Scope Underspecification and Processing

Alexander Koller, Dept. of Computational Linguistics, Saarbrücken
Joachim Niehren, Programming Systems Lab, Saarbrücken.

September 15, 1999

Overview

This reader contains material for the ESSLLI '99 course, "Scope Underspecification and Processing". It is intended as a summary of the most important points of the course and as giving pointers to material for further reading. The reader and course are aimed at a pretty broad audience; we have tried to only presuppose a very general idea of natural language processing and of first-order logic.

Underspecification is a general approach to dealing with ambiguity. In the course, we'll be particularly concerned with *scope underspecification*, which deals with scope ambiguity, a structural ambiguity of the semantics of a sentence. As scope underspecification is at least partially motivated by computational issues, we will pay particular attention to processing aspects. We're going to show how *dominance constraints* can be used for scope underspecification and how they can be processed efficiently by using *concurrent constraint programming* technology.

The reader contains material on scope underspecification (Lectures 1 and 2), concurrent constraint programming (Lectures 3 and 4), and the usage of concurrent constraint technology for processing with scope underspecification (Lecture 5).

In Lecture 1, we give a general introduction to the subjects of the course. Underspecification is a general approach to coping with ambiguity; the basic idea is to represent all readings of an ambiguous sentence compactly and to delay the enumeration of the readings for as long as possible. We explain these notions and then go into more detail about scope ambiguity. Lecture 1 is concluded with an overview of the rest of the course.

In Lecture 2, we present some formalisms for scope underspecification. Because we don't want to presuppose much prior knowledge, this chapter also contains an introduction to generalized quantifiers and (a *very* brief one) about type theory. We define dominance constraints, which can be used to describe trees and (encoded) lambda terms, and apply them to scope underspecification. Finally, we give an overview over some other scope underspecification formalisms.

In Lecture 3, we move from representing to processing meaning. We introduce concurrent constraint programming (CCP) in Oz, a rather new programming paradigm and technology mainly used for solving combinatorial problems such as scheduling and optimization. While having been developed in a totally different field, the basic ideas of CCP can be seen as closely related to those of underspecification.

In Lecture 4, we discuss programming features of Oz needed for Lecture 5.

In Lecture 5, finally, we apply concurrent constraint programming in Oz to processing with scope underspecification. We show how to solve dominance constraints based on constraint programming with finite sets. We can thereby enumerate the readings of a scope ambiguity efficiently.

The course in ESSLLI '99 is mainly based on this reader. If time permits, additional material may be presented: a demonstration of the CHORUS-system (Bodirsky et al. 1999) written in Oz and a discussion of CLLS (Egg et al. 1998). CLLS is a language of tree descriptions based on dominance constraints which features an underspecified analysis of the interaction of scope ambiguities, ellipses, and anaphora.

For further reading on concurrent constraint programming in Oz for natural language processing, we refer to a script of a lecture on the topic (Duchier et al. 1999) which was given from October 1998 to April 1999 at the Universität des Saarlandes.

An HTML version of this reader is available on the World Wide Web at `http://www.ps.uni-sb.de/Papers/abstracts/ESSLLI:99.html`. If you install the Mozart programming system `http://www.mozart-oz.org` at your site (which is free and pretty easy), you can directly execute the Oz example programs in the later chapters of the reader.

Comments are very welcome. We hope that you enjoy the course:

Alexander Koller and Joachim Niehren

(`http://www.coli.uni-sb.de/~koller` and
`http://www.ps.uni-sb.de/~niehren`)

Acknowledgments. We would like to thank all members of the CHORUS, NEP, NEGRA, and LISA project in the Collaborative Research Center (Sonderforschungsbereich) 378 at Universität des Saarlandes, who have contributed to the work reflected by this reader.

Contents

Lecture 1

Introduction

In this chapter, we give a general introduction to the subject matter of the course. First, we discuss ambiguities in general, with a specific focus on scope ambiguities. Then we introduce the notion of underspecification and describe informally how to represent scope ambiguities in an underspecified way. Next, we discuss more global aspects of underspecification, such as the general perspective of language processing from an underspecified point of view. Finally, we give a brief overview of the rest of the reader.

1.1 Ambiguities

1.1.1 Ambiguities

Sooner or later, everyone who is concerned with computational linguistics comes across the fact that ambiguities occur at all levels of linguistic analysis. The following is a (not at all exhaustive) list of possible sources of ambiguity.

(1.1)
a. Lexicon:
Mary went to the bank.

b. Syntactic attachment:
John watched the man with a telescope.

c. Coordination:
Birds eat small worms and frogs.

d. Quantifier scope:
Every man loves a woman.

e. Interaction of anaphora and ellipsis:
John likes his mother. Peter does, too.

f. Discourse:
I try to read a novel if I feel bored or I am unhappy.

The sentence in Example (a) is ambiguous in the meaning of the word *bank*; it can either mean a riverbank or a financial institution. In the syntactic analysis of Example (b), there are two different valid options where the PP *with a telescope* can be attached: it can modify either *the man*, who in this reading is identified as the man who carries a telescope, or it can modify *watched the man*, in which case it is a tool to watch the man. In Example (c), it could be only small frogs that birds eat, or it could be any kind of frogs; the ambiguity is in choosing what the conjunction coordinates. Example (d) is ambiguous between expressing that there is one woman who is loved by all man, or that for each man, there is a woman he loves, but not everyone has to love the same one. (We will explain the term "quantifier scope" in a minute.) In Example (e), it is ambiguous who it is that Peter likes; it can be either his own mother or John's. Finally, the discourse in (f) has two different readings: Either the speaker tries to read a novel under two different conditions, or she is unhappy if she does not read a novel.

From a computational point of view, ambiguities are an extremely challenging aspect of language processing. The problem is that many sentences have more than one ambiguity, and that the numbers of readings of the various ambiguities multiply if the ambiguities can be resolved independently. So a sentence containing five two-way ambiguities can have up to 32 readings. An additional inconvenience is that ambiguities can interact; for example, the sentence

(1.2) *John watched a man with his telescope. Bill did, too.*

contains three ambiguities: a PP attachment ambiguity of *with his telescope*, an ambiguity of anaphoric reference (does *his* refer to John or to the man?), and a strict/sloppy ambiguity. The sentence doesn't have $8 = 2^3$ readings, however, only six. On the one hand, the ellipsis enforces that the PP attachment from the first sentence must be taken over in the second sentence. On the other hand, we create a "copy" of the anaphor in the first sentence when we understand the second sentence; if the anaphor referred to John in the first sentence, its copy can refer either to John or to Bill, and if the anaphor referred to the man, its copy must refer to the man, as well.

There are simpler examples of ambiguity interaction, which we will look into later. For now, the really important points are that ambiguities are complex, and the total number of readings can explode exponentially with growing length of the sentence.

1.1.2 Scope ambiguities

The type of ambiguity that will be our primary concern in this text are *scope ambiguities*, as in (1.1d) above. They are typically treated on the level of semantics (although there are theories that consider them on the level of syntax or in the syntax/semantics interface, as we shall see in Lecture 2); that is, we assume that the difference between the readings is not a syntactic one, but purely a difference in meaning. Unlike e.g. lexical ambiguities, however, they are ambiguities of the *structure* of the semantic representation.

Let us assume for the time being that our semantic representation language (what we shall later call the *object language*) is ordinary first-order predicate logic. Then the two readings of (1.1d) can be written as

(1.3) $\forall x.(\mathsf{man}(x) \rightarrow \exists y.(\mathsf{woman}(y) \wedge \mathsf{love}(x, y)))$

(1.4) $\exists y.(\mathsf{woman}(y) \wedge \forall x.(\mathsf{man}(x) \rightarrow \mathsf{love}(x, y)))$

Upon closer inspection, it becomes apparent that both formulae are composed of the same "fragments", $\forall x.(\mathsf{man}(x) \rightarrow \cdot)$, $\exists y.(\mathsf{woman}(y) \wedge \cdot)$, and $\mathsf{love}(x, y)$. The difference is in the way these fragments are put together: In one reading, the fragment containing the existential quantifier gets scope over the fragment containing the universal quantifier; in the other one, this scoping relation is reversed. So the ambiguity is in which of the two quantifiers is in the scope of the other one – hence the name.

The problem carries over to the standard linguistic analysis of NPs as so-called *generalized quantifiers*, as in *Montague Grammar* (Montague 1974). A generalized quantifier is a term of higher-order logic representing the meaning of, say, *every man*, *most people*, or *Peter*. (Generalized quantifiers aren't really generalizations of quantifiers in logic like $\exists x$, but it's the standard name in formal semantics.) The scope ambiguity above is reflected by the different orders in which the two generalized quantifiers that are used to compute the meaning of the sentence (for the two NPs) are applied to the *nuclear scope* $\mathsf{love}(x, y)$. We'll come back to generalized quantifiers in more detail in the second lecture. By abuse of the word, we shall frequently just say "quantifier" to mean "generalized quantifier"; i.e., we shall use the word in its corrupted linguistic sense instead of the logical one.

More generally, not only quantifiers can participate in scope ambiguities, but also other scope-bearing objects such as negations and some verbs. For instance, the sentence (1.5) has two readings that are represented by the formulae (1.6) and (1.7).

(1.5) *Every boy does not go to the movies.*

(1.6) $\forall x.(\mathsf{boy}(x) \rightarrow \neg\mathsf{gtm}(x))$

(1.7) $\neg\forall x.(\mathsf{boy}(x) \rightarrow \mathsf{gtm}(x))$

Here, the fragments are $\forall x.(\mathsf{boy}(x) \rightarrow \cdot)$, $\neg(\cdot)$, and $\mathsf{gtm}(x)$. We'll primarily concentrate on ambiguities of quantifier scope here because all the basic ideas can be shown that way without having to worry about more than one type of scope-bearing objects.

To enumerate the readings of a sentence containing a scope ambiguity, one has to order the scope-bearing objects it contains. If there are n such objects in a sentence and they can be arranged freely, this means that the sentence has $n!$ readings from scope ambiguities alone – an exponential growth in the length of the sentence.

In Montague Grammar, enumeration of all readings of a scope ambiguous sentence was done on the level of syntax, where a special syntactic composition rule called "Quantifying In" was created for exactly that purpose. Realizing that there seem not to be any independent motivations for considering quantifier scope on the level of syntax, Cooper (1983) moved its treatment into the syntax/semantics interface by equipping a syntax tree with the so-called *Cooper Storage*, in which meanings of generalized quantifiers could be passed up the syntax tree and "discharged" whenever convenient. Semantic construction thus became a nondeterministic process. One problem of Cooper Storage was overgeneration: It would sometimes produce formulae with unbound variables. This deficiency was later repaired (Keller 1988); another algorithm for enumerating quantifier scope is (Hobbs and Shieber 1987). Most recently, this kind of analysis of scope ambiguities has received an interesting twist by employing linear logic in the syntax/semantics interface (see e.g. Dalrymple et al. 1997).

As a final aside, one very interesting interaction which scope ambiguities take part in is with ellipses in so-called *Hirschbühler sentences* (Hirschbühler 1982). Consider the following example:

(1.8) *Every man loves a woman. Several gorillas do, too.*

In processing the ellipsis, the second ("target") sentence is expanded to *Several gorillas love a woman.* This means that both the second and the first ("source") sentence contain a scope ambiguity, and if they could be resolved independently, the pair of sentences would have four different readings. But the ellipsis enforces a parallelism of the scopes of the quantifiers. So if *every man* has wide scope in the first sentence, *several gorillas* must have wide scope in the second sentence as well, and vice versa; the pair of sentences only has two readings. This interaction is accounted for, e.g., in the CLLS analysis proposed in (Egg et al. 1998).

1.2 Underspecification

1.2.1 Underspecification

More recently, however, there has been increasing interest in not enumerating the readings of a scope ambiguous sentence at all, but in describing them with one compact representation and then working with this representation instead of all the readings. This approach is called *underspecification*.

There are both computational and cognitive justifications for underspecification. Consider the following sentence, which is taken from (Hobbs 1983).

(1.9) *A politician can fool most voters on most issues most of the time, but no politician can fool all voters on every single issue all of the time.*

Each of the two sentences in this example contains four quantifiers, which means that each sentence admits $24 = 4!$ different orderings of the quantifiers. The sentences can be disambiguated independently; so together, they

$$\begin{tikzcd} \text{Syntax} \arrow[r] & \begin{array}{c}\text{Underspecified}\\ \text{sem. representation}\end{array} \arrow[r, shift left] \arrow[r] \arrow[r, shift right] & \text{Readings} \\ & \textit{metalevel} & \textit{object level} \end{tikzcd}$$

Figure 1.1: Underspecified semantics.

have $576 = 4! \cdot 4!$ readings. Some of these readings may mean the same, but they will still be distinguished in a traditional analysis of the sentence. On the other hand, you probably couldn't say which of these orderings you selected when you understood the sentence. This means that humans don't seem to enumerate readings in understanding an ambiguous sentence.

On the other hand, you probably would be able to draw conclusions from the sentence – for example, that democracies work reasonably well in controlling governments, which is what the original quote was intended to mean. A simpler example is the following inference.

(1.10) *Every man loves a woman.*
Peter is a man.
____
Peter loves a woman.

This inference is correct regardless of the exact meaning of the first sentence. From a computational point of view, we know that inference is an expensive operation; we wouldn't want to make things worse by having to execute it on each of an exponential number of readings in turn. If we had a calculus of *direct deduction* that would let us draw inferences as in the example without disambiguating the premises, we could work with just one underspecified representation for each sentence, would have to do the work only once, and might be much more efficient.

The fundamental idea of most modern approaches to underspecification (no matter on which level of linguistic description) is to add an additional layer of linguistic representation that *describes* the objects of the intended level. For instance, in underspecified semantics, we introduce a level that is between the traditional syntactic and semantic levels. The objects of this new *meta*level are (not necessarily complete) descriptions of formulae on the traditional semantic level (the *object level*) and can be derived from a traditional syntactic analysis (Fig. 1.1). So if we traditionally derived multiple semantic representations from one syntactic representation, the underspecified analysis derives *one* underspecified semantic representation from the syntax, and then it can get back all the object-level readings if necessary by enumerating them from the underspecified representation. (But this step is delayed for as long as possible.)

