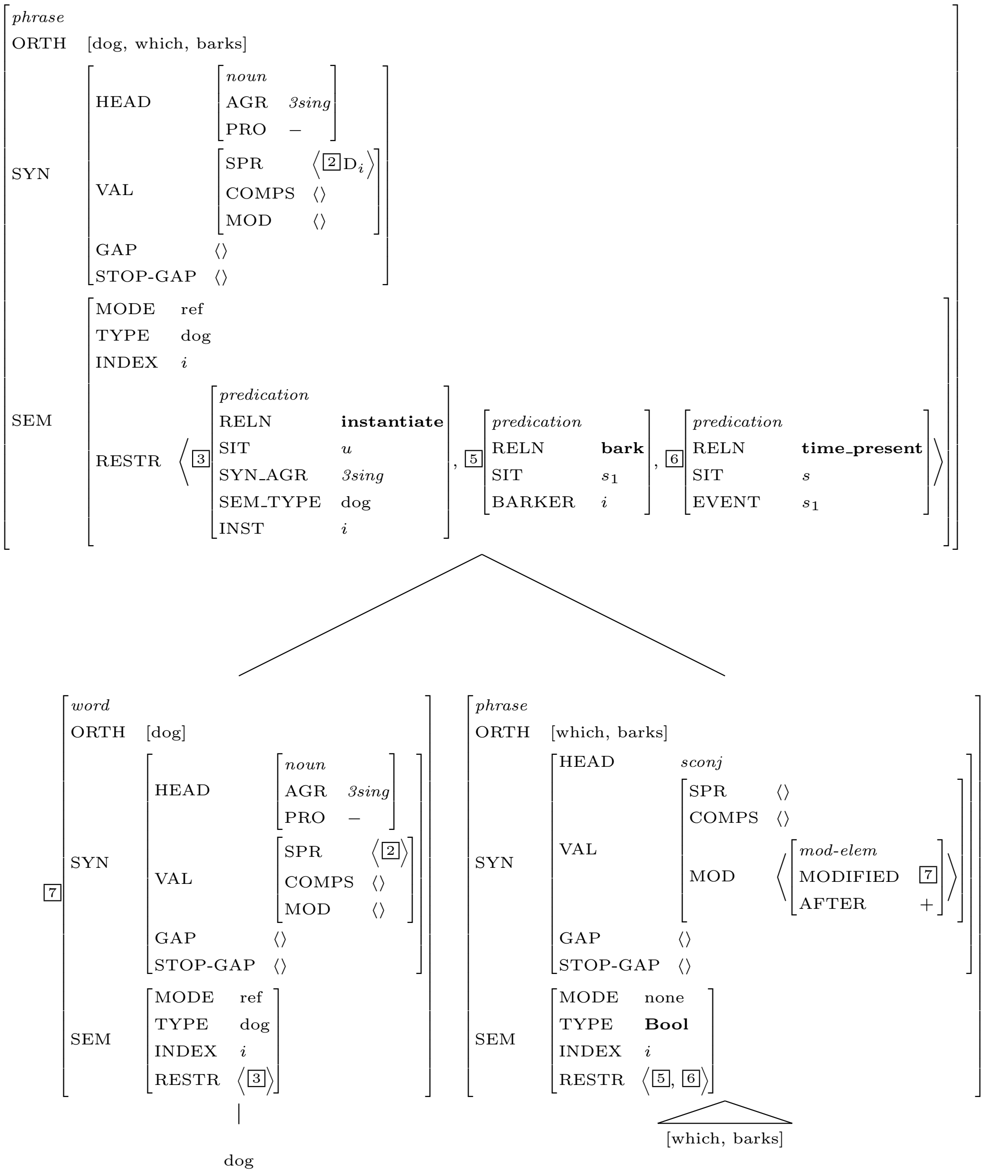
(85) $$\left[\begin{array}{ll} \textit{word} & \\ \text{ORTH} & [\text{which}] \\ \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \textit{sconj} \\ \text{VAL} & \left[\begin{array}{ll} \text{SPR} & \langle\rangle \\ \text{COMPS} & \left\langle \text{S}_t \left[\begin{array}{ll} \text{INV} & - \\ \text{GAP} & \left\langle \boxed{1}\text{NP}_i \left[\text{AGR}\ \boxed{2}\right] \right\rangle \end{array}\right] \right\rangle \\ \text{MOD} & \left\langle \left[\begin{array}{ll} \textit{mod-elem} & \\ \text{MODIFIED} & \text{NOM}_i \left[\text{AGR}\ \boxed{2}\right] \\ \text{AFTER} & + \end{array}\right] \right\rangle \end{array}\right] \\ \text{GAP} & \langle\rangle \\ \text{STOP-GAP} & \langle \boxed{1} \rangle \end{array}\right] \\ \text{SEM} & \left[\begin{array}{ll} \text{MODE} & \text{none} \\ \text{INDEX} & t \\ \text{TYPE} & \textbf{Bool} \\ \text{RESTR} & \langle\rangle \end{array}\right] \end{array}\right]$$

Note that the semantic mode is propositional, because the relative clause simply asserts the semantics of the sentence it contains applied on the nominal that precedes it. For the same reason the type is **Bool**. The semantic INDEX of the relativizer is the same as the semantic INDEX of the complement sentence.

The RESRT feature is the empty list. The reason is that when the relativizer is combined with the gappy sentence by the Head Complement Rule, the semantics of the gappy sentence is absorbed in the RESTR feature of the resulting phrase by the Semantic Compositionality Principle, and this semantic alone is enough as the semantics contribution of the relative clause.

As an example the phrase structure tree of the nominal phrase *dog which barks* is presented in (86). This tree is licensed by the Head Modifier Rule. The resulting phrase can be combined with a determiner such as *the*, *a*, etc. to form a complete noun phrase.

(86)


phrase
ORTH [dog, which, barks]
SYN
HEAD noun AGR 3sing PRO –
VAL SPR ⟨2 D_i⟩ COMPS ⟨⟩ MOD ⟨⟩
GAP ⟨⟩
STOP-GAP ⟨⟩
SEM
MODE ref
TYPE dog
INDEX i
RESTR ⟨3 predication RELN instantiate SIT u SYN_AGR 3sing SEM_TYPE dog INST i, 5 predication RELN bark SIT s1 BARKER i, 6 predication RELN time_present SIT s EVENT s1⟩
7
word
ORTH [dog]
SYN
HEAD noun AGR 3sing PRO –
VAL SPR ⟨2⟩ COMPS ⟨⟩ MOD ⟨⟩
GAP ⟨⟩
STOP-GAP ⟨⟩
SEM
MODE ref
TYPE dog
INDEX i
RESTR ⟨3⟩
dog
phrase
ORTH [which, barks]
SYN
HEAD sconj
VAL SPR ⟨⟩ COMPS ⟨⟩ MOD ⟨mod-elem MODIFIED 7 AFTER +⟩
GAP ⟨⟩
STOP-GAP ⟨⟩
SEM
MODE none
TYPE Bool
INDEX i
RESTR ⟨5, 6⟩
[which, barks]

Note that the mother phrase of the above phrase structure tree carries predications with two situation indices, namely, u and s. When the phrase is combined with a determiner by the Head Specifier Rule, at least another predication is added to the semantics with a situation index such as t, where t immediately outscopes u, and must also outscope s by the General Variable Binding Condition. It is then up to the scope resolution algorithm to equate s to some other scope in order to make the immediate outscope relation graph a singly rooted tree. The steps are already discussed in our analysis of singular countable noun phrases presented in section 6.4.

6.8.2 The Conditional Statement

The syntactic analysis we provided in chapter 5 section 5.2.5 treated *if* as a subordinate conjunction (*sconj*) with two complements, as shown below. The first complement is a sentence that serves as the condition of the conditional statement, which we subscripted by t in (87) to denote its semantic INDEX. The second complement is a preposition *then* with no semantic significance. A phrase that is formed by the Head Complement Rule with the head *if* can then act as a modifier of another sentence that follows it. We subscripted this sentence by s. This sentence serves as the consequent of the conditional statement.

(87) $\underbrace{\textit{if} \ < \text{condition} >_t \ \textit{then}}_{\text{modifier}} \ \underbrace{< \text{statement} >_s}_{\text{modified}}$

The semantic contribution of the subordinate conjunction *if* is a scopal modification predication **if** that takes two scopal features t, and s.

The feature structure description of *if* with semantic features is provided in (89). The TFOL representation of *if* is given by (88). The meaning is that w is a situation in which situation s holds provided that situation t holds as well.

(88) $\texttt{if}(w,\ t,\ s)$

(89)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{if}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \textit{sconj} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \left\langle \text{S}_{t_0},\ \text{P[then]} \right\rangle \\ \text{MOD} & \left\langle \begin{bmatrix} \textit{mod-elem} & \\ \text{MODIFIED} & \text{S}_{s_0} \\ \text{AFTER} & - \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{none} \\ \text{INDEX} & w \\ \text{TYPE} & \textbf{Bool} \\ \text{RESTR} & \left\langle \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{if} \\ \text{SIT} & w \\ \text{COND} & t \\ \text{CONS} & s \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{outscope} \\ \text{OUTER} & t \\ \text{INNER} & t_0 \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{outscope} \\ \text{OUTER} & s \\ \text{INNER} & s_0 \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix}$$

Like adverbs we have used **outscope** predications in the semantic restrictions of the word *if*.[23]

[23] The reason that **outscope** is used rather than just unifying t with t_0 and unifying s with s_0 is that there can be adverbs in the condition or consequence sentences, and by the HFP, the sentence licensed by

In chapter 5 we discussed that an alternative analysis of *if* treats it as a pre-head modifier:

(90) $\underbrace{<\text{statement}>_s}_{\text{modified}} \quad \underbrace{\textit{if}\ <\text{condition}>_t}_{\text{modifier}}$

The feature structure description of *if* corresponding to this analysis is given in (91).

(91)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{if}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \textit{sconj} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \left\langle \text{S}_{t_0} \right\rangle \\ \text{MOD} & \left\langle \begin{bmatrix} \textit{mod-elem} & \\ \text{MODIFIED} & \text{S}_{s_0} \\ \text{AFTER} & + \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{none} \\ \text{INDEX} & w \\ \text{TYPE} & \textbf{Bool} \\ \text{RESTR} & \left\langle \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{if} \\ \text{SIT} & w \\ \text{COND} & t \\ \text{CONS} & s \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{outscope} \\ \text{OUTER} & t \\ \text{INNER} & t_0 \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{outscope} \\ \text{OUTER} & s \\ \text{INNER} & s_0 \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix}$$

We provide two example sentences (92), and (93) that use *if.* In the second sentence, we subscripted *every man* and *he* with the same index i to denote that they refer to the same individual.

(92) If every bird flies then I run.

(93) Every man$_i$ sings if he$_i$ eats a cookie.

For the first sentence a narrow scope of the universal quantifier *every* is more natural, whereas the second sentence requires a wide scope of the quantifier, and the scope of the existential quantifier *a* could be either wide or narrow. The choice of wide or narrow scopes for quantifiers could be suggested by heuristics but using heuristics is not the topic of this thesis.

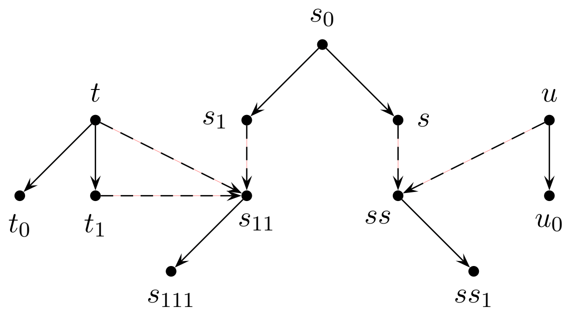
The first sentence can be parsed to form a phrase structure shown in (94).

the HMR has the same semantic INDEX as its verb, and without the **outscope** predication we would have two predications that have the same scopal feature values, which avoids the outscope graph to be resolved to a tree. This is pretty similar to the explanation we provided for adverbs in footnote 21.

(94)
$$
\begin{bmatrix}
\textit{phrase} & \\
\text{ORTH} & [\text{if, every, bird, flies, then, i, run}] \\
\text{SYN} & \ldots \\
\text{SEM} & \begin{bmatrix}
\text{MODE} & \text{prop} \\
\text{TYPE} & \textbf{Bool} \\
\text{INDEX} & s \\
\text{RESTR} & \left\langle \begin{array}{l}
\boxed{2}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{if} \\ \text{SIT} & s_0 \\ \text{COND} & s_1 \\ \text{CONS} & s \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{outscope} \\ \text{OUTER} & s_1 \\ \text{INNER} & s_{11} \end{bmatrix}, \boxed{4}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{outscope} \\ \text{OUTER} & s \\ \text{INNER} & ss \end{bmatrix}, \\
\boxed{3}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{all} \\ \text{SIT} & t \\ \text{BOUND\_VAR} & i \\ \text{QRESTR} & t_0 \\ \text{QSCOPE} & t_1 \end{bmatrix}, \boxed{4}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{dc\_obj\_add} \\ \text{SIT} & t \\ \text{OBJ} & i \\ \text{INSTPRED} & t_0 \end{bmatrix}, \boxed{5}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{instantiate} \\ \text{SIT} & t_0 \\ \text{SYN\_AGR} & \textit{3sing} \\ \text{SEM\_TYPE} & \text{bird} \\ \text{INST} & i \end{bmatrix}, \\
\boxed{6}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{fly} \\ \text{SIT} & s_{111} \\ \text{FLYER} & i \end{bmatrix}, \boxed{7}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{time\_present} \\ \text{SIT} & s_{11} \\ \text{EVENT} & s_{111} \end{bmatrix}, \boxed{8}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{dc\_obj\_get} \\ \text{SIT} & u \\ \text{OBJ} & j \\ \text{INSTPRED} & u_0 \end{bmatrix}, \\
\boxed{9}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{instantiate} \\ \text{SIT} & u_0 \\ \text{SYN\_AGR} & \textit{1sing} \\ \text{SEM\_TYPE} & \text{human} \\ \text{INST} & j \end{bmatrix}, \boxed{10}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{speaker} \\ \text{SIT} & u \\ \text{INST} & j \end{bmatrix}, \boxed{11}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{run} \\ \text{SIT} & ss_1 \\ \text{RUNNER} & j \end{bmatrix}, \\
\boxed{12}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{time\_present} \\ \text{SIT} & ss \\ \text{EVENT} & ss_1 \end{bmatrix}
\end{array} \right\rangle
\end{bmatrix}
\end{bmatrix}
$$

The graph representation of the outscope relation is shown in the diagram (95). The dashed edge from t_1 to s_{11} is the effect of the Quantifier Variable Binding Condition, whereas the dashed edge from t to s_{11} is the result of the General Variable Binding Condition. Dashed edges from s_1 to s_{11} and from s to ss are induced by the **outscope** predications. Finally the dashed edge from u to ss is for the General Variable Binding Condition.

(95)

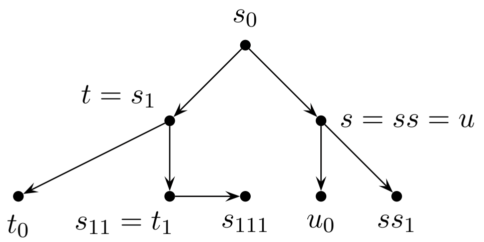


Scopes t_1, s_1, s need to be equated to other scopes, because they are not the label of any predication in the above list of predications. To satisfy the outscope constraint from u to ss, u also needs to be equated to some other scope. One possible resolution (that as we see generates the wrong TFOL representation) will result from:

(96) $t_1 = s_{11},\ t = s_1,\ s = ss = u$

Which results in the following immediate outscope relation tree:

(97)



The following TFOL sub-formulas can be generated using the predications after equating the scopes as indicated in the scope-resolved tree of (97):

(98) $if(s_0, t, s),\ all(t, i, t_0, t_1),\ i :_{t_0} bird, time\_present(t_1, (fly(s_{111}, i))$
$,\ j :_{u_0} human,\ speaker(s, j)\ \wedge\ time\_present(s, run(ss_1, j))$

And, like the process we discussed at the end of example 6.8 we can unify u_0 with s.

To form a TFOL formula structure we can replace situation values with the conjunction of the sub-formulas with the same label. This will result in the following formula. For better clarity we have placed the label as the subscript of sub-formulas.

(99) $if_{s_0}(\forall_t\ i :_{t_0} bird\ ; time\_present(t_1, fly_{s_{111}}(i)),$
$\qquad j :_s human \wedge\ speaker_s(j)\ \wedge\ time\_present_s(run_{ss_1}(j)))$

Now that the situation values have served their purpose in forming the structure of the formula, we can ignore them. Also we can use the familiar arrow notation for the `if` predicate. This gives us:

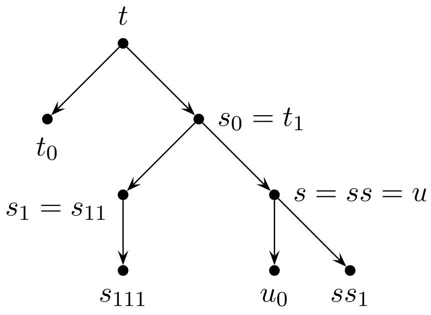
(100) $(\forall\, i : bird\, ; time\_present(fly(i))) \Rightarrow j : human \wedge\ speaker(j) \wedge time\_present(run(j)))$

A less acceptable interpretation results from the original outscope relation graph of diagram (95) by equating:

(101) $t_1 = s_0,\ s_1 = s_{11},\ s = ss,\ u = ss$

This yields the following immediate outscope relation tree:

(102)



This gives us:

(103) $\forall\,(i : bird)\, ;\ time\_present((fly(i)) \Rightarrow\ j : human \wedge speaker(j) \wedge time\_present(run(j)))$

But this means that if an arbitrary bird flies then the speaker runs. If such interpretations are undesirable, heuristics can be used to eliminate or de-prioritize them. However we do not discuss heuristics in this thesis.

Next consider the sentence (93) that we repeat below. There are two quantifiers in the sentence whose order is rather arbitrary. The scope of the existential quantifier could be either inside the conditional, or outside of it.

(104) Every man$_i$ sings if he$_i$ eats a cookie.

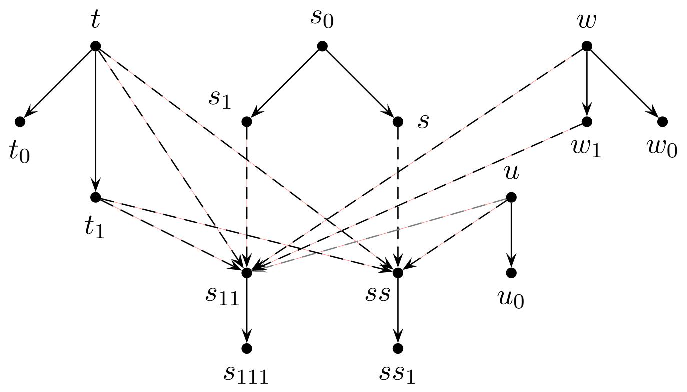
This sentence can be parsed to form the phrase structure shown in (105). We need to unify i and j if the antecedent of the pronoun *he* is *every man.*

(105)

$$
\begin{bmatrix}
\textit{phrase} & \\
\text{ORTH} & \text{[every, man, sings, if, he, eats, a, cookie]}\\
\text{SYN} & \begin{bmatrix}
\text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{FORM} & \text{fin}\\ \text{PRED} & -\\ \text{AGR} & \begin{bmatrix}\textit{3sing} & \\ \text{GEND} & \textit{masc}\end{bmatrix}\\ \text{AUX} & -\\ \text{INV} & - \end{bmatrix}\\
\text{VAL} & \begin{bmatrix}\text{SPR} & \langle\rangle\\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle\end{bmatrix}\\
\text{GAP} & \langle\rangle\\
\text{STOP-GAP} & \langle\rangle
\end{bmatrix}\\
\text{SEM} & \begin{bmatrix}
\text{MODE} & \text{prop}\\
\text{TYPE} & \textbf{Bool}\\
\text{INDEX} & ss\\
\text{RESTR} & \left\langle \begin{array}{l}
\boxed{2}\begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{all}\\ \text{SIT} & t\\ \text{BOUND\_VAR} & i\\ \text{QRESTR} & t_0\\ \text{QSCOPE} & t_1\end{bmatrix},
\boxed{3}\begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{dc\_obj\_add}\\ \text{SIT} & t\\ \text{OBJ} & i\\ \text{INSTPRED} & t_0\end{bmatrix},
\boxed{4}\begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{instantiate}\\ \text{SIT} & t_0\\ \text{SYN\_AGR} & \begin{bmatrix}\textit{3sing} & \\ \text{GEND} & \textit{masc}\end{bmatrix}\\ \text{SEM\_TYPE} & \text{male\_human}\\ \text{INST} & i\end{bmatrix},\\
\boxed{5}\begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{sing}\\ \text{SIT} & ss_1\\ \text{SINGER} & i\\ \text{SONG} & l\end{bmatrix},
\boxed{6}\begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{time\_present}\\ \text{SIT} & ss\\ \text{EVENT} & ss_1\end{bmatrix},
\boxed{7}\begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{if}\\ \text{SIT} & s_0\\ \text{COND} & s_1\\ \text{CONS} & s\end{bmatrix},
\boxed{8}\begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{outscope}\\ \text{SIT} & s_0\\ \text{OUTER} & s_1\\ \text{INNER} & s_{11}\end{bmatrix},\\
\boxed{9}\begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{outscope}\\ \text{SIT} & s_0\\ \text{OUTER} & s\\ \text{INNER} & ss\end{bmatrix},
\boxed{10}\begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{dc\_obj\_get}\\ \text{SIT} & u\\ \text{OBJ} & j\\ \text{INSTPRED} & u_0\end{bmatrix},
\boxed{11}\begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{instantiate}\\ \text{SIT} & u_0\\ \text{SYN\_AGR} & \begin{bmatrix}\textit{3sing} & \\ \text{GEND} & \textit{masc}\end{bmatrix}\\ \text{SEM\_TYPE} & \text{male\_human}\frown\text{X}\\ \text{INST} & j\end{bmatrix},\\
\boxed{12}\begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{eat}\\ \text{SIT} & s_{111}\\ \text{EATER} & j\\ \text{EATEN} & k\end{bmatrix},
\boxed{13}\begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{time\_present}\\ \text{SIT} & s_{11}\\ \text{EVENT} & s_{111}\end{bmatrix},
\boxed{14}\begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{exists}\\ \text{SIT} & w\\ \text{BOUND\_VAR} & k\\ \text{QRESTR} & w_0\\ \text{QSCOPE} & w_1\end{bmatrix},\\
\boxed{15}\begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{dc\_obj\_add}\\ \text{SIT} & w\\ \text{OBJ} & k\\ \text{INSTPRED} & w_0\end{bmatrix},
\boxed{16}\begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{instantiate}\\ \text{SIT} & w_0\\ \text{SYN\_AGR} & \begin{bmatrix}\textit{3sing} & \\ \text{GEND} & \textit{neut}\end{bmatrix}\\ \text{SEM\_TYPE} & \text{cookie}\\ \text{INST} & k\end{bmatrix}
\end{array}\right\rangle
\end{bmatrix}
\end{bmatrix}
$$

The corresponding outscope relation graph is shown below.

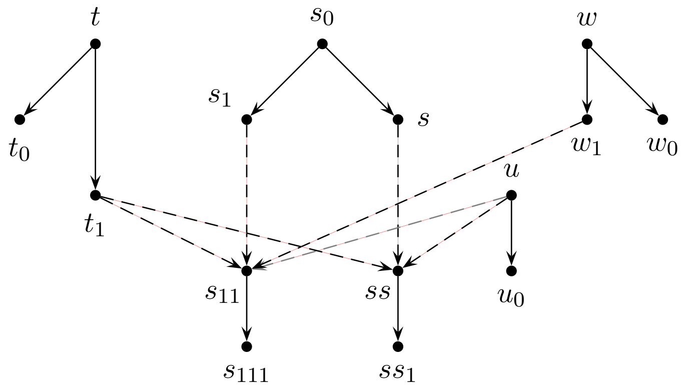
(106)



The gray dashed edge from u to s_{11} is the result of the identity $i = j$ so by the general variable binding condition s_{11} must also be in the scope of the **dc_obj_get** predication labeled with u.

Removing the dashed edges that can be derived by the transitivity of the outscope relation yields the following graph.

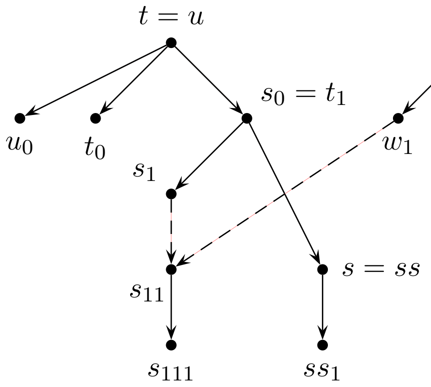
(107)



We have both s_0 and t_1 as the common ancestors of s_{11} and ss. Since the final graph must be a singly rooted tree, we cannot have a node with two edges entering. This means that either t_1 is an ancestor of s_0 or vice versa. The latter cannot be the case because s_0 has two solid edges coming out that lead to two paths two either s_{11} or ss when combined with the paths from the descendant t_1 to those nodes. So t_1 must be an ancestor of s_0. Then $s = ss$ to satisfy $s \leq ss$, because no path of solid edges can be formed from s to ss

if they are not the same. u is just the label of the instance relation the re-introduced the individual referred by the pronoun *he*. It can be equivalently equated to t or t_1. We equate it to t. So far, this will result the graph shown in (108).

(108)



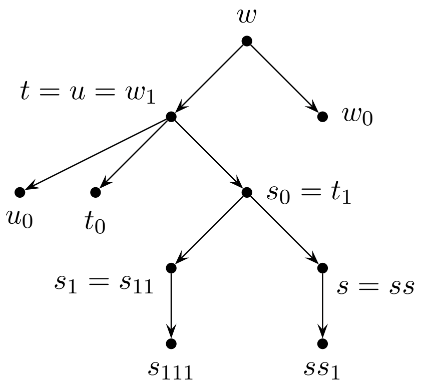
In this sentence we have two quantifiers, namely, the universal quantifier that acts upon *man* and an existential quantifier that acts upon *cookie*. The underspecified scopes of these quantifiers will lead to multiple readings of the phrase.

For the first possible reading, we can have:

(109) $w_1 = t, s_1 = s_{11}$

Which results in the following scope resolved tree:

(110)



The corresponding TFOL formula will be:

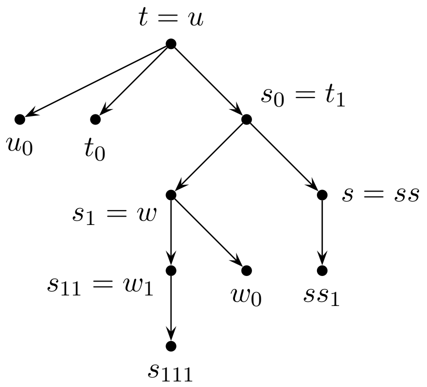
(111) $\exists k : cookie \ ; \ \forall i : male\_human \ ; \ time\_present(eat(i,k)) \Rightarrow time\_present(sing(i))$

The second possible reading can result from:

(112) $s_1 = w, w_1 = s_{11}$

With the corresponding immediate outscope tree below:

(113)



The TFOL formula for this tree will be:

(114) $\forall i : \mathit{male\_human}\ ;\ (\exists k : \mathit{cookie}\ ;\ \mathit{time\_present}(\mathit{eat}(i,k))) \Rightarrow \mathit{time\_present}(\mathit{sing}(i))$

This is indeed equivalent to:

(115) $\forall i : \mathit{male\_human}\ ;\ \forall k : \mathit{cookie}\ ;\ (\mathit{time\_present}(\mathit{eat}(i,k)) \Rightarrow \mathit{time\_present}(\mathit{sing}(i)))$

We have shown two possible scope resolutions of the graph shown in (106). Another scope resolution is possible that we do not show here, in which the scope of the existential quantifier is outside the conditional but within the scope of the universal quantifier that gives us the following formula.

(116) $\forall i : \mathit{male\_human}\ ;\ \exists k : \mathit{cookie}; (\mathit{time\_present}(\mathit{eat}(i,k))) \Rightarrow \mathit{time\_present}(\mathit{sing}(i))$

A Restriction and an Adjustment

We have presented two analysis of *if* with the same semantic restrictions and features. One analysis treated *if* as a post-head modifier and the other treated it as a pre-head modifier. There is actually is subtle difference between the semantics of the two. However, with our definition of the features and feature structure type hierarchy this cannot be established by only the semantic features of the corresponding feature structure descriptions of *if*. Consider the two sentences below.

(117) # If every man$_i$ eats a cookie then he$_i$ sings.

(118) If a man$_i$ runs then he$_i$ breathes faster.

Some native English speakers might find the sentence (117) a little bit odd if the pronoun *he* refers to the the same individual that is intended by the quantifier *every man*.

On the other hand sentence (118) is perfectly natural but it could very well mean that every man breathes fast if he runs.

To restrict a sentence like (117) to be meaningful and to add an extra possible meaning for the sentence (118) we propose the following two restrictions:

(119) Universal Conditional Restriction:
If a universal quantifier predication with label s appears after an *if* predication with label t in the semantic restriction of a syntactically well-formed parsed phrase, the outscope relation must not include $s \leq t$

(120) Existential Condition Adjustment:
If an existential quantifier predication with label s appears after an *if* predication with label t in the semantic restriction of a syntactically well-formed parsed phrase, and the outscope relation includes $t < s$ then an additional interpretation should be considered where the existential quantifier is replaced by a universal one.

6.9 Semantics of Coordinate Conjunction

We discussed the syntax of the coordinate structures in chapter 5. In this section we provide the semantic counterpart.

6.9.1 Coordinated Sentences

Two sentences can be coordinated using conjunctions such as *and* (*or*). The semantic contribution of the conjunction is an **and** (**or**) scopal predication, with two **outscope** predications. The feature structure description of *and* used to combine sentences is presented in (121).