The information on the metalevel describes the range of possible readings; so you could say it's *disjunctive* information about the meaning of the sentence

– something like "The sentence means R_1, or it means R_2." This disjunctive information doesn't have to be represented syntactically as a disjunction; in fact, that's something we want to avoid because a disjunction is not a very compact representation.

An even more important distinction, however, is between disjunctive information on the metalevel and on the object level. A naive attempt at "optimizing" the representation might be to eliminate the metalevel and represent ambiguity as object-level disjunction (e.g. of predicate logic). Unfortunately, this can lead to unwanted interactions between the new disjunctions and the actual semantic representations, as the following example (involving a lexical ambiguity) shows.

(1.11) *Mary goes to the bank.*

(1.12) *Mary does not go to the bank.*

Generally, we'd like to assign meaning to these sentences systematically; if a sentence means φ, we want the corresponding negated sentence to mean $\neg\varphi$. The naive, object-level disjunctive analysis of the sentence (1.11) would be something like

$$go(m, b_1) \vee go(m, b_2),$$

where b_1 and b_2 stand for the two different meanings of the word *bank*. Then because of the negation rule, we would assign sentence (1.12) the meaning

$$\neg(go(m, b_1) \vee go(m, b_2)).$$

However, this is not the same as the disjunction of the *real* meanings of the sentence, which would be

$$\neg go(m, b_1) \vee \neg go(m, b_2).$$

So if we want a closed representation of the meaning of an ambiguous sentence, we *need* the metalevel because *ambiguity is disjunctive information on the metalevel.*

1.2.2 Scope Underspecification: The General Idea

Now let's take a look at how to apply underspecification to scope ambiguities.

Most recent approaches to scope underspecification (e.g. Underspecified DRT (Reyle 1993), Hole Semantics (Bos 1996), and CLLS/dominance constraints (Egg et al. 1998)) describe the semantics of a sentence by first saying what material the semantics contains and then imposing constraints on the way this material can be arranged.

As an example, Fig. 1.2 displays a graphic representation of such a description. It specifies that the semantic representation of a reading of the sentence should contain the three fragments we identified earlier. Furthermore, it contains dotted lines which stand for the "has scope over" relation (also called the

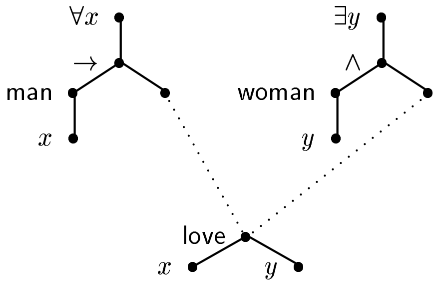


Figure 1.2: An underspecified representation of the meaning of Example 1.1d.

outscopes relation). Here, we see that both upper fragments must have scope over the nuclear scope, but there is no line between the two upper fragments, so their relative scope isn't specified. However, as the described object must be a well-formed formula, we know that one of these fragments must always be within the scope of the other one. This latter condition is enforced by different formal means in different formalisms; for example, Hole Semantics requires a one-to-one "plugging" of fragments into "holes" of formulae, whereas CLLS is really a language of tree descriptions and exploits that trees cannot branch in the bottom-up direction. More on that tomorrow.

A very interesting commonality of the three formalisms mentioned above is that each uses graphs that look very much like the one in Fig. 1.2. Each of them assigns these graphs different formal meaning, but the similarity is not entirely superficial; for example, one can encode both UDRT and Hole Semantics graphs in CLLS. Besides these three, there have been several other influential approaches to scope underspecification. The oldest of them is Quasi Logical Form (QLF, Alshawi and Crouch 1992); some others are Muskens's (1995) underspecified semantics and Minimal Recursion Semantics (MRS, Copestake et al. 1997).

Scope ambiguities seem to lend themselves very well to underspecification. It may not be straightforward to represent a referential ambiguity in a compact way, and it can be argued that a human really decides quickly what's perceived as the antecedent for an anaphor. Generally, underspecification may not be adequate for all classes of ambiguity. But as we have seen above, this doesn't seem to happen for scope ambiguities of any complexity, so underspecification seems to be a very natural way to represent them.

1.2.3 Underspecified View of the World

To conclude the introduction to underspecification in general, we will now present the view of the world of language processing from a radically under-

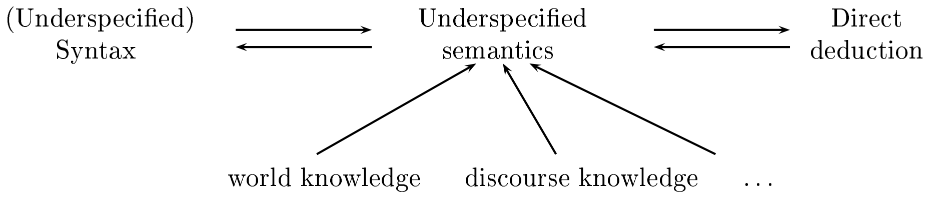


Figure 1.3: Underspecified view of the world.

specified perspective. See Fig. 1.3.

The guiding idea of this view is that language processing has to deal with incomplete information more often than not. Possible sources of incomplete information are not just the obvious missing or misunderstood words in spoken input; other examples are ambiguities (e.g. of scope or anaphoric reference) or ellipses (where entire VPs are missing). Still, the goal is to determine the intended meaning of an utterance as far as necessary to extract the information relevant in the situation.

In this context, syntax and semantics can only contribute to the full determination of the meaning of a sentence; we also have to take other sources of information, such as discourse and world knowledge, into account.

In such an architecture, we give underspecified semantics center stage: It is at this level that we want to collect and process all the information we have about the meaning of a sentence. Processing of a sentence happens as follows. First, a syntax component will parse the input sentence as completely as possible. As the sentence can contain missing words or syntactic ambiguities (e.g. of PP attachment), we can't expect to be able to determine the syntax completely; so we use an underspecified syntax formalism to describe the set of possible syntactic analyses. Now we transfer the partial syntactic information into an underspecified semantic description. From now on, we try to integrate more sources of information to make this description more specific. For example, discourse and world knowledge can be combined with the semantic information by direct deduction; this may give us more information about the semantics, fill in blanks, or exclude readings. Furthermore, reasoning on the semantic level may give us hints about the actual syntactic structure, so information might percolate back to the syntactic level from the semantic level.

Because there are actual ambiguities which can't be resolved further, we can't hope to determine the meaning of a sentence completely by this process. If a rough description of the meaning is good enough for what we want to do with the it (e.g. some inferences), we can stop; otherwise we'll have to disambiguate, i.e. enumerate readings. The key idea is that we want to do as many "cheap" inferences as we can before doing any "expensive" case-distinction steps. This is exactly the same idea that is the foundation of *Constraint Programming*, where these classes of operations have the names *propagation* and *distribution*. More on CP will be said in the last three lectures of this course.

1.3 Overview

In conclusion of this first chapter, let's have a look at the program for the rest of the course.

Lecture 2 has the title "Scope and Trees". We are going to look into scope ambiguities and some formalisms for scope underspecification in some more detail. We will give a brief overview of type theory and the theory of generalized quantifiers; then we'll discuss Cooper Storage (which is not an underspecification formalisms, but helps understand the problems), QLF (a historically important underspecification formalism), and Hole Semantics (the most transparent of an important class of modern underspecification formalisms).

Another approach to take to the problem of scope underspecification which we shall speak about in the second lecture is to consider formulae as trees and then describe these with an appropriate logic. In that respect, we will first review terms of type theory can be seen as trees. Then we will introduce the language of *dominance constraints*, which is a logic whose models are trees; we can take a dominance constraint to describe the set of terms which are encoded by trees that satisfy the constraint. In this way, we can use dominance constraints for scope underspecification, and we will show how this is done. The material in the first two lectures is derived from (Koller 1999).

In Lecture 3, "Concurrent Constraint Programming in Oz", we move towards the "processing" part of the title of the course. We will discuss Concurrent Constraint Programming (CCP), a programming paradigm for solving combinatorial problems such as scheduling or optimization. The general problem it considers is to find assignments of values to variables that satisfy a given set of constraints. Traditionally, this is done by generating such models and then testing if they satisfied the constraint (by brute-force search). The basic idea of CCP is that information about the values of variables can be held in a *constraint store*, and concurrent processes called *propagators* can watch the store and contribute information to it. This is a process of adding simple (i.e. computationally cheap) inferences to the store. Only when propagation can't contribute anything new does one search step take place; then propagation starts again. In this way, a search space can sometimes be reduced dramatically, which is essential for difficult problems.

After this general overview of CCP, Lecture 4, "More on Oz", introduces the basic constructs of the CCP language Oz, as implemented in the Mozart programming system (Smolka 1995; Oz Development Team 1999). Next to common data structures such as numbers, lists, records, etc., Oz also offers first-class procedures and concurrent threads. We also take a look at finite domain and finite set constraints, as well as constructive disjunction.

Finally, we put our new knowledge about CCP to use in Lecture 5, "Solving Dominance Constraints". We show how dominance constraints, which we saw in Lecture 2 to be a powerful formalism for scope underspecification, can be represented, processed, and solved using constraint programming. The implementation encodes a dominance constraint as a constraint on variables over finite sets of integers which is simply a syntactic variation of a valid Oz program.

The material for lectures 3 to 5 in this reader is an adapted version of parts of an introductory course on Oz for computational linguists by Denys Duchier, Claire Gardent, and Joachim Niehren at the University of the Saarland (Duchier et al. 1999). Lecture notes for this course can be found on the World Wide Web at `http://www.ps.uni-sb.de/~niehren/oz-course.html`.

1.4 Summary

- *Ambiguities* occur on all levels of linguistic analysis. They are a challenge to automatic language processing because ambiguities in the same sentence multiply, yielding a number of readings exponential in the number of ambiguities.
- One type of ambiguity is the *scope ambiguity*. Scope ambiguities are ambiguities of the structure of the semantic representation of a sentence. They occur whenever a sentence contains multiple scope-bearing objects which can be ordered independently.
- *Underspecification* is an approach to coping with ambiguity which aims to represent all ambiguities by a single, compact description of all readings. Then any work is done with the description instead of the readings, and their enumeration is delayed for as long as possible. There are both cognitive and computational motivations for doing this.
- *Scope underspecification* is typically done by specifying the semantic material of a sentence and imposing some constraints on how this material can be composed. Many scope underspecification formalisms use diagrams as in Fig. 1.2, but each assigns them different meaning.
- *Constraint programming* is a programming paradigm that was developed in the context of combinatorial problems. Incomplete information about a problem is kept in a so-called constraint store and used to guide the search for complete solutions ("propagate and distribute", as opposed to "generate and test"). CP shares a common underlying intuition with underspecification and can be used for efficient processing of underspecification.

Lecture 2

Scope and Trees

Today, we are going to discuss the problem of scope underspecification in more detail. Our key point of this section is to show how to use *dominance constraints* for scope underspecification. The language of dominance constraints is a logic whose models are trees; the variables of these formulae denote nodes of a tree. Further, we are going to look into other approaches to scope ambiguity – Cooper Storage, QLF, and Hole Semantics –, the latter two of which are underspecification formalisms as well.

2.1 Generalized Quantifiers

As we have tried not to presuppose too much prior knowledge about logic or semantics, we will first give a brief introduction to *type theory* and the theory of *generalized quantifiers* before we delve into the details of this chapter. We will provide as much material on these issues as necessary to understand the rest of the chapter, but it's clear that we can only touch on the surface of these topics, and we recommend a closer look at both. The standard formal semantics textbook in this area is (Gamut 1991); Blackburn and Bos (1999) also have a very readable introduction from the perspective of computational semantics.

2.1.1 The basic problem

In the 60s, semanticists first became interested in a *compositional* analysis of meaning. The idea of compositionality is usually attributed to Frege and is generally taken to mean that "the meaning of an expression is a function of the meanings of its parts". For example, if you want to determine the meaning of a sentence, you'd first determine the meanings of the top NP and VP and then combine these in a uniform way. Compositionality is nice because it encourages a clean semantic construction, where all NPs are basically treated in the same way etc., so you can essentially "read the semantics off the syntax tree".

Unfortunately, if we use first-order predicate logic to represent meaning, we can't easily construct these representations compositionally. One problem is that NPs can end up in very different places throughout a formula:

(2.1) *Peter likes a woman.*

(2.2) $\exists x.\mathsf{woman}(x) \wedge \mathsf{like}(\underline{\mathsf{peter}}, x)$

(2.3) *Every man likes a woman.*

(2.4) $\exists x.\mathsf{woman}(x) \wedge \underline{\forall y.(\mathsf{man}(y) \rightarrow}\mathsf{like}(\underline{y}, x))$

(2.5) $\underline{\forall y.(\mathsf{man}(y) \rightarrow}\exists x.\mathsf{woman}(x) \wedge \mathsf{like}(\underline{y}, x))$

In (2.2), the semantic representation of (2.1), the semantics of the underlined NP has been reduced to a single constant peter. In (2.3), on the other hand, the underlined NP is represented by much more (and very different) "semantic material", which is distributed all over the formula.

At first sight, this makes *Peter* and *every man*, which fill exactly the same role syntactically, so different semantically that it seems impossible to model semantic construction compositionally. To do it anyway, we will treat both NPs as generalized quantifiers. But first, we need to lay some formal groundwork.

2.1.2 Type Theory

First-order predicate logic (FOL) is severely restricted in its expressive power in that it only allows variables (and quantification) denoting individuals, and only constants denoting individuals and relations between individuals. *Type theory* or *higher-order logic* (HOL) is a generalization of FOL that allows both variables and constants denoting any kind of function involving individuals and truth values.

Type theory splits the world into classes by distinguishing objects of different *types*. A type α is a term of the following syntax:

$$\begin{array}{rcl} \alpha & ::= & e \\ & | & t \\ & | & \langle \alpha, \alpha' \rangle \end{array}$$

Every type denotes a distinct subset of the universe. The objects denoted by type e are individuals; they are just the kind of basic entities that a first-order variable can denote. Objects of type t are truth values (true and false). The denotation of a type $\langle \alpha, \beta \rangle$ is the set of functions that take objects of type α as arguments and output objects of type β.

The syntactic objects of HOL are *terms*; every well-formed term can be assigned exactly one type. Terms are defined as follows:

- All constants and variables of type α are terms of type α.
- If M and M' are terms of type t, then $\neg M$ and $M \wedge M'$ are terms of type t.

- If M is a term of type t and x is a variable of arbitrary type α, then $\forall x.M$ is a term of type t.
- If M is a term of type $\langle \alpha, \beta \rangle$ and M' is a term of type α, then $M(M')$ is a term of type β.
- If M is a term of type β and x is a variable of type α, then $\lambda x.M$ is a term of type $\langle \alpha, \beta \rangle$.

The intuition is that the logical connectives work as in FOL (with the other familiar connectives definable in the usual way). An *application* $M(M')$ is really something like application of a function to an argument. An *abstraction* $\lambda x.M$ is intuitively a function that inserts its argument wherever x appears in M and then evaluates the result. Clearly, abstraction is most interesting if M contains free occurrences of x, but that doesn't have to be the case. You can think of the x in an abstraction as a formal argument of a procedure in a programming language. In fact, lambda calculus is the foundation of an entire programming paradigm, so-called *functional programming*, including languages like Lisp or SML. The most fascinating aspect of lambda calculus is that its definition is extremely simple, but (its untyped variety) is still expressive enough to encode a Turing machine.

For example, if f and g are constants of type $\langle e, e \rangle$ and a is a constant of type e, then $f(g(a))$ is a term of type e, and $\lambda x^e.f(g(x))$ is a term of type $\langle e, e \rangle$. On the other hand, $f(g)$ is not a term because f expects an argument of type e, and g is of type $\langle e, e \rangle$. Finally, if P is a constant of type $\langle e, t \rangle$ (that is, the equivalent of a FOL predicate), x is a variable of type e, and F a constant of type t, then $\forall x.P(x) \to F$ is a well-formed term of type t.

HOL terms can be assigned a semantics that's compatible with the standard FOL semantics and the application/abstraction intuition. We won't do so here and refer to the textbooks mentioned above.

As in first-order logic, it's interesting to have a syntactic test for checking whether two expressions have the same denotation. An (incomplete, but essential) way for doing so is testing for so-called$\alpha\beta\eta$-equivalence. The idea here is that if it is possible to rewrite the terms using a given set of *reduction* rules until they are reduced to the same term, they denote the same function.

These rewrite rules look as follows:

$$\begin{array}{lrcll}
(\alpha) & \lambda x.M & \to & \lambda y.M[y/x] & (y \text{ not free in } M) \\
(\beta) & (\lambda x.M)(M') & \to & M[M'/x] & \\
(\eta) & \lambda x.M(x) & \to & M &
\end{array}$$

In simply typed lambda calculus, $\alpha\beta\eta$-equivalence of terms is decidable. But of course, it doesn't say anything about the logical connectives, only about application and abstraction.

2.1.3 Generalized Quantifiers

After this brief excursion, let us return to the problem of compositional analysis of NP meaning. In this section, we're going to present a uniform way to represent the semantics of an NP which can be used in a compositional semantic construction. We won't really talk about semantic construction, though; we're still only laying foundations.

In higher-order logic, the meaning of a verb is a function that takes one or more arguments of type e and then returns a truth value (type t). For example, an intransitive verb is assigned type $\langle e, t\rangle$; a transitive verb is assigned type $\langle e, \langle e, t\rangle\rangle$.

The first idea of combining the meanings of an intransitive verb (type $\langle e, t\rangle$) and of an NP is that the NP denotes an individual (type e); so we can just apply the verb semantics to the NP semantics. This works for a sentence like

(2.6) *Peter sleeps.*

The semantics we get is just $\mathsf{sleep}(\mathsf{peter})$.

But the examples we have seen earlier make it clear that this analysis doesn't carry very far. An additional problem is that most NPs (*every man, two girls*) don't really denote single individuals. A more flexible analysis, then, is to uniformly analyze the semantics of NPs as terms of type $\langle\langle e, t\rangle, t\rangle$ – so-called *generalized quantifiers*. A term of type $\langle\langle e, t\rangle, t\rangle$ describes a set of properties; the intuition behind this *type-raised* analysis of NPs is that the meaning of an NP is the set of all properties that apply to all the individuals described by the NP. Consider, for example,

(2.7) *Every man sleeps.*

The semantics we give to the NP *every man* here is

$$\lambda P.\forall x.\mathsf{man}(x) \to P(x),$$

where P is a variable of type $\langle e, t\rangle$, and x is a variable of type e. This term denotes the set of all properties that every man has. It's a term of type $\langle\langle e, t\rangle, t\rangle$, so we can apply it to sleep; semantically, this means to verify if sleeping is a property that every man has. Incidentally, we can apply β-reduction to simplify the term:

$$\begin{aligned} & (\lambda P.\forall x.\mathsf{man}(x) \to P(x))(\mathsf{sleep}) \\ \to_\beta \quad & \forall x.\mathsf{man}(x) \to \mathsf{sleep}(x). \end{aligned}$$

In other words, we have obtained the same meaning that we had originally intended. But note that the application has reversed; we applied the verb semantics to the NP semantics before, and now we apply the NP semantics to the verb semantics.

We can take the decomposition of the sentence meaning one step further if we assign independent meaning to *determiners*. In the analysis of (2.7), we could analyze *every* as

$$\lambda P \lambda Q \forall x.P(x) \to Q(x)$$

and *man* just as man; then the meaning of *every man* as used above can be obtained just by applying the determiner meaning to the noun meaning. This works for other determiners as well, even for ones like *most*, which can't be represented in a first-order formula. We call the term the determiner meaning is first applied to the *restriction* of the generalized quantifier, and the argument that the entire generalized quantifier is applied to, its *scope*. (In the example, the restriction was man, and the scope was sleep.)

The type-raised analysis of NPs even works for proper names. We just replace the individual by all of its properties. That is, we analyze *Peter* not as peter, but as

$$\lambda P.P(\mathsf{peter}).$$

Again, P is a variable of type $\langle e,t\rangle$. So as before, we can apply the meaning of *Peter* to the meaning of *sleeps* (to stay with our earlier example); β-reduction will then simplify the result to our original analysis:

$$\begin{array}{rl} & (\lambda P.P(\mathsf{peter}))(\mathsf{sleep}) \\ \to_\beta & \mathsf{sleep}(\mathsf{peter}). \end{array}$$

This means that interpreting NPs as sets of properties, which looks strange at first, gives us a uniform analysis of all kinds of NPs. In the light of the examples from the beginning of the section, this is a very surprising result.

2.1.4 Generalized Quantifiers and Transitive Verbs

A problem comes up when we try to analyze sentences with transitive verbs. The problem is that a transitive verb is analyzed as something of type $\langle e,\langle e,t\rangle\rangle$, and we can't use this as an argument for a generalized quantifier. We'll present an analysis using something Blackburn and Bos (1999) call "Montague's trick" because it originates in (Montague 1974) and involves a step that looks surprising at best and like a hack at worst. In Section 2.5, we'll present an analysis that doesn't use Montague's trick overtly, but produces the same results.

The idea behind Montague's trick is to apply the transitive verb meaning to as many variables of type e as necessary to give the result type t, and then to abstract just once over a type e variable each time a quantifier is applied. Saying the same in some more detail, each NP is assigned a unique index i, and when the verb gets an NP argument syntactically, the verb is applied to the variable x_i, of type e (and not, as above, the entire NP to the verb). When all arguments of the verb have been bound, the result will have type t. Then we can apply the NP meanings (of type $\langle\langle e,t\rangle,t\rangle$) to this term; but to give the argument the correct type ($\langle e,t\rangle$), we first have to abstract over a variable. Of course, it has to be the *correct* variable; so if we're trying to apply the NP with index i, we first have to abstract over x_i. We repeat this for all NPs, in any order (which is where scope ambiguities come from). Montague's trick is that the λx_i's "fall from the sky", seemingly unmotivated.

Here's an example to make this clearer. Consider again the earlier example

(2.8) *Every man loves a woman.*

Let's say *every man* gets index 1, and *a woman* gets index 2. Now the first step to constructing the meaning of the sentence is to apply love, the meaning of the verb, to these two variables:

$$\mathsf{love}(x_2)(x_1)$$

This is a term of type t. In the next step, we want to apply one of the quantifiers; let's take *a woman* for now. Before we can apply the quantifier, we first have to abstract over x_2 to give the argument suitable type. This looks as follows:

$$\begin{array}{ll} & (\lambda P.\exists y.\mathsf{woman}(y) \wedge P(y))(\lambda x_2.\mathsf{love}(x_2)(x_1)) \\ \to_\beta & \exists y.\mathsf{woman}(y) \wedge (\lambda x_2.\mathsf{love}(x_2)(x_1))(y) \\ \to_\beta & \exists y.\mathsf{woman}(y) \wedge \mathsf{love}(y)(x_1) \end{array}$$

Again, we have something of type t, and by abstracting over x_2 prior to the application, we have made sure that the variable y introduced by the generalized quantifier ends up in the correct argument position of love. Now we do the same for the other quantifier (which was connected to the variable x_1):

$$\begin{array}{ll} & (\lambda Q.\forall x.\mathsf{man}(x) \to Q(x))(\lambda x_1.\exists y.\mathsf{woman}(y) \wedge \mathsf{love}(y)(x_1)) \\ \to_\beta & \forall x.\mathsf{man}(x) \to (\lambda x_1.\exists y.\mathsf{woman}(y) \wedge \mathsf{love}(y)(x_1))(x) \\ \to_\beta & \forall x.\mathsf{man}(x) \to \exists y.\mathsf{woman}(y) \wedge \mathsf{love}(y)(x) \end{array}$$

The end result is a term of type t, and it's just the first-order formula that we intended to have as one of the two different meanings of the sentence initially.

You'll notice that in constructing this formula, we first applied the quantifier for *a woman* to $\mathsf{love}(x_2)(x_1)$, and then we applied the quantifier for *every man* to the result. We get the other reading of the sentence by reversing the order of application:

$$\begin{array}{ll} & (\lambda Q.\forall x.\mathsf{man}(x) \to Q(x))(\lambda x_1.\mathsf{love}(x_2)(x_1)) \\ \to_\beta & \forall x.\mathsf{man}(x) \to (\lambda x_1.\mathsf{love}(x_2)(x_1))(x) \\ \to_\beta & \forall x.\mathsf{man}(x) \to \mathsf{love}(x_2)(x) \end{array}$$

$$\begin{array}{ll} & (\lambda P.\exists y.\mathsf{woman}(y) \wedge P(y))(\lambda x_2.\forall x.\mathsf{man}(x) \to \mathsf{love}(x_2)(x)) \\ \to_\beta & \exists y.\mathsf{woman}(y) \wedge (\lambda x_2.\forall x.\mathsf{man}(x) \to \mathsf{love}(x_2)(x))(y) \\ \to_\beta & \exists y.\mathsf{woman}(y) \wedge \forall x.\mathsf{man}(x) \to \mathsf{love}(y)(x) \end{array}$$

2.2 Cooper Storage

Historically, Montague's trick first appears in the "Quantifying In" rule of *Montague Grammar*, an approach of seminal importance to quantifier scope and much else. Montague defined a categorial grammar for a fragment of English. Using the Quantifying-In syntax rule, the syntax of a scope ambiguous sentence could be derived in several different ways, each of which gave rise to one of the

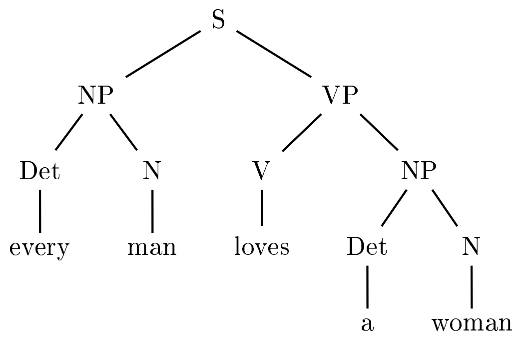


Figure 2.1: Syntax tree for *Every man loves a woman.*

readings because Montague's trick was applied in different orders to the various quantifiers. For an in-depth overview of Montague's work, see also (Partee and Hendriks 1997).

A major conceptual problem with Montague's analysis is that it assumes a syntactic ambiguity for analyzing scope ambiguities which doesn't seem to be justifiable in any other way; this ambiguity is really on the level of semantics. An early attempt to capture scope ambiguity semantically was the *Cooper storage* (Cooper 1975; Cooper 1983). In this section, we are going to briefly explain how it works, and then we will discuss some problems that it has.

Cooper storage takes as its input an (unambiguous) syntactic analysis of a sentence. Its output is a HOL formula that represents the meaning of the sentence. It operates nondeterministically so it can produce multiple readings for a semantically ambiguous sentence. We assume here that the syntax of a sentence is given as a phrase structure tree (but the basic mechanism would work with other grammar formalisms, too). The example we'll work with is Fig. 2.1, the syntax tree of *Every man loves a woman.*

The fundamental idea of Cooper Storage is to associate with each node of the syntax tree two values: the ordinary *semantic content* of the tree below that node, and a *quantifier store* for remembering generalized quantifiers that still have to be applied. Formally, semantic contents are (higher-order) terms; quantifier stores are sets of pairs $\langle i, M\rangle$ of an index i and a term M of type $\langle\langle e,t\rangle,t\rangle$. These values are computed in a bottom-up fashion; computation terminates when all nodes have obtained a content and the quantifier store of the root is empty. You can think of the quantifier store as a record-keeping device for Montague's trick; whenever an NP gets a new index i and a verb is applied to the new variable x_i, the real NP meaning is put on the store under the index i.