(121)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{and}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \textit{pred-co-conj} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \textit{sem-pred-co-conj} & \\ \text{MODE} & \text{none} \\ \text{INDEX} & w \\ \text{TYPE} & \textbf{Bool} \\ \text{COMPONENT1} & s_1 \\ \text{COMPONENT2} & s_2 \\ \text{RESTR} & \left\langle \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{and} \\ \text{SIT} & w \\ \text{COMPONENT1} & t_1 \\ \text{COMPONENT2} & t_2 \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{outscope} \\ \text{OUTER} & t_1 \\ \text{INNER} & s_1 \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{outscope} \\ \text{OUTER} & t_2 \\ \text{INNER} & s_2 \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix}$$

We already provided the coordination rules in section 5.3.1 of chapter 5. We just need to add the semantics, which is done in the rule below:

(122) Coordination Rule for Sentences Final Version:

$$\text{S}\begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix}\text{INV } \boxed{1}\end{bmatrix} \\ \text{GAP} & \langle\rangle \end{bmatrix} \\ \text{SEM} & \begin{bmatrix}\text{INDEX} & w\end{bmatrix} \end{bmatrix} \longrightarrow$$

$$\text{S}\begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix}\text{INV } \boxed{1}\end{bmatrix} \\ \text{GAP} & \langle\rangle \end{bmatrix} \\ \text{SEM} & \begin{bmatrix}\text{INDEX} & s_1\end{bmatrix} \end{bmatrix} \begin{bmatrix} \text{ORTH} & [\text{and}] \\ \text{HEAD} & \textit{pred-co-conj} \\ \text{SEM} & \begin{bmatrix} \textit{sem-pred-co-conj} & \\ \text{INDEX} & w \\ \text{COMPONENT1} & s_1 \\ \text{COMPONENT2} & s_2 \end{bmatrix} \end{bmatrix} \text{S}\begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix}\text{INV } \boxed{1}\end{bmatrix} \\ \text{GAP} & \langle\rangle \end{bmatrix} \\ \text{SEM} & \begin{bmatrix}\text{INDEX} & s_2\end{bmatrix} \end{bmatrix}$$

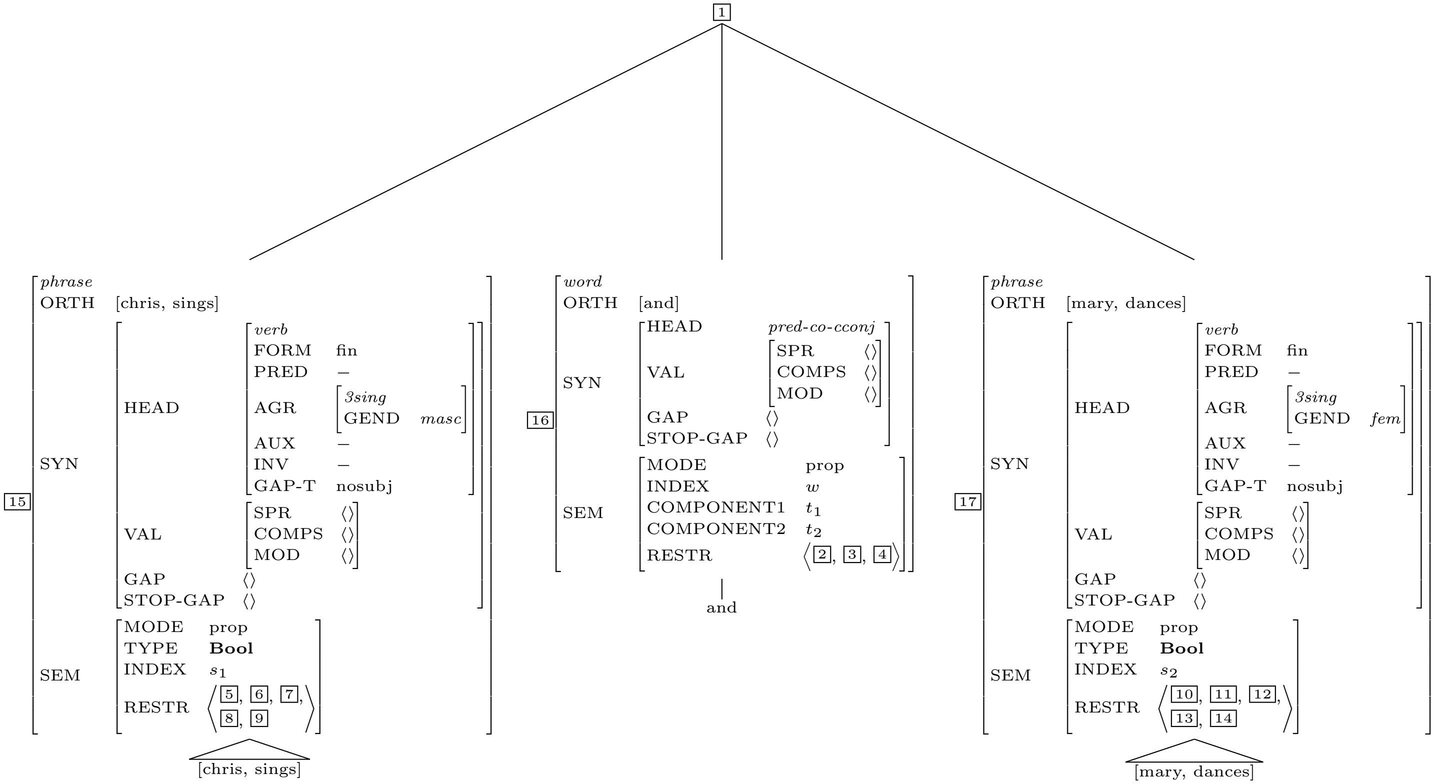
The scope resolution process of *and* is very similar to *if*, because both are logical connectives. So we do not go through the details. Rather we present the phrase structure tree of the sentence below, and provide its TFOL representation after scope resolution.

(123) Chris sings and Mary dances.

(124)

$$\boxed{1}\begin{bmatrix} \textit{phrase} & \\ \text{ORTH} & [\text{chris, sings, and, mary, dances}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{FORM} & \text{fin} \\ \text{AUX} & - \\ \text{GAP-T} & \text{nosubj} \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \\ \text{GAP} & \langle\rangle \\ \text{STOP-GAP} & \langle\rangle \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{prop} \\ \text{TYPE} & \textbf{Bool} \\ \text{INDEX} & w \\ \text{RESTR} & \left\langle \begin{array}{l} \boxed{2}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{and} \\ \text{SIT} & w \\ \text{COMPONENT1} & t_1 \\ \text{COMPONENT2} & t_2 \end{bmatrix}, \boxed{3}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{outscope} \\ \text{OUTER} & t_1 \\ \text{INNER} & s_1 \end{bmatrix}, \boxed{4}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{outscope} \\ \text{OUTER} & t_2 \\ \text{INNER} & s_2 \end{bmatrix}, \\ \boxed{5}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{dc\_obj\_add} \\ \text{SIT} & u \\ \text{OBJ} & i \\ \text{INSTPRED} & u_1 \end{bmatrix}, \boxed{6}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{instantiate} \\ \text{SIT} & u_1 \\ \text{SYN\_AGR} & \begin{bmatrix} \textit{3sing} & \\ \text{GEND} & \textit{masc} \end{bmatrix} \\ \text{SEM\_TYPE} & \text{male\_human} \\ \text{INST} & i \end{bmatrix}, \boxed{7}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{name} \\ \text{SIT} & u \\ \text{NAME} & \text{chris} \\ \text{NAMED} & i \end{bmatrix}, \\ \boxed{8}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{sing} \\ \text{SIT} & s_{11} \\ \text{SINGER} & i \\ \text{SONG} & k \end{bmatrix}, \boxed{9}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{time\_present} \\ \text{SIT} & s_1 \\ \text{EVENT} & s_{11} \end{bmatrix}, \boxed{10}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{dc\_obj\_add} \\ \text{SIT} & x \\ \text{OBJ} & j \\ \text{INSTPRED} & x_1 \end{bmatrix}, \\ \boxed{11}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{instantiate} \\ \text{SIT} & x_1 \\ \text{SYN\_AGR} & \begin{bmatrix} \textit{3sing} & \\ \text{GEND} & \textit{fem} \end{bmatrix} \\ \text{SEM\_TYPE} & \text{female\_human} \\ \text{INST} & j \end{bmatrix}, \boxed{12}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{name} \\ \text{SIT} & x \\ \text{NAME} & \text{mary} \\ \text{NAMED} & j \end{bmatrix}, \boxed{13}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{dance} \\ \text{SIT} & s_{22} \\ \text{DANCER} & j \end{bmatrix}, \\ \boxed{14}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{time\_present} \\ \text{SIT} & s_2 \\ \text{EVENT} & s_{22} \end{bmatrix} \end{array} \right\rangle \end{bmatrix} \end{bmatrix}$$

(125)


1
15
phrase
ORTH [chris, sings]
SYN
HEAD
verb
FORM fin
PRED –
AGR
3sing
GEND masc
AUX –
INV –
GAP-T nosubj
VAL
SPR ⟨⟩
COMPS ⟨⟩
MOD ⟨⟩
GAP ⟨⟩
STOP-GAP ⟨⟩
SEM
MODE prop
TYPE Bool
INDEX s_1
RESTR ⟨5, 6, 7, 8, 9⟩
[chris, sings]
16
word
ORTH [and]
SYN
HEAD pred-co-cconj
VAL
SPR ⟨⟩
COMPS ⟨⟩
MOD ⟨⟩
GAP ⟨⟩
STOP-GAP ⟨⟩
SEM
MODE prop
INDEX w
COMPONENT1 t_1
COMPONENT2 t_2
RESTR ⟨2, 3, 4⟩
and
17
phrase
ORTH [mary, dances]
SYN
HEAD
verb
FORM fin
PRED –
AGR
3sing
GEND fem
AUX –
INV –
GAP-T nosubj
VAL
SPR ⟨⟩
COMPS ⟨⟩
MOD ⟨⟩
GAP ⟨⟩
STOP-GAP ⟨⟩
SEM
MODE prop
TYPE Bool
INDEX s_2
RESTR ⟨10, 11, 12, 13, 14⟩
[mary, dances]

The TFOL representation of the semantics of this sentence (without situation labels) will be:

(126) $(i : male\_human \wedge name(i, chris) \wedge time\_present(sing(i, k))) \;\; \wedge$
$\qquad (j : female\_human \wedge name(j, mary) \wedge time\_present(dance(j)))$

where k refers to the song that is being sung, which is underspecified. Underspecified individuals can be thought of as being implicitly existentially quantified.

6.9.2 Semantics of Coordinated Count Nouns with Multiple Inheritance

Two singular countable noun phrases can be conjoined by the connective *and* to form a coordinated countable noun phrase. As noted in section 6.6, countable nouns in fact refer to types in the domain type hierarchy. So, a coordinated countable noun phrase can be thought to refer to a type, which is constructed by the type composition operator that we discussed about in chapter 2 and 3. The resulting type will be a type that inherits from the two constituent types.

(127) Coordination Rule for Noun Phrases (com3sg):
If $\tau, \sigma \in \mathbf{NatTyp}$ and $\tau \nparallel \sigma$:

$$\begin{bmatrix} \textit{phrase} & \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \boxed{0}\begin{bmatrix} \textit{noun} & \\ \text{AGR} & \textit{3sing} \\ \text{PRO} & - \\ \text{TYPE\_DEF} & + \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \left\langle \boxed{1} \right\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \\ \text{GAP} & \langle\rangle \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{INDEX} & i \\ \text{TYPE} & \tau * \sigma \end{bmatrix} \end{bmatrix} \longrightarrow$$

$$\begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{HEAD} & \boxed{0} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \left\langle \boxed{1} \right\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \\ \text{GAP} & \langle\rangle \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{INDEX} & i \\ \text{TYPE} & \tau \end{bmatrix} \end{bmatrix} \begin{bmatrix} \text{ORTH} & [\text{and}] \\ \text{HEAD} & \begin{bmatrix} \textit{nom-co-conj} & \\ \text{CONJ\_TYPE} & \text{com3sg} \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \textit{sem-nom-co-conj} & \\ \text{INDEX} & w \\ \text{RESTR} & \langle\rangle \end{bmatrix} \end{bmatrix} \begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{HEAD} & \boxed{0} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \left\langle \boxed{1} \right\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \\ \text{GAP} & \langle\rangle \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{INDEX} & j \\ \text{TYPE} & \sigma \end{bmatrix} \end{bmatrix}$$

The coordination rule to construct types with multiple inheritance is provided in (127). This rule only combines 3rd person singular countable nouns that is why we have named it com3sg. We need to make sure that the constituents are carrying types, i.e., they are count nouns. This is done by checking the TYPE_DEF feature, which must have the value +.

The feature structure description of *and* is shown in the above rule. No semantic restriction is contributed by this connective in this usage.

The count nouns participating in the above rule need to share their determiner. Determiners share their semantic INDEX with the nominals that follow them. As a result the indices of the two count nouns are unified automatically.

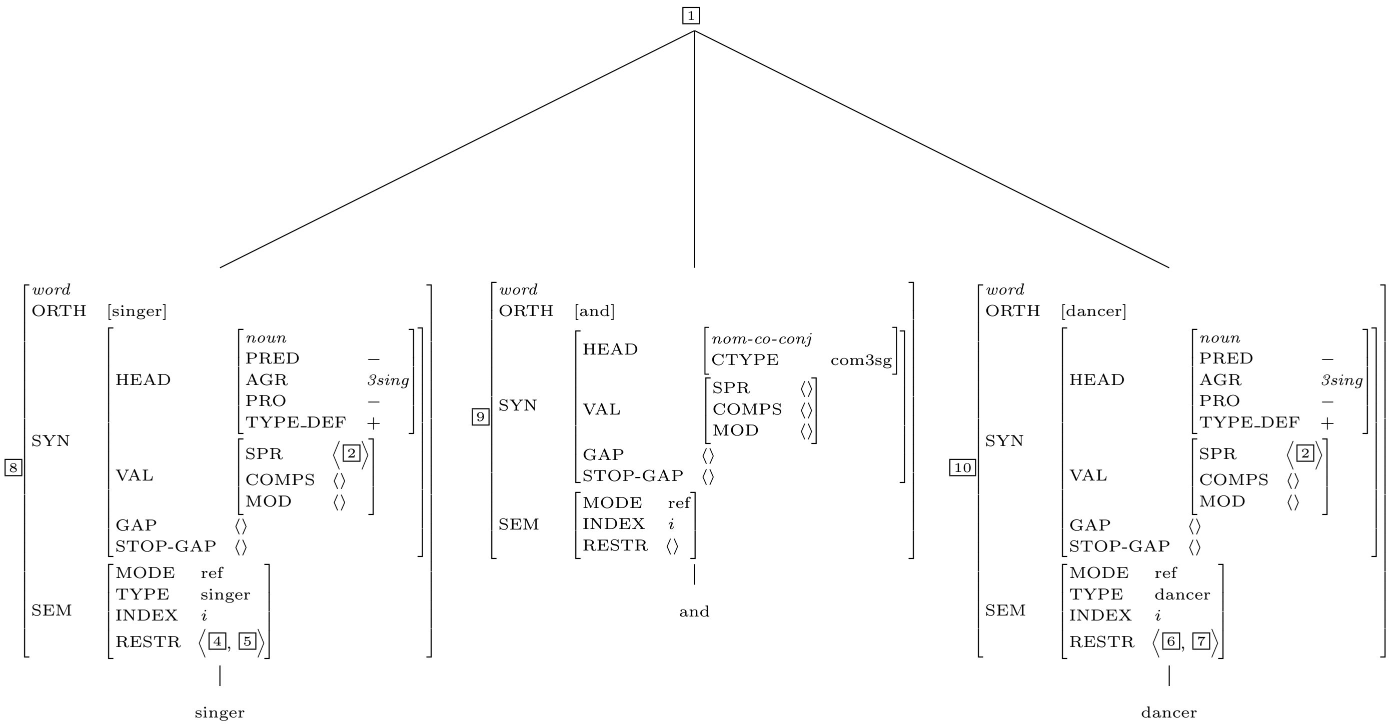
With this coordination rule we can parse the countable noun phrase below. The phrase structure tree of this phrase is shown in (130).

(128) singer and dancer

(129)

$$\boxed{1}\begin{bmatrix} \textit{phrase} & \\ \text{ORTH} & \text{[singer, and, dancer]} \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{noun} & \\ \text{AGR} & \textit{3sing} \\ \text{PRO} & - \\ \text{TYPE\_DEF} & + \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \left\langle \boxed{2}\text{D} \right\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \\ \text{GAP} & \langle\rangle \\ \text{STOP-GAP} & \langle\rangle \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{ref} \\ \text{TYPE} & \text{singer*dancer} \\ \text{INDEX} & i \\ \text{RESTR} & \left\langle \boxed{4}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{instantiate} \\ \text{SIT} & s \\ \text{SYN\_AGR} & \textit{3sing} \\ \text{SEM\_TYPE} & \text{singer} \\ \text{INST} & i \end{bmatrix}, \boxed{5}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{sing} \\ \text{SIT} & s \\ \text{SINGER} & i \\ \text{SONG} & k \end{bmatrix}, \boxed{6}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{instantiate} \\ \text{SIT} & s \\ \text{SYN\_AGR} & \textit{3sing} \\ \text{SEM\_TYPE} & \text{dancer} \\ \text{INST} & i \end{bmatrix}, \boxed{7}\begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{dance} \\ \text{SIT} & s \\ \text{DANCER} & i \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix}$$

(130)


1
8
word
ORTH [singer]
SYN
HEAD
noun
PRED –
AGR 3sing
PRO –
TYPE_DEF +
VAL
SPR ⟨2⟩
COMPS ⟨⟩
MOD ⟨⟩
GAP ⟨⟩
STOP-GAP ⟨⟩
SEM
MODE ref
TYPE singer
INDEX i
RESTR ⟨4, 5⟩
singer
9
word
ORTH [and]
SYN
HEAD
nom-co-conj
CTYPE com3sg
VAL
SPR ⟨⟩
COMPS ⟨⟩
MOD ⟨⟩
GAP ⟨⟩
STOP-GAP ⟨⟩
SEM
MODE ref
INDEX i
RESTR ⟨⟩
and
10
word
ORTH [dancer]
SYN
HEAD
noun
PRED –
AGR 3sing
PRO –
TYPE_DEF +
VAL
SPR ⟨2⟩
COMPS ⟨⟩
MOD ⟨⟩
GAP ⟨⟩
STOP-GAP ⟨⟩
SEM
MODE ref
TYPE dancer
INDEX i
RESTR ⟨6, 7⟩
dancer

6.10 Semantics of Copulas and Auxiliary Verbs

Semantics of copular and auxiliary verbs is very straightforward in that there is little or no semantic contribution at all. For example the feature structure description of the lexical entry of the copular verb *be* is provided below.

(131)
$$\begin{bmatrix} \textit{lexeme} & \\ \text{ORTH} & [\text{be}] \\ \text{ARG-ST} & \left\langle \text{NP}, \begin{bmatrix} \text{SYN} & \begin{bmatrix}\text{HEAD} & \begin{bmatrix}\text{PRED} & +\end{bmatrix}\end{bmatrix} \\ \text{SEM} & \begin{bmatrix}\text{INDEX} & s\end{bmatrix}\end{bmatrix}\right\rangle \\ \text{SEM} & \begin{bmatrix}\text{MODE} & \text{prop}\\ \text{INDEX} & s \\ \text{RESTR} & \langle\rangle\end{bmatrix}\end{bmatrix}$$

This lexical entry is subject to the auxiliary verb constraints NN-NI, N-NI, NN-I, N-I that we provided in chapter 5. With this taken into account, we are now able to see the reason why we have chosen to provide a non-empty list of specifiers for adjectives. As a result of the auxiliary verb constraints above, an auxiliary or a copular verb shares its SPR with its complement. We call this *subject sharing*. Adjectives have a modification predication with the SUBJECT role identical to the semantic index of the only element of the SPR list. With the semantics of *be* above, this enables the grammar to pass the subject of the verb *be* to the specifier element of the adjective which is then assigned the role of the individual that is being modified. So a sentence like (132) can be parsed.

(132) Mary is happy.

(133)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{is}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{FORM} & \text{fin} \\ \text{PRED} & - \\ \text{AGR} & \textit{3sing} \\ \text{AUX} & + \\ \text{INV} & - \end{bmatrix} \\ \text{VAL} & \begin{bmatrix}\text{SPR} & \left\langle \boxed{1} \right\rangle\end{bmatrix} \end{bmatrix} \\ \text{ARG-ST} & \left\langle \boxed{1}\text{NP}_i \text{ , } \begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix}\text{PRED} & +\end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \left\langle \boxed{1} \right\rangle \\ \text{COMPS} & \langle\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix} \right\rangle \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{prop} \\ \text{INDEX} & s \\ \text{RESTR} & \left\langle \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{time\_present} \\ \text{SIT} & t \\ \text{EVENT} & s \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix}$$

The feature structure description of *is* is presented in (133). We should note that tense predications can be added in the process of transforming a lexical entry of a verb into a word.

The feature structure description of *happy* is shown below.

(134)

$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{happy}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{adj} & \\ \text{PRED} & + \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \left\langle \text{NP}_i \right\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \left\langle \begin{bmatrix} \textit{mod-elem} & \\ \text{MODIFIED} & \text{NOM}_i \\ \text{AFTER} & - \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{none} \\ \text{TYPE} & \textit{all} \\ \text{INDEX} & s \\ \text{RESTR} & \left\langle \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{happy} \\ \text{SIT} & s \\ \text{SUBJECT} & i \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix}$$

Now *is* and *happy* can be combined together by the Head Complement Rule to form the phrase structure shown in (135). As can be seen the subject of the verb *is* now has the same index as the individual that is assigned with the subject role in the **happy** predication. This will easily combine with *Mary* by the Head Specifier Rule to form a complete sentence.

(135)

$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{is, happy}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{FORM} & \text{fin} \\ \text{PRED} & - \\ \text{AGR} & \textit{3sing} \\ \text{AUX} & + \\ \text{INV} & - \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \left\langle \boxed{1}\text{NP}_i \right\rangle \end{bmatrix} \end{bmatrix} \\ \text{ARG-ST} & \left\langle \boxed{1}\text{NP}_i \right\rangle \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{prop} \\ \text{INDEX} & s \\ \text{RESTR} & \left\langle \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{time\_present} \\ \text{SIT} & t \\ \text{EVENT} & s \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{happy} \\ \text{SIT} & s \\ \text{SUBJECT} & i \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix}$$

6.10.1 Semantics of Negation

We studied the syntactic properties of negation using the N-NI, and N-I constraints on the auxiliary verbs in chapter 5. We need to add semantics to these constraints. First, the feature structure description of the word *not* is presented in (136). The word *not* carries the negation semantics with its predications and bring it to the phrase it participates in. In this thesis we treat the **not** predication, as a narrow scopal predication. The effect is that all the negation acts like an adverb and negates only the verb. Thus all quantifiers are outside the negated scope.

(136)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{not}] \\ \text{SYN} & \begin{bmatrix}\text{HEAD} & \textit{adv-pol}\end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{prop} \\ \text{INDEX} & s \\ \text{RESTR} & \left\langle \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{not} \\ \text{SIT} & s \\ \text{NEGATED\_PRED} & s_1 \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{outscope} \\ \text{OUTER} & s_1 \\ \text{INNER} & s_2 \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix}$$

However, using negation as a narrow scopal predication is just a choice in this thesis that reduces the number of readings of a sentence.[24] The user of the grammar must be informed about this choice. As far as the theory is concerned, there is no need to treat **not** as a narrow scopal predication, in which case the negation could affect the quantifiers present in the sentence. Heuristics can be used to eliminate or prioritize the readings in case of multiple quantifiers and negations in a sentence, however we do not study these heuristics in this thesis.

The modified N-NI and N-I constraints are:

- Negated and Non-Inverted (N-NI) Version II:

  The feature structure description of a word that is associated with a negated but non-inverted auxiliary verb must be unifiable with:

[24] This choice is consistent with the choice of adv_{pol} category for *not* in [67] as a special kind of adverb and the treatment of adverb predications as narrow scopal predications in this thesis.

(137)
$$\begin{bmatrix}
\text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{AUX} & + \\ \text{INV} & - \end{bmatrix} \\ \text{VAL} & \begin{bmatrix}\text{SPR} & \left\langle \boxed{1} \right\rangle\end{bmatrix}\end{bmatrix} \\
\text{SEM} & \begin{bmatrix}\text{INDEX} & s \\ \text{MODE} & \text{prop}\end{bmatrix} \\
\text{ARG-ST} & \left\langle \boxed{1}\text{NP}\ , \begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{not}] \\ \text{SYN} & \begin{bmatrix}\text{HEAD} & \textit{adv-pol}\end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{prop} \\ \text{INDEX} & s \\ \text{RESTR} & \left\langle \begin{array}{l} \begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{not} \\ \text{SIT} & s \\ \text{NEGATED\_PRED} & s_1\end{bmatrix}, \\ \begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{outscope} \\ \text{OUTER} & s_1 \\ \text{INNER} & s_2\end{bmatrix} \end{array} \right\rangle \end{bmatrix}\end{bmatrix}, \begin{bmatrix}\text{SYN} & \begin{bmatrix}\text{VAL} & \begin{bmatrix}\text{SPR} & \left\langle\boxed{1}\right\rangle \\ \text{COMPS} & \langle\rangle\end{bmatrix}\end{bmatrix}\end{bmatrix} \right\rangle
\end{bmatrix}$$

- Negated and Inverted (N-I) Version II:

  The feature structure description of a word that is associated with a negated and inverted auxiliary verb must be unifiable with:

(138)
$$\begin{bmatrix}
\text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{AUX} & + \\ \text{INV} & + \end{bmatrix} \\ \text{VAL} & \begin{bmatrix}\text{SPR} & \langle\rangle\end{bmatrix}\end{bmatrix} \\
\text{SEM} & \begin{bmatrix}\text{INDEX} & s \\ \text{MODE} & \text{ques}\end{bmatrix} \\
\text{ARG-ST} & \left\langle \boxed{1}\text{NP}\ , \begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{not}] \\ \text{SYN} & \begin{bmatrix}\text{HEAD} & \textit{adv-pol}\end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{prop} \\ \text{INDEX} & s \\ \text{RESTR} & \left\langle \begin{array}{l} \begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{not} \\ \text{SIT} & s \\ \text{NEGATED\_PRED} & s_1\end{bmatrix}, \\ \begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{outscope} \\ \text{OUTER} & s_1 \\ \text{INNER} & s_2\end{bmatrix} \end{array} \right\rangle \end{bmatrix}\end{bmatrix}, \begin{bmatrix}\text{SYN} & \begin{bmatrix}\text{VAL} & \begin{bmatrix}\text{SPR} & \left\langle\boxed{1}\right\rangle \\ \text{COMPS} & \langle\rangle\end{bmatrix}\end{bmatrix}\end{bmatrix} \right\rangle
\end{bmatrix}$$

Using the lexical entry for *not*, and the new N-NI constraint on the copular verb *be* we can parse the following sentence. The resulting phrase structure is shown in (140).

(139) Mary is not sad.

(140)

$$\boxed{1}\left[\begin{array}{lll}
\textit{phrase} & & \\
\text{ORTH} & \multicolumn{2}{l}{\text{[mary, is, not, sad]}} \\
\text{SYN} & \multicolumn{2}{l}{\left[\begin{array}{ll}
\text{HEAD} & \left[\begin{array}{ll}
\textit{verb} & \\
\text{FORM} & \text{fin} \\
\text{PRED} & - \\
\text{AGR} & \left[\begin{array}{ll}\textit{3sing} & \\ \text{GEND} & \textit{fem}\end{array}\right] \\
\text{AUX} & + \\
\text{INV} & - \\
\text{GAP-T} & \text{nosubj}
\end{array}\right] \\
\text{VAL} & \left[\begin{array}{ll}
\text{SPR} & \langle\rangle \\
\text{COMPS} & \langle\rangle \\
\text{MOD} & \langle\rangle
\end{array}\right] \\
\text{GAP} & \langle\rangle \\
\text{STOP-GAP} & \langle\rangle
\end{array}\right]} \\
\text{SEM} & \multicolumn{2}{l}{\left[\begin{array}{ll}
\text{MODE} & \text{prop} \\
\text{TYPE} & \textbf{Bool} \\
\text{INDEX} & s_0 \\
\text{RESTR} & \left\langle
\begin{array}{l}
\boxed{2}\left[\begin{array}{ll}
\textit{predication} & \\
\text{RELN} & \textbf{dc\_obj\_add} \\
\text{SIT} & s_1 \\
\text{OBJ} & j_0 \\
\text{INSTPRED} & s_2
\end{array}\right],
\boxed{3}\left[\begin{array}{ll}
\textit{predication} & \\
\text{RELN} & \textbf{instantiate} \\
\text{SIT} & s_2 \\
\text{SYN\_AGR} & \left[\begin{array}{ll}\textit{3sing} & \\ \text{GEND} & \textit{fem}\end{array}\right] \\
\text{SEM\_TYPE} & \text{female\_human} \\
\text{INST} & j_0 \\
\text{LOCATION} & \text{(1,2)}
\end{array}\right],
\boxed{4}\left[\begin{array}{ll}
\textit{predication} & \\
\text{RELN} & \textbf{name} \\
\text{SIT} & s_1 \\
\text{NAME} & \text{mary} \\
\text{NAMED} & j_0
\end{array}\right], \\
\boxed{5}\left[\begin{array}{ll}
\textit{predication} & \\
\text{RELN} & \textbf{time\_present} \\
\text{SIT} & s_3 \\
\text{EVENT} & s_4
\end{array}\right],
\boxed{6}\left[\begin{array}{ll}
\textit{predication} & \\
\text{RELN} & \textbf{not} \\
\text{SIT} & s_0 \\
\text{NEGATED\_PRED} & s_5
\end{array}\right],
\boxed{7}\left[\begin{array}{ll}
\textit{predication} & \\
\text{RELN} & \textbf{outscope} \\
\text{SIT} & s_6 \\
\text{OUTER} & s_5 \\
\text{INNER} & s_4
\end{array}\right], \\
\boxed{8}\left[\begin{array}{ll}
\textit{predication} & \\
\text{RELN} & \textbf{sad} \\
\text{SIT} & s_4 \\
\text{SUBJECT} & j_0
\end{array}\right]
\end{array}
\right\rangle
\end{array}\right]}
\end{array}\right]$$

A final note on the auxiliary and copular verbs is that the constraints NN-NI and N-NI must set the semantic mode of the verb to 'prop' and the constraints NN-I and N-I that apply on the inverted auxiliary verbs need to set the semantic mode of the verb to 'ques'. The reason is simply that inverted sentences are questions rather than propositions.

6.11 Semantics of *Wh*-Questions

Question words like *Who* and *What* contribute to the semantics by adding a special predication whose duty is to find an individual that matches the sentence that follow the question words. We call this new predication **find**. We treat it as a quantifier predication that needs to be paired with an **instantiate** predication. This **instantiate** predication adds type information for the individual that is sought. For *Who* the type needs to be *human*, whereas for *What* it needs to be $all \curvearrowright X - human$. An **outscope** predication is also provided to ensure that the outscope resolution algorithm makes correct choices. For example, the feature structure description of the question word *Who* is given in (141).

(141)
$$\left[\begin{array}{ll} \textit{word} & \\ \text{ORTH} & \text{[who]} \\ \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \textit{qword} \\ \text{VAL} & \left[\begin{array}{ll} \text{SPR} & \langle\rangle \\ \text{COMPS} & \left\langle \text{S}_u \left[\begin{array}{ll} \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \left[\begin{array}{ll} \text{INV} & - \\ \text{GAP-TYPE} & \text{gsubj} \end{array}\right] \\ \text{GAP} & \langle \boxed{1}\text{NP}_i \rangle \end{array}\right] \end{array}\right] \right\rangle \\ \text{MOD} & \langle\rangle \end{array}\right] \\ \text{GAP} & \langle\rangle \\ \text{STOP-GAP} & \langle \boxed{1} \rangle \end{array}\right] \\ \text{SEM} & \left[\begin{array}{ll} \text{MODE} & \text{ques} \\ \text{INDEX} & s \\ \text{TYPE} & \textbf{Bool} \\ \text{RESTR} & \left\langle \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{find} \\ \text{SIT} & s \\ \text{BOUND\_VAR} & i \\ \text{QRESTR} & s_0 \\ \text{QSCOPE} & t \end{array}\right], \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{instantiate} \\ \text{SIT} & s_0 \\ \text{SEM\_TYPE} & \text{human} \\ \text{INST} & i \end{array}\right], \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{outscope} \\ \text{OUTER} & s \\ \text{INNER} & u \end{array}\right] \right\rangle \end{array}\right] \end{array}\right]$$

(142) Who is not sad?