The semantic content of a terminal node can be taken from the lexicon; the quantifier store of terminals is always empty. If an internal node has no NP children, its semantic content is the semantic content of its children, applied to each other (in the correct order); its quantifier store is the union of the children's quantifier stores. If it does have NP children, we can either determine its content and store in this way, too, or we can apply the *storage* rule to queue the quantifier for later application on the store. Finally, for S nodes, we have the choice between usual application, NP storage (if there is an NP child), and *retrieval* of quantifiers from the store.

Storage works as follows. Let A be any internal node with an NP child B; let's call the other child C. Let M_B and M_C be the contents and Δ_B and Δ_C the quantifier stores of the nodes B and C. Then pick a new index i. The semantic content of A can be

$$M_C(x_i),$$

and its quantifier store can be

$$\Delta_B \cup \Delta_C \cup \{\langle i, M_B \rangle\}.$$

Conversely, *retrieval* works as follows. Let A be an S node with content M and quantifier store Δ, and let $\langle i, M' \rangle \in \Delta$. Then A can also have the content

$$M'(\lambda x_i.M)$$

and quantifier store

$$\Delta - \{\langle i, M \rangle\}.$$

Let's consider an example for illustration. Fig. 2.2 shows a cooper-storage analysis of *Every man loves a woman* which is complete except for the values associated with the root; we'll discuss those presently. As you can see, all the preterminal nodes of the tree have the obvious semantic contents, and their quantifier stores are empty. Now the contents of the two NP nodes are just the applications of their Det daughters to their N daughters, and their quantifier stores are still empty. In the third step, we compute the meaning of the VP node. This node has one NP daughter, so we apply the storage rule (say, with index 1), which assigns the VP node the content $\mathsf{love}(x_1)$ and puts the NP meaning into the quantifier store with index 1.

Now, because it's an internal node with an NP child, the content and store of the root (S) node can be obtained by application of the storage rule (say, with index 2). The result is

$$\mathsf{love}(x_1)(x_2), \{\langle 1, \lambda Q \exists y.\mathsf{woman}(y) \wedge Q(y) \rangle, \langle 2, \lambda Q \forall x.\mathsf{man}(x) \rightarrow Q(x) \rangle\}.$$

What we need, however, is a value for the root where the quantifier store is empty. So we have to take the quantifiers out of the store by application of the retrieval rule; the order in which we take them out will determine the scope

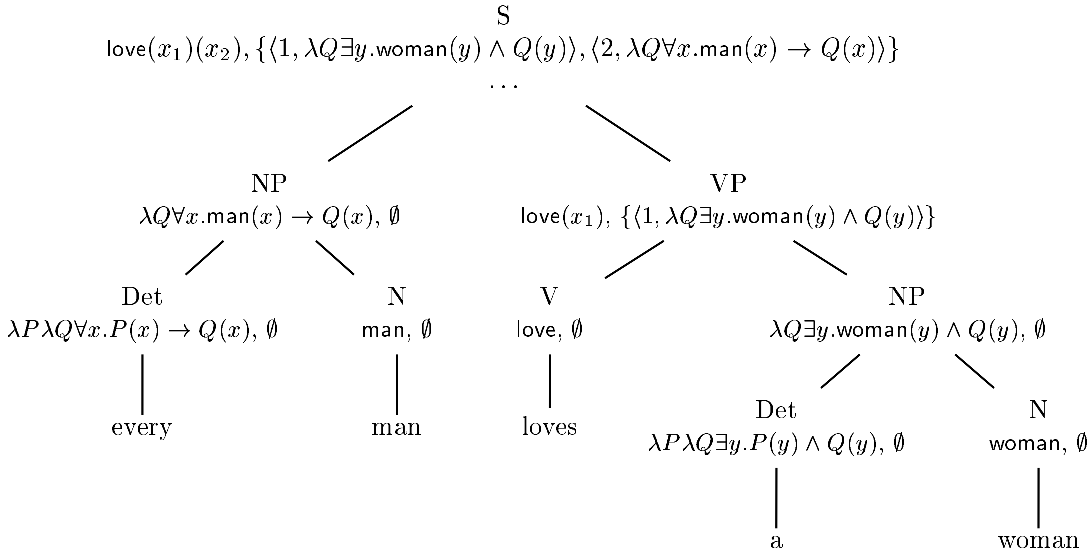


Figure 2.2: Cooper-storage analysis of the running example.

reading. First, let's start with the woman NP (index 1); then we can assign the following values to the root:

$$\begin{array}{ll} & (\lambda Q \exists y.\mathsf{woman}(y) \wedge Q(y))(\lambda x_1.\mathsf{love}(x_1)(x_2)), \{\langle 2, \lambda Q \forall x.\mathsf{man}(x) \to Q(x)\rangle\} \\ \to_\beta & \exists y.\mathsf{woman}(y) \wedge (\lambda x_1.\mathsf{love}(x_1)(x_2))(y), \{\langle 2, \lambda Q \forall x.\mathsf{man}(x) \to Q(x)\rangle\} \\ \to_\beta & \exists y.\mathsf{woman}(y) \wedge \mathsf{love}(y)(x_2), \{\langle 2, \lambda Q \forall x.\mathsf{man}(x) \to Q(x)\rangle\} \end{array}$$

Then we have to take the quantifier with index 2 from the store:

$$\begin{array}{ll} & (\lambda Q \forall x.\mathsf{man}(x) \to Q(x))(\lambda x_2.\exists y.\mathsf{woman}(y) \wedge \mathsf{love}(y)(x_2)), \emptyset \\ \to_\beta & \forall x.\mathsf{man}(x) \to (\lambda x_2.\exists y.\mathsf{woman}(y) \wedge \mathsf{love}(y)(x_2))(x), \emptyset \\ \to_\beta & \forall x.\mathsf{man}(x) \to \exists y.\mathsf{woman}(y) \wedge \mathsf{love}(y)(x), \emptyset \end{array}$$

This completes the derivation of the first reading: We have found a way to derive the root a content and an empty store. The other reading is obtained by first taking the NP with index 2 out of the store and then the one with index 1; we invite you to verify this. Generally, the retrieval operations can be done in any order, which means that Cooper storage will always generate $n!$ readings for a sentence with n quantifiers. So we could say that the semantic representation we have obtained before retrieving anything is a representation of all the readings that doesn't commit to any one of them.

$$(\lambda P \lambda Q \forall x.P(x) \to Q(x))(\mathsf{man})(\lambda x_1.\cdot) \qquad (\lambda P \lambda Q \exists y.P(y) \wedge Q(y))(\mathsf{woman})(\lambda x_2.\cdot)$$

$$\mathsf{love}(x_2)(x_1)$$

Figure 2.3: Underspecified description of the meaning of *Every man loves a woman* as a lambda term.

2.3 Towards Underspecification

Unfortunately, Cooper storage can overgenerate. Consider the following famous example from (Hobbs and Shieber 1987):

(2.9) *Every researcher of a company saw most samples.*

This sentence contains three quantifiers, but it has only five readings. Cooper storage will generate six (= 3!) readings, the sixth of which is

$$\forall x.\mathsf{res}(x) \wedge \mathsf{of}(x_3)(x) \wedge \forall z.\mathsf{comp}(z) \to \mathsf{most}(\mathsf{sample})(\lambda y.\mathsf{saw}(y)(x)).$$

This reading is obviously nonsense; it still contains the variable x_3, which should have been bound by the quantifier with index 3 (*a company*) and is now free. What has happened here is that the necessary scope relations between the quantifiers are more complex than Cooper storage can represent; it's not just any permutation at the sentence level.

One way out of this problem was proposed by Keller (1988) with his "Nested Cooper Storage". The difference to ordinary Cooper storage is that the Storage rule of Nested Cooper Storage doesn't just place the meanings of the NP children into the quantifier store, but the entire pair of meaning and term store associated with the NP; i.e., the store can be nested deeply. Retrieval is adjusted accordingly. Nested Cooper Storage generates exactly the five correct readings in the example. (But now it's important that we really have a choice whether we want to store or apply an NP; this wasn't really necessary for ordinary Cooper storage, and we always stored NPs in the example.) Another algorithm for generating quantifier scope is (Hobbs and Shieber 1987).

A fundamental problem with all of these approaches, however, is that they can only *generate* all readings. As we have seen, a scope ambiguous sentence can have an exponential number of readings, so this can be very expensive. As we have argued in the introduction, it is more reasonable both from a cognitive and from a computational point of view to *describe* the set of readings in a compact way and then to work with this description instead of with all readings for as long as possible.

Cooper Storage does describe readings compactly, but the description is rather implicit and, as we have seen, not very expressive. What we are really after is a description roughly as in Fig. 2.3 which says which fragments a formula contains and allows to specify how they must be arranged – with a notion of "fragment" as in the introduction, where we split the corresponding first-order formulae. We have already argued in the introduction that this is what most modern underspecification formalisms do and will spend the next two sections to give the diagram a formal meaning. We want to speak about the *structure* of a lambda term; we will take this really seriously and speak about *trees*, which make this structure explicit.

2.4 Trees and Dominance Constraints

In this section, we will first define what a *tree* is. Then we will embed trees into first-order model structures, so-called *lambda structures*; in addition to specifying a tree, they define some relations between tree nodes, in particular, a *binding relation*. Finally, we define the syntax and semantics of *dominance constraints*; this is a logic which is interpreted over lambda structures. We shall see in the next section how to apply dominance constraints for scope underspecification.

In one form or another, dominance constraints have been used very widely throughout computational linguistics. The first occurrence we're aware of is in (Marcus et al. 1983), where they were used for incremental parsing. They're important for combining tree-adjoining grammars with unification grammars (Vijay-Shanker 1992), they're at the heart of many current scope underspecification formalisms (as we shall see), and they're used for the analysis of discourse (Gardent and Webber 1998). Their formal properties are rather well-understood, as well; Backofen et al. (1995) axiomatized them in first-order predicate logic, and Koller et al. (1998) investigated the complexity of various logical languages over dominance constraints. Finally, Duchier and Gardent (1999), Koller et al. (1998), and Duchier and Niehren (1999) have investigated how to solve dominance constraints (i.e. enumerate their models) efficiently. We'll say something about this in Lecture 5.

An extension of dominance constraints is the Constraint Language for Lambda Structures (CLLS), which also allows to talk about anaphora and ellipses. We will not go into these aspects of CLLS here, but refer the reader to (Egg et al. 1998), where this language is motivated and defined.

Before we start, a word on notation. We write $\mathbb{N}$ for the set of natural numbers $1, 2, 3, \dots$ and $\mathbb{N}_0$ for $\mathbb{N} \cup \{0\}$. If A is a set, we write A^* for the word monoid over A, that is, all words of length ≥ 0 that are built from symbols in A. As usual, we write the empty word ϵ, and we write concatenation of two words π_1, π_2 by juxtaposition $\pi_1 \pi_2$.

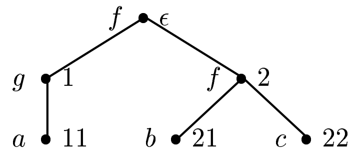


Figure 2.4: Tree corresponding to $f(g(a), f(b, c))$.

2.4.1 Trees

Trees are one of the most successful data structures in computational linguistics and computer science. The standard way of thinking about trees is as directed graphs that have a unique *root* such that there is exactly one path from the root to any other node of the tree. This condition implies some other properties, e.g. that they are acyclic and that there is no node with two incoming edges. Tree nodes are typically decorated with labels (e.g. S or NP in a syntax tree), and sometimes edges are decorated with so-called features.

Here, we will employ a slightly different definition of a tree. First, we assume a signature Σ of node labels. Each of the *labels* (or *constructors*) in this signature is assigned an *arity* by an arity function $\mathsf{ar} : \Sigma \to \mathbb{N}_0$. The only restriction we impose on the signature is that it must contain at least two different constructors, one of which must be nullary; otherwise, there would be no finite trees.

Now we define a *tree domain* Δ to be a nonempty subset of $\mathbb{N}^*$ such that

1. Δ is *prefixed-closed*: Whenever for any $\pi_1, \pi_2 \in \mathbb{N}^*$, $\pi_1\pi_2 \in \Delta$, it must also hold that $\pi_1 \in \Delta$.

2. Δ is closed under the *left-sibling* relation: Whenever $\pi i \in \Delta$ and $i > 1$, it must also hold that $\pi(i-1) \in \Delta$.

Finally, we can define a *constructor tree* to be a pair (Δ, σ) of a tree domain Δ and a *labeling function*

$$\sigma : \Delta \to \Sigma$$

such that for any $\pi \in \Delta$, $\pi i \in \Delta \Leftrightarrow 1 \leq i \leq \mathsf{ar}(\sigma(\pi))$. A *finite* constructor tree is a constructor tree whose domain is finite.

This sounds a bit complicated at first, but it is really very simple. First, consider Fig. 2.4, and let's see what the pair (Δ, σ) that models this tree looks like. We have annotated the nodes in the diagram with words over $\mathbb{N}^*$. These words are the *paths* in the tree domain; so $\Delta = \{\epsilon, 1, 11, 2, 21, 22\}$. Paths indicate a sequence of steps in the tree starting at the root. For example, the path 21 means to start at the root, then move to the second child of the root, and then to move to the first child of that node. This correspondence means that the elements of the tree domain can be identified with the set of nodes of a tree.

The labeling function σ assigns labels to the elements of the tree domain; we have annotated the tree with these values. So for example, $\sigma(\epsilon) = f$, $\sigma(1) = g$, etc.