(143)
$$\boxed{1}\left[\begin{array}{ll} \textit{phrase} & \\ \text{ORTH} & \text{[who, is, not, sad]} \\ \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \textit{qword} \\ \text{VAL} & \left[\begin{array}{ll} \text{SPR} & \langle\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{array}\right] \\ \text{GAP} & \langle\rangle \\ \text{STOP-GAP} & \langle\rangle \end{array}\right] \\ \text{SEM} & \left[\begin{array}{ll} \text{MODE} & \text{ques} \\ \text{TYPE} & \textbf{Bool} \\ \text{INDEX} & t \\ \text{RESTR} & \left\langle \begin{array}{l} \boxed{2}\left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{find} \\ \text{SIT} & t \\ \text{BOUND\_VAR} & i \\ \text{QRESTR} & t_0 \\ \text{QSCOPE} & \textit{scope} \end{array}\right], \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{instantiate} \\ \text{SIT} & t_0 \\ \text{SEM\_TYPE} & \text{human} \\ \text{INST} & i \end{array}\right], \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{outscope} \\ \text{OUTER} & t \\ \text{INNER} & u \end{array}\right], \\ \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{time\_present} \\ \text{SIT} & u' \\ \text{EVENT} & u'_1 \end{array}\right], \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{not} \\ \text{SIT} & u \\ \text{NEGATED\_PRED} & u' \end{array}\right], \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{sad} \\ \text{SIT} & u'_1 \\ \text{SUBJECT} & i \end{array}\right] \end{array} \right\rangle \end{array}\right] \end{array}\right]$$

With this description of *Who* we can parse the sentence (142). The phrase structure of the resulting phrase is given in (143).

This can be easily converted to TFOL formula below, after removing the situation labels (we have also removed the tense predication). We use the notation ∃? for this new quantifier.

(144) $\exists?(i : human)\ ;\ \neg sad(i)$

6.12 Converting Scope Resolved Predications to TFOL

We have described in some in examples through this chapter how to convert a list of scope-resolved predications to a Typed First Order Logic (TFOL) formula. In this section we present a general procedure to convert arbitrary scope-resolved list of predications to a simple TFOL formula.

This procedure can convert the list of predications of a well-formed sentence or a list of well-formed sentences.

We break the procedure down to six different phases. In all the phases we maintain a bag $\mathcal{B}$ of TFOL sub-formulas.

All the references to the discourse context by the **dc_obj_get** must already be resolved and the indices that refer to the same individual must be unified.

1. In the first phase, we equate the situation indices not used as arguments of tense or modification predications to their parent indices (from top to bottom of the tree representation of the immediate outscope relation).

2. In the second phase, we remove the **dc_obj_get**, and **dc_obj_add** predications.

3. In the third phase, we group all the predications with the same SIT value together. We label each group with the common SIT value of its predications.

4. In the fourth phase we process the predications in each group. We mark a predication after being processed. For every group with label x we do the following:

   (a) For every unmarked predication other than the tense and scopal predications:

      i. we create a TFOL sub-formula with exactly the same arguments as the predication with the same relation name. For instantiate predication we do not need to include the syntactic agreement argument (SYN_AGR feature).

ii. we mark the predication as processed.

(b) We make a conjunction of the sub-formulas created in the previous step.

(c) For the tense predication in the group, we create a tense sub-formula with the predicate symbol same as the relation name of the tense predication with two arguments.

(d) We set x the first argument of the tense sub-formula, we also label this sub-formula with x.

(e) We set the conjunction generated in step b as the second argument of the tense sub-formula, and we add the resulting sub-fomula to the bag $\mathcal{B}$, and we mark the tense predication as marked.

5. In the fifth phase, we repeat the following steps until there is no unmarked predication in the list.

   (a) We find an unmarked scopal predication $\mathcal{PR}(s, s_1, ..., s_n, j_1, ..., j_m)$ with SIT value s and scopal feature values $s_1, ..., s_n$, and *individual* features $j_1, ..., j_m$ such that there is no unmarked predication with SIT value s_i for $1 \leq i \leq n$

   (b) We create a sub-formula $\mathcal{P}$ with the same number of arguments as $\mathcal{PR}$. We set s as its first argument. For each scopal feature s_i, we find a conjunction sub-formula C_i labeled s_i from the bag $\mathcal{B}$. We then set C_i as the $(i + 1)$'th argument of $\mathcal{P}$, and remove C_i from the bag $\mathcal{B}$. The *individual* arguments appear in $\mathcal{P}$ with the same values and order.

   (c) If there is a sub-formula C_s with the same label s in the bag $\mathcal{B}$, then we remove it from the bag and conjoin it with $\mathcal{P}$.

   (d) We label the sub-formula $\mathcal{P}$ by s and add it to $\mathcal{B}$, and mark the predication $\mathcal{PR}(s, s_1, ..., s_n, j_1, ..., j_m)$ as processed.

6. In the sixth phase we conjoin all the sub-formulas in the bag $\mathcal{B}$. The result is the TFOL representation of the whole list of predications.

We can replace tense sub-formulas with their second arguments if we do not need the tense information. As well, we can remove the situation arguments, if we wish to have no situation argument in the final formula.

6.13 Conclusion

In this chapter we introduced the semantic features of expressions. We presented the semantic principles and constraints, variable binding conditions, that were closely related to MRS. And finally we gave an algorithm how to derive a TFOL formula representing the semantics of a well-formed sentence, or a list of sentences. Although we have used types in the feature TYPE, we have not yet shown how this information can be used in the grammar to reduce undesired ambiguity. We will do so by introducing *guards* for the HPSG grammar. This is covered in the next chapter.

Our contributions in this chapter were:

- incorporation of discourse analysis by introducing discourse predications.
- several binding conditions and constraints that are necessary to construct the correct structure of the predicate calculus formula that carries the semantics of the parsed sentence:
    - Quantifier Restriction Constraint, Quantifier Scope Constraint, Quantifier Variable Binding Condition to deal with the structuring of the formulas with quantifier.
    - Narrow Scope Variable Binding Condition that was introduced to handle the structure of the formulas with scopal modifier predicates correctly.

Chapter 7

Type Restrictions and Guards

In this chapter we introduce and use features in our HPSG type hierarchy that enable us to incorporate the type restrictions to the grammar. By using type restrictions, the grammar will not parse some semantically ill formed phrases, which results in less complexity and ambiguity.

Type restrictions are implemented by using *guards* in the grammar. A *guard* is a condition that must hold for an expression to combine with another. In section 7.1 we provide guards for specifiers and complements of a syntactic head. Section 7.3 introduces guards for modifiers. And in section 7.4 we study how guards can be used to state the restrictions of the missing elements or gaps in a phrase. Then in section 7.5 we study how guards can be used to control the formation of nominal phrases[1] with relative clauses. Finally in section 7.7 we show how type restrictions effect the process of antecedent resolution in a discourse.

7.1 Type Restrictions of Arguments as Guards

In chapter 2 we introduced the idea of type restrictions as means for ensuring the appropriateness of the semantic type of an expression for the role it plays in the larger phrase. For example the semantic type of the noun *Mary* is appropriate for the *subject* of the verb *walk* in sentence (1a) below, whereas the type of the noun *book* is inappropriate for the same verb in sentence (1b).

[1] A nominal phrase is a noun phrase that is missing its specifier, such as *book that I gave you*.

(1) a. Mary walks.

b. * The book walks.

We formulated the type restrictions of an expression as a set of ordered pairs of *roles* and *types*[2], for each syntactic role that the expression assigns to each of its constituent expressions. For example in (1a) *Mary* is an argument of the verb *walks* (and a constituent of the verb phrase) to which the role of *subject* is assigned.

In the terminology of HPSG, arguments of an expression ξ are those constituent expressions that are listed in the SPR list together with the COMPS list of the feature structure description of the expression ξ.[3]

We can encode the type restrictions together with other extra restrictions of the arguments of an expression in lists that are parallel to SPR and COMPS. We name these lists SPR_GUARDS and COMPS_GUARDS. The size of these lists are exactly the same as the size of SPR and COMPS. Each element in SPR or COMPS is in correspondence to an element in SPR_GUARDS or COMPS_GUARDS respectively, and contains a list of *guards*. Each *guard* is a boolean expression that must hold for an expression to be accepted as an argument.

The SPR_GUARDS and COMPS_GUARDS features are declared by the feature structure type *syn-cat*, as shown in table C.1 of chapter 4.

The first guard that we study in this chapter is the *type guard*. The type guard must be the first element of the list of guards for any constituent expression (including arguments). It ensures that an expression is of the specific type <expected_type>, before taking part in a larger phrase. Type guards are of the form:

(2) : <expected_type>

To use and check the guards before a phrase is formed by combining a head to its arguments, we need to refine the Head Specifier Rule and the Head Complement Rule.

Notation 7.1 In our new rules, we need to refer to the first element of a list (*head*) and the rest of its elements (*tail*). For this purpose we use the notation $\langle H|T\rangle$, where H is the first element or the *head* of the list, and T is the rest of the elements or the *tail* of the list.

[2] We refer to domain types rather than grammar entity types.

[3] The reader might remember from chapter 5 that for words and lexemes there is a special feature ARG-ST (for argument structure) that contains the list of arguments.

(3) Head Specifier Rule (HSR), Final Version:

$$\begin{bmatrix} \textit{phrase} & \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\,\rangle \\ \text{SPR\_GUARDS} & \langle\,\rangle \end{bmatrix} \end{bmatrix}$$

$$\rightarrow \boxed{1}\begin{bmatrix}\text{SEM}\begin{bmatrix}\text{TYPE } \tau\end{bmatrix}\end{bmatrix} \;\mathbf{H}\begin{bmatrix}\text{HEAD} & \begin{bmatrix}\text{PRED} & -\end{bmatrix}\\ \text{VAL} & \begin{bmatrix}\text{SPR} & \left\langle\boxed{1}\right\rangle \\ \text{SPR\_GUARDS} & \left\langle\left\langle :\sigma \mid \mathit{Tail}\right\rangle\right\rangle\end{bmatrix}\end{bmatrix}$$

where $\tau \sqsubseteq \sigma$, and all the guards in $Tail$ are satisfied.

This modification is made possible by the result that we got from theorem 3.76 of chapter 3, where the satisfaction of type restrictions is expressed using the subtype relation.

The feature structure description of the verb *walks* can then have its SPR_GUARD feature set to an appropriate value like $\langle\langle: \mathit{human}\rangle\rangle$ as shown in (4).

(4)
$$\begin{bmatrix}
\textit{word} & \\
\text{ORTH} & [\text{walks}] \\
\text{SYN} & \begin{bmatrix}
\text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{FORM} & \text{fin} \\ \text{PRED} & - \\ \text{AGR} & \textit{3sing} \\ \text{AUX} & - \\ \text{INV} & - \\ \text{GAP-T} & \text{nosubj} \end{bmatrix} \\
\text{VAL} & \begin{bmatrix} \text{SPR} & \left\langle \text{NP}_i \right\rangle \\ \text{SPR\_GUARDS} & \left\langle\left\langle :\textit{human}\right\rangle\right\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \\
\text{GAP} & \langle\rangle \\
\text{STOP-GAP} & \langle\rangle
\end{bmatrix} \\
\text{SEM} & \begin{bmatrix}
\text{MODE} & \text{prop} \\
\text{TYPE} & \mathbf{Bool} \\
\text{INDEX} & t \\
\text{RESTR} & \left\langle \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \mathbf{walk} \\ \text{SIT} & t \\ \text{WALKER} & i \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \mathbf{time\_present} \\ \text{SIT} & s \\ \text{EVENT} & t \end{bmatrix} \right\rangle
\end{bmatrix}
\end{bmatrix}$$

Then the modified HSR rule does not license the bizarre sentence (1b), because:

(5) $\mathit{book} \not\sqsubseteq \mathit{human}$

Likewise we can check the type of an expression before it combines with a head as a complement. For this we need to revise the Head Complement Rule. The new rule combines a head with n complements, and checks all the guards simultaneously.

(6) Head Complement Rule (HCR), Final Version:

$$\begin{bmatrix} \textit{phrase} & \\ \text{VAL} & \begin{bmatrix} \text{COMPS} & \langle\ \rangle \\ \text{COMPS\_GUARDS} & \langle\ \rangle \end{bmatrix} \end{bmatrix}$$

$$\rightarrow \ \mathbf{H}\begin{bmatrix} \text{VAL} & \begin{bmatrix} \text{COMPS} & \langle \boxed{1}, ..., \boxed{n} \rangle \\ \text{COMPS\_GUARDS} & \Big\langle \big\langle :\sigma_1|Tail_1 \big\rangle, ..., \big\langle :\sigma_n|Tail_n \big\rangle \Big\rangle \end{bmatrix} \end{bmatrix} \ \boxed{1} \ ... \ \boxed{n}$$

, where $\boxed{i} = \begin{bmatrix} \text{SEM} & \begin{bmatrix} \text{TYPE} & \tau_i \end{bmatrix} \end{bmatrix}$, and $\tau_i \sqsubseteq \sigma_i$, and all the guards in $Tail_i$ are satisfied, for $1 \leq i \leq n$

Now the feature structure description of the verb *paints*, for example, can be adjusted to accommodate complement guards as shown in (7).

(7)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & \text{[paints]} \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{FORM} & \text{fin} \\ \text{PRED} & - \\ \text{AGR} & \textit{3sing} \\ \text{AUX} & - \\ \text{INV} & - \\ \text{GAP-T} & \text{nosubj} \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle \text{NP}_i \rangle \\ \text{SPR\_GUARDS} & \langle\langle \text{:human} \rangle\rangle \\ \text{COMPS} & \langle \text{NP}_j \rangle \\ \text{COMPS\_GUARDS} & \langle\langle \text{:physical} \rangle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \\ \text{GAP} & \langle\rangle \\ \text{STOP-GAP} & \langle\rangle \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{prop} \\ \text{TYPE} & \textbf{Bool} \\ \text{INDEX} & t \\ \text{RESTR} & \left\langle \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{paint} \\ \text{SIT} & t \\ \text{PAINTER} & i \\ \text{PAINTED} & j \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{time\_present} \\ \text{SIT} & s \\ \text{EVENT} & t \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix}$$

With this lexical entry for the verb *paints*, HCR cannot license the verb phrase *paints a joke* in the sentence below.

(8) * He paints a joke.

The reason is that the type restrictions of the complement of the verb is not satisfied:

(9) $joke \not\sqsubseteq physical$

In section 5.4 we introduced the feature ARG-ST that contains all the arguments of a syntactic head. The value of the ARG-ST list is the concatenation of the lists SPR and COMPS. In this section we introduced guards for each argument in the SPR and COMPS list. So, it makes sense to have a parallel feature ARG-ST-GUARDS that contains the guards for all of the arguments in the ARG-ST list.

The ARG-ST-GUARDS feature is declared by the *lex-sign* feature structure type. Its value is the result of concatenating the SPR_GUARDS and COMPS_GUARDS. By using this feature we can efficiently describe the constraints on auxiliary verbs with type restrictions in section 7.2.

(10) ARG-ST-GUARDS = SPR_GUARDS $\oplus$ COMPS_GUARDS

We have not yet described why we have a list of guards for each argument rather than just one guard. Later in section 7.5 we will see a list of guards is necessary for the lexical entries of relativizers.

7.2 Type Restrictions of Copulas and Auxiliary Verbs

In section 5.5 of chapter 5 we provided an analysis of copulas and auxiliary verbs with four constraints, namely, NN-NI, NN-I, N-NI, N-I constraints. These constraints focused on the ARG-ST that made it possible to easily state that an argument is moved from SPR to COMPS in an inverted sentence. As before, we include copulas in the auxiliary verbs in our analysis, and we refer to both groups by the term *auxiliary verb*.

In inverted sentences, we need to move the corresponding guards of the moved subject to the COMPS_GUARDS list. With the use of ARG-ST-GUARDS, this is very simple. We need to modify the auxiliary verb constraints such that they use ARG-ST-GUARDS to enable the proper evaluation of guards.

Before we present the modified constraints, we need to discuss about the subject sharing in auxiliary verbs and introduce a new guard *no_comps_head*.

7.2.1 Subject Sharing and Guards

As we discussed in section 5.5, the subject of the embedded VP or predicative complement of auxiliary verbs is the same as the subject of the auxiliary verb, in a mechanism called *subject sharing* [67].

So, it makes perfect sense that the guards of the subject of the VP or predicative complement be shared with the auxiliary verb as well. That is, not only the subject of the complement, but also its guards are shared by the auxiliary verb.

Subject sharing makes the subject of the auxiliary verb dependent on its complement. In other words, before the complement is combined with an auxiliary verb, we do not have enough information about the syntactic and semantic restrictions of its subject. Thus we prefer a precedence of HCR over HSR with auxiliary verbs.

This can be easily done by the use of a special guard: *no_comps_head* that we place in the SPR_GUARDS of the auxiliary verbs. This guard checks whether the COMPS list of the syntactic head of the current rule is empty or not. It passes only if the COMPS list is empty. So it makes sure no complements are left to be combined with the head.

With the presence of this new guard in the SPR_GUARD list of auxiliary verbs, HSR can only combine subjects with auxiliary verbs if their complement has already been added by HCR. So all the information necessary, including the type restrictions of the subject of the auxiliary verb, is known.

- Non-Negated and Non-Inverted (NN-NI) Final Version:

  The feature structure description of a word that is associated with a non-negated and non-inverted auxiliary verb must be unifiable with:

  (11)
$$\left[\begin{array}{ll} \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \left[\begin{array}{ll} \textit{verb} & \\ \text{AUX} & + \\ \text{INV} & - \end{array}\right] \\ \text{VAL} & \left[\begin{array}{ll}\text{SPR} & \left\langle \boxed{1} \right\rangle\end{array}\right] \end{array}\right] \\ \text{ARG-ST} & \left\langle \boxed{1}\text{NP}\ , \left[\begin{array}{ll}\text{SYN} & \left[\begin{array}{ll}\text{VAL} & \left[\begin{array}{ll} \text{SPR} & \left\langle\boxed{1}\right\rangle \\ \text{SPR\_GUARDS} & \boxed{2} \\ \text{COMPS} & \langle\rangle \end{array}\right]\end{array}\right]\end{array}\right]\right\rangle \\ \text{ARG-ST-GUARDS} & \left\langle \boxed{2} \oplus \left\langle \text{no\_comps\_head}\right\rangle, \left\langle \text{:all}\right\rangle \right\rangle \end{array}\right]$$

- Non-Negated and Inverted (NN-I) Final Version:

  The feature structure description of a word that is associated with a non-negated but inverted auxiliary verb must be unifiable with:

(12)
$$\left[\begin{array}{ll} \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \left[\begin{array}{ll} \textit{verb} & \\ \text{AUX} & + \\ \text{INV} & + \end{array}\right] \\ \text{VAL} & \left[\begin{array}{ll} \text{SPR} & \langle\rangle \end{array}\right] \end{array}\right] \\ \text{ARG-ST} & \left\langle \boxed{1}\text{NP}\ ,\ \left[\begin{array}{ll} \text{SYN} & \left[\begin{array}{ll} \text{VAL} & \left[\begin{array}{ll} \text{SPR} & \left\langle \boxed{1} \right\rangle \\ \text{SPR\_GUARDS} & \boxed{2} \\ \text{COMPS} & \langle\rangle \end{array}\right] \end{array}\right] \end{array}\right] \right\rangle \\ \text{ARG-ST-GUARDS} & \left\langle \boxed{2}\ ,\ \left\langle \text{:all} \right\rangle \right\rangle \end{array}\right]$$

- Negated and Non-Inverted (N-NI) Final Version:

  The feature structure description of a word that is associated with a negated but non-inverted auxiliary verb must be unifiable with:

(13)
$$\left[\begin{array}{ll} \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \left[\begin{array}{ll} \textit{verb} & \\ \text{AUX} & + \\ \text{INV} & - \end{array}\right] \\ \text{VAL} & \left[\begin{array}{ll} \text{SPR} & \left\langle \boxed{1} \right\rangle \end{array}\right] \end{array}\right] \\ \text{SEM} & \left[\begin{array}{l} \text{INDEX } s \end{array}\right] \\ \text{ARG-ST} & \left\langle \boxed{1}\text{NP}\ ,\ \left[\begin{array}{ll} \textit{word} & \\ \text{ORTH} & [\text{not}] \\ \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \textit{adv-pol} \end{array}\right] \\ \text{SEM} & \left[\begin{array}{ll} \text{MODE} & \text{prop} \\ \text{INDEX} & s \\ \text{RESTR} & \left\langle \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{not} \\ \text{SIT} & s \\ \text{NEGATED\_PRED} & s_1 \end{array}\right], \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{outscope} \\ \text{OUTER} & s_1 \\ \text{INNER} & s_2 \end{array}\right] \right\rangle \end{array}\right] \end{array}\right]\ ,\ \left[\begin{array}{ll} \text{SYN} & \left[\begin{array}{ll} \text{VAL} & \left[\begin{array}{ll} \text{SPR} & \left\langle \boxed{1} \right\rangle \\ \text{COMPS} & \langle\rangle \end{array}\right] \end{array}\right] \end{array}\right] \right\rangle \\ \text{ARG-ST-GUARDS} & \left\langle \boxed{2} \oplus \left\langle \text{no\_comps\_head} \right\rangle, \left\langle \text{:all} \right\rangle, \left\langle \text{:all} \right\rangle \right\rangle \end{array}\right]$$

- Negated and Inverted (N-I) Final Version:

  The feature structure description of a word that is associated with a negated and inverted auxiliary verb must be unifiable with:

(14)
$$\begin{bmatrix}
\text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{verb} & \\ \text{AUX} & + \\ \text{INV} & + \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \end{bmatrix} \end{bmatrix} \\
\text{SEM} & \begin{bmatrix} \text{INDEX } s \end{bmatrix} \\
\text{ARG-ST} & \left\langle \boxed{1}\text{NP}\ ,\ \begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{not}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \textit{adv-pol} \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{prop} \\ \text{INDEX} & s \\ \text{RESTR} & \left\langle \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{not} \\ \text{SIT} & s \\ \text{NEGATED\_PRED} & s_1 \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{outscope} \\ \text{OUTER} & s_1 \\ \text{INNER} & s_2 \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix},\ \begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{VAL} & \begin{bmatrix} \text{SPR} & \langle \boxed{1} \rangle \\ \text{COMPS} & \langle\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix} \right\rangle \\
\text{ARG-ST-GUARDS} & \left\langle \boxed{2}\ ,\ \langle\text{:all}\rangle, \langle\text{:all}\rangle \right\rangle
\end{bmatrix}$$

7.3 Type Restrictions of Modifiers

In the first section of this chapter we described type restrictions of two kinds of arguments, namely, specifiers and complements. These are located in the valence feature of expressions. We limited the expressions that can combine with a head by HSR and HCR. There is one valence feature remaining which is MOD for modifiers, and there is one rule left that can combine a head with a modifier, which is HMR that we need to restrict in order to avoid the formation of semantically ill expressions like below.

(15) a. * a tall joke

b. * a happy television

The reason for the invalidity of the above expressions is that the adjectives used are not proper for the nominals that follow them. Or to turn the focus on the modifiers (which is the focus of HMR), the nominals are not appropriate for the modifiers. That is, *tall* requires a nominal of type *physical* and *happy* requires a nominal of type *animal* for example.

For the lexical entries of modifiers, MOD is a non-empty list that contains the feature structure description of the modified expression. We can easily include the type restriction of the modified expression by using the MOD-GUARD feature of *mod-elem*. This feature contains the guards that restrict the combination of the modifier with a modified expression.

The first element of the guard list must specify the expected type of the modified expression, like the guard lists of SPR and COMPS. As an example the feature structure description of the adjective *tall* is given below.

(16)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{tall}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \textit{adj} & \\ \text{PRED} & + \end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle \text{NP}_i \rangle \\ \text{SPR\_GUARDS} & \langle \boxed{1} \rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \left\langle \begin{bmatrix} \textit{mod-elem} & \\ \text{MODIFIED} & \text{NOM}_i \\ \text{AFTER} & - \\ \text{MOD-GUARD} & \boxed{1}\langle :\text{physical} \rangle \end{bmatrix} \right\rangle \end{bmatrix} \\ \text{GAP} & \langle\rangle \\ \text{STOP-GAP} & \langle\rangle \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{none} \\ \text{TYPE} & \text{all} \\ \text{INDEX} & s \\ \text{RESTR} & \left\langle \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{tall} \\ \text{SIT} & s \\ \text{SUBJECT} & i \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix}$$

The reason that the SPR_GUARDS is non-empty and contains the same list as MOD-GUARD is related to *subject sharing* if the modifier is preceded by a copular verb. We will discuss more about this shortly.

We need to modify the Head Modifier Rule such that it checks the guards before the phrase can be formed:

(17) Head Modifier Rule (HMR), Pre-Head, Final Version:

$$\begin{bmatrix}\textit{phrase}\end{bmatrix} \rightarrow \begin{bmatrix} \text{COMPS} & \langle\rangle \\ \text{MOD} & \left\langle \begin{bmatrix} \textit{mod-elem} & \\ \text{MODIFIED} & \boxed{1} \\ \text{AFTER} & - \\ \text{MOD-GUARD} & \langle : \sigma \mid \textit{Tail} \rangle \end{bmatrix} \right\rangle \end{bmatrix} \quad \mathbf{H}\boxed{1}\begin{bmatrix} \text{SYN} & \begin{bmatrix} \text{COMPS} & \langle\rangle \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{TYPE} & \tau \end{bmatrix} \end{bmatrix}$$

where $\tau \sqsubseteq \sigma$, and all the guards in *Tail* are satisfied.

(18) Head Modifier Rule (HMR), Post-Head, Final Version:

$$\begin{bmatrix}\textit{phrase}\end{bmatrix} \rightarrow \mathbf{H}\boxed{1}\begin{bmatrix}\text{SYN} & \begin{bmatrix}\text{COMPS} & \langle\rangle\end{bmatrix}\\ \text{SEM} & \begin{bmatrix}\text{TYPE} & \tau\end{bmatrix}\end{bmatrix} \begin{bmatrix}\text{COMPS} & \langle\rangle \\ \text{MOD} & \left\langle \begin{bmatrix}\textit{mod-elem} & \\ \text{MODIFIED} & \boxed{1}\\ \text{AFTER} & + \\ \text{MOD-GUARD} & \left\langle : \sigma \mid \textit{Tail}\right\rangle\end{bmatrix}\right\rangle\end{bmatrix}$$

where $\tau \sqsubseteq \sigma$, and all the guards in $Tail$ are satisfied.

These new rules do not license an invalid phrase like *tall joke*, however a valid phrase like *tall building* is licensed.

7.3.1 Predicative Modifiers and Subject Sharing

The guards for the specifier of predicative modifiers is the same as MOD-GUARD (as can be seen in (16)). This is for the predicative use of modifiers in which they can appear after the copular verbs such as *be*. As mentioned in section 6.10 the SPR element of adjectives (and other predicative modifiers, such as prepositional modifiers) can be shared by the auxiliary or copular verbs in *subject sharing*. And in section 7.2 we shared the subject guards of the auxiliary verb complements with the subject guards of the auxiliary verbs. So we need to pass the modified guards of the modifier to the subject guards of the auxiliary verb if it precedes the modifier. This can be easily done by setting the specifier guards of the modifiers equal to their modified guards list. Note however that SPR_GUARDS is a list of list of guards, whereas MOD-GUARD is list of guards. So the SPR_GUARDS of predicative modifiers is a list that contains another list which is exactly the same as the value of the MOD-GUARD feature.

For example the guard list ⟨*:physical*⟩ of the adjective *tall* , as shown in (16), is passed to the SPR-GUARDS of the verb *is* in the verb phrase (19a). This will prevent the sentence (19b) from being parsed, whereas the sentence (19c) is parsed.

(19) a. is tall

b. * The joke is tall.

c. The building is tall.

7.4 Type Restrictions of Gaps

As we described in section 5.6, there should be lexical rules in place that remove an argument from SPR, or COMPS list and put it in the GAP list. This was necessary to parse phrases that missed a constituent. As we mentioned in the first section the size of SPR and SPR_GUARDS must be the same. So is true for the size of COMPS and COMPS_GUARDS. It is therefore natural to also remove the guards corresponding to the SPR or COMPS list that is being removed.

To complete the semantic analysis of long distance dependencies we avoid losing the type restriction information (and other corresponding guards) we place the guards in the GAP_GUARDS list. GAP_GUARDS feature is declared by the *syn-cat* feature structure type, where the GAP feature is also declared. The size of the GAP_GUARD list is exactly the same as the size of the GAP list. GAP_GUARDS feature contains the guards of the gaps that are present in the GAP list.

(20)
$$\begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{painted}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix}\textit{verb} & \\ \text{FORM} & \text{fin} \\ \text{PRED} & - \\ \text{AUX} & - \\ \text{INV} & - \\ \text{GAP-T} & \text{nosubj}\end{bmatrix} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle \text{NP}_i \rangle \\ \text{SPR\_GUARDS} & \langle\langle \text{:human}\rangle\rangle \\ \text{COMPS} & \langle\rangle \\ \text{COMPS\_GUARDS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{bmatrix} \\ \text{GAP} & \langle \text{NP}_j\rangle \\ \text{GAP\_GUARDS} & \langle\langle\text{:physical}\rangle\rangle \\ \text{STOP-GAP} & \langle\rangle \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{MODE} & \text{prop} \\ \text{TYPE} & \textbf{Bool} \\ \text{INDEX} & t \\ \text{RESTR} & \left\langle \begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{paint} \\ \text{SIT} & t \\ \text{PAINTER} & i \\ \text{PAINTED} & j\end{bmatrix}, \begin{bmatrix}\textit{predication} & \\ \text{RELN} & \textbf{time\_past} \\ \text{SIT} & s \\ \text{EVENT} & t\end{bmatrix}\right\rangle \end{bmatrix} \end{bmatrix}$$

For example, the feature structure description of the verb *painted* that misses its complement (object) is shown in (20). The information in the GAP_GUARDS list can be used to avoid parsing a phrase like (21).

(21) * the joke which I painted

The reason is that GAP_GUARDS contains the list $\langle : physical\rangle$ that means the missing element is a *physical* object. So the semantic role of the missing element cannot be assigned to the nominal *joke* because there is a type mismatch, i.e., $joke \not\sqsubseteq physical$.

7.5 Guards for Relativizers

Relativizers are modifiers with gappy sentence complements. So their analysis of guards relies on our discussion of modifiers and gaps in the last two sections.