Finite constructor trees are even more intuitive than that, though: They correspond uniquely to closed terms over the signature, and vice versa. The tree in the diagram was built from the signature $\Sigma = \{f^2, g^1, a^0, b^0, c^0\}$, where we have indicated arities as superscript numbers. A well-formed term of this signature is $f(g(a), f(b, c))$. ($f(a)$, for instance, would not be well-formed because f must have two arguments.) Now compare this term to the tree; you will notice that the two objects have exactly the same structure. It was to enforce this correspondence between trees and well-formed terms that we required every node to have exactly as many children as the arity of its label says. The constructors determine the structure of the tree, hence the name.

From now on, we shall use "tree" as an abbreviation for "finite constructor tree".

2.4.2 Lambda Structures

Given a tree (Δ, σ), we can define first-order model structures with domain Δ which allow us to speak about interesting relations in trees. In addition to relations which can be read off the underlying tree easily, a *lambda structure* will define a λ-*binding function*, which we will use for modeling lambda terms.

Assume a signature Σ as above, with special constructors $\mathsf{var}^0, \mathsf{lam}^1, @^2 \in \Sigma$. A lambda structure L over the tree (Δ, σ) is a triple (Δ, λ, I), where $\lambda : \Delta \rightsquigarrow \Delta$ is a partial function mapping nodes π with $\sigma(\pi) = \mathsf{var}$ to nodes π' with $\sigma(\pi') = \mathsf{lam}$, and I is an interpretation function which assigns relations to a fixed set of predicate symbols. The predicate symbols we are interested in here are the *dominance* predicate $\triangleleft^*$, the *inequality* predicate $\neq$, the *binding* predicate $\lambda(\cdot) = \cdot$, and, for each label $f^n \in \Sigma$, the $(n+1)$-ary *labeling* predicate $:f$. We shall use the same symbols for the predicates and their interpretations; there will be no danger of confusion.

Now we define the relations assigned to the predicate symbols by I. If $f^n \in \Sigma$, the *labeling* relation $\pi{:}f(\pi_1, \ldots, \pi_n)$ holds iff $\sigma(\pi) = f$ and for all $1 \leq i \leq n$, $\pi_i = \pi i$. The *dominance* relation $\pi \triangleleft^* \pi'$ holds iff π is a prefix of π'. The *inequality* relation $\pi \neq \pi'$ holds iff π and π' are different. Finally, the *binding* relation $\lambda(\pi) = \pi'$ holds iff the binding function λ is defined on π and maps it to π'.

The interpretation function is completely determined by the underlying tree and the binding function. For example, the dominance relation induced by the tree in Fig. 2.4 (together with any binding function) contains 14 pairs of nodes, including $(\epsilon, 1)$, $(\epsilon, 21)$, $(2, 2)$, etc.; a labeling relation satisfied by that tree is e.g. $2{:}f(21, 22)$.

We can use lambda structures to model lambda terms by equipping the parse tree of a lambda term with a binding relation between variables and their binders. We obtain such a parse tree if we use the binary constructor @ we introduced above for modeling application, the unary constructor lam

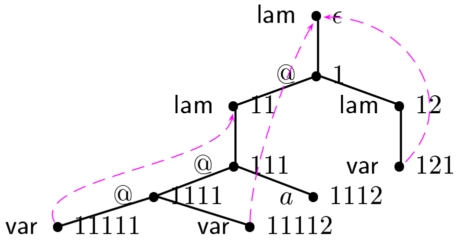


Figure 2.5: Lambda structure for $\lambda x.(\lambda F.(F(x))(a))(\lambda y.x)$.

for signifying abstraction, and the nullary constructor var for variables. As an example, Fig. 2.5 shows the lambda structure corresponding to the term $\lambda x.(\lambda F.(F(x))(a))(\lambda y.x)$. In the diagram, pairs of nodes mapped to each other by the binding function are connected by a dashed arrow. So lambda structures correspond to lambda terms up to α-equivalence (i.e. consistent renaming of variables).

2.4.3 Dominance Constraints

Now it is straightforward to build a syntax for a logic for talking about lambda structures. We fix a set of variables $X, Y, \ldots$ (which are supposed to denote tree nodes) and define a *dominance constraint* φ to be built according to the following abstract syntax:

$$\begin{array}{rcl} \varphi & ::= & X{:}f(X_1, \ldots, X_n) \\ & | & X \triangleleft^* Y \\ & | & \lambda(X) = Y \\ & | & X \neq Y \\ & | & \varphi \wedge \varphi'. \end{array}$$

An atomic dominance constraint φ is satisfied by a lambda structure $L = (\Delta, \lambda, I)$ and a variable assignment α into Δ iff α maps the participating variables into the relation to which I maps the respective predicate symbol. An arbitrary dominance constraint is a conjunction of atomic dominance constraints and satisfied by (L, α) iff (L, α) satisfies every conjunct; we also call L a *solution* of the constraint.

For instance, the dominance constraint

$$X_1{:}\mathsf{lam}(X_2) \wedge X_2{:}@(X_3, X_4) \wedge X_3 \triangleleft^* X_5 \wedge X_5{:}\mathsf{var} \wedge \lambda(X_5) = X_1 \qquad (2.10)$$

is satisfied by the lambda structure in Fig. 2.5 together with the variable assignment $\alpha(X_1) = \epsilon, \alpha(X_2) = 1, \alpha(X_3) = 11, \alpha(X_4) = 12, \alpha(X_5) = 11112$.

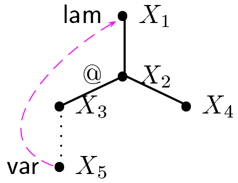


Figure 2.6: Constraint graph for (2.10).

Usually, we are not interested in *all* solutions of a dominance constraint. An atomic dominance constraint $X \lhd^* Y$ can be satisfied with X and Y having an arbitrary distance from each other in the tree, which means that most dominance constraints have an infinite number of solutions. What we're really after is a *constructive solution*. A solution (L, α) of a dominance constraint φ is called constructive iff every node in the domain of L is the α-image of a variable in φ.

The abstract syntax we have just defined is perfect for formal purposes; however, it can easily become unreadable for humans. To this end, we employ *constraint graphs*. A constraint graph is a directed graph with node labels and three kinds of edges: solid edges, dotted edges, and dashed arrows. Nodes of the graph stand for variables in a constraint; node labels together with solid edges stand for labeling constraints, dotted lines stand for dominance constraints, and dashed arrows stand for binding constraints. In addition, a constraint graph represents an inequality constraint between any two variables corresponding to labeled graph nodes.

For example, the constraint (2.10) can be drawn as the constraint graph in Fig. 2.6. (But note that the constraint graph also represents some inequality constraints, i.e. $X_1 \neq X_2$, $X_2 \neq X_5$, etc.)

Constraint graphs are rather similar to the lambda structures satisfying them (compare Fig. 2.6 to Fig. 2.5). In particular, you get a constructive solution by simply arranging the fragments in the constraint graph in a tree-like fashion and then identifying the ends of any remaining dominance edges. Note, however, that constraint graphs are objects of the syntactic level of dominance constraints, whereas trees are objects of the semantic level. The nodes of a constraint graph represent variables of a constraint, which in turn can denote nodes in a tree. So it is important to keep them apart.

2.5 Scope Underspecification Using Dominance Constraints

Now let's return to scope ambiguities and put the dominance constraints we have just defined to use for scope underspecification.

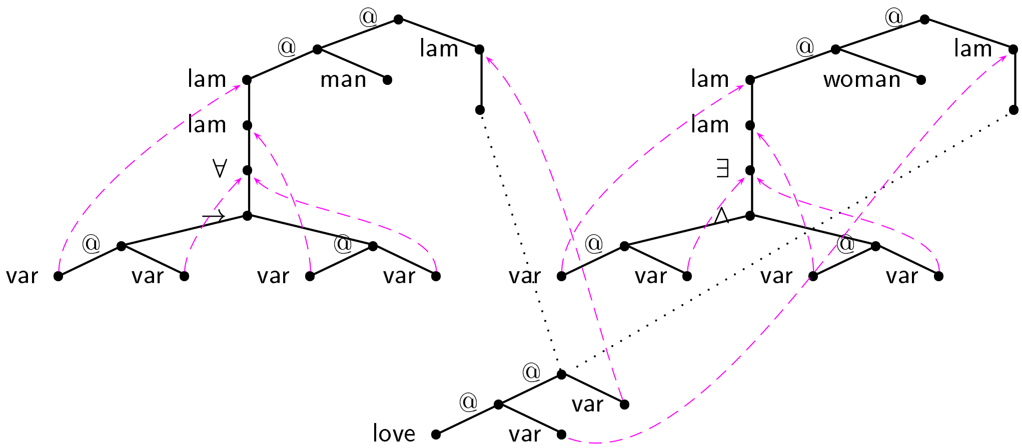
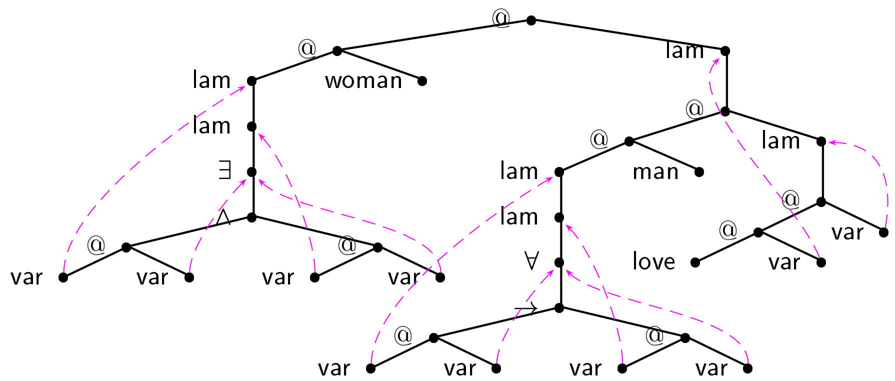


Figure 2.7: Constraint graph for *Every man loves a woman.*

The intended semantic representation language for single readings is higher-order logic. Using our new knowledge of dominance constraints, we can fix a signature containing all the constants we're interested in plus the logical connectives $\rightarrow^2$, $\neg^1$, etc., and try to interpret Fig. 2.3 as a dominance constraint graph describing the lambda structure corresponding to the lambda term we want.

The only remaining problem is the modeling of lambda binding, but we can solve it by introducing binding constraints. The result is the constraint graph in Fig. 2.7; the constraint it represents has exactly two constructive solutions, shown in Fig. 2.8. Converted back to lambda terms, they are just the two readings we wanted. (These are the only diagrams where we'll ever spell out the tree structure of a determiner meaning; in the future, we'll abbreviate them as little triangles labeled with the determiner.)

Why are these the only two constructive solutions? The constraint graph specifies the two generalized quantifiers and the nuclear scope of the sentence, and expresses that the nuclear scope has to be in the scope of both quantifiers. It doesn't say anything about the order of the quantifiers. But beause the tree part of a lambda structure can't branch in the bottom-up direction, one of the two quantifiers must dominate the other one; so there are two structurally different solutions. Because we only want constructive solutions, they could only contain "material" that had been "mentioned" in the constraint.



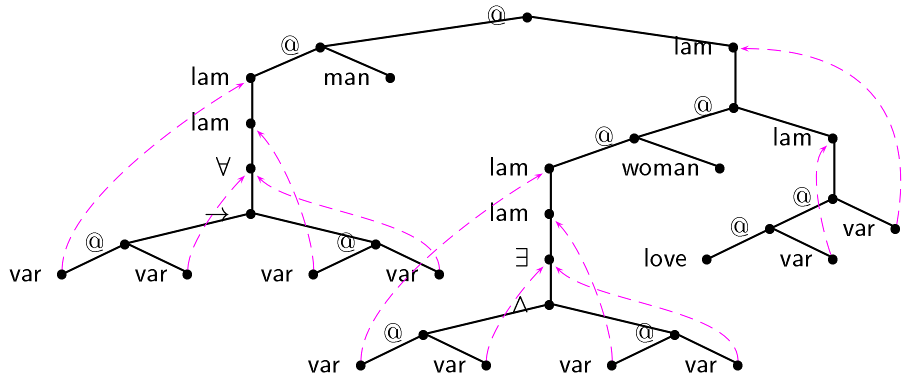


Figure 2.8: Constructive solutions of the constraint in Fig. 2.7.

So we can give a clean formal meaning to the intuitive scope underspecification diagrams we had earlier by using dominance constraints.

It's interesting to observe how this analysis implements "Montague's trick". Here we know from the start what material the semantic representation is going to be built of; there's no abstraction "falling from the sky". There's also no need to remember NPs in a store because the new λ is firmly connected to the rest of the NP meaning. We can do this because we can treat the λ-term fragments that we used informally in Fig. 2.3 in a formally sound way here – as fragments of trees. Finally, we don't have to worry about variable names because we have binding constraints that tell us from the start what the correct binders for variables are.

Now let's see what hepps with (2.11), which was a problem for Cooper storage.

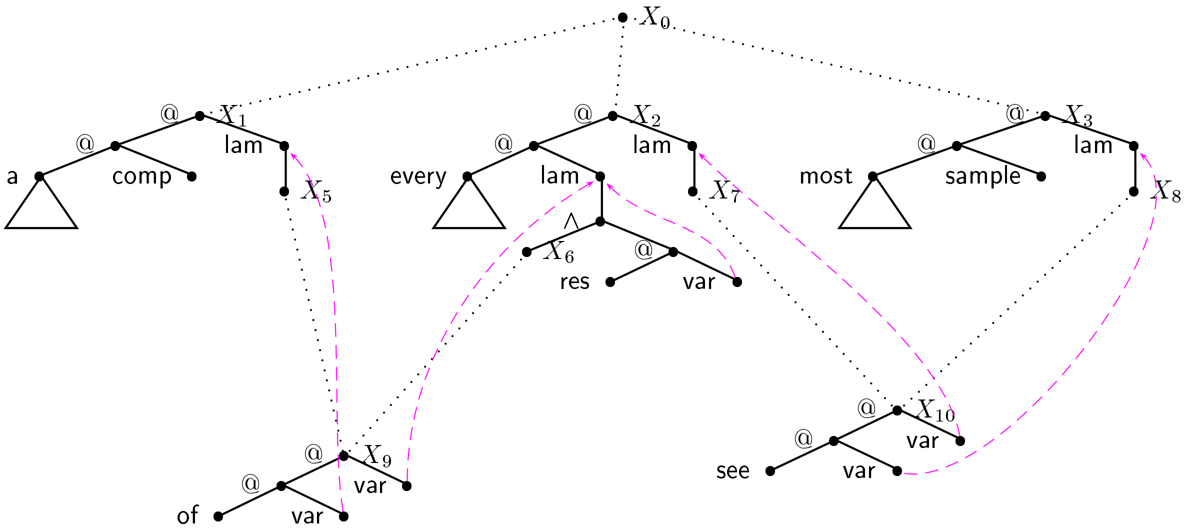


Figure 2.9: Constraint graph for *Every researcher of a company saw most samples.*

(2.11) *Every researcher of a company saw most samples.*

The dominance constraint graph describing this sentence is shown in Fig. 2.9). It has *two* nodes that have two incoming dominance edges (X_9 and X_{10}), corresponding to two nuclear scopes (for the verb and for the preposition). Again, we must choose which of the two dominating nodes should dominate the other in order to disambiguate the constraint. If we choose that X_5 dominates X_6, X_8 can go in three places: either above X_5, or between X_5 and X_2, or below X_7. If we choose that X_6 dominates X_5, X_8 can go in two different places: either above X_2, or below X_7. This makes for a total of five constructive solutions, corresponding to the five readings of the sentence.

So far, we haven't talked about a syntax/semantics interface generating dominance constraints from a syntactic analysis. It is not difficult to build such an interface; see the last section of (Koller et al. 1999).

2.6 Other Approaches to Scope Underspecification

In conclusion of this chapter, we will now give brief and informal introductions to two other formalisms for scope underspecification. From the wide variety of formalisms that we have listed in the first chapter, the two we pick for a closer look are Quasi Logical Form and Hole Semantics. The former is of seminal importance for the field and has a broad coverage of linguistic phenomena. The latter is representative of a family of underspecification formalisms that is probably the most influential at this time. The most popular member of this family is UDRT (Reyle 1993; Schiehlen 1997), but Hole Semantics is much more accessible, and its basic ideas are essentially the same. (Alshawi et al. 1992) and (Bos 1996) are warmly recommended for further reading.

2.6.1 Quasi Logical Form

QLF (Alshawi and Crouch 1992) was the first formalism for semantic underspecification that was implemented and used for real-world applications. It was continually developed over several years to meet the demands of a growing linguistic coverage. The original syntax looks rather intimidating. Therefore, we have adopted a heavily simplified version for our exposition here. For the original, we refer the reader to (Alshawi et al. 1992), a comprehensive summary of QLF and its applications.

The underlying idea of the formalism is to provide an underspecified representation of quantifier raising. In a QLF representing a sentence, the terms representing NPs are arguments of the VPs whose syntactic arguments they are. Each of them is identified by a unique index, and different scope relations can be represented by specifying an order on indices in special scoping lists. In order to be able to represent difficult sentences like (2.11), scoping lists can also occur in nested positions in a term. In an unresolved QLF, these lists are unspecified; they are represented as uninstantiated variables. To ensure that logical formulae can be derived from fully resolved QLFs, there is the constraint that for every index, the term it identifies must appear inside the scoping list that contains the index. Disambiguation means instantiation of the scoping lists.

By way of example, consider the (simplified) QLF representation of our running example, repeated here as (2.12).

(2.12) *Every man loves a woman.*

(2.13) $\texttt{\_s:love(term(+m,}\,\forall,\ \lambda X.\texttt{man}(X)),$
$\qquad\qquad \texttt{term(+w,}\,\exists,\ \lambda Y.\texttt{woman}(Y)))$

In the QLF, we find the two NPs represented as two `terms` that are arguments of their syntactic mother, the `love` VP. Each `term` has a unique index, given as its first argument; for the NP quantifying over men, it is `+m`, for the one quantifying over women, it is `+w`. The type of quantifier (e.g. universal or existential) is stored as the second argument; and the restriction of the quantifier is placed in the third argument.

The `love` formula is prefixed with a scoping list that is, at this point, unspecified and represented by the variable `_s`. Due to the free-variable constraint we mentioned above, any fully resolved QLF that can be derived from (2.13) must instantiate `_s` with a list that contains both `+w` and `+m`. This can be done in either order, yielding the two readings (2.14) and (2.16) below. If you imagine that a scoping list `[+m,+w]` means to first retrieve the *woman* and then the *man* quantifier from a Cooper store, the QLFs correspond to HOL terms which can be β-reduced to (2.15) and (2.17), respectively.

(2.14) $\texttt{[+m,+w]:love(term(+m},\ \forall,\ \lambda X.\texttt{man}(X)),$
$\qquad\qquad\texttt{term(+w},\ \exists,\ \lambda Y.\texttt{woman}(Y)))$

(2.15) $\forall x.\mathsf{man}(x) \to \exists y.(\mathsf{woman}(y) \wedge \mathsf{love}(x,y))$

(2.16) $\texttt{[+w,+m]:love(term(+m},\ \forall,\ \lambda X.\texttt{man}(X)),$
$\qquad\qquad\texttt{term(+w},\ \exists,\ \lambda Y.\texttt{woman}(Y)))$

(2.17) $\exists y.\mathsf{woman}(y) \wedge \forall x.(\mathsf{man}(x) \to \mathsf{love}(x,y))$

The evolutionary, application-oriented development of QLF has the positive effect of leading to a very wide coverage of linguistic phenomena. But the downside of this is that some formal aspects of QLF are patchwork needed to make things work, instead of consequences of an overall vision. One particular inconvenience is that unlike most modern approaches to underspecification, QLF does not provide a clean separation between object and meta level; elements of both are distributed all over an underspecified representation. This makes the representation a bit intransparent; in addition, it makes the task of designing a calculus for direct deduction even more difficult than it inherently is.

2.6.2 Hole Semantics

Hole Semantics was developed by Bos (1996) and is a general framework for creating an underspecified representation language from a non-underspecified object language. Bos himself applies it to predicate logic and DRT; his "DRT unplugged" essentially agrees with UDRT, with which it shares the underlying perspective on scope ambiguities.

Hole Semantics is based on underspecification pictures such as Figure 2.3, which we repeat below as Fig. 2.10 in a slightly adjusted format, but gives them a different technical interpretation than with dominance constraints. Formulae occurring in the nodes of such an underspecified representation (UR) are taken from the object language; but any subformula can be replaced by a so-called *hole* (h_0, h_1, h_2 in the picture). The function of holes is that other formulae can be *plugged* into them to obtain a larger formula. The dotted lines in the graph are drawn from holes to formulae, and they express that the formulae must be subformulae of the formulae into whose holes they will be plugged. To take care of problems that can arise when the same formula occurs more than once in the graph, each fragment is given a unique identity, its *label* (l_1, l_2, l_3 in the picture). The graph can be represented as an upper semilattice specifying a partial order on holes and labels, and disambiguation means to make this order more specific.

The object-language formulae a UR represents can be obtained from so-called *admissible pluggings*. A plugging is a bijection between holes and labels, and it is called admissible if it agrees with the partial order on labels and holes. An admissible plugging P induces a object-language formula by starting at the (unique) top formula of the UR and subsequently replacing holes h by formulae $P(h)$.

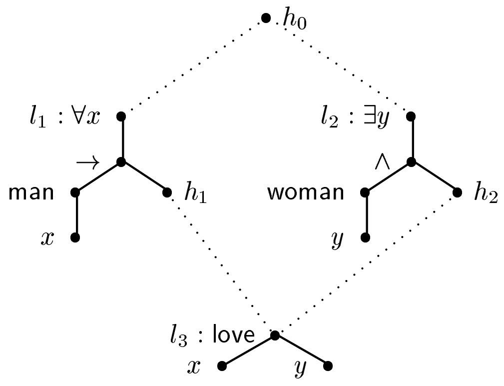


Figure 2.10: A scope ambiguity in Hole Semantics.

To see an example for such a plugging, we have equipped Fig. 2.10 with explicit holes and labels. The example is in "Predicate Logic Unplugged", the instantiation of Hole Semantics to *first-order* logic. The UR presented in this picture has exactly two admissible pluggings. They are shown as (2.18) and (2.20), along with the predicate logic formulae they induce.

(2.18) $\{h_0 = l_1, h_1 = l_2, h_2 = l_3\}$

(2.19) $\forall x.\mathsf{man}(x) \rightarrow \exists y.(\mathsf{woman}(y) \wedge \mathsf{love}(x, y))$

(2.20) $\{h_0 = l_2, h_2 = l_1, h_1 = l_3\}$

(2.21) $\exists y.\mathsf{woman}(y) \wedge \forall x.(\mathsf{man}(x) \rightarrow \mathsf{love}(x, y))$

Hole Semantics and its kin cope easily with sentences like (2.11), using basically the same pictures as the dominance constraint analysis.

2.7 Summary

- *Generalized quantifiers* can be used for a compositional semantic construction (which derives semantic representations from a syntactic analysis). A generalized quantifier is a λ-*term* of *type* $\langle\langle e, t\rangle, t\rangle$; it represents the meaning of an NP in the sentence.

- *Cooper storage* can be used to enumerate the readings of a scope ambiguity without artificially analyzing a sentence as syntactically ambiguous. Cooper storage has overgeneration problems that are a consequence of its lack of expressive power.

- *Dominance constraints* are a logic whose models are *lambda structures*. A dominance constraint can be regarded as a (partial) description of its *constructive solutions*. Lambda structures can be used for modeling λ-terms.

- This means that dominance constraints can be used for partial (= underspecified) descriptions of λ-terms. A human-readable form of dominance constraints, *constraint graphs*, look almost like the intuitive pictures we used on Monday for describing the readings of a scope ambiguity informally.

- *QLF* and *Hole Semantics* are other important formalisms for scope underspecification. QLF is an application-oriented formalism of seminal importance for the field, but lacks the formal elegance and the separation of object and meta language of modern underspecification formalisms. Hole Semantics is representative of a very popular class of formalisms. It allows formulae with *holes* into which other formulae can be *plugged* and represents this with diagrams very similar to dominance constraint graphs.

Lecture 3

Concurrent Constraint Programming in Oz

3.1 Relation to Underspecification

Concurrent constraint programming is a modern technology which can be used to solve complex combinatoric problems efficiently. Typical applications of concurrent constraint programming in industry include scheduling and configuration.

Underspecification and constraint programming can be seen as two sides of the same coin. Underspecification is concerned with ambiguity in natural language which a constraint programmer would consider more generally as disjunctive information in a combinatoric problem. The idea of underspecification is to delay the enumeration of readings of an ambiguous sentence for as long as possible. A more general idea underlies concurrent constraint programming, where combinatoric problems are solved in such a way that case distinctions are delayed for as long as possible.

3.1.1 Towards processing underspecified semantics

What have we done so far? In the first two lectures we have learned about underspecification in semantics of natural language. We have discussed several formalisms in which to represent the meaning of sentence containing scope ambiguities. We have advocated formulas of higher-order logic (HOL) for representing meaning, and partial descriptions of HOL-formulas for representing meaning in an underspecified manner. As partial descriptions of HOL-formulas, we have proposed tree descriptions based on dominance constraints. The idea was to consider a HOL-formula as a tree (the tree of its abstract syntax) and to describe this tree partially.

Of course, when doing computational linguistics it is not sufficient to represent the meaning of a sentence in theory. The goal is to provide algorithms

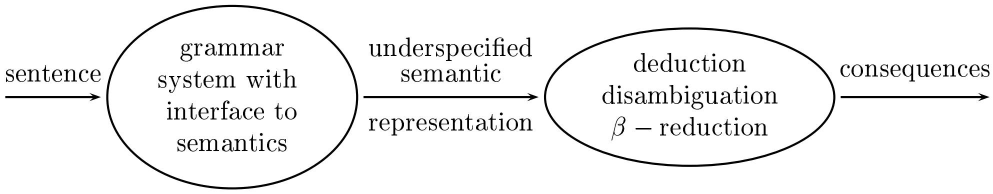


Figure 3.1: Architecture for natural language processing

and implementations thereof that can derive semantic representations and compute its consequences (see Figure 3.1). As we argued before, the semantics of a sentence is best represented in an underspecified manner because of scope ambiguities. So, the question is how to compute underspecified semantic representations from a sentence and how to derive its consequences.

In this lecture, we are mainly concerned with semantics rather than with syntax. Therefore, we assume the existence of some magician who is doing the syntactical work for us. We can pass a sentence to the magician who then returns its syntactic structure. From this is it is not difficult to compute an underspecified semantic representation. We discussed in the previous lectures how syntax and semantics are related in principle.

In practice, the magician will be some grammar system (LFG, HPSG, dependency grammar), i.e. a parser into which a *syntax-semantics interface* is integrated. Compared to the complexity of parsing, a syntax-semantics interface is usually quite easy to design. We will therefore omit the details in this course. Instead, we assume that the grammar system provides us with an underspecified semantic representations in form of a dominance constraint (which describes a HOL-formula that in turn represents the meaning of the input sentence).

3.1.2 Disambiguation is constraint solving

So what remains to be done? We would like to compute the consequences of an underspecified representation. In fact this question is very difficult and can not be answered in this lecture. Suppose that we would have a fully specified meaning representation in first-order logic then we would still need a theorem prover for computing all its consequences. This is not what we are going to do in this course.

Instead, we approach a more basic question which concerns underspecification independently of how it is approached. The problem is that an underspecified representation does not explicitly represent the set of all possible meanings. So the question is how to disambiguate an underspecified representation efficiently, i.e. how to enumerate the set of readings it represents in polynomial time depending of the size of this set. One might argue that disambiguation contradicts the main idea of underspecification which is to delay disambiguation for as long as possible. But earlier or later during natural language processing, one can be forced to disambiguate at least partially. In our approach to underspecification based on tree descriptions, disambiguation amounts to solving dominance constraints.