There are three important types in a nominal phrase with a relativizer. The first type is the type τ of the individual i that the nominal represents. The second is the type that the relativizer expects for the nominal it modifies (σ). And the third is the type of the missing constituent of the gappy complement sentence of the relativizer (ω). As an example see the nominal phrase below for example:

(22) $\underbrace{\text{student}}_{i:\tau}$ $\underbrace{\text{who}}_{\sigma=human}$ $\underbrace{\rule{1.5em}{0.4pt}}_{\omega}$ passed the course

The nominal plays two roles, first as the modified expression of the relativizer, which requires:

(23) $\tau \sqsubseteq \sigma$

, and second as the missing constituent of the gappy sentence, so the type restriction of the gap applies to the nominal:

(24) $\tau \sqsubseteq \omega$

Relativizers fall into three groups with respect to what they modify, namely:

- a human, such as *who*, *whom.*
- a non-human object or animal, such as *which.*
- both a human or non-human object, such as *that.*

With the groups we outlined above, σ can be only chosen from the set:

(25) $\{human, all \curvearrowright X - human, all\}$

We will study the type restrictions of each group separately in the following sections.

7.5.1 Human Relativizers

In this case, $\sigma = human$. Since we assumed in section 3.3 that *human* with all of its super types are triangular types, any arbitrary set is either disjoint, or a subtype or a super-type of *human*. As (23), and (24) indicate, ω cannot be disjoint with σ, and it must be true that either $\sigma \sqsubseteq \omega$ or $\omega \sqsubseteq \sigma$.

If we assume $least(\sigma, \omega)$ is the least general type among σ and ω, then (23), and (24) can be condensed to:

(26) $\tau \sqsubseteq least(\sigma, \omega)$

$least(a, b)$ is also a guard that ensures that either $a \sqsubseteq b$ or $b \sqsubseteq a$. Otherwise it should fail.

The feature structure description of *who* with type restrictions can be presented in the lexicon as follows:

(27)
$$\left[\begin{array}{ll} \textit{word} & \\ \text{ORTH} & [\text{who}] \\ \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \textit{sconj} \\ \text{VAL} & \left[\begin{array}{ll} \text{SPR} & \langle\rangle \\ \text{COMPS} & \left\langle \text{S}_t \left[\begin{array}{ll} \text{INV} & - \\ \text{GAP} & \left\langle \boxed{1}\text{NP}_i \left[\text{AGR}\ \boxed{2}\right]\right\rangle \\ \text{GAP-GUARDS} & \left\langle\left\langle :\omega|Tail\right\rangle\right\rangle \end{array}\right]\right\rangle \\ \text{COMPS\_GUARDS} & \left\langle\left\langle :\textbf{Bool},\ least(human,\omega)=\rho\right\rangle\right\rangle \\ \text{MOD} & \left\langle \left[\begin{array}{ll} \textit{mod-elem} & \\ \text{MODIFIED} & \text{NOM}_i\left[\text{AGR}\ \boxed{2}\right] \\ \text{AFTER} & + \\ \text{MOD-GUARD} & \left\langle :\rho|Tail\right\rangle \end{array}\right]\right\rangle \end{array}\right] \\ \text{GAP} & \langle\rangle \\ \text{STOP-GAP} & \left\langle\boxed{1}\right\rangle \end{array}\right] \\ \text{SEM} & \left[\begin{array}{ll} \text{MODE} & \text{none} \\ \text{TYPE} & \textbf{Bool} \\ \text{INDEX} & t \\ \text{RESTR} & \langle\rangle \end{array}\right] \end{array}\right]$$

The complement guard list of *who* is a list with two elements. The first element is necessarily the type of the complement, which is **Bool** as the complement is a sentence. The second guard calculates the least general type among *human* and ω, and the result is saved in ρ. Later, ρ is used as the modified guard of the modifier. Other guards of the gap are also included in the guard list of the modified element, because the modified expression plays the same role as the gap, so all the remaining guards corresponding to the gap must also be satisfied by the modified expression.

The fact that HMR requires an empty list of complements for the modifier guarantees that the complement sentence is combined first with *who* by HCR and ρ is known as a result, before using *who* as a modifier.

Using this lexical entry, the grammar can parse a phrase like (28a) or (28b) but not a phrase like (28c).

(28) a. student who is on the roof.

b. student who passed the course.

c. * car who I painted.

7.5.2 Non-human Relativizers

In this case we have $\sigma = \mathit{all} \curvearrowright X - \mathit{human}$. Restrictions (23), and (24) can be re-written as:

(29) $\tau \sqsubseteq \mathit{all} \curvearrowright X - \mathit{human}$

(30) $\tau \sqsubseteq \omega$

Let us define a function $\mathit{snh}(\omega)$ for calculating the specializable subtype of ω that is disjoint from *human* by:

```
function snh($\omega$:$\mathbf{BasTyp}$) : $\mathbf{BasTyp}$
/* $Y, Y'$ are free variables not used anywhere else */
begin
  if is_natural($\omega$) then return $\omega \curvearrowright Y - \mathit{human}$
  else if $\omega = \omega' - \mathit{human}$, for a natural $\omega'$ then return $\omega' \curvearrowright Y - \mathit{human}$
  else if $\omega = \omega' \curvearrowright Y$ or $\omega = \omega' \curvearrowright Y - \mathit{human}$ then return $\omega' \curvearrowright Y' - \mathit{human}$
  end if
end
```

Program 7.1: The *snh* function to calculate the specializable non-human subtype

Theorem 7.2 Restrictions (29) and (30) are equivalent to:

(31) $\tau \sqsubseteq \mathit{snh}(\omega)$

(The proof of this theorem is provided in appendix B.)

It can be observed that if $\omega \sqsubseteq human$, the return value of $snh(\omega)$ cannot be specialized to a non-bottom type. So we would like this function to act as a guard too, and fail if $\omega \sqsubseteq human$.

Then the feature structure description of *which* with type restrictions can be defined in the lexicon by:

(32)
$$\left[\begin{array}{ll}
\multicolumn{2}{l}{\textit{word}} \\
\text{ORTH} & [\text{which}] \\
\text{SYN} & \left[\begin{array}{ll}
\text{HEAD} & \textit{sconj} \\
\text{VAL} & \left[\begin{array}{ll}
\text{SPR} & \langle\rangle \\
\text{COMPS} & \left\langle \text{S}_t \left[\begin{array}{ll}
\text{INV} & - \\
\text{GAP} & \left\langle \boxed{1}\text{NP}_i \left[\text{AGR } \boxed{2}\right]\right\rangle \\
\text{GAP-GUARDS} & \left\langle\left\langle :\omega | Tail \right\rangle\right\rangle
\end{array}\right]\right\rangle \\
\text{COMPS\_GUARDS} & \left\langle\left\langle :\textbf{Bool},\ snh(\omega) = \rho \right\rangle\right\rangle \\
\text{MOD} & \left\langle \left[\begin{array}{ll}
\multicolumn{2}{l}{\textit{mod-elem}} \\
\text{MODIFIED} & \text{NOM}_i \left[\text{AGR } \boxed{2}\right] \\
\text{AFTER} & + \\
\text{MOD-GUARD} & \left\langle :\rho | Tail \right\rangle
\end{array}\right]\right\rangle
\end{array}\right] \\
\text{GAP} & \langle\rangle \\
\text{STOP-GAP} & \left\langle \boxed{1} \right\rangle
\end{array}\right] \\
\text{SEM} & \left[\begin{array}{ll}
\text{MODE} & \text{none} \\
\text{TYPE} & \textbf{Bool} \\
\text{INDEX} & t \\
\text{RESTR} & \langle\rangle
\end{array}\right]
\end{array}\right]$$

Using this lexical entry the grammar can license phrases like (33a), and (33b) but not semantically ill phrases like (33c), and (33d).

(33) a. car which is on the street

b. car which you painted

c. * car which speaks to me

d. * student which speaks to me

7.5.3 Neutral Relativizers

A neutral relativizer like *that* can modify both a human or a non-human individual. The analysis of this relativizer is the easiest. We should simply have $\tau \sqsubseteq \omega$ and no other restriction is needed. The feature structure description of *that* is given below.

(34)
$$\left[\begin{array}{ll}
\textit{word} & \\
\text{ORTH} & [\text{that}] \\
\text{SYN} & \left[\begin{array}{ll}
\text{HEAD} & \textit{sconj} \\
\text{VAL} & \left[\begin{array}{ll}
\text{SPR} & \langle\rangle \\
\text{COMPS} & \left\langle \text{S}_t \left[\begin{array}{ll}
\text{INV} & - \\
\text{GAP} & \left\langle \boxed{1}\text{NP}_i \left[\text{AGR}\ \boxed{2}\right]\right\rangle \\
\text{GAP-GUARDS} & \left\langle\left\langle :\omega|Tail\right\rangle\right\rangle
\end{array}\right]\right\rangle \\
\text{COMPS\_GUARDS} & \left\langle\left\langle :\mathbf{Bool}\right\rangle\right\rangle \\
\text{MOD} & \left\langle\left[\begin{array}{ll}
\textit{mod-elem} & \\
\text{MODIFIED} & \text{NOM}_i\left[\text{AGR}\ \boxed{2}\right] \\
\text{AFTER} & + \\
\text{MOD-GUARD} & \left\langle :\omega|Tail\right\rangle
\end{array}\right]\right\rangle
\end{array}\right] \\
\text{GAP} & \langle\rangle \\
\text{STOP-GAP} & \left\langle\boxed{1}\right\rangle
\end{array}\right] \\
\text{SEM} & \left[\begin{array}{ll}
\text{MODE} & \text{none} \\
\text{TYPE} & \mathbf{Bool} \\
\text{INDEX} & t \\
\text{RESTR} & \langle\rangle
\end{array}\right]
\end{array}\right]$$

This will license the phrase (35a), but the phrase (35b) cannot be parsed.

(35) a. student that speaks to me

b. * car that speaks to me

7.6 A Note on Coordination Rules

In this thesis we only studied the coordination of sentences and countable noun phrases. Sentences do not have any arguments, so the SPR_GUARD, and COMPS_GUARDS for sentences are empty. Moreover, sentences cannot be used as modifiers. So sentences do not have any guards, and the coordination rule of sentences does not need to say anything about type restrictions. The countable nouns that we studied in this thesis do not have any complements. However, they have a non-empty SPR list that contains a determiner. We have used meaningful types only for nouns and sentences. For all other parts of speech, including determiners the type has been simply *all*. So the countable nouns should have a non-empty SPR_GUARD list that is simply equal to <<: *all* >>. And the coordination rule for countable noun phrases does not need any change either.[4]

[4] If coordination is used on constituents with arguments or on modifiers, the guards for the mother phrase must be the result of the combination of the guards of the coordinated constituents. However for the purposes of conciseness we do not study such cases in this thesis.

7.7 Effective Use of Guards with Antecedent Resolution

Using type restrictions in parsing standalone sentences reduces the number of interpretations by rejecting semantically ill ones. In this section we show how type restrictions can be used with multiple sentences in a discourse with the process of antecedent resolution.

The fact that we have used specializable types for pronouns reduces the problem of checking a type restriction involving a pronoun to a satisfiability problem with a variable. If the restriction is satisfied the value of the answer to the variable is carried by the semantics of the pronoun in the entire discourse, as a result of the unification of the variable to its answer. This is particularly helpful in antecedent resolution. Consider the sentences below for example.

(36) a. I heard a song in a car.

b. It was red.

The semantic restriction of the sentence (36a) introduces two context individuals to the discourse context (DC):

$$\mathbf{DC} = \left\langle \text{a song, a car} \right\rangle = \left\langle \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{instantiate} \\ \text{SIT} & s_1 \\ \text{SYN\_AGR} & \left[\begin{array}{ll} \textit{3sing} & \\ \text{GEND} & \textit{neut} \end{array}\right] \\ \text{SEM\_TYPE} & \text{song} \\ \text{INST} & i \end{array}\right], \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{instantiate} \\ \text{SIT} & s_2 \\ \text{SYN\_AGR} & \left[\begin{array}{ll} \textit{3sing} & \\ \text{GEND} & \textit{neut} \end{array}\right] \\ \text{SEM\_TYPE} & \text{car} \\ \text{INST} & j \end{array}\right] \right\rangle$$

The semantic restriction of the second sentence is:

$$\left\langle \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{dc\_obj\_get} \\ \text{SIT} & t \\ \text{OBJ} & k \\ \text{INSTPRED} & t_1 \end{array}\right], \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{instantiate} \\ \text{SIT} & t_1 \\ \text{SYN\_AGR} & \left[\begin{array}{ll} \textit{3sing} & \\ \text{GEND} & \textit{neut} \end{array}\right] \\ \text{SEM\_TYPE} & \tau \curvearrowright \text{Y-human} \\ \text{INST} & k \end{array}\right], \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{time\_past} \\ \text{SIT} & t' \end{array}\right], \left[\begin{array}{ll} \textit{predication} & \\ \text{RELN} & \textbf{red} \\ \text{SIT} & t' \\ \text{SUBJECT} & k \end{array}\right] \right\rangle$$

, where τ is an answer to:

(37) $all \curvearrowright X - human \sqsubseteq physical$

So τ is a most general subtype of *physical* that is disjoint with *human.* Possible values of τ include *inanimate*, *beluga*, *dog*, *bird.*

For any of the possible values of τ, the **dc_obj_get** predication cannot retrieve i from the discourse context for k, because the requirements of the **instantiate** predications for i and k are incompatible.

In other words, if the pronoun *it* refers to *a song*, then its expected type is *song*, so we must have:

(38) $\tau \curvearrowright Y - human \sqsubseteq song$

which is not satisfiable for any possible value of τ. In fact the two types are disjoint.

So the only choice for the antecedent of the pronoun *it* will be *a car*, which requires

(39) $\tau \curvearrowright Y - human \sqsubseteq car$

which is satisfiable for $\tau = inanimate$.

Thus the number of the candidate interpretations of the second sentence (and the whole discourse) is reduced from two to one, by rejecting the implausible interpretation that *the song* was *red*.

In general, a pronoun with the semantic type ω can be resolved to an antecedent of type σ, if the following restriction holds or is satisfiable.

(40) $\omega \sqsubseteq \sigma$

7.8 Conclusion

Our contribution in this chapter was the incorporation of the type restrictions into the HPSG formalism. Using the type restrictions will enable us to reduce the undesired ambiguity by restricting the licensing of the phrases by the modified HSR, HCR, and HMR. The ambiguity is also reduced in the process of antecedent resolution where the possible referents of pronouns are restricted by the appropriateness of the types of pronouns and their candidate referents.

In the next chapter we will discuss how this grammar can be implemented, by using Prolog on top of ProFIT and some ideas from CHRG.

Part III: Implementation and Application

In the final part of this thesis we discuss the implementation of the system, and show how the user interface of our system can be used. We present a simple example to show how the system can be applied.

Chapter 8

Implementation and Application

In this chapter we describe how we have implemented the grammar and a system that uses this grammar to parse simple software specifications.

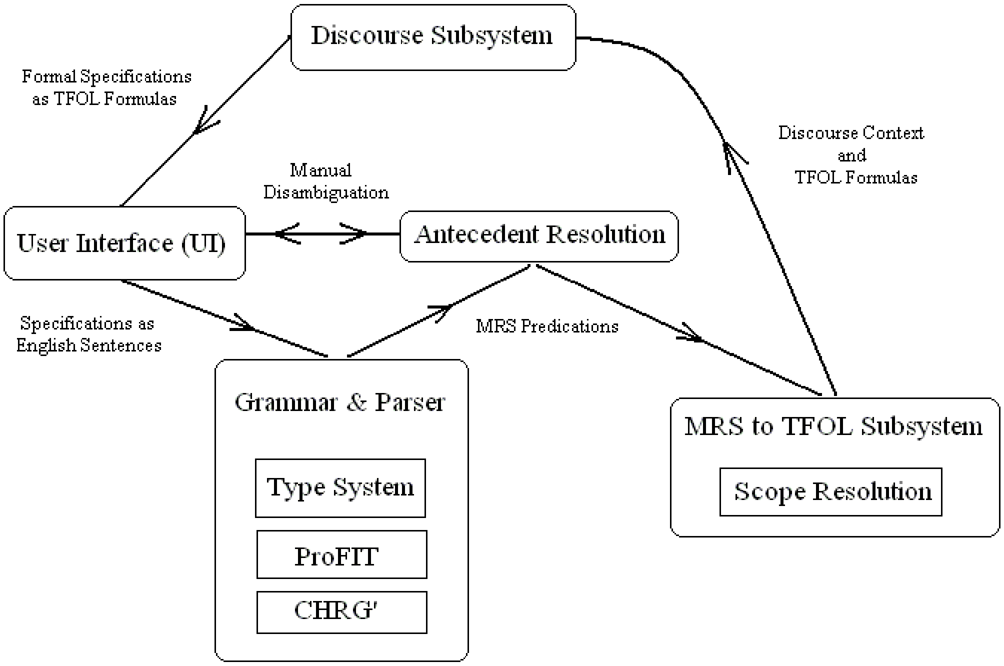


Figure 8.1: General outline of the system implementation

Our system consists of:

- the type system that we developed in chapters 2, and 3

- a User Interface (UI) to get user input
- a subsystem to maintain the current discourse context and the TFOL representation of the sentences already processed.
- the grammar and a parser to parse the sentences entered by the user
- an antecedent resolution subsystem, that communicates with the user through a user interface in case manual disambiguation is needed
- a subsystem to convert the flat predication representation to TFOL representation, which contains a scope resolution mechanism.

We chose to use the Prolog language [15] as the main language of our system. In section 8.1 we describe the reason for this choice.

We have already provided pseudo code for our type system in chapter 3, the implementation is just a conversion of the pseudo code to a concrete language, in this case, Prolog.

For the user interface we chose to use Java to avoid dependencies over UI packages in Prolog that are not as standard as Java.

Maintaining the current discourse context and the TFOL formulas so far is very straightforward by using Prolog dynamic predicates.

Antecedent resolution is also straightforward, and involves a few calls to the type system to perform subtype checking. In case the user feedback is needed the UI is used to let the user choose the proper antecedent.

In section 1 we describe the implementation of the grammar and the parser. Then in section 2 we describe the user interface of our system. Finally in section 3, we propose some potential applications for our system.

8.1 Implementation of the Grammar and the Parser

As mentioned in chapter 4, HPSG is a unification based or constraint based formalism. So we prefer a language that facilitates the use of unification and constraints. For this reason we chose Prolog language as the base to implement the grammar. We used SWI Prolog [74] for being freely available, its efficiency, and its large collection of libraries.

HPSG heavily uses feature structures, and for this we used the ProFIT implementation [30] that was used as part of the ATTEMPTO project [35, 33].

8.1.1 Brief Syntax of ProFIT

In ProFIT, type labels in feature structure (descriptions) are specified by a '<' prefix. Feature names and feature values are separated by a '!' operator, and the conjunction or unification operator is '&'.

As an example consider the syntactic segment of the word feature structure description[1] of the determiner *a* below.

(1)
$$\left[\begin{array}{ll} \textit{word} & \\ \text{ORTH} & [\text{a}] \\ \text{SYN} & \left[\begin{array}{ll} \text{HEAD} & \left[\begin{array}{ll} \textit{det} & \\ \text{AGR} & \textit{3sing} \\ \text{COUNT} & + \end{array}\right] \\ \text{VAL} & \left[\begin{array}{ll} \text{SPR} & \langle\rangle \\ \text{COMPS} & \langle\rangle \\ \text{MOD} & \langle\rangle \end{array}\right] \\ \text{GAP} & \langle\rangle \\ \text{STOP-GAP} & \langle\rangle \end{array}\right] \end{array}\right]$$

This feature structure description can be represented in ProFIT by the following structure.

```
<word &
  orth![a] &
  syn!(
    head!(
      <det &
      agr!@a3sing &
      count!<plus
    ) &
    val!(
      spr![] &
      comps![] &
      mod![]
    ) &
    gap![] &
    stop_gap![]
  )
```

[1] We use feature structure description here because the value of the GEND feature in *3sing* is underspecified.

`@a3sing` is a template in ProFIT that we use for *3sing*. A template is simply an abbreviation of another structure, which should be defined earlier. For example `a3sing` is defined by the following ProFIT statement, where `t3sing` is the type that we use for *3sing*.[2]

```
a3sing :=  <t3sing & per!3 & num!<sg .
```

Consistent to the style introduced by [30], the values of the boolean type (+, −) are defined as types themselves, that is, `plus` and `minus` are defined as two distinct subtypes of the type `bool`. Similarly `sg` and `pl` for singular and plural respectively, are defined as two subtypes of the `num_enum` type. In general if the values of a type are specifically from a known range of enumerated items, each of the items is defined as a subtype.

Describing the complete syntax of the ProFIT language is not our goal in this section and is in fact outside the scope of the thesis. The interested reader can refer to [30] for more information.

8.1.2 The Parser and CHRG′

The implementation of ATTEMPTO uses a top-down parser. However, we used a bottom up implementation to parse all the sub-phrases in parallel and to provide all the alternative interpretations. Also we had a goal of not distancing from a constraint based parser implementation, Constraint Handling Rules Grammar (CHRG) [14] that is implemented on top of Constraint Handling Rules (CHR) [32].

At first we wanted to use CHRG combined with ProFIT directly, however we encountered a limitation as a result of the dependency of CHRG on CHR. The problem is with the underspecification in lexical entries and licensed phrases and efficiency.

For efficiency we have decided to use feature structure descriptions for words and phrases rather than word or phrase structures. For example, the CASE feature of a proper noun like *John* is left underspecified. This reduces the memory needed to store the information of this name to half. Otherwise we would have to provide lexical entries with two distinguished CASE values or lexical rules to create two modified copies of a proper noun lexical entry, one with CASE *acc*, and the other with CASE *nom*.

On the other hand, we would like an underspecified structure to be unifiable with a more specific version of the structure. To use the built-in Prolog unification mechanism, we need

[2] Identifiers (including type names, and template names) in ProFIT cannot start with a digit.

to model underspecification by free variables that carry implicit universal quantifiers with them.

If grammar rules are to be modeled by CHR rules, as done in CHRG, then word feature structure descriptions and phrase feature structure descriptions need to be modeled by constraints in the constraint store. However, free variables in the constraints of the CHR constraint store do not have the universally quantified semantics. That is we cannot enjoy the built-in unification mechanism of Prolog at the same time as the constraint handling mechanism of CHR with the efficiency that is gained by underspecification in the lexicon.

As an example of this problem, consider the following sentence and let us focus on the Head Specifier Rule (HSR). The feature structure description of the the verb *walks*, requires a CASE *nom* specifier. By the HSR the CASE value of the specifier of *walks* and the proper noun *John* need to be unified, which is impossible in CHR, and hence in CHRG.

(2) John walks.

The same kind of problem happens with the underspecified GEND value of the *3sing* agreement features. If CHRG is to be used then each lexical entry (or the lexical rules that operate on them) must provide specific gender values for the feature GEND. This will double the amount of the memory needed to handle words and phrases. When combined with CASE feature and other features that are usually underspecified this creates a big burden in terms of the amount of memory needed.

For this reason we have implemented a new CHR implementation that we call CHR′ based solely on Prolog, and a new CHRG that we call CHRG′ on top of CHR′. Other than allowing the underspecified constraints, CHRG′ is semantically a subset of CHRG that only provides the propagation operator (::>), which suffices to implement a bottom up grammar.

To avoid an extra mechanism to enforce grammar principles we apply all the principle to each rule, which makes the rule longer than its HPSG version. For example the HSR rule in our implementation is provided by the piece of code shown in figure 8.2.

Tags are implemented by free Prolog variables. These variables are unified with the expression that follow them after the '&' operator, and can be reused elsewhere in the same statement.

`exp` is a CHRG′ constraint, with arity 4. This constraint is used to represent the feature structure (description) of an expression with some extra information, namely, the name of the rule that licensed it, its children in the phrase structure tree, and the list of words

```
exp(TAG1 & orth!EntryList1 & sem!(type!TYPE1 & restr!RESTR1) &
    syn!(gap!GAP1 & gap_guards!GAP1_Guards), _, _, _),
exp(HEAD & orth!EntryList2 & syn!(head!(TAG2 & pred!<minus) &
    sem!(SEM1 & type!TYPE2 & restr!RESTR2 & mode!MODE & index!INDEX ) &
    val!(spr![TAG1] & spr_guards![SprGuards] &
         comps!COMPS & comps_guards!CompsGuards &
         mod!MOD)
    & gap!GAP2 & gap_guards!GAP2_Guards & stop_gap![]) & , _, _, _)
  ::>
    and_list([
      compatible_form(TAG2),
      validateGuards(HEAD, TYPE1, SprGuards),
      copy_sem(SEM1, SEM2, RESTR),
      append(GAP1, GAP2, GAP0),
      append(GAP1_Guards, GAP2_Guards, GAP_Guards0),
      list_subtraction(GAP0, GAP_Guards0, [], GAP, GAP_Guards),
      append(EntryList1, EntryList2, EntryList),
      append(RESTR1, RESTR2, RESTR)
    ])
    |
    exp(<phrase & orth!EntryList & syn!(head!TAG2 & val!(
        spr![] & spr_guards![] & comps!COMPS & comps_guards!CompsGuards & mod!MOD)
        & gap!GAP & gap_guards!GAP_Guards & stop_gap![]) & sem!SEM2
         , head_specifier_rule, [TAG1, HEAD], [EntryList1, EntryList2]).
```

Figure 8.2: The implementation of the HSR rule with CHRG$'$ and ProFIT

corresponding to each child. The feature structure is the first argument of `exp`. The second argument is a Prolog atom that is the name of the grammar rule that licensed the expression. The third argument is the list of subconstituent expressions (child expressions). And finally the fourth argument is the list of list of words for each child expression.

Like CHRG we have used the '|' operator to separate the *guard* with the output constraint. A guard in CHRG is a Prolog expression that must evaluate to true before the rule can be applied.

The important duties of the guard of the above CHRG′ rule are maintaining the list of semantic restrictions (predications), ensuring the semantic compositionality principle, verifying the specifier guards, realizing the gap principle, and maintaining the list of words for the mother phrase. The `compatible_form` predicate ensures that the head is finite for verbs. And `copy_sem(SEM1, SEM2, RESTR)` predicate just copies the semantic feature structure description `SEM1` into `SEM2` while leaving the semantic restriction feature of the copy an open variable `RESTR`, which is later filled by the last `append` predicate that implements the semantic compositionality principle.

8.1.3 Scope Resolution

In our implementation we have used a simple backtracking solution that given an outscope relation graph with the binding conditions and constraints, tries all possible combinations of sub-trees to build the final formula tree. Although this approach works fine for small sentences and suits well for academic purposes, it may not scale well to longer sentences.

Very efficient scope resolution algorithms are developed by Niehren et al. [57], Fuchss et al. [36], which can be adjusted and replace the current scope resolution algorithm to gain more efficiency.

8.2 The User Interface

The main window of the user interface is partitioned into two main panels. The left hand side panel shows the English specifications that are entered by the user and are parsed or being parsed. Some information are provided with smaller font between each word. This information includes:

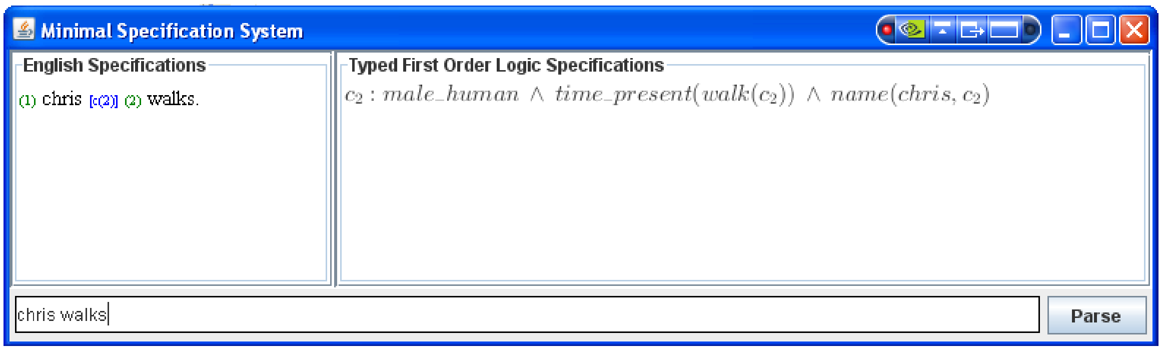
- The location of the word in the whole text, which is a natural number starting with one.
- The constant symbol that is associated with the context object that the word represents, if the word is a noun or a pronoun.

The words in this section are capable of being highlighted and responding to clicks, which as we will see later are helpful for antecedent resolution, or informing the user about an error.

The panel in the right contains the TFOL formulas that are translated from the English specifications. Each formula in the right is associated with the equivalent English sentence in the left with the same order.

The bottom section of the main window contains a text field to enter a new English sentence, and a button that the user should click in order for the typed sentence to be parsed. If the parse is successful the result will be added to the two specification panels.

Figure 8.3 shows the main window of the user interface with the parse result of the sentence *chris walks*.


Minimal Specification System
English Specifications
Typed First Order Logic Specifications
Parse

(1) chris [c(2)] (2) walks.

$c_2 : male\_human \wedge time\_present(walk(c_2)) \wedge name(chris, c_2)$

chris walks

Figure 8.3: The main window of the user interface

8.2.1 Automatic Antecedent Resolution

The system is able to resolve antecedents automatically if the correct antecedent can be inferred from the type restrictions of the verbs and the types of the nouns involved. For example suppose *teach* is a verb that is defined only for professors. And *Veronica* is defined in the lexicon to be an individual of type *professor*, and *Cindy* is an individual of type human. Suppose the input English sentences to the system are:

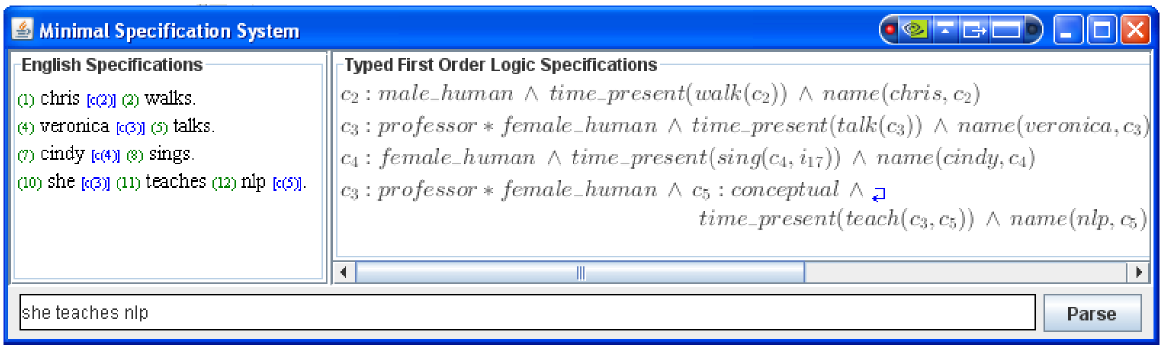
(3) a. Chris walks.

b. Veronica talks.

c. Cindy sings.

d. She teaches NLP.

Without considering types, the antecedent of the pronoun *She* could refer to both *Veronica* or *Cindy*. But type restrictions will narrow the choice of the antecedent to only *Veronica*. This is correctly handled in our system, as shown in figure 8.4.



Minimal Specification System

English Specifications

(1) chris [c(2)] (2) walks.
(4) veronica [c(3)] (5) talks.
(7) cindy [c(4)] (8) sings.
(10) she [c(3)] (11) teaches (12) nlp [c(5)].

Typed First Order Logic Specifications

$c_2 : male\_human \wedge time\_present(walk(c_2)) \wedge name(chris, c_2)$

$c_3 : professor * female\_human \wedge time\_present(talk(c_3)) \wedge name(veronica, c_3)$

$c_4 : female\_human \wedge time\_present(sing(c_4, i_{17})) \wedge name(cindy, c_4)$

$c_3 : professor * female\_human \wedge c_5 : conceptual \wedge$ ↵
$time\_present(teach(c_3, c_5)) \wedge name(nlp, c_5)$

she teaches nlp

Figure 8.4: Automatic antecedent resolution

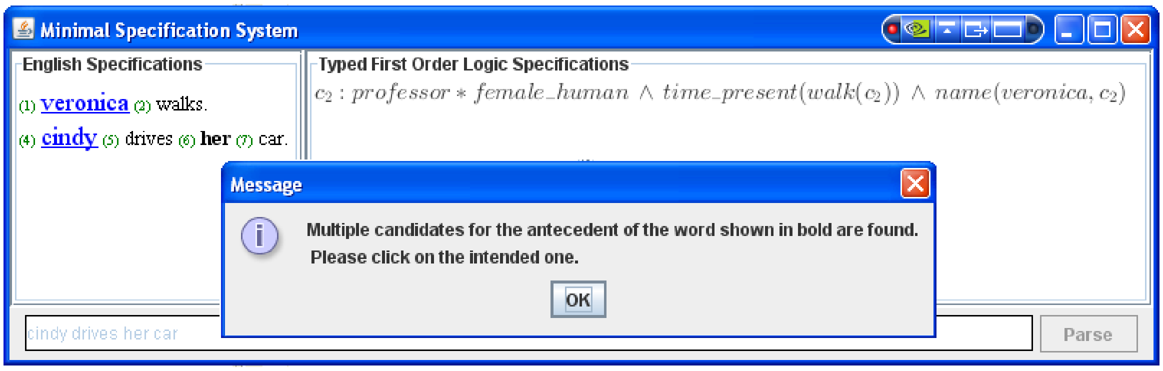
8.2.2 Manual Antecedent Resolution

If type restrictions cannot single out an antecedent for a reference, then the referent and the candidate antecedents are shown in the English specification panel in bold font, and the user is asked to click on the appropriate antecedent. This happens with the following input sentences, where the antecedent of the possessive pronoun *her* cannot be automatically determined:

(4) a. Veronica walks.

b. Cindy drives her car.

The user interface looks like what is shown in figure 8.5. The candidate antecedents are bold and underlined, while the referent is just bold. The user must click on one of the candidates for the parse to continue.


Minimal Specification System
English Specifications
Typed First Order Logic Specifications
Message
OK
Parse

(1) **veronica** (2) walks.

(4) **cindy** (5) drives (6) **her** (7) car.

$c_2 : professor * female\_human \wedge time\_present(walk(c_2)) \wedge name(veronica, c_2)$

Multiple candidates for the antecedent of the word shown in bold are found.
Please click on the intended one.

cindy drives her car

Figure 8.5: Manual antecedent resolution

8.2.3 Manual Choice among Multiple Readings

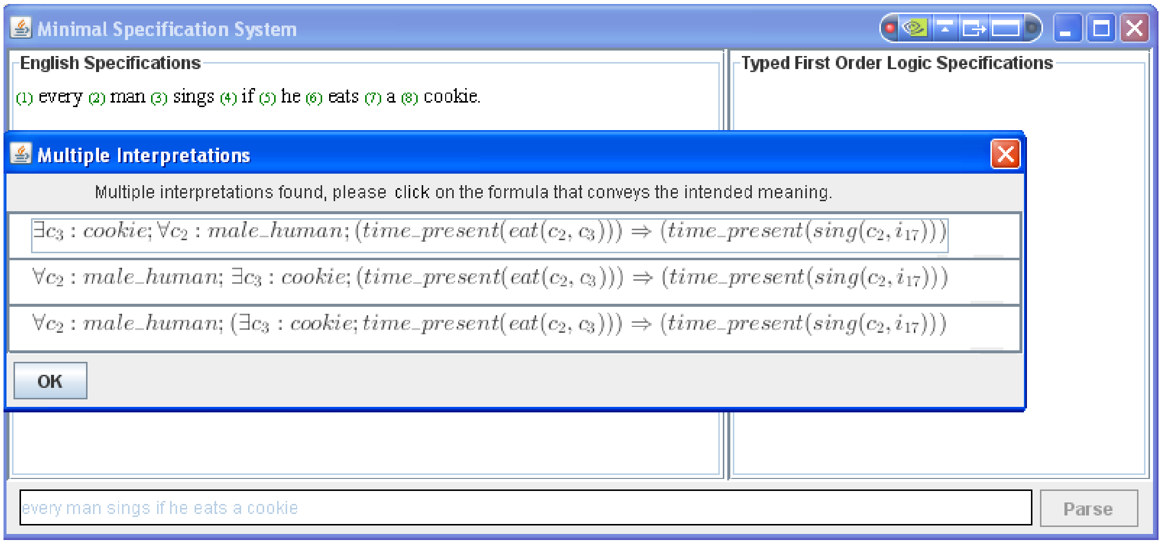
It is possible that an English sentence has more than one readings even after antecedent resolution. Multiple readings could be due to different orderings of quantifiers, or different structuring of the translated TFOL formulas. In such cases, all the possible TFOL readings of the sentence are presented to the user, and the user should select the intended reading. An example is the following sentence, that has multiple TFOL readings due to quantifiers and different formula structuring. The user interface of this situation is shown in figure 8.6.

(5) Every man sings if he eats a cookie.

In our system we simplify the output formulas by converting them to a canonical form and remove the duplicate interpretations.

8.2.4 Handling New Lexical Entries

The system is capable of adding new lexical entries to its lexicon. This can happen if a word that is not defined in the lexicon is encountered in a sentence. We have only implemented



Minimal Specification System
English Specifications
Typed First Order Logic Specifications
Multiple Interpretations
OK
Parse

(1) every (2) man (3) sings (4) if (5) he (6) eats (7) a (8) cookie.

Multiple interpretations found, please click on the formula that conveys the intended meaning.

$\exists c_3 : cookie; \forall c_2 : male\_human; (time\_present(eat(c_2, c_3))) \Rightarrow (time\_present(sing(c_2, i_{17})))$

$\forall c_2 : male\_human; \exists c_3 : cookie; (time\_present(eat(c_2, c_3))) \Rightarrow (time\_present(sing(c_2, i_{17})))$

$\forall c_2 : male\_human; (\exists c_3 : cookie; time\_present(eat(c_2, c_3))) \Rightarrow (time\_present(sing(c_2, i_{17})))$

every man sings if he eats a cookie

Figure 8.6: Manual choice of the intended reading among multiple interpretations

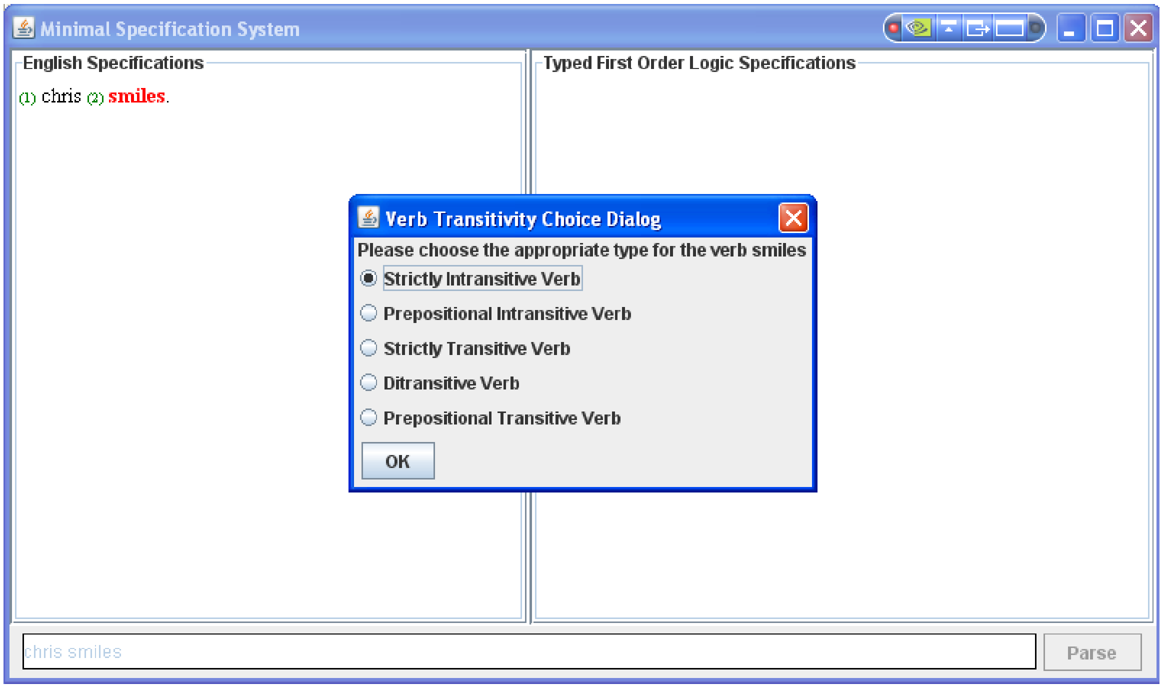
the addition of new verbs and new nouns. However, addition of other parts of speech is indeed easy.

Upon encountering an unknown word, the system asks the user if it is a verb or a noun. For verbs, the user is asked to choose the specific type of the verb among Strictly Intransitive, Prepositional Intransitive, Strictly Transitive, Ditransitive and Prepositional Transitive. This is shown in figure 8.7 for the following sentence, where the verb *smiles* has not been defined before.

(6) Chris smiles.

Then the user is asked to fill the details of the base form of the new verb in the form shown in figure 8.8. The user can enter the predications associated with the base form of the verb. Up to 3 predications are supported in the current implementation of the user interface, however there is no theoretical neither practical limit for this. For every predications, we have a maximum of 3 arguments with their roles. User should specify the argument type by selecting one of “None”, “Individual”, “Situation” options. If “Individual” is selected, the argument will be an individual index. If “Situation” is selected, the argument will be a situation index. “None” should be selected for a value that is not an index, or for a role that is not used.

After filling the required information, a new lexical entry is added, and all forms of the



Minimal Specification System

English Specifications

(1) chris (2) **smiles**.

Typed First Order Logic Specifications

Verb Transitivity Choice Dialog

Please choose the appropriate type for the verb smiles

● Strictly Intransitive Verb

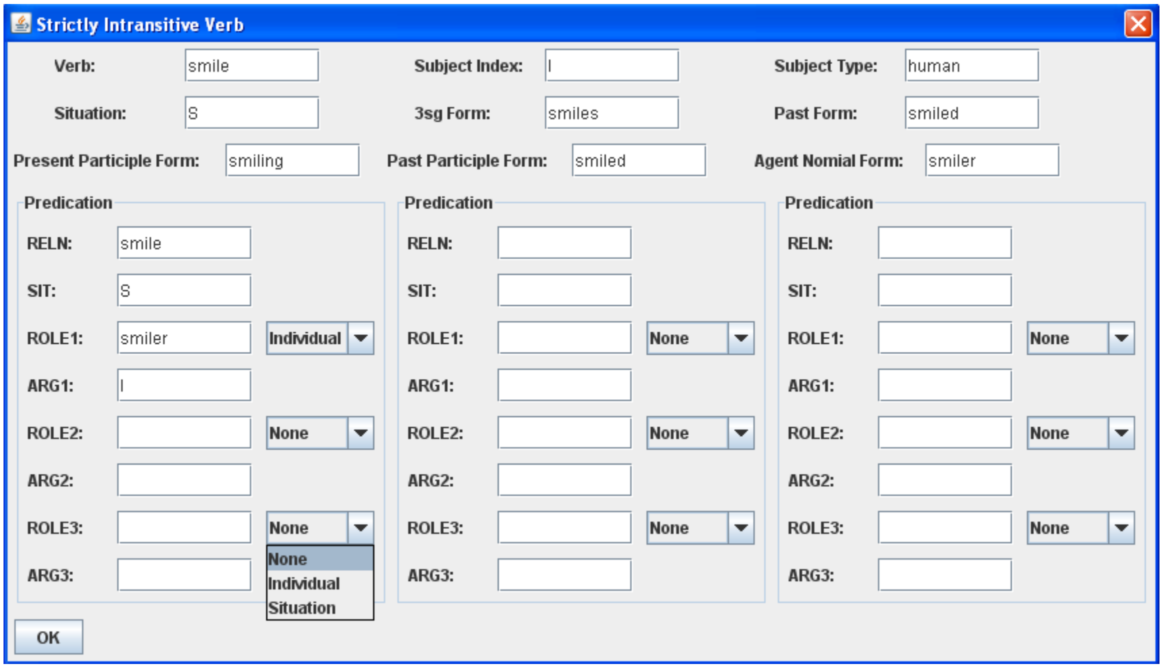
○ Prepositional Intransitive Verb

○ Strictly Transitive Verb

○ Ditransitive Verb

○ Prepositional Transitive Verb

Figure 8.7: Asking the transitivity of a new verb



Strictly Intransitive Verb

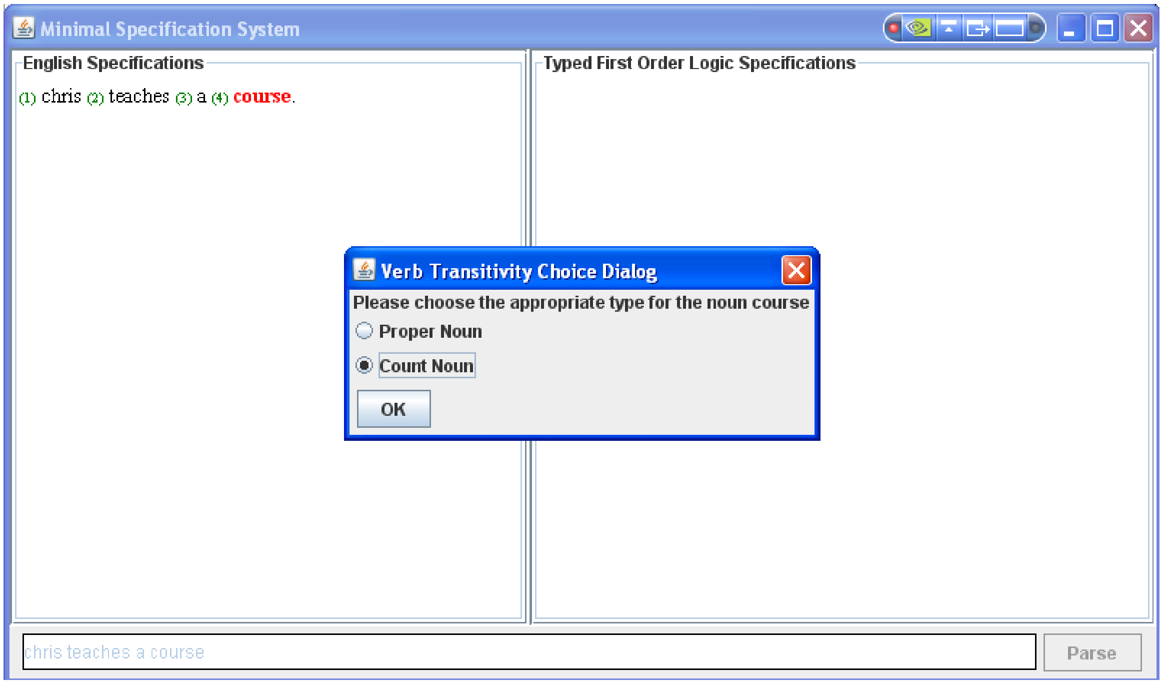
Verb:	smile	Subject Index:	I	Subject Type:	human
Situation:	S	3sg Form:	smiles	Past Form:	smiled
Present Participle Form:	smiling	Past Participle Form:	smiled	Agent Nomial Form:	smiler

	Predication			Predication			Predication	
RELN:	smile		RELN:			RELN:		
SIT:	S		SIT:			SIT:		
ROLE1:	smiler	Individual	ROLE1:		None	ROLE1:		None
ARG1:	I		ARG1:			ARG1:		
ROLE2:		None	ROLE2:		None	ROLE2:		None
ARG2:			ARG2:			ARG2:		
ROLE3:		None	ROLE3:		None	ROLE3:		None
ARG3:			ARG3:			ARG3:		

Figure 8.8: Asking the details of a new verb

verb become known to the system.

If a new noun is encountered, the user is asked to specify if the noun is a proper noun or a count noun.[3] Figures 8.9 and 8.10 show the detection of the new noun *course* and acquisition of its lexical definition.


Minimal Specification System
English Specifications
Typed First Order Logic Specifications
Verb Transitivity Choice Dialog
OK
Parse

(1) chris (2) teaches (3) a (4) course.

Please choose the appropriate type for the noun course

○ Proper Noun

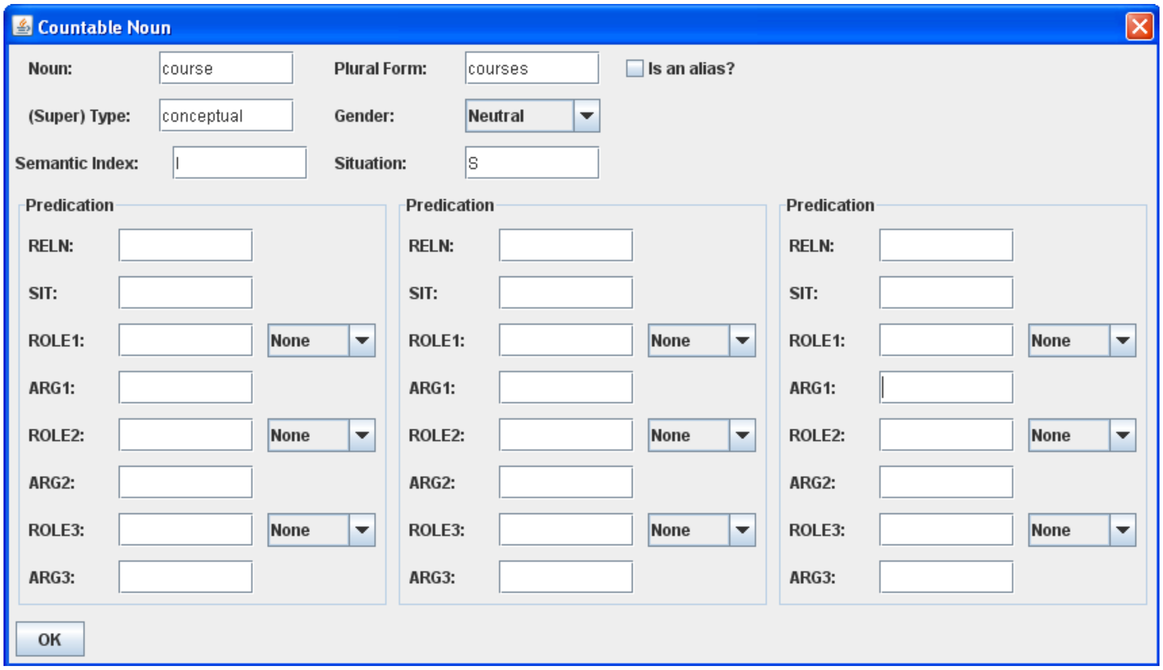
● Count Noun

chris teaches a course

Figure 8.9: Encountering a new noun

As said in previous chapters, a count noun carries a reference to a domain type. The user interface asks the user to specify if this type is a new type or an alias of a type already existing in the domain type hierarchy. This is achieved by the “Is an alias” check box. If this box is checked, it means the count noun refers to an existing type. This type should be entered in the “(Super) Type” text field. If the count noun introduces a new type into the domain type hierarchy, then this field specifies its immediate super-type. The **instantiate** and discourse predications are automatically added. If extra and special predications are needed for a noun, they can be entered in the dialog box shown in figure 8.10. “Semantic Index” and “Situation” input fields are only relevant in case additional predications should be entered. “Semantic Index” is the individual index used to refer to the context object

[3] Mass nouns are not considered in this thesis, as their analysis is very demanding and outside the scope of this thesis.



Noun: course
Plural Form: courses
□ Is an alias?
(Super) Type: conceptual
Gender: Neutral
Semantic Index: I
Situation: S

Predication: RELN:, SIT:, ROLE1: None, ARG1:, ROLE2: None, ARG2:, ROLE3: None, ARG3:

Predication: RELN:, SIT:, ROLE1: None, ARG1:, ROLE2: None, ARG2:, ROLE3: None, ARG3:

Predication: RELN:, SIT:, ROLE1: None, ARG1:, ROLE2: None, ARG2:, ROLE3: None, ARG3:

Figure 8.10: Asking the details of a new noun

introduced by the count noun, and “Situation” is the situation index that refers to the event of its presence.

8.3 Application

The grammar and parser and the interactive user interface that we developed can be used to enter Controlled English specifications in a sense that has been also used in Attempto project [35, 33]. The advantages of our system could be summarized as:

- Our system uses a dynamic ontology that can be enhanced when entering specifications.

- It is more difficult to enter an invalid specification as a result of type restrictions. In other words, our system is more mistake resistant.

- The use of domain types results in less candidates for antecedent resolution. Thus choosing an interpretation of a sentence is more automatic and preciser than the Attempto project.

One potential application of our system is parsing use cases. Use cases describe the behavior of a system when it interacts with the outer world. The system can be as small as a vending machine or as large as a whole organization. Since the format of use cases is plain text with simple grammar, it could be possible to use a Controlled Language (e.g., based on the grammar we developed in this thesis) with a parser (e.g. the parser we developed in this thesis) to parse and process the use cases.

Since we have incorporated semantics into our grammar, each sentence that is parsed by our system carries a meaningful semantics. Combined with reasoning resources (e.g., theorem provers and model builders that we have not considered in this thesis) we will have a stronger system for maintaining use cases, that detects inconsistencies (by theorem provers) or completeness (by model builders). This architecture is similar to the one proposed by Schwitter et al. in [68], however our system has the advantage of using a type system.

In use cases, verbs usually indicate operations that are performed by some entities (*actors* in the use case terminology). In our grammar each verb has a type restriction, which specifies the acceptable types of the verbs arguments. In other words type restrictions of verbs determine the types of the entities that are involved in the corresponding operation.

The presence of this *strong typing* in the controlled language results in preciser use cases by rejecting some invalid use cases that contain some type mismatches. Next we will see how this can help maintaining use cases and process re-engineering.

8.3.1 A Sample of Structural Change and Affected Use Case Identification

Example 8.1 Suppose the *system under discussion* or SuD (which is the system for which we are writing a use case) is part of Citizenship and Immigration Canada, where a hypothetical change of the organization occurs. We consider a use case that describes the business process of issuing a study permit for an international student. The change of organization is first done on the domain model, and we see that with our type system the affected use case can be identified, because it cannot be parsed any longer. This will automatically identify the affected use cases, that also need to be modified.

The partial current status of the organization is given by the domain type hierarchy shown in figure 8.11, and the following use case. The facts here are adjusted and simplified for ease of illustration.

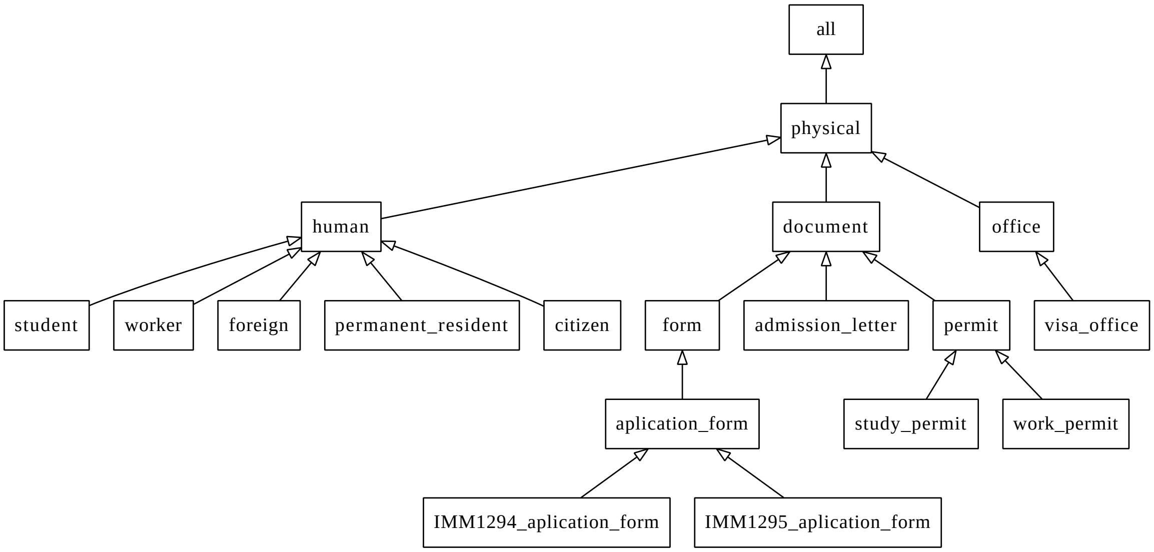


Figure 8.11: Initial domain type hierarchy of the study permit example

Since each student falls exactly in one of foreign, permanent resident, citizen, we can consider the domain type of the word *student* as a general (abstract) type.[4] That is:

$student : spec(student)$

Same goes for a worker:

$worker : spec(worker)$

The adjective *foreign* can be defined in the lexicon with the type restriction:

$\{(modified, foreign)\}$

With this type restriction, the type of the phrase *foreign student* will be:

$spec(student * foreign)$

This demonstrates how a modifier can add information to the type of the general (abstract) modified expression.

In the following use case the phrases *admission letter*, *application form*, *visa office* are defined as multi word lexical entries, because we have not discussed nouns (such as *visa* in *visa office*) that can act as modifiers. These noun modifiers can be declared in the lexicon with non-empty MOD and MOD-GUARD features. The analysis would be very similar to adjectives. Moreover *letter*, *form*, and *office* can be defined as general classes, i.e., with

[4] See section 2.3.3.

specializable types in the lexicon.

Preconditions:

A university in Canada admitted a foreign student.

The student received an admission_letter from the university.

Main Success Scenario:

1. The student obtains an IMM1294_application_form and She/he fills it.
2. She/he attaches the admission_letter to the application_form.
3. She/he submits the application_form to a visa_office.
4. The visa_office verifies the admission_letter.
5. It records the application_form.
6. It issues a study permit for the student.
7. It sends the permit to the student.

The type restrictions of the verbs, i.e. suitable argument types of operations in the use case are as follows. These can be thought of as operation signatures of the domain. Some entries of the following table is from common sense.

Table 8.1: Operation signatures

Operation	**Subject Type**	**Object 1 Type**	**Object 2 Type**
admit	university	student	
receive	human	document	
obtain	human	document	
fill	human	form	
attach	human	document	(to) document
submit	foreign student	IMM1294_application_form	(to) visa_office
submit	foreign worker	IMM1295_application_form	(to) visa_office
verify	office	document	
record	office	document	
issue	visa_office	study_permit	(for) foreign student
issue	visa_office	work_permit	(for) foreign worker
send	office	document	(to) human

Now suppose for expediting the process of applications the visa office decides to have two separate units with different addresses, namely a study permit unit and a work permit unit. According to the new rules, students should submit their application to the study permit unit and workers should submit their application to the work permit unit. This will result in the following changes in the operation signature.

Table 8.2: The modified operation signatures of the Study Permit example

Operation	**Subject Type**	**Object 1 Type**	**Object 2 Type**
submit	foreign student	IMM1294_application_form	(to) study_permit_unit
submit	foreign worker	IMM1295_application_form	(to) work_permit_unit
issue	study_permit_unit	study_permit	(for) foreign student
issue	work_permit_unit	work_permit	(for) foreign worker

The new domain type hierarchy is shown in figure 8.12. The additional domain facts are:

Every visa_office has a study_permit_unit.
Every visa_office has a work_permit_unit.

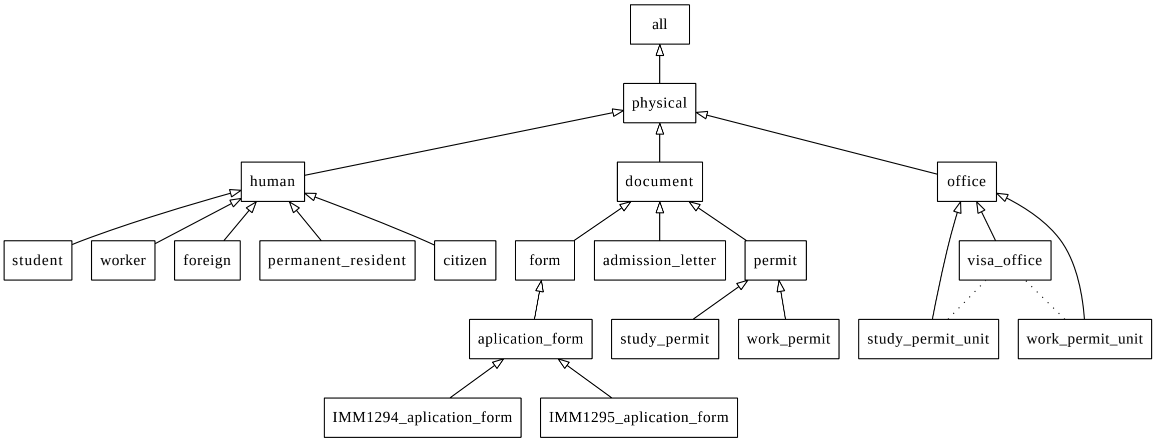


Figure 8.12: The modified domain type hierarchy of the study permit example

With this change that has affected the type restrictions in our system, the previous use case cannot be parsed anymore. This is in correspondence with the fact that with the new regulations a student cannot submit a study permit application (IMM1294) simply to a visa office. Instead she/he should submit the application to the study permit unit.

So the use cases that are maintained by our interactive parser/analyzer, are not just arbitrary pieces pf plain text but are sensitive to the domain model (ontology).

This is in analogous to a change in the design of a piece of software code that could initially result in some compilation errors through the rest of the code. The errors actually point out to the affected areas of the source code that need to be fixed. This is useful because the system automatically points out to the parts that need to be adapted to the new design.

By using the type-aware parser we not only reject some invalid use cases at the time of entry, but also we are able to detect the use cases that need to be modified as a result of some change in the domain model of a system. The main success scenario of our example should be then modified to:

Main Success Scenario:
1. The student obtains an IMM1294_application_form and fills it.
2. She/he attaches the admission_letter to the application_form.
3. She/he submits the application_form to a study_permit_unit.
4. The study_permit_unit verifies the admission_letter.
5. It records the application_form.
6. It issues a study permit for the student.
7. It sends the permit to the student.