Hence, our goal is to solve dominance constraints efficiently. The problem of solving dominance constraints is *NP-complete* as shown in (Koller et al. 1998). In other words, solving dominance constraints is a combinatoric problem which is much harder than one might think at first sight: we cannot expect the existence of an algorithm which solves dominance constraints in polynomial time in general. This does not mean however that we cannot built a solver which is efficient for those dominance constraints representing underspecified semantics.

A good approach to solve combinatorial problems is concurrent constraint programming (Saraswat et al. 1991; Smolka 1994; Smolka 1995). We will show how to use concurrent constraint programming with sets in order to solve dominance constraints (Duchier and Gardent 1999; Duchier and Niehren 1999; Koller et al. 1998). As an implementation platform, we will employ the Mozart system of the programming language Oz (Oz Development Team 1999). The rest of this lecture is devoted to concurrent constraint programming technology. In the next two lecture, we will present further features of Oz and then the solver for dominance constraints.

3.2 What is Constraint Programming

Constraint programming is a method for solving combinatoric problems, which comes with a well-developed technology. Combinatoric problems are traditionally formulated as logical formulas that are called constraints. Solving combinatoric problems is inherently difficult because of the disjunctive character of combinatorics.

3.2.1 Applications

Typical applications of constraint programming include optimization problems of industrial relevance such as:

- scheduling,
- time tabling,
- configuration.

Recently, many new challenging applications have been investigated at universities:

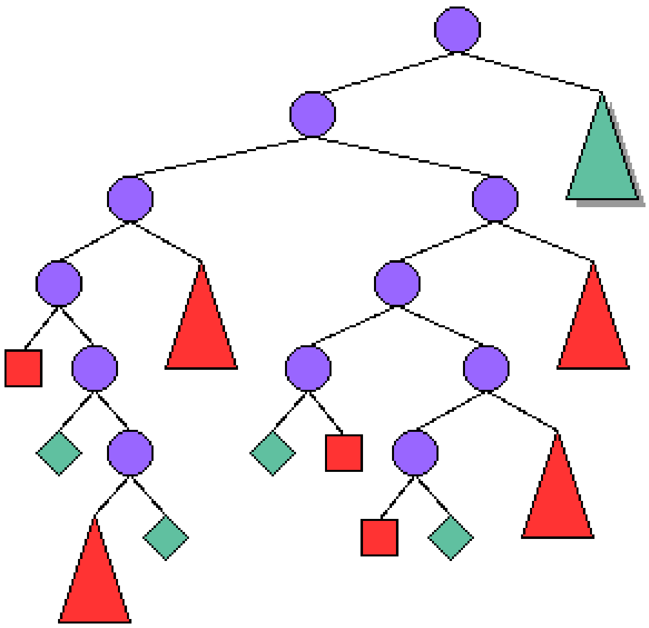
- deduction and reasoning
- knowledge representation

- processing of natural language

3.2.2 The Problem: Combinatoric Explosion

The naive way of solving combinatoric problems can be paraphrased as 'generate and test': In a first step one enumerates all combinations from which one selects all solutions in the second step. In most cases however, 'generate and test' is simply not feasible. This is obvious if the set of combinations is infinite. But even if it is finite then it is usually very large, i.e. exponentially large in size of the problem description. In this case, the generation step runs into a combinatoric explosion (from which it usually returns only several billions of years later).

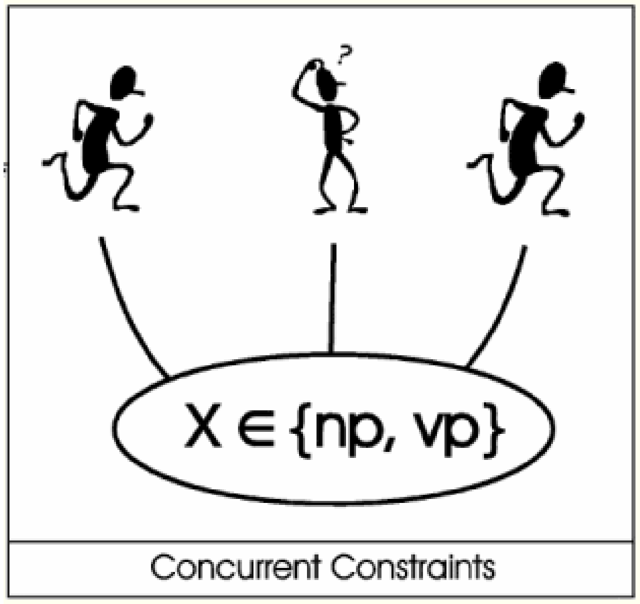
The process of generation spawns a search tree. The inner nodes in this tree correspond to case distinctions performed during generation and the leaves of the tree to the set of all possible combinations. There are two kinds of leaves: solution and failure leaves which can be distinguished by a test procedure. The problem is that a search tree grows exponentially in its depths. For instance, if we have a combinatoric problem with 15 variables each of which has 4 possible values then we obtain a search tree of depth 15 which contains 4ˆ15 = 10.737.410.000.000.000 nodes. The search tree below is rather small but only since its depth is small too.



3.2.3 The Method: Propagate and Distribute

Suppose that you are now convinced that we can not simply generate the full search tree in usual combinatoric problems. So what can we do instead? The idea is to delay case distinctions for as long as possible. Instead we do simple inferences first and hope that we can thereby prune the search tree, i.e. avoid to visit all its nodes. This is the general method of constraint programming which can be paraphrased as 'propagate and distribute'. A propagation step restricts the set of possible solutions by a simple inference. A distribution step executes a case distinction by which the set of possible solutions is restricted further. Of course, propagation and distribution steps have to be iterated. In order to delay case distinctions for as long as possible, a distribution step has to wait until the propagation process is terminated. This is since distribution is considered expensive whereas propagation is not.

In concurrent constraint programming, propagation is organized as a concurrent process. The idea is to store simple information in a common constraint store such that it can be observed by all concurrent propagators. Whenever a propagator can make an inference then is adds its results to the constraint store. Thereby another propagator may become triggered and so on. One can imagine a constraint store with its propagators as follows:



3.2.4 What is Oz and who is Mozart?

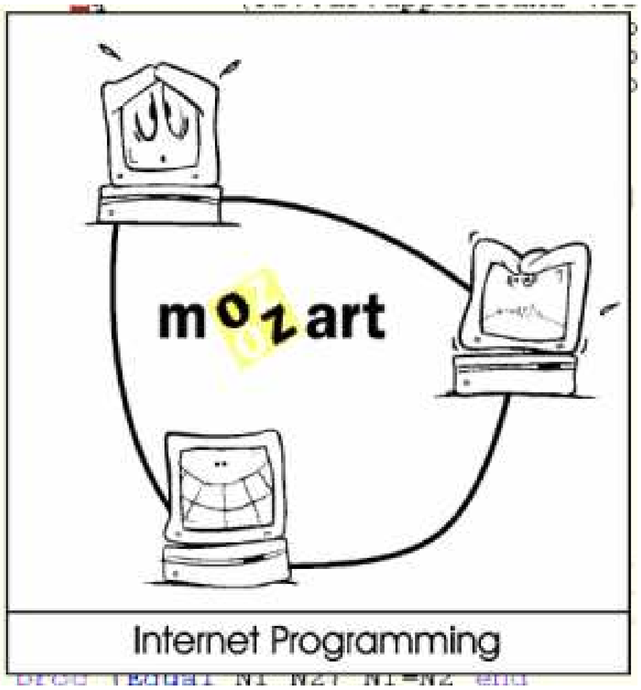
A concurrent constraint programming system provides a set of procedures for defining propagators and all machinery for running propagation and distribution. The programmer simply models his problem by defining sets of propagators and a strategy for distribution. The rest is done by the compiler and emulator of the programming system.

Oz is a concurrent constraint programming system which has been developed by the Programming Systems Lab in Saarbrücken led by Gert Smolka.

The most recent Oz version is Mozart 1.0.1[1] The Mozart system was developed by the Mozart consortium which integrate the programming systems lab in Saarbrücken, the programming systems lab at SICS (Swedish Institute of Computer Science) led by Seif Haridi, and Peter Van Roy's group at the Universite catholique de Louvain. The Mozart system is freely available, extensively documented, and fully operational.

Oz unifies ideas originating from logic programming in Prolog and functional programming in Lisp or SML. Oz provides the most innovative technology compared to other constraint programming languages on the market (ILOG, CHIP). This makes Oz a good foundation for building innovative applications in computational linguistics and artificial intelligence.

Beyond concurrent constraint programming, Mozart supports Internet programming similar to Java. Mozart is also well-suited for building multi agent systems and sophisticated graphical user interfaces.



[1] http://www.mozart-oz.org/

3.3 Solve a Combinatorical Problem in Oz

Our next goal is to build a constraint solver for the following problem which is given by an equation system with variables denoting integers.

$$X, Y, Z \in \{1, \ldots, 7\}$$
$$X + Y = 3 * Z$$
$$X - Y = Z$$

A solution of this problem is an assignment of variables `X, Y, Z` to natural numbers which satisfies the given arithmetic constraints.

3.3.1 Bits of a Constraint Solver

We next show how to solve this problem in Oz. We define the following constraint which can be added directely to the constraint store

```
[X Y Z] ::: 1#7
```

and define the following set of propagator over this constraint store:

```
X + Y =: 3*Z
X - Y =: Z
```

Here we make use of Oz-variables whose syntax is given by words with leading capital letters. The first line states that `X, Y, Z` are so called finite domain variable, i.e. variables for an integer in a finite domain (here, between 1 and 7).

Next, we use a predefined distribution strategy which takes the actual restrictions on `X, Y, Z` into account:

```
{FD.distribute naive [X Y Z]}
```

We represent a solution as a record (called feature tree in computational linguistics):

```
solution(x:2 y:1 z:1)
```

This record is built from integers and Oz-atoms which are words beginning with a lower case letter. The solution record has the label `solution` and three features `x, y, z`.

3.3.2 Observing Propagation

It might be instructive to observe propagation independently from distribution. Propagation relies on the concept of a constraint store which is simply a set of simple constraints on values of variables. New information can be added to the constraint store by propagation. Propagation is done by propagators. These are agents observing the constraint store and getting active whenever they are able to add information. The Oz programmer can observe the constraint store by using the *Oz Browser*. For instance, feed the following Oz-code into the Oz-compiler:

```
declare X Y Z in [X Y Z] ::: 1#10
{Browse [X Y Z]}
```

This declares three new variables `X Y Z` for integers in the domain 1, ..., 10 and browses whatever the constraint store knows about their values. When new information is added the browser updates its output. For instance, you may feed the propagator:

```
2 * Y =: Z
```

This propagator tells the constraint store new information on upper and lower bounds of Y and Z whenever possible. For example, it adds the information that `Y` must be at most 5 and `Z` must be at least 2 to the constraint store. However, it cannot tell the constraint store to remove odd numbers from the interior of the domain of Z. We next might feed a new propagator stating that X is strictly smaller than Y:

```
X <: Y
```

One of the effect of this propagator is that 1 is removed from the lower bound of Y. This reactivates the observing propagators `2 * Y =: Z` which excludes 2 and 3 from the domain of Z.

3.3.3 Composing the Solver

Oz supports encapsulated search. As in Prolog is sufficies to only specify a problem and let it be solved by the search engine of the programming language. In contrast to Prolog, search is encapsulated in Oz. This means that a search problem has always to be encapsulated into a predicate which has to be passed explicitly to a search engine. As a consequence of encapsulation, Oz permits standard programming in the usual style (i.e. as SML, Lisp, or Scheme).

In order to use encapsulated search, we have to encasulate the above propagators and distributor into a predicate. The procedure `Equations` describes exactly the solutions of the problem considered above.

```
declare
proc{Equations Sol}
   X Y Z
in
   Sol = solution(x:X y:Y z:Z)
   [X Y Z] ::: 1#7
   X + Y =: 3*Z
   X - Y =: Z
   {FD.distribute naive [X Y Z]}
end
```

The definition of `Equations` in Oz not only specifies a set of objects but also describes how these objects can be searched by propagation and distribution. For computing its solutions in Oz, it is sufficient to pass the definition of `Equations` to the Oz-Explorer.

```
{Explorer.all Equations}
{Explorer.one Equations}
```

3.3.4 Was this a good Example?

- Yes, because it was so simple.
- No, since there are much better solvers in this case (Gauss elimination algorithm).

Constraint programming yields good solvers only if no direct algorithm for solving your problem is available.

3.3.5 Questions

- Why are there three colons in the statement `[X Y Z] ::: 1#7`?

  If you want restrict the domain of a single FD variable then you write `X :: 1#7` with two colons. But if you want to restrict the domains of all variables of some list like `[X Y Z]`, then you need to write three colons.

- Is the name `solution` in the example program `Equations` arbitrary?

  Yes, you may choose whatever Oz-atom instead.

- What is the difference between the statements `X + Y =: 3*Z` and `X + Y = 3*Z`?

  Be careful, this is very different! The first statement `X + Y =: 3*Z` hides an application of a procedure which builds a propagator for the equation $X + Y = 3 * Z$. The second statement `X+Y=3*Z` is executed by first evaluating the arithmetic expressions `X+Y` and `3*Z` if the values of `X`, `Y`, and `Z` are specified and then unifying the results.

- Why does the Explorer come up with a yellow diamond in the following program instead of searching for a solution?

  ```
  declare
  proc{Equations Sol}
     X Y Z
  in
     Sol = solution(x:X y:Y z:Z)
     {FD.distribute naive [X Y Z]}
     [X Y Z] ::: 1#7
     X + Y =: 3*Z
     X - Y =: Z
  end

  {Explorer.one Equations}
  ```

The problem is that the distributor {`FD.distribute naive [X Y Z]`} blocks the execution of all subsequent statements. The distributor waits until the variables `X, Y, Z` have to denote integers in a finite domain. This will never happen since the execution of the statement `[X Y Z] ::: 1#7` is blocked by the distributor itself. So we have a deadlock.