This is just an example of a document that is sensitive to an ontology (domain model). Applications of such documents are not limited to use cases. They can be used for other purposes such as software documentation, inline source code documentation, or user manuals of products. Using such domain model sensitive documents ensures that the whole documentation maintains its integrity with regards to an ontology.

8.3.2 Relevance of Narrow Scope Variable Binding Condition to Semantic Analysis of Use Cases

In our framework the Narrow Scope Variable Binding Condition is necessary for extracting the semantics of any natural language sentence that has an adverb or a prepositional modifier that modifies a verb phrase or a sentence. The reason is illustrated in the example of analyzing sentence (71) of chapter 6. Since use cases are written in natural language, use of adverbs or prepositional phrases modifying verbs or sentences is inevitable. So the controlled language that is chosen for documenting the use cases should cover adverbs and prepositional modifiers. For example, consider this simple use case of an automated Call Center System (CCS) of a bank.

Example 8.2 The system under discussion is an automated call center system that receives phone calls from the customers of a bank named TbBank and forwards the calls to the appropriate personnel.

Main Success Scenario:
1. A customer calls TbBank during office hours.
2. CCS receives the call.
3. CCS plays a recorded message that asks the customer to enter a staff member extension number.
4. The customer enters an extension number with the phone's keypad.
5. CCS forwards the call to a staff member with that extension number.
6. A conversation between the staff member and the customer starts.
7. CCS ends the call after the conversation is over.

The first sentence of this use case uses a prepositional modifier *during office hours.* The lexical entry for the preposition *during* is presented below.

$$(7)\quad \begin{bmatrix} \textit{word} & \\ \text{ORTH} & [\text{during}] \\ \text{SYN} & \begin{bmatrix} \text{HEAD} & \textit{prep} \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \langle\rangle \\ \text{COMPS} & \left\langle \text{NP}_u \right\rangle \\ \text{COMPS\_GUARDS} & \left\langle\left\langle \text{:time\_period}\right\rangle\right\rangle \\ \text{MOD} & \left\langle \begin{bmatrix} \text{MODIFIED} & \text{S}_t \\ \text{AFTER} & + \\ \text{MOD-GUARD} & \textbf{Bool} \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{INDEX} & s \\ \text{RESTR} & \left\langle \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{during} \\ \text{SIT} & s \\ \text{EVENT1} & t' \\ \text{EVENT2} & u' \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{outscope} \\ \text{OUTER} & t' \\ \text{INNER} & t \end{bmatrix}, \begin{bmatrix} \textit{predication} & \\ \text{RELN} & \textbf{outscope} \\ \text{OUTER} & u' \\ \text{INNER} & u \end{bmatrix}, \right\rangle \end{bmatrix} \end{bmatrix}$$

The predication **during** is a scopal modification predication that we declare as a narrow scopal predication. With this declaration, the only possible TFOL semantic representation of the first sentence of the use case will be:

(8) c_1 : bank $\wedge$ name(TbBank, c_1)
$\wedge$ $\exists c_2$: customer ; `during`(time_present(call(c_2, c_1)), office_hours)

The predication **during** is translated to a TFOL predicate symbol `during` with two arguments. The first argument in this example is the event of calling, and the second argument is the event of office hours.

Without the Narrow Scope Variable Binding Condition the sentence could be parsed to the extra erroneous reading:

(9) $\exists c_2$: customer ; `during`(
c_1 : bank $\wedge$ name(TbBank, c_1) $\wedge$ time_present(call(c_2, c_1)), office_hours)

This erroneous reading implies that the naming of the bank also occurs during the office hours which is wrong and not implied by the natural language sentence. In this instance, the error happens even if the **outscope** predications from the definition are removed.

8.3.3 Tense Predications and Use Cases

Each sentence in a use case has a step number. A use case comprises of a series of steps. Time can be thought of as the ordering of these steps. In this light, tense predications can be adjusted to refer to the steps in use cases. For example *time_present* could mean "at the current step in the use case", and *time_past* could mean "at a previous step in this use case or at the time of the entry to the current use case (refers to the precondition)". Although very important, we cannot cover this topic in this thesis. It will remain as a future work.

8.4 Conclusion

In this chapter we described how we have implemented our interactive parser and how its user interface can be used. We also provided some potential application of our parser in the area of domain modeling. Our contributions in this chapter were:

- The CHRG$'$ language for implementing unification based typed feature structure grammars.
- Development of a user interface capable of interacting with the user by click sensitive text used in manual antecedent resolution. The user interface supports entering new lexical items.

Chapter 9

Conclusions and Future Work

9.1 Conclusion

In the first part of this thesis we advocated the use of a strong type system for the application domain of a grammar. We noted requirement texts specifically use cases as a special kinds of text that are remarkably dependent on a domain model or ontology. We discussed how a type system can enhance the accuracy of parsing sentences by reducing ambiguity and rejecting sentences with type mismatches. Thus we promoted the use of a grammar that is *domain type aware* for parsing such texts.

We first explored some of the existing grammars that apply a type system and discussed their limitations. We then developed a type system that could be seen as an extension of the typed lambda calculus. We used our notion of type restrictions as a method of expressing *semantic selection restrictions*. We demonstrated how the type system with the concept of type restrictions for each lexical entry can reduce ambiguity in parsing and in semantic analysis of English sentences.

We designed our type system in such a way that it is capable of handling multiple inheritance efficiently, without the need of type encoding (encodings such as described by Fall in [31]). We called the types that inherit from multiple parents in the type hierarchy *composite types*. Composite types together with simple atomic types comprise the set of *natural types*. As part of the type system, we introduced the specializable types inspired by the incomplete types introduced by Dahl [23] and negated types, and showed how they can be useful to map the domain type of pronouns and general (abstract) nouns. We extended the concept of subtypes to these extended types.

We then elaborated on the details of the type system and studied its characteristics by several theorems. We provided a high level implementation of the type system in pseudocode with efficient algorithms that reduced the subtype checking among composite, specializable, and negated types to subtype checking among simple types.

Along the way we proposed alternative but equivalent views of type restrictions. In the final view, satisfaction of type restrictions is proven equivalent to subtype checking among the actual types of the arguments of an expression and the expected type of its arguments.

In the second part of the thesis we set out for combining the developed type system with HPSG grammar formalism based on Sag et al. [67]. In doing so we first introduced the HPSG framework, and introduced its internal type system for dealing with grammar entities. Then we described the syntax part of the grammar. Finally we explained the semantic component of the grammar.

In the semantic component we had several contributions, such as the incorporation of discourse analysis by introducing *discourse predications*, and several binding conditions and constraints that are necessary to construct the correct structure of the predicate calculus formula that carries the semantics of the parsed sentence. These constraints and binding conditions were namely, *Quantifier Restriction Constraint*, *Quantifier Scope Constraint*, *Quantifier Variable Binding Condition*, *Narrow Scope Variable Binding Condition*. Narrow binding condition was introduced to handle the structure of the formulas with scopal modifier predicates correctly.

We then showed how the type system can be integrated to the grammar by using new features TYPE, SPR_GUARDS, COMPS_GUARDS, MOD_GUARD. This is incorporated to the grammar that was developed in the two previous chapters. We briefly showed how the grammar integrated with the type system can be used for antecedent resolution.

In the third part of the thesis, we discussed how we implemented the grammar and the parser. We described how we implemented our bottom-up parser by a variant of CHRG [14] that we call CHRG$'$. We showed why the original CHRG cannot be used. Then demonstrated with a few examples how the user interface of our program can be used. We showed that our system is able to capture new lexical entry definitions such as new verbs and new nouns. We also argued that the system can be applied to parsing requirement texts or use cases by an example.

We believe our approach can be used in programs such as *RavenFlow* [64] that maintain requirements, to enhance its capabilities to parse and analyze requirements. For example in [65], on page 19 the user is cautioned to beware of situations where the antecedent of a syntactically ambiguous pronoun could be construed automatically by their system as referring to a wrong antecedent. This error is due to *RavenFlow*'s heuristics, which map pronouns to the nearest noun (matching their case, gender and number), are insufficient, because types are ignored. Using a domain type aware grammar like the one we developed in this thesis can alleviate these problems.

An example is the following piece of requirement where the pronoun *he* is automatically and incorrectly interpreted as referring to *no analysis* by the *RavenFlow* software:

"If the ECO manager determines that no analysis is needed, then he enters the ECOR on the weekly Change Review Board agenda. Otherwise, the ECO manager sends the incomplete ECOR to the lead engineer."[1]

If our type system is used, then such wrong interpretations would be automatically rejected.

9.2 Contributions

Our main contributions can be summarized by:

- An efficient application-domain type system, capable of dealing with multiple inheritance, specializable and negated types at the same time, with the ability of modifying the type hierarchy on the fly with no significant cost.
- Implementing the notion of type restrictions with our type system.
- Incorporating discourse predications to deal with the discourse.
- Discovering necessary constraints and binding conditions to allow structurally correct predicate calculus formulas.
- Integrating the type system to the HPSG framework with the modified MRS.
- Applying the grammar with the domain type system for better antecedent resolution.

[1] Taken from RavenFlow Requirements Writing Guide [65].

9.3 A Limitation

For anaphora resolution, our system finds the candidate antecedents by matching syntactic feature agreement and type compatibility. This works fine for referents and antecedents that are not modified by an adjective or a relative clause. However, with the presence of such modifiers further semantic analysis is needed to rule out some of the antecedents.

Consider the following discourse as an example:

(1) a. A red car is on a street.

b. A blue car is on the street.

c. The red car is speeding.

Our system suggests the *blue car* introduced by sentence (1b) (as well as the *red car* introduced by sentence (1a)) as a candidate antecedent of *the red car* in sentence (1c). This is due to the limitation that our system does not analyze adjectives in finding the right antecedent.

A similar situation happens with relative clauses as shown in the following example, where it is needed to analyze the relative clauses for ruling out the wrong antecedents.

(2) a. A red car is on a street.

b. A blue car is on the street.

c. The car which is red is speeding.

This task in general requires automated reasoning as the sentences grow in complexity. For example consider the following scenario:

(3) a. Chris speaks only English.

b. John knows only the French language.

c. Mary is speaking to John and John is speaking to Mary.

d. John understands what Mary says.

e. The person who is not speaking French is jealous.

Our system suggests *Chris*, *John* and *Mary* as candidates of *the person who is not speaking French*, although *John* and *Mary* should have been ruled out.

9.4 Future Work

An interesting future direction of research would be to apply a reasoning component for use in a complete requirement analysis system. This reasoning component could be integrated to anaphora resolution mechanism to alleviate the limitation that we discussed in the previous section. Also, the reasoning component can be integrated with the scope resolution subsystem to weed out semantically duplicate formulas with different tree structures, and also to answer queries, and to check the consistency and completeness of the entered specifications.

Another good research direction would be to study on the correspondence of the semantics of tense predications and the steps in use cases.

Among the possible applications of our system, a very interesting one is ontology-sensitive source code documentations, or user manuals of different products.

Another possible extension is to develop rich user interfaces for entering and maintaining the specifications. For example we have not implemented the modification of the specifications that are already entered. This would be certainly necessary in a real application. A related future work is to develop verbalizers for TFOL specifications. With this feature it would be possible for the user to modify the TFOL specifications and have the updated English specifications as a result of the verbalizer automatically.

Along with the lines of the user interface, it is very desirable to have a look-ahead parser like the one proposed and implemented by Schwitter et al. [68] to help the user enter English sentences more easily into the system by informing her/him what kind of word (i.e., part part of speech) is expected to be typed next. This will alleviate the need of training to learn the details of the controlled language that is described by the grammar.

Appendix A

General Subtype Checking Algorithm

We assume all negated types use triangular types whose parents are also triangular. V and X are free and different variables. We assume the jump expressions in A, B are reduced to their simplest form. The return value of the function is the *nil* set if the formula does not hold or is not satisfiable. If the formula holds or is satisfiable and no variables occur in A, then the return value is an empty set. If a variable occurs in A, and the formula is satisfiable, then the return value is the set of answers (specializable or not).

Program A.1: The general algorithm of subtype checking

```
function subtype(A : BasTyp, B : BasTyp) :  set of answers
begin
  if is_natural(A) and is_natural(B) then
    return subtype_natural(A, B)
  /* By theorem 3.38 */
  if A = $\tau \curvearrowright V$ and is_natural(B) and B = $\sigma$ then
    if not disjoint($\tau$, $\sigma$) then
      return $\{\tau * \sigma \curvearrowright V_h\}$
    else
      return $nil$
    end if
  end if
  /* By theorem 3.40 */
  if is_natural(A) and A = $\tau$ and B = $\sigma \curvearrowright V$ then
    return subtype($\tau$, $\sigma$)
  end if
  /* By theorem 3.45 and remark 3.46 */
  if A = $\tau \curvearrowright V$ and B = $\sigma \curvearrowright V$ then
    $\mu \leftarrow$ most_general_common_subtype($\tau$, $\sigma$)
    if $\mu \neq nil$ then
      return $\{\mu \curvearrowright V_h\}$
    else
      return $nil$
    end if
  end if
  /* By theorem 3.47 */
  if A = $\tau \curvearrowright V$ and B = $\omega \curvearrowright X$ then
    return subtype(A, $\omega$)
  end if
  /* By theorem 3.50 */
  if A = $\tau - \sigma$ and is_natural($\tau$) and is_natural(B) and B = $\omega$ then
    if subtype($\tau$, $\sigma$) then
      return $nil$
    return is_partition($\tau$, $\omega$, $\sigma$) or subtype($\tau$, $\omega$)
  end if
  /* By theorem 3.51 */
  if is_natural(A) and A = $\tau$, and B = $\omega - \sigma$ and is_natural($\omega$) then
    if subtype($\omega$, $\sigma$) then
      return $nil$
    return disjoint($\tau$, $\sigma$) and subtype($\tau$, $\omega$)
  end if
  /* By theorem 3.52 */
  if A = $\tau - \sigma$ and B = $\omega - \sigma$ and is_natural($\tau$) and is_natural($\omega$) then
    if subtype($\omega$, $\sigma$) or subtype($\tau$, $\sigma$) then
```

```
    return $nil$
  return is_partition($\tau$, $\omega$, $\sigma$) or subtype($\tau$, $\omega$)
end if
/* By theorem 3.53 */
if A = $\tau \curvearrowright V - \sigma$ and is_natural(B) and B = $\omega$ then
  $Result \leftarrow \emptyset$
  if is_partition($p$, $\omega$, $\sigma$) then
    if subtype($p$, $\tau$) then
      $Result \leftarrow Result \cup \{p\}$
    end if
  end if
  /* By theorem 3.38 the output of the following */
  /* recursive call is a singleton */
  $\{\mu \curvearrowright X\} \leftarrow$ subtype($\tau \curvearrowright V$, $\omega$)
  $M \leftarrow$ most_general_common_subtypes_not_contained_by($\mu$, $\mu$, $\sigma$)
  $Result \leftarrow Result \cup M$
  $n = |Result|$
  if $n > 0$ then
    return $Result$
  else
    return $nil$
end if
/* By theorem 3.54 */
if A = $\tau - \sigma$ and B = $\omega \curvearrowright V$ and is_natural($\tau$) then
  return subtype($\tau - \sigma$, $\omega$)
end if
/* By theorem 3.55 and remark 3.57 */
if A = $\tau \curvearrowright V - \sigma$ and B = $\omega \curvearrowright V$ then
  return most_general_common_subtypes_not_contained_by($\tau$, $\omega$ $\sigma$)
end if
/* By theorem 3.58 */
if A = $\tau \curvearrowright V - \sigma$ and B = $\omega \curvearrowright X$ then
  return subtype(A, $\omega$)
```

```
end if
/* By theorem 3.59 */
if is_natural(A) and A = $\tau$ and B = $\omega \curvearrowright V - \sigma$ then
  if disjoint($\tau$, $\sigma$) then
    return subtype(A, $\omega \curvearrowright V$)
  else
    return $nil$
  end if
end if
/* By theorem 3.60 and remark 3.63 */
if A = $\tau \curvearrowright V$ and B = $\omega - \sigma$ and is_natural($\omega$) then
  /* By theorem 3.38 the output of the following */
  /* recursive call is a singleton */
  $\{\mu \curvearrowright X\} \leftarrow$ subtype(A, $\omega$)
  $M \leftarrow$ most_general_disjoint_subtypes($\mu$, $\sigma$)
  $n = |M|$
  if $n > 0$ then
    $Result \leftarrow \emptyset$
    for each $\mu_i \in M$ do
      $Result \leftarrow Result \cup \{\mu_i \curvearrowright V_{h+i}\}$
    end for
    return $Result$
  else
    return $nil$
  end if
end if
/* By theorem 3.64 and remark 3.65*/
if A = $\tau \curvearrowright V$ and B = $\omega \curvearrowright V - \sigma$ then
  $Result \leftarrow \emptyset$
  $\mu \leftarrow$ most_general_common_subtype($\tau$, $\omega$)
  if $\mu \neq nil$ then
    $N \leftarrow$ most_general_disjoint_subtypes($\mu$, $\sigma$)
    for each $\mu_i \in N$ do
```

```
        $Result \leftarrow Result \cup \{\mu_i \curvearrowright V_{h+i}\}$
      end for
    end if
    if $|Result| > 0$ then
      return $Result$
    else
      return $nil$
  end if
  /* By theorem 3.67 */
  if A = $\tau \curvearrowright V$ and B = $\omega \curvearrowright X - \sigma$ then
    return subtype(A, $\omega - \sigma$)
  end if
  /* By theorem 3.68 */
  if A = $\tau \curvearrowright V - \sigma$ and B = $\omega - \sigma$ and is_natural($\omega$) then
    if subtype($\omega$, $\sigma$) then
      return $nil$
    return subtype(A, $\omega$)
  end if
  /* By theorem 3.69 */
  if A = $\tau - \sigma$ and B = $\omega \curvearrowright V - \sigma$ and is_natural($\tau$) then
    return subtype(A, $\omega - \sigma$)
  end if
  /* By theorem 3.70 and remark 3.71 */
  if A = $\tau \curvearrowright V - \sigma$ and B = $\omega \curvearrowright V - \sigma$ then
    return most_general_common_subtypes_not_contained_by($\tau$, $\omega$, $\sigma$)
  end if
  /* By theorem 3.72 */
  if A = $\tau \curvearrowright V - \sigma$ and B = $\omega \curvearrowright X - \sigma$ then
    return subtype(A, $\omega - \sigma$)
  end if
  /* This should not happen:*/
  print "Warning : an unknown case encountered!"; return $nil$
end
```

Appendix B

Proof of Theorems

Proof of Theorem 3.38:

- (a) By applying theorem 3.24 $\tau \nparallel \sigma$ is proved. For the second conjunct we observe that if we set $\omega = \tau * \sigma$ and $R = \sqsubseteq$ in definition 3.32, then the first condition of being the most general specializable answer (statement (7) of chapter 3) is trivially true. The second condition (statement (8) of chapter 3) can be re-written as:

  (1) $\forall \tau_0 \in \mathbf{NatTyp} \; . \; (\tau_0 \sqsubseteq \tau \;\wedge\; \tau_0 \sqsubseteq \sigma) \Longleftrightarrow \tau_0 \sqsubseteq \tau * \sigma$

  which holds because of theorem 3.14. Thus both conditions hold and by definition 3.32 $\tau * \sigma$ is the most general answer of V and the proof of (a) is complete. □

- (b) First we observe that since τ and σ are not disjoint, by definition of composite types (definition 3.7), $\tau * \sigma$ is a well-formed composite type. Next we prove that

  (2) $\tau \curvearrowright V \sqsubseteq \sigma$

  is satisfiable by showing that $\tau * \sigma$ is an answer for V. If we replace $\tau * \sigma$ for V in (2) we get:

  (3) $\tau \curvearrowright (\tau * \sigma) \sqsubseteq \sigma$

  The left hand side of $\sqsubseteq$ in the above statement is equal to $\tau * \sigma$ by definition of jump expressions (definition 3.16). So the truth value of (3) is equal to the following statement:

(4) $(\tau * \sigma) \sqsubseteq \sigma$

which holds by theorem 3.9. So (4) and consequently (3) are true, and (2) is satisfiable. Now part (a) is applicable and the value of most general answer will be equal to what is needed. □

Proof of Theorem 3.40:

- ⇒) Suppose $\tau \sqsubseteq \sigma \curvearrowright V$ is satisfiable, then by definition 3.23 there should be a natural type ω such that

  (5) $\omega \sqsubseteq \sigma$

  (6) $\tau \sqsubseteq \sigma \curvearrowright \omega$

  By definition of jump expressions (definition 3.16), $\sigma \curvearrowright \omega = \omega$ and (6) becomes:

  (7) $\tau \sqsubseteq \omega$

  By applying the transitivity of $\sqsubseteq$ on (5) and (7) we get:
  $\tau \sqsubseteq \omega \sqsubseteq \sigma$
  Hence:
  $\tau \sqsubseteq \sigma$ □

- ⇐) Suppose $\tau \sqsubseteq \sigma$, then V can be set to σ and

  (8) $\tau \sqsubseteq \sigma \curvearrowright V$

  becomes:
  $\tau \sqsubseteq \sigma \curvearrowright \sigma$, which is equal to:
  $\tau \sqsubseteq \sigma$, which is true by the supposition.
  So, σ is an answer to (8) and thus (8) is satisfiable. □

Proof of Theorem 3.45:

- a) Suppose v_1 is an answer to V satisfying $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright V$, then we have:
  $\tau \curvearrowright v_1 \sqsubseteq \omega \curvearrowright v_1$

This requires:

$v_1 \sqsubseteq \tau$

$v_1 \sqsubseteq \omega$

This means v_1 is a common subtype of τ, ω, and hence is a subtype of the most general common subtype of τ and ω. □

- b) Suppose μ is the most general subtype of τ, ω, and that v_1 is an arbitrary subtype of μ, then we have:

  $v_1 \sqsubseteq \mu$

  So,

  $v_1 \sqsubseteq \tau$

  $v_1 \sqsubseteq \omega$

  Hence the following jump expressions are well-formed according to the definition of jump expressions (definition 3.16), and they are both equal to v_1:

  $\tau \curvearrowright v_1$

  $\omega \curvearrowright v_1$

  So, we can write:

  $\tau \curvearrowright v_1 \sqsubseteq \omega \curvearrowright v_1$

  This means that v_1 is an answer to V satisfying $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright V$ □

Proof of Theorem 3.47:

- a) Suppose $(v_1,\ x_1)$ is an answer to $(V,\ X)$ satisfying $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright X$, then we have:

  (9) $\tau \curvearrowright v_1 \sqsubseteq \omega \curvearrowright x_1$

  According to the definition of jump expressions (definition 3.16) we have:

  (10) $x_1 \sqsubseteq \omega$

  (11) $\omega \curvearrowright x_1 = x_1$

  So we can substitute x_1 for $\omega \curvearrowright x_1$ in (9) and we get:

  (12) $\tau \curvearrowright v_1 \sqsubseteq x_1$

  Now by the transitivity of $\sqsubseteq$ and applying it on (12), (10) we get:

(13) $\tau \curvearrowright v_1 \sqsubseteq \omega$

This means that v_1 is an answer to V satisfying $\tau \curvearrowright V \sqsubseteq \omega$ $\square$

- b) Suppose v_1 is an answer to V satisfying $\tau \curvearrowright V \sqsubseteq \omega$ this means:

  (14) $\tau \curvearrowright v_1 \sqsubseteq \omega$

  According to the definition of jump expressions (definition 3.16) we can write: $\omega \curvearrowright \omega = \omega$, and we can re-write (14) as:

  (15) $\tau \curvearrowright v_1 \sqsubseteq \omega \curvearrowright \omega$

  And this means that (v_1, ω) is an answer to (V, X) satisfying $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright X$ $\square$

Proof of Theorem 3.48:

Suppose $\tau - \sigma \sqsubseteq \omega$ (assumption $\Longrightarrow$)
Immediately according to definition 3.26 for $\tau - \sigma$ to be a well-formed type expression it is required that:

(16) $\tau \not\sqsubseteq \sigma$

Since σ is triangular by the hypotheses of the theorem we are left with the following cases due to definition 3.37:

(17) $\tau \sqsubseteq \sigma$

(18) $\sigma \sqsubset \tau$

(19) $\tau \parallel \sigma$

But (17) cannot be the case because it contradicts with (16).

Suppose (19) is the case, that is, τ and σ are disjoint. Then any instance x of τ is not an instance of σ and hence is an instance of $\tau - \sigma$, so $\tau \sqsubseteq \tau - \sigma$, but on the other hand any instance of $\tau - \sigma$ must also be an instance of τ, that is, $\tau - \sigma \sqsubseteq \tau$, and we have:
$\tau - \sigma = \tau$
so the assumption $\Longrightarrow$ can be re-written as:
$\tau \sqsubseteq \omega$

which is one disjunct in the proposition that we want to prove, and so the theorem is proved in this case.

So the only remaining case is (18), that is : $\sigma \sqsubset \tau$, this means that τ is a natural super type of σ and by the hypotheses of this theorem it is required that τ is a triangular type. This means for the natural type ω we have the following cases by definition 3.37:

(18.1) $\omega \sqsubseteq \tau$

(18.2) $\tau \sqsubset \omega$

(18.3) $\omega \parallel \tau$

– (18.3) cannot be the case. The reason is that according to definition 3.26, $I_{\tau-\sigma} \neq \emptyset$, so there is an element $x : \tau - \sigma$, which by the assumption ($\Longrightarrow$) and an application of (SUB) we must also have $x : \omega$, but $x : \tau - \sigma$ requires that $x : \tau$, so x is an instance of both τ and ω and so these two types cannot be disjoint.

– If (18.2) is the case, then the proposition of the theorem is trivially proved.

– If (18.1) is the case, we break down the cases of the triangular type σ against ω, and we will have:

(18.1.1) $\omega \sqsubseteq \sigma$

(18.1.2) $\sigma \sqsubset \omega$

(18.1.3) $\omega \parallel \sigma$

–– Assume (18.1.1), then:
$I_{\tau-\sigma} \neq \emptyset$ (by definition 3.26 of negated types)
$\Rightarrow$ there is an instance $x : \tau - \sigma$
$\Rightarrow x : \tau \ \wedge \ \neg(x : \sigma)$ (by applying (NEG))
$\Rightarrow x : \omega$ (by assumption ($\Longrightarrow$)).
$\Rightarrow x : \sigma$ (by applying (SUB) on the assumption (18.1.1) : $\omega \sqsubseteq \sigma$)
And we have a contradiction because x is an instance of σ while at the same time it is not an instance of σ. So this case is not possible.

$--$ Assume (18.1.2), that is, $\sigma \sqsubset \omega$, then by combining the assumption ($\Longrightarrow$) with assumption (18.1) we get:
$\tau - \sigma \sqsubseteq \omega \sqsubseteq \tau$
Now if an arbitrary instance x of τ is not an instance of ω then we have:

(20) $x : \tau \;\wedge\; \neg(x : \omega)$

then x cannot be an instance of σ because by assumption (18.1.2), if that was the case then by applying (SUB) x would be an instance of ω too, which is contradictory to (20), so:
$\neg(x : \sigma)$
and since x was assumed to be an arbitrary instance of τ, and what we derived above, by the definition of negated types we have:
$x : \tau - \sigma$
and by assumption ($\Longrightarrow$) and applying (SUB):
$x : \omega$
which is again contradictory to (20).
So every instance of τ is an instance of ω and thus:
$\tau \sqsubseteq \omega$
which is one disjunct of the proposition of this theorem, and the theorem is proved in this case.

Note that in this case we have $\tau = \omega$, because of what we proved above and the assumption (18.1)

$--$ The only remaining sub-case is (18.1.3) in which we have the following assumptions:
assumption ($\Longrightarrow$) : $\quad \tau - \sigma \sqsubseteq \omega$
assumption (18) : $\sigma \sqsubset \tau$
assumption (18.1) : $\omega \sqsubseteq \tau$
assumption (18.1.3) : $\omega \parallel \sigma$

For this sub-case we will prove here that τ can be partitioned into σ and ω, which is one disjubct of the proposition of this theorem. Both σ and ω are disjoint subtypes of τ by the above assumptions. So according to definition 3.35, we only need to prove:
$I_\tau = I_\sigma \cup I_\sigma$

which can be broken down into two parts:
(part i) $I_\sigma \cup I_\omega \subseteq I_\tau$
(part ii) $I_\tau \subseteq I_\sigma \cup I_\omega$

The first part is true because σ and ω are subtypes of τ and by corollary 3.2 we have $I_\sigma \subseteq I_\tau$ and $I_\omega \subseteq I_\tau$ so the union $I_\sigma \cup I_\omega$ must also be a subset of I_τ by set theory results.

For the second part, suppose x is an arbitrary instance of τ then we have either:

(21) $x : \sigma$

(22) $\neg(x : \sigma)$

If (21) is the case, then $x \in I_\sigma$ and by set theory results also $x \in I_\sigma \cup I_\omega$.

If (22) is the case, then by the definition of negated types $x : \tau - \sigma$, because x was assumed an arbitrary instance of τ, and by (22) x is not an instance of σ.
then by assumption ($\Longrightarrow$) we have:
$x : \omega$
$\Rightarrow x \in I_\omega$
$\Rightarrow x \in I_\sigma \cup I_\omega$ (by set theory results)

So in both cases (21) and (22) we proved that the arbitrary member x of I_τ is also a member of $I_\sigma \cup I_\omega$. So we have:
$I_\tau \subseteq I_\sigma \cup I_\omega$

And $I_\tau = I_\sigma \cup I_\sigma$ by combining the two parts (i) and (ii) above.

So the proof that τ can be partitioned into σ and ω is now complete. That is:
$\tau = (\omega \mid \sigma)$
which is one disjunct in the proposition that we need to prove.

So for all cases, at least one disjunct of the proposition is true, and the theorem is proved.

Proof of Theorem 3.49:

Case 1) Suppose $\tau \sqsubseteq \omega$, then we have:

(23) $I_\tau \subseteq I_\omega$

But according to theorem 3.27, $\tau - \sigma \sqsubseteq \tau$ and by applying corollary 3.2 we get:

(24) $I_{\tau-\sigma} \subseteq I_\tau$

Now by combining (23) and (24) and the transitivity of $\subseteq$ we get:
$I_{\tau-\sigma} \subseteq I_\omega$
And by corollary 3.2 we get:
$\tau - \sigma \sqsubseteq \omega$
And the theorem is proved in this case.

Case 2) Suppose $\tau = (\omega \mid \sigma)$, then for an arbitrary instance x of $\tau - \sigma$ we have:
$x : \tau - \sigma$
$\Rightarrow x : \tau \ \wedge \ \neg(x : \sigma)$
Now since τ is partitioned into σ and ω and every instance of τ is either an instance of σ (which is not the case for x here) or an instance of ω:
$\Rightarrow x : \omega$ So we proved every arbitrary instance of $\tau - \sigma$ is also an instance of ω that is:
$\tau - \sigma \sqsubseteq \omega$
And the theorem is proved in this case too.

Proof of Theorem 3.50:

By combining theorems 3.48 and 3.49.

Proof of Theorem 3.51:

- $\Longrightarrow$) Suppose $\tau \sqsubseteq \omega - \sigma$, this means any instance of τ is an instance of $\omega - \sigma$ which by definition 3.26 of negated types must be an instance of ω but not an instance of σ.
  When separated clearly, we have:
  any instance of τ is an instance of ω that is, $\tau \sqsubseteq \omega$
  any instance of τ is not an instance of σ so there is no common instance between these two types, that is, $\tau \parallel \sigma$.
  The first conjunct in the proposition immediately follows because of the well-formedness of $\omega - \sigma$. $\square$

- $\Longleftarrow$) Suppose $\omega \not\sqsubseteq \sigma \ \wedge \ \tau \sqsubseteq \omega \ \wedge \ \tau \parallel \sigma$, then any arbitrary instance x of τ we have:

  $x : \tau$

  then by the $\tau \sqsubseteq \omega$ part of assumption ($\Longleftarrow$) and an application of (SUB) we get:

  (25) $x : \omega$

  also by the $\tau \parallel \sigma$ part of assumption ($\Longleftarrow$) we know that there is no common instance of τ and σ, so:

  (26) $\neg(x : \tau)$

  Now since $\omega \not\sqsubseteq \sigma$, $\omega - \sigma$ is a well-formed type and by combining (25) and (26) and applying the definition 3.26 of negated types we get:

  $x : \omega - \sigma$

  So any arbitrary instance of τ is an instance of $\omega - \sigma$, that is:

  $\tau \sqsubseteq \omega - \sigma \quad \square$

Proof of Theorem 3.52:

- $\Longrightarrow$) Suppose $\tau - \sigma \sqsubseteq \omega - \sigma$:

  Because of the well-formedness requirement of $\tau - \sigma$ and $\omega - \sigma$ we immediately get:

  $\tau \not\sqsubseteq \sigma \ \wedge \ \omega \not\sqsubseteq \sigma$

  We also know that $\omega - \sigma \sqsubseteq \omega$ by theorem 3.27, so we have:

  $\tau - \sigma \sqsubseteq \omega - \sigma \sqsubseteq \omega$

  and by the transitivity of $\sqsubseteq$ we get:

  $\tau - \sigma \sqsubseteq \omega$

  Now theorem 3.48 can apply and we get:

  $\tau = (\omega \mid \sigma) \ \vee \ \tau \sqsubseteq \omega \quad \square$

- $\Longleftarrow$) Suppose $\tau \not\sqsubseteq \sigma \ \wedge \ \omega \not\sqsubseteq \sigma \ \wedge \ (\tau = (\omega \mid \sigma) \ \vee \ \tau \sqsubseteq \omega)$, by applying theorem 3.49 we get:

  (27) $\tau - \sigma \sqsubseteq \omega$

  Assumption ($\Longleftarrow$) requires either of the following to be true:

  (28) $\tau = (\omega \mid \sigma)$

(29) $\tau \sqsubseteq \omega$

- Assume (28) is the case, that is, $\tau = (\omega \mid \sigma)$, then by definition 3.35 of partitions, it is required that ω and σ are disjoint. So by the definition 3.6 of disjoint types we know that the instance sets of ω and σ are disjoint. Hence by set theory results we get:

$I_\omega - I_\sigma = I_\omega$

And using corollary 3.2 we get:

$\omega - \sigma = \omega$

So in (27), we can replace ω by $\omega - \sigma$ and we get:

$\tau - \sigma \sqsubseteq \omega - \sigma$

And in this case the theorem is prove.

- Now assume (28) is the case, that is, $\tau \sqsubseteq \omega$, then by corollary 3.2 we get:

$I_\tau \subseteq I_\omega$

Now for any arbitrary instance of $\tau - \sigma$ we have:

$x : \tau - \sigma$

So,

(30) $\Rightarrow x : \tau \;\wedge\; \neg(x : \sigma)$ (by applying (NEG))

$\Rightarrow x : \tau$

$\Rightarrow x : \omega$ (by applying (SUB) on $\tau \sqsubset \omega$)

And (30) $\Rightarrow \neg(x : \sigma)$

By combining the above two statements and the definition of instance sets we get:

$\Rightarrow x \in I_\omega \;\wedge\; x \notin I_\sigma$

$\Rightarrow x \in I_\omega - I_\sigma$ (by set theory results)

$\Rightarrow x \in I_{\omega - \sigma}$ (by definition of negated types)

$\Rightarrow x : \omega - \sigma$ (by definition of instance sets)

That is, any arbitrary instance of $\tau - \sigma$ is also an instance of $\omega - \sigma$, so:

$\tau - \sigma \sqsubseteq \omega - \sigma$ □

Proof of Theorem 3.53:

- i) Suppose v_1 is an answer to $\tau \curvearrowright V - \sigma \sqsubseteq \omega$, that is, v_1 should satisfy it. So we have:

(31) $\tau \curvearrowright v_1 - \sigma \sqsubseteq \omega$

but according to definition of jump expressions (definition 3.16) we get:

(32) $v_1 \in \mathbf{NatTyp}$

(33) $v_1 \sqsubseteq \tau$

(34) $\tau \curvearrowright v_1 = v_1$

Using (34) we can substitute v_1 for $\tau \curvearrowright v_1$ in (31) which yields:
$v_1 - \sigma \sqsubseteq \omega$
By well-formedness of the above formula we get:

(35) $v_1 \not\sqsubseteq \sigma$

Now by applying theorem 3.48 we obtain:
$v_1 = (\omega \mid \sigma) \;\vee\; v_1 \sqsubseteq \omega$

- If the first case is true, then obviously v_1 is an answer to $V = (\omega \mid \sigma)$, and by (32) and (33), this answer is also a natural subtype of τ, and the proposition holds for this case. □
- If the second case is true, then $\tau \curvearrowright v_1 \sqsubseteq \omega$ is also true, because $\tau \curvearrowright v_1 = v_1$. This means v_1 is an answer to $\tau \curvearrowright V \sqsubseteq \omega$, and because of (35) this answer is not a subtype of σ and the proposition holds for this case as well. □

- ii) Suppose v_1 is either an answer to the equation $V = (\omega \mid \sigma)$ that is also a natural subtype of τ or an answer to $\tau \curvearrowright V \sqsubseteq \omega$ which is not a subtype of σ. We break down the disjunction into two cases:

  - Suppose v_1 is an answer to $V = (\omega \mid \sigma)$ that is a natural subtype of τ, then:

    (36) $v_1 = (\omega \mid \sigma)$

    (37) $v_1 \sqsubseteq \tau$

    Note that it is impossible for v to be a subtype of σ, otherwise the partition requires that ω be the bottom type, which contradicts the assumption that it is a natural type. By applying theorem 3.49 on (36) we obtain:

(38) $v_1 - \sigma \sqsubseteq \omega$

By definition of jump expressions and (37) $\tau \curvearrowright v_1$ is a well-formed jump expression equal to v_1 and by substituting $\tau \curvearrowright v_1$ for v_1 in (38) we get:

(39) $\tau \curvearrowright v_1 - \sigma \sqsubseteq \omega$

So v_1 is an answer to $\tau \curvearrowright V - \sigma \sqsubseteq \omega$ □

- Suppose v_1 is an answer to $\tau \curvearrowright V \sqsubseteq \omega$, which is not a subtype of σ, then we have:

  $\tau \curvearrowright v_1 \sqsubseteq \omega$

  Since $\tau \curvearrowright v_1 = v_1$ we get:

  $v_1 \sqsubseteq \omega$

  Since $v_1 \not\sqsubseteq \sigma$ theorem 3.49 can apply and we obtain:

  $v_1 - \sigma \sqsubseteq \omega$

  By the equality $\tau \curvearrowright v_1 = v_1$ we get:

  $\tau \curvearrowright v_1 - \sigma \sqsubseteq \omega$

  So v_1 is an answer to $\tau \curvearrowright V - \sigma \sqsubseteq \omega$ □

Proof of Theorem 3.54:

- i) Suppose $\tau - \sigma \sqsubseteq \omega \curvearrowright V$ is satisfiable, then there should exist a natural type v_1 such that:

  (40) $v_1 \sqsubseteq \omega$

  (41) $\tau - \sigma \sqsubseteq \omega \curvearrowright v_1$

  Since $\omega \curvearrowright v_1 = v_1$ we get:

  (42) $\tau - \sigma \sqsubseteq v_1$

  By the transitivity of $\sqsubseteq$ and (42), (40) we will have:

  $\tau - \sigma \sqsubseteq \omega$ □

- ii) Suppose $\tau - \sigma \sqsubseteq \omega$ holds, then there exists a natural type $v_1 = \omega$, which trivially respect $v_1 \sqsubseteq \omega$ for which we have:

  $\tau - \sigma \sqsubseteq v_1$

Since $\omega \curvearrowright v_1 = v_1$ we get:
$\tau - \sigma \sqsubseteq \omega \curvearrowright v_1$
So $\tau - \sigma \sqsubseteq \omega \curvearrowright V$ is satisfiable. □

Proof of Theorem 3.55:

- i) Suppose v_1 is an answer to V satisfying $\tau \curvearrowright V - \sigma \sqsubseteq \omega \curvearrowright V$ then we have:
  $\tau \curvearrowright v_1 - \sigma \sqsubseteq \omega \curvearrowright v_1$
  which by the definition of jump expressions (definition 3.16) is equivalent to:

  (43) $v_1 - \sigma \sqsubseteq v_1$

  and requires:
  $v_1 \sqsubseteq \tau$
  $v_1 \sqsubseteq \omega$
  The above requirements indicate that v_1 is a common subtype of τ and ω.
  Since (43) holds, it should be a well-formed formula, and every one of its sub-expressions must also be well-formed. In particular, $v_1 - \sigma$ must be well-formed, which by the definition of negated types (definition 3.26 it is required that:
  $v_1 \not\sqsubseteq \sigma$ □

- ii) Suppose v_1 is a common subtype of τ and ω which is not a subtype of σ. Then we have:
  And by the transitivity of $\sqsubseteq$ we get:

  (44) $v_1 \sqsubseteq \tau$

  (45) $v_1 \sqsubseteq \omega$

  (46) $v_1 \not\sqsubseteq \sigma$

  By using (44) , (45) and the definition of jump expressions (definition 3.16) we obtain that the following jump expressions are well-formed, and are both equal to v_1:

  (47) $\tau \curvearrowright v_1$

  (48) $\omega \curvearrowright v_1$

Now since we have $v_1 \not\sqsubseteq \sigma$ (46) theorem 3.27 applies and we obtain:

(49) $v_1 - \sigma \sqsubseteq v_1$

And by using (47), (48) and substituting $\tau \curvearrowright v_1$ for the v_1 on the left side of (49) and substituting $\omega \curvearrowright v_1$ for the v_1 on the right side of (49) we get:

$\tau \curvearrowright v_1 - \sigma \sqsubseteq \omega \curvearrowright v_1$

So v_1 is an answer to $\tau \curvearrowright V - \sigma \sqsubseteq \omega \curvearrowright V$ □

Proof of Theorem 3.58:

- i) Suppose $(v_1,\ \omega_1)$ is an answer to $\tau \curvearrowright V - \sigma \sqsubseteq \omega \curvearrowright X$, that is:

  $\tau \curvearrowright v_1 - \sigma \sqsubseteq \omega \curvearrowright \omega_1$

  Since $\omega \curvearrowright \omega_1 = \omega_1$, and ω_1 is required to be a subtype of ω for $\omega \curvearrowright \omega_1$ to be a well-formed expression, we get:

  $\tau \curvearrowright v_1 - \sigma \sqsubseteq \omega_1 \sqsubseteq \omega$

  And by the transitivity of $\sqsubseteq$ we obtain:

  $\tau \curvearrowright v_1 - \sigma \sqsubseteq \omega$

  So v_1 is an answer to $\tau \curvearrowright V - \sigma \sqsubseteq \omega$ □

- ii) Suppose v_1 is an answer to $\tau \curvearrowright V - \sigma \sqsubseteq \omega$, then we have:

  $\tau \curvearrowright v_1 - \sigma \sqsubseteq \omega$

  Since $\omega \sqsubseteq \omega$, $\omega \curvearrowright \omega$ is a well-formed jump expression equal to ω, we can substitute $\omega \curvearrowright \omega$ for ω in the above formula which yields:

  $\tau \curvearrowright v_1 - \sigma \sqsubseteq \omega \curvearrowright \omega$

  That is, (v_1, ω) is an answer to $\tau \curvearrowright V - \sigma \sqsubseteq \omega \curvearrowright X$ □

Proof of Theorem 3.59:

- i) Suppose $\tau \sqsubseteq \omega \curvearrowright V - \sigma$ is satisfiable then for a natural type v_1 we have:

  $v_1 \sqsubseteq \omega$

  $\tau \sqsubseteq \omega \curvearrowright v_1 - \sigma$

  But according to the definition of jump expressions (definition 3.16) $\omega \curvearrowright v_1 = v_1$ and we get:

$\tau \sqsubseteq v_1 - \sigma$

Now by applying theorem 3.51 we obtain:

(50) $\tau \parallel \sigma$

(51) $\tau \sqsubseteq v_1$

Since $\omega \curvearrowright v_1 = v_1$ we can substitute $\omega \curvearrowright v_1$ for v_1 in the last formula and we get:

$\tau \sqsubseteq \omega \curvearrowright v_1$

This indicates that v_1 is also an answer to $\tau \sqsubseteq \omega \curvearrowright V$

Together with what we derived in (50) it proves the proposition. $\Box$

- ii) Suppose $\tau \parallel \sigma$, and that v_1 is an answer to $\tau \sqsubseteq \omega \curvearrowright V$ then we get:

(52) $v_1 \sqsubseteq \omega$

(53) $\tau \sqsubseteq \omega \curvearrowright v_1$

Since $\omega \curvearrowright v_1 = v_1$, we get:

(54) $\tau \sqsubseteq v_1$

Note that it is impossible that $v_1 \sqsubseteq \sigma$ because in that case every subtype of ω, including τ must also be a subtype of σ, but according to the supposition, $\tau \parallel \sigma$. So,

$v_1 \not\sqsubseteq \sigma$

Applying theorem 3.51 on (54) and the supposition of (ii) that $\tau \parallel \sigma$ we obtain:

(55) $\tau \sqsubseteq v_1 - \sigma$

Since $\omega \curvearrowright v_1 = v_1$ we can substitute $\omega \curvearrowright v_1$ for v_1 in (55) which yields:

(56) $\tau \sqsubseteq \omega \curvearrowright v_1 - \sigma$

This means that v_1 is also an answer to $\tau \sqsubseteq \omega \curvearrowright V - \sigma$ $\Box$

Proof of Theorem 3.60:

- i) Suppose $\tau \curvearrowright V \sqsubseteq \omega - \sigma$ is satisfiable and v_1 is one of its answers, then:

$\tau \curvearrowright v_1 \sqsubseteq \omega - \sigma$

Since $\tau \curvearrowright v_1 = v_1$ we get:

$v_1 \sqsubseteq \omega - \sigma$

By applying theorem 3.51 we get:

(57) $v_1 \parallel \sigma$

(58) $v_1 \sqsubseteq \omega$

Again since $\tau \curvearrowright v_1 = v_1$, the last formula can be re-written as:

$\tau \curvearrowright v_1 \sqsubseteq \omega$

That is, v_1 is also an answer to $\tau \curvearrowright V \sqsubseteq \omega$

And by (57) this answer is disjoint with σ. □

- ii) Suppose v_1 is an answer to $\tau \curvearrowright V \sqsubseteq \omega$ that is disjoint with σ, then we have:

  (59) $v_1 \parallel \sigma$

  (60) $\tau \curvearrowright v_1 \sqsubseteq \omega$

  Since $\tau \curvearrowright v_1 = v_1$, the last formula can be written as:

  (61) $v_1 \sqsubseteq \omega$

  By applying theorem 3.51 on (59) and (61) we get:

  (62) $v_1 \sqsubseteq \omega - \sigma$

  Again since $\tau \curvearrowright v_1 = v_1$ we can derive:

  (63) $\tau \curvearrowright v_1 \sqsubseteq \omega - \sigma$

  That is, v_1 is also an answer to $\tau \curvearrowright V \sqsubseteq \omega - \sigma$ □

Proof of Theorem 3.61:

By contradiction: Suppose $\mu_0 \sqsubseteq \mu$ is not disjoint with σ, that is, there is an instance $x : \mu_0$, which is also an instance of σ. But by axiom (SUB), x is also an instance of μ, and we have an object that is both an instance of μ and σ, so μ, σ are not disjoint, which contradicts the supposition of the theorem.

Proof of Theorem 3.64:

- i) Suppose v_1 is an answer to $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright V - \sigma$, then we have:

  $\tau \curvearrowright v_1 \sqsubseteq \omega \curvearrowright v_1 - \sigma$

  This requires v_1 to be a subtype of both τ and ω for $\tau \curvearrowright v_1$, and $\omega \curvearrowright v_1$ to be well-formed.

  Since $\tau \curvearrowright v_1 = \omega \curvearrowright v_1 = v_1$ we get:

  $v_1 \sqsubseteq v_1 - \sigma$

  By applying theorem 3.51 we get:

  $v_1 \sqsubseteq v_1$, which trivially holds, and

  $v_1 \parallel \sigma$

  This proves that v_1 is a common subtype of τ and ω that is disjoint with σ □

- ii) Suppose v_1 is a common subtype of τ and ω that is disjoint with σ then we have:

  $v_1 \sqsubseteq v_1$, which is trivially true, and

  $v_1 \parallel \sigma$

  And by applying theorem 3.51 we derive:

  $v_1 \sqsubseteq v_1 - \sigma$

  Since v_1 is a subtype of τ and ω, according to definition of jump expressions (definition 3.16), the expressions $\tau \curvearrowright v_1$, $\omega \curvearrowright v_1$ are well-formed and are equal to v_1, and we can re-write the above formula as:

  $\tau \curvearrowright v_1 \sqsubseteq \omega \curvearrowright v_1 - \sigma$

  That is, v_1 is an answer to $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright V - \sigma$ □

Proof of Lemma 3.66:

According to corollary 3.2 it is enough to prove:

$I_{\omega_1 - \sigma} \subseteq I_{\omega - \sigma}$

For an arbitrary object $x \in I_{\omega_1 - \sigma}$ we have:

$x \in I_{\omega_1} \cap I'_{\sigma}$

$\Longrightarrow x \in I_{\omega_1} \wedge x \notin I_{\sigma}$

$\Longrightarrow x \in I_{\omega_1} \wedge x \notin I_{\sigma}$

Since $\omega_1 \sqsubseteq \omega$ we have $I_{\omega_1} \subseteq I_{\omega}$, and we can derive:

$\Longrightarrow x \in I_{\omega} \wedge x \notin I_{\sigma}$ (by definition of subsets in set theory, and their application on x)

$\Longrightarrow x \in I_\omega \cap I'_\sigma$ (from set theory results)
$\Longrightarrow x \in I_{\omega-\sigma}$ (by the definition of negated types)
So, any arbitrary member x of $I_{\omega_1-\sigma}$, is also a member of $I_{\omega-\sigma}$
And by the definition of subsets in set theory we get:
$I_{\omega_1-\sigma} \subseteq I_{\omega-\sigma}$

Proof of Theorem 3.67:

- i) Suppose $(v_1,\ \omega_1)$ is an answer to $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright X - \sigma$, then we have:

  (64) $\tau \curvearrowright v_1 \sqsubseteq \omega \curvearrowright \omega_1 - \sigma$

  By the definition of jump expressions we have $\omega \curvearrowright \omega_1 = \omega_1$ and by substituting ω_1 for $\omega \curvearrowright \omega_1$ in (64) we get:

  (65) $\tau \curvearrowright v_1 \sqsubseteq \omega_1 - \sigma$

  By lemma 3.66 we have:

  (66) $\omega_1 - \sigma \sqsubseteq \omega - \sigma$

  By the transitivity of $\sqsubseteq$ and (65) and (66) we derive:

  (67) $\tau \curvearrowright v_1 \sqsubseteq \omega - \sigma$

  That is, v_1 is an answer to $\tau \curvearrowright V \sqsubseteq \omega - \sigma$ $\Box$

- ii) Suppose v_1 is an answer to $\tau \curvearrowright V \sqsubseteq \omega - \sigma$, then we have:
  $\tau \curvearrowright v_1 \sqsubseteq \omega - \sigma$
  Since $\omega \curvearrowright \omega = \omega$ we can substitute $\omega \curvearrowright \omega$ for ω in the above formula and we get:
  $\tau \curvearrowright v_1 \sqsubseteq \omega \curvearrowright \omega - \sigma$
  This means that $(v_1,\ \omega)$ is an answer to $\tau \curvearrowright V \sqsubseteq \omega \curvearrowright X - \sigma$ $\Box$

Proof of Theorem 3.68:

- i) Suppose v_1 is an answer to $\tau \curvearrowright V - \sigma \sqsubseteq \omega - \sigma$ then we have:

  (68) $\tau \curvearrowright v_1 - \sigma \sqsubseteq \omega - \sigma$

By theorem 3.27 we have:

(69) $\omega - \sigma \sqsubseteq \omega$

By the transitivity of $\sqsubseteq$ and applying it on (68) and (69) we derive:

(70) $\tau \curvearrowright v_1 - \sigma \sqsubseteq \omega$

That is, v_1 is an answer to $\tau \curvearrowright V - \sigma \sqsubseteq \omega$ $\Box$

- ii) Suppose v_1 is an answer to $\tau \curvearrowright V - \sigma \sqsubseteq \omega$, that is:

(71) $\tau \curvearrowright v_1 - \sigma \sqsubseteq \omega$

Since $\tau \curvearrowright v_1 = v_1$ according to the definition of jump expressions (definition 3.16), we can substitute v_1 for $\tau \curvearrowright v_1$ in (71) and we get:

(72) $v_1 - \sigma \sqsubseteq \omega$

Now for an arbitrary object x suppose $x \in I_{v_1 - \sigma}$ then we have:

(73) $x : v_1 - \sigma$

By applying (NEG) we get:

(74) $x : v_1 \wedge \neg(x : \sigma)$

So.

(75) $\neg(x : \sigma)$

By applying (SUB) on (72) we get:

(76) $x : \omega$

Now by applying (NEG) on (76) and (75) we get:

(77) $x : \omega - \sigma$

That is:

(78) $x \in I_{\omega - \sigma}$

So any arbitrary instance of $I_{v_1-\sigma}$ is also an instance of $I_{\omega-\sigma}$ and by set theory results we get:

(79) $I_{v_1-\sigma} \subseteq I_{\omega-\sigma}$

And by corollary 3.2 we get:

(80) $v_1 - \sigma \sqsubseteq \omega - \sigma$

Since $\tau \curvearrowright v_1 = v_1$ according to the definition of jump expressions (definition 3.16). we can substitute $\tau \curvearrowright v_1$ for v_1 in (80) and we get:

(81) $\tau \curvearrowright v_1 - \sigma \sqsubseteq \omega - \sigma$

That is, v_1 is also an answer to $\tau \curvearrowright V - \sigma \sqsubseteq \omega - \sigma$ □

Proof of Theorem 3.69:

- i) Suppose $\tau - \sigma \sqsubseteq \omega \curvearrowright V - \sigma$ is satisfiable then for a natural type v_1 we have:

  (82) $v_1 \sqsubseteq \omega$

  (83) $\tau - \sigma \sqsubseteq \omega \curvearrowright v_1 - \sigma$

  Since by the definition of jump expressions (definition3.16) we have $\omega \curvearrowright v_1 = v_1$ we can substitute v_1 for $\omega \curvearrowright v_1$ in the above formula which yields:

  (84) $\tau - \sigma \sqsubseteq v_1 - \sigma$

  The well-formedness of the above formula (for the jump expression $v_1 - \sigma$) requires that:

  (85) $v_1 \not\sqsubseteq \sigma$

  Now by applying lemma 3.66 on (82) and (85) we get:

  (86) $v_1 - \sigma \sqsubseteq \omega - \sigma$

  By the transitivity of $\sqsubseteq$ and (84) and (86) we derive:
  $\tau - \sigma \sqsubseteq \omega - \sigma$ □

- ii) Suppose $\tau - \sigma \sqsubseteq \omega - \sigma$ holds.

  By the definition of jump expressions (definition 3.16) we have $\omega \curvearrowright \omega = \omega$ and we can substitute $\omega \curvearrowright \omega$ for ω in the above formula and we get:

  $\tau - \sigma \sqsubseteq \omega \curvearrowright \omega - \sigma$

  This means there is a an answer $v_1 = \omega$ for V satisfying:

  $\tau - \sigma \sqsubseteq \omega \curvearrowright V - \sigma$

  That is, $\tau - \sigma \sqsubseteq \omega \curvearrowright V - \sigma$ is satisfiable. □

Proof of Theorem 3.70:

- a) Suppose v_1 is an answer to V satisfying $\tau \curvearrowright V - \sigma \sqsubseteq \omega \curvearrowright V - \sigma$, then we have:

  (87) $\tau \curvearrowright v_1 - \sigma \sqsubseteq \omega \curvearrowright v_1 - \sigma$

  which must be a well-formed formula, and thus contain well-formed (sub)expressions. So $\tau \curvearrowright v_1$, and $\omega \curvearrowright v_1$ must be well-formed and according to the definition of jump expressions (definition 3.16) this requires:

  (88) $v_1 \sqsubseteq \tau$

  (89) $v_1 \sqsubseteq \omega$

  What remains to be proved is that $v_1 \not\sqsubseteq \sigma$. But again according to the definition of jump expressions $\tau \curvearrowright v_1 = v_1$, and $\omega \curvearrowright v_1 = v_1$, and it is valid to substitute v_1, v_1 for $\tau \curvearrowright v_1$, $\omega \curvearrowright v_1$ in (87), which yields:

  (90) $v_1 - \sigma \sqsubseteq v_1 - \sigma$

  Again, since the above formula and any of its sub-expressions must be well-formed, we can claim that $v_1 - \sigma$ is well-formed, and according to the definition of negated types (definition 3.26) it is required that:

  (91) $v_1 \not\sqsubseteq \sigma$

  Now (88), (89) prove that v_1 is a common subtype of τ, ω, and (91) states that v_1 is not a subtype of σ □

- b) Suppose v_1 is a common subtype of τ, ω which is not a subtype of σ, then by the definition of negated types (definition 3.26), $v_1 - \sigma$ is well-formed, and since $\sqsubseteq$ is a reflexive relation we can claim:

  (92) $v_1 - \sigma \sqsubseteq v_1 - \sigma$

  Now since v_1 is a subtype of τ, and ω, then according to the definition of jump expressions (definition 3.16), $\tau \curvearrowright v_1$, and $\omega \curvearrowright v_1$ are well-formed jump expressions which are both equal to v_1, so we can substitute $\tau \curvearrowright v_1$ for the v_1 on the left hand side of $\sqsubseteq$ in (92, and also substitute $\omega \curvearrowright v_1$ for the v_1 on the right hand side of $\sqsubseteq$ in the same formula, preserving its truth. This yields:

  (93) $\tau \curvearrowright v_1 - \sigma \sqsubseteq \omega \curvearrowright v_1 - \sigma$

  And this means that v_1 is an answer to V satisfying:
  $\tau \curvearrowright V - \sigma \sqsubseteq \omega \curvearrowright V - \sigma \quad \Box$

Proof of Theorem 3.72:

- a) Suppose (v_1, x_1) is an answer to (V, X) satisfying $\tau \curvearrowright V - \sigma \sqsubseteq \omega \curvearrowright X - \sigma$ then we have:

  (94) $\tau \curvearrowright v_1 - \sigma \sqsubseteq \omega \curvearrowright x_1 - \sigma$

  According to the definition of jump expressions (definition 3.16) for the above formula to be a well-formed one, it is required that $\omega \curvearrowright x_1$ be well-formed and that requires:

  (95) $x_1 \sqsubseteq \omega$

  And $\omega \curvearrowright x_1$ is equal to x_1, and the subexpression $\omega \curvearrowright x_1 - \sigma$ in (94) is equal to $x_1 - \sigma$. So we can substitute $x_1 - \sigma$ for $\omega \curvearrowright x_1 - \sigma$ in (94) and this yields the following formula, preserving its validity:

  (96) $\tau \curvearrowright v_1 - \sigma \sqsubseteq x_1 - \sigma$

  According to the definition of negated types (definition 3.26) the well-formedness of $x_1 - \sigma$ requires:

  (97) $x_1 \not\sqsubseteq \sigma$

The assumption of the theorem already contains:

(98) $\omega \not\sqsubseteq \sigma$

Now by applying theorem 3.52 on (95), (97), (98) we obtain:

(99) $x_1 - \sigma \sqsubseteq \omega - \sigma$

By applying the transitivity of $\sqsubseteq$ on (96), (99) we get:

(100) $\tau \curvearrowright v_1 - \sigma \sqsubseteq \omega - \sigma$

So, v_1 is an answer to V satisfying $\tau \curvearrowright V - \sigma \sqsubseteq \omega - \sigma$ □

- b) Suppose v_1 is an answer to V satisfying $\tau \curvearrowright V - \sigma \sqsubseteq \omega - \sigma$, then we have:

  (101) $\tau \curvearrowright v_1 - \sigma \sqsubseteq \omega - \sigma$

  Since by the definition of jump expressions (definition 3.16) we have $\omega \curvearrowright \omega = \omega$ we can substitute $\omega \curvearrowright \omega$ for ω in (101) and we get:

  (102) $\tau \curvearrowright v_1 - \sigma \sqsubseteq \omega \curvearrowright \omega - \sigma$

  So, (v_1, ω) is an answer to (V, X) satisfying $\tau \curvearrowright V - \sigma \sqsubseteq \omega \curvearrowright X - \sigma$ □

Proof of Theorem 7.2:

We prove the theorem by the following 4 lemmas that refer to these formulas:

(103) $\tau \sqsubseteq all \curvearrowright X - human$

(104) $\tau \sqsubseteq \omega$

(105) $\tau \sqsubseteq snh(\omega)$

Each lemma covers a possible case for τ. I.e., τ can be natural, negated but not specializable, specializable but not negated, and negated and specializable.

Lemma B.1 For a natural type τ, the requirements (103) and (104) are equivalent to (105).

Proof: For a natural τ:

- $\Rightarrow$ Suppose (103) and (104) hold we need to prove: $\tau \sqsubseteq snh(\omega)$

  (103) implies that there is a natural type x such that $\tau \sqsubseteq x - human$. Then by theorem 3.51 we get:

  (106) $x \not\sqsubseteq human$

  (107) $\tau \sqsubseteq x$

  (108) $\tau \parallel human$

  - Assume ω is natural, then $snh(\omega) = \omega \curvearrowright Y - human$, for a free variable Y. We need to prove: $\tau \sqsubseteq \omega \curvearrowright Y - human$

    If we apply theorem 3.14 to (104) and (107) we get:

    (109) $\tau \sqsubseteq x * \omega$

    On the other hand if $x * \omega \sqsubseteq human$, we will have:

    (110) $\tau \sqsubseteq x * \omega \sqsubseteq human \Rightarrow \tau \sqsubseteq human \Rightarrow \tau \nparallel human$

    , which is a contradiction. So:

    (111) $x * \omega \not\sqsubseteq human$

    And by theorem 3.51 we get:

    (112) $\tau \sqsubseteq x * \omega - human$

    , which means $\tau \sqsubseteq snh(\omega) = \omega \curvearrowright Y - human$ has at least the answer $x * \omega$ and hence is satisfiable. □

  - Assume $\omega = \omega' - human$, for a natural ω' then $snh(\omega) = \omega' \curvearrowright Y - human$, for a free variable Y. We must prove $\tau \sqsubseteq \omega' \curvearrowright Y' - human$

    (104) states that $\tau \sqsubseteq \omega' - human$. By theorem 3.51 this implies:

    (113) $\tau \sqsubseteq \omega'$

    Then by substituting ω' for ω and using the above statement instead of (104) in the proof of the previous case up to (112) we derive:

    (114) $\tau \sqsubseteq x * \omega' - human$

    , which means $\tau \sqsubseteq \omega' \curvearrowright Y' - human$ is satisfiable with an answer equal to $x * \omega'$ □

- Assume $\omega = \omega' \curvearrowright Y$ for a free variable Y, then $snh(\omega) = \omega' \curvearrowright Y' - human$ for a free variable Y'. We need to prove $\tau \sqsubseteq \omega' \curvearrowright Y' - human$

  (104) holds and is equivalent to $\tau \sqsubseteq \omega' \curvearrowright Y$, which must be satisfiable. Supose y is one of its answers, then we have:

  (115) $\tau \sqsubseteq \omega' \curvearrowright y$

  , which requires:

  (116) $y \sqsubseteq \omega'$

  and is equivalent to:

  (117) $\tau \sqsubseteq y$

  Applying the theorem 3.14 to the above result and (107) we get:

  (118) $\tau \sqsubseteq x * y$

  Since (108) requires that τ be disjoint from *human*, $x * t$ cannot be a subtype of *human* and we have:

  (119) $x * y \not\sqsubseteq human$

  And by theorem 3.51 we obtain

  (120) $\tau \sqsubseteq x * y - human$

  Since $x * y \sqsubseteq \omega'$ we can rewrite the above statement as:

  (121) $\tau \sqsubseteq \omega' \curvearrowright x * y - human$

  , which means that $\tau \sqsubseteq snh(\omega) = \omega' \curvearrowright Y' - human$ has at least the answer $x * y$ and hence is satisfiable. □

- Assume $\omega = \omega' \curvearrowright Y - human$, then $snh(\omega) = \omega' \curvearrowright Y' - human$ for a free variable Y'. We need to prove $\tau \sqsubseteq \omega' \curvearrowright Y' - human$

  By (108) we already know that τ and *human* are disjoint. By theorem 3.59 $\tau \sqsubseteq \omega' \curvearrowright Y - human$ is equivalent to $\tau \sqsubseteq \omega' \curvearrowright Y$. So all the statements for the previous case still apply, and the proof is exactly the same. □

- $\Leftarrow$ Suppose $\tau \sqsubseteq snh(\omega)$ we need to prove that (103) and (104) hold.

  - Assume ω is natural. Then $snh(\omega) = \omega \curvearrowright Y - human$, and:

    $\tau \sqsubseteq snh(\omega) = \omega \curvearrowright Y - human$ is satisfiable. Suppose it has an answer x. Then we have:

(122) $\tau \sqsubseteq \omega \curvearrowright x - human$

, which requires:

(123) $x \sqsubseteq \omega$

and is equivalent to:

(124) $\tau \sqsubseteq x - human$

This means that x is an answer to $\tau \sqsubseteq all \curvearrowright X - human$, and hence (103) holds. So theorem 3.51 obtains the results (106), (107), (108). By combining (107) and (123) and the transitivity of $\sqsubseteq$ we get:

(125) $\tau \sqsubseteq \omega$

, which proves (104). □

- Assume $\omega = \omega' - human$, for a natural ω' then $snh(\omega) = \omega' \curvearrowright Y - human$, for a free variable Y, and $\tau \sqsubseteq snh(\omega) = \omega' \curvearrowright Y - human$ is satisfiable. Suppose it has an answer x. Then by a proof very similar to the previous case and replacing ω' for ω, (103) is proved, and we can derive

  (126) $\tau \sqsubseteq \omega'$

  Then by theorem 3.51 we obtain:

  (127) $\tau \sqsubseteq \omega' - human = \omega$

  , which proves (104). □

- Assume $\omega = \omega' \curvearrowright Y$ for a free variable Y, then $snh(\omega) = \omega' \curvearrowright Y' - human$ for a free variable Y' and we have:
  $\tau \sqsubseteq snh(\omega) = \omega' \curvearrowright Y' - human$ is satisfiable. Suppose x is one of its answers, then:

  (128) $\tau \sqsubseteq \omega' \curvearrowright x - human$

  , which requires:

  (129) $x \sqsubseteq \omega'$

  and is equivalent to:

  (130) $\tau \sqsubseteq x - human$

  This means that x is an answer to $\tau \sqsubseteq all \curvearrowright X - human$, and hence (103) holds. So theorem 3.51 obtains the results (106), (107), (108). By combining (107) and (129) and the transitivity of $\sqsubseteq$ we get:

(131) $\tau \sqsubseteq \omega'$

, which implies $\tau \sqsubseteq \omega' \curvearrowright Y$ has a trivial answer ω' and thus is satisfiable, and this proves (104). □

- Assume $\omega = \omega' \curvearrowright Y - human$ for a free variable Y, then $snh(\omega) = \omega' \curvearrowright Y' - human$ for a free variable Y', which is the same value for the previous case. The proof is the same as the previous case till the statement $\tau \sqsubseteq \omega' \curvearrowright Y$ is derived. There is only one additional step needed afterwards. Since by the result (107) that is obtained in the proof we know that x and $human$ are disjoint and by theorem 3.59 $\tau \sqsubseteq \omega' \curvearrowright Y$ (that is proved) is equivalent to $\tau \sqsubseteq \omega' \curvearrowright Y - human$. So (104) holds in this case too. □

■

Lemma B.2 For a specializable type $\tau = \tau' \curvearrowright V$, the requirements (103) and (104) are equivalent to (105).

Proof: For a specializable $\tau = \tau' \curvearrowright V$:
The equations (103), (104), (105) hold if and only if there is a natural subtype of τ', τ_1 for V that satisfies (103), (104), (105). So the proof is similar to the above after replacing τ with τ_1.

■

Lemma B.3 For a negated τ which is equal to $\tau' - human$ for a natural type τ', the requirements (103) and (104) are equivalent to (105).

Proof: For a negated τ which is equal to $\tau' - human$ for a natural type τ':
In this case, the well-formed-ness of $\tau' - human$ requires that:

(132) $\tau' \not\sqsubseteq human$

And (103) trivially holds by the choice of $X = \tau'$. So we need to only prove:

(133) $\tau = \tau' - human \sqsubseteq \omega \Leftrightarrow \tau \sqsubseteq snh(\omega)$

We break down the theorem to the cases we have for ω in $snh(\omega)$.

- Assume ω is natural. So, $snh(\omega) = \omega \curvearrowright Y - human$, and we have:
  $\tau' - human \sqsubseteq \omega$
  $\Longleftrightarrow \tau' = (\omega \mid human) \ \vee \ \tau' \sqsubseteq \omega$ (by theorem 3.50)
  If $\tau' \sqsubseteq \omega$ then $\omega \not\sqsubseteq human$, because its negation would imply $\tau' \sqsubseteq human$, which is a contradiction to (132). If $\tau' = (\omega \mid human)$ then ω and $human$ are disjoint and necessarily $\omega \not\sqsubseteq human$. So:
  $\Longleftrightarrow \tau' = (\omega \mid human) \ \vee \ \tau' \sqsubseteq \omega$ and $\omega \not\sqsubseteq human$
  $\Longleftrightarrow \tau' - human \sqsubseteq \omega - human$ (by theorem 3.52)
  $\Longleftrightarrow \tau' - human \sqsubseteq \omega \curvearrowright Y - human$ (by theorem 3.69)
  $\Longleftrightarrow \tau \sqsubseteq snh(\omega) \quad \square$

- Assume $\omega = \omega' - human$ for a natural type ω'. Then $snh(\omega) = \omega' \curvearrowright Y - human$ and the well-formed-ness of ω requires:

  (134) $\omega' \not\sqsubseteq human$

  We have:
  $\tau' - human \sqsubseteq \omega$
  $\Longleftrightarrow \tau' - human \sqsubseteq \omega' - human$
  $\Longleftrightarrow \tau' - human \sqsubseteq \omega' \curvearrowright Y - human$ (by theorem 3.69)
  $\Longleftrightarrow \tau \sqsubseteq snh(\omega)$

- Assume $\omega = \omega' \curvearrowright Y$ for a free variable Y, then $snh(\omega) = \omega' \curvearrowright Y' - human$, for a free Y'. The well-formed-ness of ω requires that:

  (135) $\omega' \sqsubseteq \omega$

  We have:
  $\tau' - human \sqsubseteq \omega$
  $\Longleftrightarrow \tau' - human \sqsubseteq \omega' \curvearrowright Y$
  $\Longleftrightarrow \tau' - human \sqsubseteq \omega' \curvearrowright y$ (supposing y is an answer)
  $\Longleftrightarrow \tau' - human \sqsubseteq y \wedge y \sqsubseteq \omega'$
  $\Longleftrightarrow \tau' = (y \mid human) \ \vee \ \tau' \sqsubseteq y$ (by theorem 3.50) and $y \sqsubseteq \omega'$
  If $\tau' \sqsubseteq y$ then $y \not\sqsubseteq human$, because it would imply $\tau' \sqsubseteq human$, which is a contradiction to (132). If $\tau' = (y \mid human)$ then y and $human$ are disjoint and necessarily $y \not\sqsubseteq human$. So:

$\Longleftrightarrow \tau' = (y \mid human) \ \vee \ \tau' \sqsubseteq y$ (by theorem 3.50) and $y \sqsubseteq \omega'$ and $y \not\sqsubseteq human$
$\Longleftrightarrow \tau' - human \sqsubseteq y - human$ (by theorem 3.52) and $y \sqsubseteq \omega'$
$\Longleftrightarrow \tau' - human \sqsubseteq \omega' \curvearrowright Y' - human$ (supposing y is an answer)
$\Longleftrightarrow \tau \sqsubseteq snh(\omega)$ $\quad\square$

- Assume $\omega = \omega' \curvearrowright Y - human$ for a free variable Y, then $snh(\omega) = \omega' \curvearrowright Y' - human$, for a free Y'. The well-formed-ness of ω requires that:

  (136) $\omega' \sqsubseteq \omega$

  We have:
  $\tau' - human \sqsubseteq \omega$
  $\Longleftrightarrow \tau' - human \sqsubseteq \omega' \curvearrowright Y - human$
  $\Longleftrightarrow \tau' - human \sqsubseteq \omega' \curvearrowright y - human$ (supposing y is an answer)
  $\Longleftrightarrow \tau' - human \sqsubseteq y \wedge y \sqsubseteq \omega' \wedge y \not\sqsubseteq human$
  $\Longleftrightarrow \tau' = (y \mid human) \ \vee \ \tau' \sqsubseteq y$ (by theorem 3.50) and $y \sqsubseteq \omega'$ and $y \not\sqsubseteq human$
  $\Longleftrightarrow \tau' - human \sqsubseteq y - human$ (by theorem 3.52) and $y \sqsubseteq \omega'$
  $\Longleftrightarrow \tau' - human \sqsubseteq \omega' \curvearrowright Y' - human$ (supposing y is an answer)
  $\Longleftrightarrow \tau \sqsubseteq snh(\omega)$ $\quad\square$

■

Lemma B.4 For a negated and specializable τ which is equal to $\tau' \curvearrowright V - human$ for a natural type τ', and a free variable V, the requirements (103) and (104) are equivalent to (105).

PROOF: For a negated and specializable τ which is equal to $\tau' \curvearrowright V - human$ for a natural type τ', and a free variable V:
The equations (103), (104), (105) hold if and only if there is a natural subtype of τ', τ_1 for V such that $\tau_1 \not\sqsubseteq human$ that satisfies (103), (104), (105). So the proof is similar to the above after replacing τ with τ_1.

■

Appendix C

Feature Types and their Permissible Values

In table C.1, for each feature structure type we have shown an AVM with the features introduced by that type, however, instead of values, we provided permissible value types or a set of permissible values for that feature. If the set of permissible values is a singleton, the only element is shown rather than the set. Permissible value types are italicized, except for **BasTyp** which is the type that we brought from chapters two and three to allow for domain specific type hierarchies for *domain entities* (rather than *grammar entities* whose type hierarchy is provided in figure 4.1). Our contributions or modifications are underlined.

The reader will probably notice that in table C.1 some subtypes repeat a feature of their super-type. In that case, a new feature is not introduced, but rather a constraint is imposed on its admissible values. For example the only acceptable value for the feature NUM of a feature structure of type *2sing* is *2*.

Table C.1: Feature Types and Permissible Values

Type	Features with their Types or Permissible Values	Direct Super-type
sign	$\begin{bmatrix} \text{ORTH} & \textit{list of string} \\ \text{SYN} & \textit{syn-cat} \\ \text{SEM} & \textit{sem-cat} \end{bmatrix}$	*feat-struct*

Table C.1: Feature Types and Permissible Values (cont.)

Type	Features with their Types or Permissible Values	Direct Super-type
expression		*sign*
lex-sign	$\begin{bmatrix}\text{ARG-ST} & \textit{list of expression}\\ \underline{\text{ARG-ST-GUARDS}} & \underline{\textit{list of list of guards}}\end{bmatrix}$	*sign*
word		*lex-sign*
lexeme		*lex-sign*
syn-cat	$\begin{bmatrix}\text{HEAD} & \textit{pos}\\ \text{VAL} & \textit{val-cat}\\ \text{GAP} & \textit{list of expression}\\ \underline{\text{GAP-GUARDS}} & \underline{\textit{list of list of guards}}\\ \text{STOP-GAP} & \textit{list of expression}\end{bmatrix}$	*feat-struct*
sem-cat	$\begin{bmatrix}\text{MODE} & \left\{\text{prop, ques, dir, ref, ana, none}\right\}\\ \text{INDEX} & \textit{index}\\ \underline{\text{TYPE}} & \underline{\textbf{BasTyp}}\\ \text{RESTR} & \textit{list of predication}\end{bmatrix}$	*feat-struct*
<u>*sem-det*</u>	$\begin{bmatrix}\text{QRESTR} & \textit{situation}\\ \text{QSCOPE} & \textit{situation}\end{bmatrix}$	*sem-cat*
<u>*sem-nom-co-conj*</u>	$\begin{bmatrix}\text{COMPONENT1} & \textit{individual}\\ \text{COMPONENT2} & \textit{individual}\end{bmatrix}$	*sem-cat*
<u>*sem-pred-co-conj*</u>	$\begin{bmatrix}\text{COMPONENT1} & \textit{situation}\\ \text{COMPONENT2} & \textit{situation}\end{bmatrix}$	*sem-cat*

Table C.1: Feature Types and Permissible Values (cont.)

Type	Features with their Types or Permissible Values	Direct Super-type
val-cat	$\left[\begin{array}{ll}\text{SPR} & \textit{list of expression}\\ \underline{\text{SPR-GUARDS}} & \underline{\textit{list of list of guards}}\\ \text{COMPS} & \textit{list of expression}\\ \underline{\text{COMPS-GUARDS}} & \underline{\textit{list of list of guards}}\\ \text{MOD} & \textit{list of mod-elem}\end{array}\right]$	*feat-struct*
$\underline{\textit{mod-elem}}$	$\left[\begin{array}{ll}\text{MODIFIED} & \text{expression}\\ \text{AFTER} & \left\{+,\ -\right\}\\ \underline{\text{MOD-GUARD}} & \underline{\textit{list of guards}}\end{array}\right]$	*feat-struct*
pos	$\left[\begin{array}{ll}\text{FORM} & \left\{\begin{array}{l}\text{fin, base, prp, psp,}\\ \text{nform, aform, ...}\end{array}\right\}\\ \text{PRED} & \left\{+,\ -\right\}\end{array}\right]$	*feat-struct*
adj		*pos*
adv		*pos*
adv-pol		*pos*
qword		*pos*
prep		*pos*
agr-pos	$\left[\text{AGR}\quad \textit{agr-cat}\right]$	*pos*
verb	$\left[\begin{array}{ll}\text{AUX} & \left\{+,\ -\right\}\\ \text{INV} & \left\{+,\ -\right\}\\ \underline{\text{GAP-TYPE}} & \underline{\left\{\text{gsubj, gnosubj}\right\}}\end{array}\right]$	*agr-pos*

Table C.1: Feature Types and Permissible Values (cont.)

Type	Features with their Types or Permissible Values	Direct Super-type
noun	$\begin{bmatrix} \text{CASE} & \{\text{nom, acc}\} \\ \text{PRO} & \{+, -\} \\ \underline{\text{TYPE-DEF}} & \underline{\{+, -\}} \end{bmatrix}$	*agr-pos*
det	$\begin{bmatrix} \text{COUNT} & \{+, -\} \end{bmatrix}$	*agr-pos*
conj		*pos*
$\underline{\textit{co-conj}}$		*conj*
$\underline{\textit{nom-co-conj}}$	$\begin{bmatrix} \text{CONJ-TYPE} & \{\text{minpl, com3sg, proxim}\} \end{bmatrix}$	*co-conj*
$\underline{\textit{pred-co-conj}}$		*co-conj*
$\underline{\textit{sub-conj}}$		*conj*
agr-cat	$\begin{bmatrix} \text{PER} & \{\mathit{1}, \mathit{2}, \mathit{3}\} \\ \text{NUM} & \{\text{sg, pl}\} \end{bmatrix}$	*feat-struct*
3sing	$\begin{bmatrix} \text{PER} & \mathit{3} \\ \text{NUM} & \text{sg} \\ \text{GEND} & \{\text{fem, masc, neut}\} \end{bmatrix}$	*agr-cat*
non-3sing		*agr-cat*
1sing	$\begin{bmatrix} \text{PER} & \mathit{1} \\ \text{NUM} & \text{sg} \end{bmatrix}$	*non-3sing*
non-1sing		*non-3sing*

Table C.1: Feature Types and Permissible Values (cont.)

Type	Features with their Types or Permissible Values	Direct Super-type
2sing	$\begin{bmatrix} \text{PER} & \mathit{2} \\ \text{NUM} & \text{sg} \end{bmatrix}$	*non-1sing*
plural	$\begin{bmatrix} \text{NUM} & \text{pl} \end{bmatrix}$	*non-1sing*

Bibliography

[1] Russell J. Abbott. Program design by informal english descriptions. *Commun. ACM*, 26(11):882–894, 1983.

[2] Hassan Aït-Kaci and Roger Nasr. Login: A logic programming language with built-in inheritance. *J. Log. Program.*, 3(3):185–215, 1986.

[3] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. *J. Log. Program.*, 16(3):195–234, 1993.

[4] Hiyan Alshawi, editor. *The Core Language Engine (ACL-MIT Series in Natural Language Processing).* The MIT Press, May 1992.

[5] Hiyan Alshawi, Doug Arnold, Rolf Backofen, David Carter, Jeremy Lindop, Klaus Netter, Junichi Tsujii, and Hans Uszkoreit. Eurotra 6/1: Rule formalism and virtual machine design study. final report. Technical report, SRI International, Cambridge,, 1991.

[6] Paul Bennett. *A Course in Generalized Phrase Structure Grammar.* UCL Press, University College London, Gower Street, London, WC1E 6BT, 1995.

[7] BIM-SEMA. ALEP System Documentation: The ALEP Linguistic Subsystem, Version 1.0. Commission of the European Communities, March 1993.

[8] Dines Bjørner. *Software Engineering 2, Specification of Systems and Languages.* Springer-Verlag Berlin Heidelberg, New York, 2006.

[9] Patrick Blackburn, Johan Bos, and Kristina Striegnitz. *Learn Prolog Now!* College Publications (15 Jun 2006), 2006.

[10] Peter Bollen. SBVR: A Fact-Oriented OMG Standard. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *OTM Workshops*, volume 5333 of *Lecture Notes in Computer Science*, pages 718–727. Springer, 2008.

[11] Robert D. Borsley. *Modern Phrase Structure Grammar.* Blackwell Publishers Ltd, 108 Cowley Road, Oxford, OX4 1JF, UK, 1996.

[12] Bob Carpenter. *The logic of typed feature structures.* Cambridge University Press, New York, NY, USA, 1992.

[13] Bob Carpenter. *Type-Logical Semantics.* The MIT Press, 1998.

[14] Henning Christiansen. CHR Grammars. *TPLP*, 5(4-5):467–501, 2005.

[15] William F. Clocksin and Chris S. Mellish. *Programming in Prolog: Using the ISO Standard.* Springer, 2003.

[16] Alistair Cockburn. *Writing Effective Use Cases.* Addison-Wesley Professional, 2000.

[17] Alain Colmerauer. Metamorphosis grammars. In *Natural Language Communication with Computers*, pages 133–189, London, UK, 1978. Springer-Verlag.

[18] Ann Copestake. *Implementing Typed Feature Structure Grammars.* CSLI Publications, Stanford, 2002.

[19] Ann Copestake, Dan Flickinger, Rob Malouf, Susanne Riehemann, and Ivan A. Sag. Translation using minimal recursion semantics. In *In Proceedings of the Sixth International Conference on Theoretical and Methodological Issues in Machine Translation*, 1995.

[20] Ann Copestake, Dan Flickinger, Carl Pollard, and Ivan A. Sag. Minimal recursion semantics: An introduction. *Research on Language and Computation*, 3(4):281–332, December 2005.

[21] Ole-Johan Dahl and Kristen Nygaard. How Object-Oriented Programming started. Available online at `http://heim.ifi.uio.no/~kristen/FORSKNINGSDOK_MAPPE/F_OO_start.html`.

[22] Ole-Johan Dahl and Kristen Nygaard. Simula: an ALGOL-based simulation language. *Commun. ACM*, 9(9):671–678, 1966.

[23] Veronica Dahl. On database systems development through logic. *ACM Trans. Database Syst.*, 7(1):102–123, 1982.

[24] Veronica Dahl. Incomplete types for logic databases. *Applied Mathematics Letters*, 4(3):25–28, 1991.

[25] Mary Dalrymple. *Lexical Functional Grammar (Syntax and Semantics, Volume 34).* 2001.

[26] Roberto Di Cosmo, François Pottier, and Didier Rémy. Subtyping recursive types modulo associative commutative products. In *Seventh International Conference on Typed Lambda Calculi and Applications (TLCA'05)*, volume 3461 of *Lecture Notes in Computer Science*, pages 179–193, Nara, Japan, April 2005. Springer Verlag.

[27] Simon C. Dik. *The Theory of Functional Grammar, Part 1: The Structure of the Clause, second, revised edition.* Mouton de Gruyter, Berlin. Newyork, 1997.

[28] Bruce Eckel. *Thinking in Java.* Prentice Hall, 3rd edition, 2003.

[29] Gregor Erbach. Multi-dimensional inheritance. In H. Trost, editor, *Proceedings of KONVENS '94*, pages 102 – 111. Springer, 1994.

[30] Gregor Erbach. ProFIT: Prolog with Features, Inheritance and Templates. In *Proceedings of the seventh conference on European chapter of the Association for Computational Linguistics*, pages 180–187, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[31] Andrew Fall. *Reasoning with Taxonomies.* PhD thesis, Simon Fraser University, 1996.

[32] Thom W. Frühwirth. Theory and Practice of Constraint Handling Rules. *J. Log. Program.*, 37(1-3):95–138, 1998.

[33] Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. Attempto Controlled English for Knowledge Representation. In Cristina Baroglio, Piero A. Bonatti, Jan Małuszyński, Massimo Marchiori, Axel Polleres, and Sebastian Schaffert, editors, *Reasoning Web, Fourth International Summer School 2008*, number 5224 in Lecture Notes in Computer Science, pages 104–124. Springer, 2008.

[34] Norbert E. Fuchs and Rolf Schwitter. Specifying logic programs in controlled natural language. Technical report, 1995.

[35] Norbert E. Fuchs and Rolf Schwitter. Attempto Controlled English (ACE). *CoRR*, cmp-lg/9603003, 1996. informal publication.

[36] Ruth Fuchss, Alexander Koller, Joachim Niehren, and Stefan Thater. Minimal recursion semantics as dominance constraints: Translation, evaluation, and analysis. In *42th Meeting of the Association for Computational Linguistics*, pages 247–254, July 2004.

[37] Gerald Gazdar, Ewan Klein, Geoffrey K. Pullum, and Ivan A. Sag. *Generalized Phrase Structure Grammar.* Basil Blackwell, Oxford; Harvard University Press, Cambridge, Massachusetts, 1985.

[38] Gerald Gazdar and Geoffrey K. Pullum. *Generalized Phrase Structure Grammars : A Theoretical Synopsis.* Indiana University Linguistics Club, Bloomington, 1982.

[39] Robert B. Grady. *Practical software metrics for project management and process improvement.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[40] Carl A. Gunter. *Semantics of Programming Languages, Structures and Techniques.* The MIT Press, 1992.

[41] Robert F. Hadley. A natural language query system for a prolog database. Master's thesis, Simon Fraser University, 1983.

[42] Fritz Henglein. Syntactic Properties of Polymorphic Subtyping. TOPPS Technical Report (D-report series) D-293, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark, May 1996.

[43] My Hoang and John C. Mitchell. Lower bounds on type inference with subtypes. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 176–185, New York, NY, USA, 1995. ACM.

[44] International Institute of Business Analysis. Guide to the Business Analysis Body of Knowledge, draft material for review and feedback, release 1.6 draft. Available online at `http://www.theiiba.org/Content/NavigationMenu/Learning/BodyofKnowledge/Version16/BOKV1_6.pdf`, 2006.

[45] David Johanson and Paul Postal. *Arc Pair Grammar.* 1980.

[46] Hans Kamp and Uwe Reyle. *From Discourse to Logic.* Kluwer Academic Publishers, Dordrecht, 1993.

[47] Jerrold J. Katz and Jerry A. Fodor. The Structure of a Semantic Theory. *Language*, 39:170–210, 1963.

[48] Jerrold J. Katz and Paul M. Postal. *An Integrated Theory of Linguistic Descriptions.* The MIT Press, Cambridge, Massachusetts, 1964.

[49] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition).* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[50] Mark H. Linehan. Sbvr use cases. In *RuleML '08: Proceedings of the International Symposium on Rule Representation, Interchange and Reasoning on the Web*, pages 182–196, Berlin, Heidelberg, 2008. Springer-Verlag.

[51] Chris. S. Mellish. Implementing systemic classification by unification. *Comput. Linguist.*, 14(1):40–51, 1988.

[52] Chris. S. Mellish. Graph-encodable description spaces. Technical Report ESPRIT Basic Research Action DYANA Deliverable R3.2.B, DIKU, University of Copenhagen, University of Edinburgh, Scotland, 1991.

[53] Chris S. Mellish. Term-encodable description spaces. In D. R. Brough, editor, *Logic Programming - New Frontiers*, pages 189–207. Intellect, Oxford, 1992.

[54] John C. Mitchell. Coercion and type inference. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 175–185, New York, NY, USA, 1984. ACM.

[55] Richard Montague. The Proper Treatment of Quantification in Ordinary English. In Richmond Thomason, editor, *Formal Philosophy: Selected Papers of Richard Montague*, pages 247–270. Yale University Press, New Haven, CT, 1973.

[56] Hanspeter Mössenböck. Twin – A Design Pattern for Modeling Multiple Inheritance. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *Ershov Memorial Conference*, volume 1755 of *Lecture Notes in Computer Science*, pages 358–369. Springer, 1999.

[57] Joachim Niehren and Stefan Thater. Bridging the gap between underspecification formalisms: Minimal recursion semantics as dominance constraints. In *41st Meeting of the Association of Computational Linguistics*, pages 367–374, July 2003.

[58] Johan Nordlander. Polymorphic subtyping in O'Haskell. In *APPSEM Workshop on Subtyping and Dependent Types in Programming*, 2000.

[59] Object Management Group. Semantics of Business Vocabulary and Business Rules (SBVR), v1.0, 2008.

[60] Carl Pollard and Ivan A. Sag. *Information-Based Syntax and Semantics: Vol. 1: Fundamentals.* CSLI Lecture Notes no. 13. Center for the Study of Language and Information (distributed by the University of Chicago Press), Stanford, 1987.

[61] Carl Pollard and Ivan A. Sag. *Head-driven phrase structure grammar.* University of Chicago Press, Chicago & London, 1994.

[62] François Pottier. Simplifying subtyping constraints: a theory. *Information & Computation*, 170(2):153–183, November 2001.

[63] James Pustejovsky. *The Generative Lexicon.* MIT Press, 1998.

[64] Ravenflow Inc. RAVEN-Professional 5.1 User Guide. Available online at `http://www.Ravenflow.com`, 2009.

[65] Ravenflow Inc. Requirements-Writing Guide. Available online at `http://www.Ravenflow.com`, 2009.

[66] John R. Ross. Auxiliaries as main verbs. In W. Todd, editor, *Studies in Philosophical Linguistics 1.* Great Expectations Press, Evanston, Ill, 1969.

[67] Ivan A. Sag, Thomas Wasow, and Emily Bender. *Syntactic Theory: A Formal Introduction.* Center for the Study of Language and Information, Stanford, 2nd edition, 2003. ISBN: 1-57586-399-5.

[68] Rolf Shwitter, Anna Ljungberg, and David Hood. ECOLE: A Look-ahead Editor for a Controlled Language. In *EAMT-CLAW03, Controlled Translation, Joint Conference combining the 8th International Workshop of the European Association for Machine*

Translation and the 4th Controlled Language Application Workshop, May 15–17,, pages 141–150, Ireland, May 2003. Dublin City University.

[69] Ghan Bir Singh. Single versus multiple inheritance in Object Oriented Programming. *SIGPLAN OOPS Mess.*, 6(1):30–39, 1995.

[70] Bjarne Stroustrup. *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[71] Bill Venners. Multiple Inheritance and Interfaces, A Conversation with Scott Meyers, Part I. Available online at `http://www.artima.com/intv/abcs.html`, 2002.

[72] Gerd Wagner, Sergey Lukuchev, Norbert E. Fuchs, and Silvie Spreeuwenberg. First-Version Controlled English Rule Language. Available online at `http://rewerse.net/deliverables/m12/i1-d2.pdf`, 2005.

[73] Uriel Weinreich. *Explorations in Semantic Theory*. Mouton, The Hague, Paris, 1972.

[74] Jan Wielemaker. SWI-Prolog 5.6.60 Reference Manual. Available online at `http://gollem.science.uva.nl/SWI-Prolog/Manual/`, 2008.

[75] Terry Winograd. *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language*. PhD thesis, Massachusetts Institute of Technology, 1971.

[76] Richard H. Wojcik, Philip Harrison, and John Bremer. Using bracketed parses to evaluate a grammar checking application. In *Proceedings of the 31st annual meeting on Association for Computational Linguistics*, pages 38–45, Morristown, NJ, USA, 1993. Association for Computational Linguistics.

Index