The yellow diamond displayed by the Explorer means that the search process is blocked forever.

You can resolve the problem putting the distributor into its own thread, i.e. by replacing it with `thread` {`FD.distribute naive [X Y Z]`} `end`.

- I found the following call of the explorer in some document. What's wrong with this?

  ```
  {Explorer one(Equations)}
  ```

  This is the old syntax of Oz 2.0 which is no longer valid in Mozart 1.0.1. There the syntax for calling the Explorer is slightly different. You have to use the more consistent notation {`Explorer.one Equations`} instead.

3.3.6 Exercise

Write a solver for the equation SEND+MORE=MONEY, where every letter stands for a distinct digit between 0 and 9 and such that leading digits are distinct from 0.

3.4 Summary

- Underspecification and constraint programming are two sides of the same coin. The main idea of both is to delay case distinctions for as long as possible.
- Disambiguation of underspecified descriptions can be seen as constraint solving.
- The main problem of constraint solving is the danger of combinatoric explosion.
- The basic method of concurrent constraint programming is 'propagate and distribute', in contrast to 'generate and test'.
- Propagation is an efficient concurrent process. Propagation is typically incomplete from a logical point of view. Completeness can be obtained by adding distribution to propagation.

Lecture 4

More on Oz

The purpose of this lecture is to improve our Oz-programming skills. We will present those programming concepts needed for writing the solver of dominance constraints in the next lecture. We introduce the data structures provided by Oz which are similar to those in SML, then turn to first-order unification as in Prolog, and finally present features for concurrent constraint programming: finite domain constraints, finite set constraints, and disjunctive propagators.

4.1 Data Structures

We first introduce the data structures provided by Oz (see The Oz Base Environment). A data-structure allows to store values of some (data) type and provides the standard procedures for munching these values.

We take the viewpoint of functional programming as in SML which is quite distinct from the concept of constraint programming. However, functional programming provides a good platform on which to base a constraint programming system. The idea of functional programming is to organize computation purely in terms of values, types, and functional procedures which compute functions between values of some types.

4.1.1 Values and Types

Up to now we have seen several values used in Oz: numbers, atoms, records, and lists. There are more values and types in Oz. A still incomplete list of values and types is the following:

- A number is either an integer or a float (rational number).
- An atom is a word.
- A Boolean value is either true or false.
- The unit is a constant value without particular meaning (a dummy).

- A record aterm of the form Lab(F1:V1 ... Fn:Vn) where:
  - the label Lab is an atom, the unit, or a Boolean.
  - the features F1, ... , Fn are pairwise distinct atoms or integers.
  - the fields V1, ..., Vn are arbitrary values.
  - $n \geq 0$, ie. a record may be an atom, the unit, or a Boolean.
- A tuple is a record with only integer features.
- A list is a tuple which is either the atom nil or a tuple |(1:V 2:L) where | is an atom, V a value, and L a list. The atom | is sometimes called 'cons'.
- A procedure is a value.

4.1.2 Syntax for Values

Oz provides a lot syntactical alternative for describing the same value. We here present some typical descriptions, each of which determines some value completely.

- Integers are described as `0, 1, ~1, 2, 3` etc and floats by `0.0, 1.0, ~1.1` etc.
- Atoms are described by words starting with lower case letter like `thisIsAnAtom` or by a word in backwards quotes like `'case'`, `'true'` and `'ThisIsAnAtom'`.
- The Booleans and the unit are described by the keywords `true`, `false`, `unit`.
- Typical description for tuples and records are the following:

  ```
  plus(5 times(5 ~10))

  address(street:'Talstrasse'
          name:unit(first:hans
                    second:kamp))

  det(phon:a number:singular)
  ```

  In the first tuple, we have left out the features; it's a syntactically sugared version of `plus(1:5 2:times(1:5 2:~10))`.

  The values of a record at some feature can be selected by using the selection function that is denoted by a dot. For instance, the atom `singular` is described by the expression

  ```
  det(phon:a number:singular).number
  ```

- Typical descriptions of lists are: `1|2|3|nil`, `[1 2 3]`, and `nil`. Note however that `[ ]` does not describe the empty list!

- A description of a procedure computing the square function is:

  ```
  fun{$ X} X*X end
  ```

  The symbol `$` simply means that this procedure is anonymous, i.e. is not yet given a name. The syntax for the application of procedures uses curly brackets. For instance, the number `9` is described by the following application whose evaluation computes the square of `3`:

  ```
  {fun{$ X} X*X end 3}
  ```

4.1.3 Global and Local Variables

A variable in Oz describes a value of an arbitrary type. Variables in Oz are logic variable whose value cannot cannot be changed.

The Oz programming interface comes with a lot of predefined global variables such as `List` and `Number`. The values of both variables are records containing the standard functions for lists and numbers. For instance, a procedure for multiplication `Number.'*'` can be selected from the record `Number` at feature `'*'`. The expression `X*X` in turn is nothing else than syntactic sugar for the application {`Number.'*' X X`}.

Local variables can be introduced in Oz by using expression of the form `local ... in ... end`. The following piece of code describes a record which contains two number, the squares of 3 and 4.

```
local
   Square = fun{$ X} X*X end
in
   record(s3:{Square 3} s4:{Square 4})
end
```

The scope of a local variable is restricted by the local-end-expression in which its is introduced. For instance, the local variable `Square` cannot be accessed any further.

There is also a way for introducing new global variables in the programming interface by using the keyword `declare`. For instance we can declare the variable `X` and assign the value `2` to `X` as follows.

```
declare
X=2
```

Global variables are local with respect to the Oz-programming interface in which they were declared. Global variable can be accessed during a complete programming session with the same programming interface.

4.1.4 Browsing Values and Types

The Oz-Browser is a output tool provided by the Oz programming interface. The Oz-Browser is written in Oz itself and available via the global variable

`Browse`. For instance, we can browse the value of the global variable `X` above by executing:

```
{Browse X}
```

Evaluating the application {`Browse X`} simply evokes the side effect of browsing the value of `X`. Note that the execution of {`Browse X`} does not return a value in contrast to {`Square X`}. The reason is that `Browse` denotes a relational procedure which in contrast to a functional procedure (such as `Square`) does not return a output value when applied (see section procedures).

The Browser allows you to observe the values denoted by Oz-variables in its scope. For instance, feed the following lines to the emulator.

```
declare

R = address(street:'Talstrasse'
            name:unit(first:hans
                      second:kamp))
L = [1 2 3 4 5]

T = pair(L R F)

F = fun{$ X} X*X end

in

{Browse [R L T F]}
{Browse ['Browsing fun{$ X} X*X end yields <P/2>' F]}
```

When browsing the value of procedure named `F` a string is displayed meaning that `F` denotes a procedure with 2 arguments, an explicit one for input and an implicit one for output. The reason is that every functional procedure with n arguments is treated internally as a relational procedure with $n+1$ arguments.

The types of values can be checked in Oz dynamically, as illustrated by the following examples.

```
{Browse {IsRecord R}}
{Browse {IsRecord F}}

{Browse {Or {IsRecord ~100}
            {IsBool ~100}}}
{Browse {And {And
              {IsNumber ~100}
              {IsInt ~100}}
             {IsFloat ~100}}}

{Browse {Not {IsRecord false}}}
```

```
{Browse {IsRecord {IsRecord false}}}

{Browse {And
         {And
          {IsList L}
          {IsTuple L}}
         {IsRecord L}}}
```

There also exists a predefined procedure in Oz which computes the type of a given value. This is the procedure `Value.status`. When applied, it return not only the type of its input argument but also its actual status which may be either determined, kinded, or free.

```
{Browse [{Value.status R}
         {Value.status T}
         {Value.status L}
         {Value.status F}]}
```

For functional programming, we'd better deal only with values of status 'determined', in order to avoid suspensions (blocking computations).

4.1.5 Procedures

A *functional procedure* is a procedure which computes a function from values to a value, possibly depending on global values. Evaluating an application of a functional procedure means to pass the input values for its arguments, to compute the output value in function of the input values and the values of its global variables, and finally to output the output value (in case of termination).

As an example, we consider a description of the functional procedure called `SquareList`. When applied, this procedure inputs a list of integers and output the list of squares of these integers.

```
declare
fun{SquareList Ints}
   case Ints
   of     I|Is then I*I | {SquareList Is}
   elseof nil then nil
   end
end

{Browse {SquareList [1 2 3 4 5]}}
{Browse {SquareList {SquareList [1 2 3 4 5]}}}
```

Here, we use an alternative syntax for giving a name to a functional procedure. The following two forms are equivalent descriptions:

```
fun{SquareList Ints} ... end
SquareList = fun{$ Ints} ...end
```

Oz supports syntax for functional and relational procedures. Internally however, there are relational procedures only. A relational procedure behaves like a functional one except that it does not return an output value. Oz supports the following syntax for relational procedures (an anonymous and a named variant):

```
P1 = proc {$ X Y Z} ... end
proc {P2 U V} ... end
```

Applying a relational procedures usually has a side effect such as browsing a value. For instance, the following relational procedure browses the value of its argument twice.

```
proc {$ X} {Browse X} {Browse X} end
```

The output behaviour of a functional procedure can be simulated by a relational procedure which raises a side effect on a logic variable (see section unification). In fact, Oz supports functional procedure in that it provides functional descriptions of relational procedures. The description of functional procedure with n arguments is translated into a description of a relational procedure with $n+1$ arguments, where the last arguments serves as an output argument. For instance, the descriptions of the functional procedure `fun{Square X} X*X end` and its application `Y={Square 3}` are translated as follows:

```
fun{Square X} X*X end  ==>  proc{Square X Out} Out=X*X end
Y={Square 3}           ==>  {Square 3 Y}
```

Executing the application `{Square 3 Y}` has a side effect: the value `9` is assigned to the previously free variable `Y`.

4.1.6 Records

Records are the central data structure in Oz. Records are equally important in computational linguistics, where they are called feature trees. For instance, one might wish to represent the English word `girl` and its features as the following record:

```
word(cat:noun phon:[girl] subcat:determiner)
```

The main operation on records is feature selection which allows to access a field belonging to some feature. Feature selection is denoted by a dot. For instance:

```
{Browse word(cat:noun phon:[girl] subcat:determiner).phon}
{Browse word(cat:noun phon:[girl] subcat:determiner).phon.1}
```

Note that feature selection is a very efficient operation in Oz which can be done in constant time. A record is implemented as a *hash table* whose keys are the features of the record.

The base environment of Oz is provided by a set of records that are also called *modules*. Global variables denoting modules `Number`, `Record`, `List`, `FD`, and many more. For instance if you want to see the functionality provided for finite domains or records in Oz then simply browse the modules `FD` and `Record`.

```
{Browse FD}
{Browse Record}
```

This also explains the syntax of `FD.distribute` in our introductory example: a procedure for distribution is selected from the record `FD`. For further information on records, we refer to 'The Oz Base Environment'.

4.1.7 Lists

Lists are another important data structure in Oz similarly to Lisp. Therefore, much functionality for lists is provided in the Oz-module `List`. Again, we only give some examples here and refer to documentation 'The Oz Standard Modules' for further information.

Here is an example of a list which might be obtained by reading lexical information on natural language from some file:

```
declare

WordReps=[[mary noun nil]
  [john noun nil]
  [girl noun determiner]
  [nice adjective nil]
  [pretty adjective nil]
  [the determiner nil]
  [laughs verb noun]
  [meets verb [noun noun]]
  [kisses verb [noun noun]]
  [embarrasses verb [noun noun]]
  [thinks verb [verb noun]]
  [is verb [adjective noun]]
  [met adjective nil]
  [kissed adjective nil]
  [embarrassed adjective nil]]
```

As proposed above, one might wish to represent the features of a word in a more accessible way by using a record rather than a list. For instance, the record `word(cat:noun phon:[mary] subcat:nil)` is more readable than the list `[mary noun nil]`. More importantly, it is possible to select a feature of a word in the record representation in constant time, whereas it takes linear time in the number of features in the list representation.

Given the list of list `WordReps` above, we can compute a list of records `Words` by converting all represention in `WordReps`. This can be done by using the functional procedure `Map`:

```
declare

fun{Convert [P C S]}
   word(phon:[P] cat:C subcat:S)
end

Words = {Map WordReps Convert}

in

{Browse Words}
```

Note that the procedure `Map` is provided by the module `List`. Indeed, `Map` is identical to `List.map`, as shown when feeding:

```
{Browse Map==List.map}
```

Here, we apply the predefined functional procedure `==`, which compares two Oz-values for equality and returns its result as a Boolean value.

Next, we might want to filter all verbs out of the lexicon `Words`. This can be done by using the procedure `Filter` also defined in the module `List`:

```
declare

Verbs = {Filter Words fun{$ W}
 W.cat == verb
      end}
{Browse Verbs}
```

4.1.8 Concurrent Threads

Concurrency is an way to organize computation based on the notion of concurrent processes. Concurrency is well-known from operating systems like UNIX which support multi-tasking in order to administrate multiple windows each of which runs in its own process. Oz supports concurrent computation on a high level of abstraction. The presentation of concurrency in this reader stays at the very surface of the phenomenon.

A process in Oz is called a *thread*. A thread is created when executing a sequences of Oz-statement sequentially. A thread may block until more information becomes available. At first sight blocking may seem to be a programming error. For instance, consider:

```
declare  F
X={F 2}
```

```
{Browse 'this thread blocks'}
{Browse variables(x:X f:F number:1)}
```

When feeding this piece of code at once, nothing is browsed. The problem is that the value of the variable F is unknown such that the application of {F 2} has to blocks. All followup statements of the same thread (code sequence) are also blocked until the free variable F gets assigned a value (i.e. gets bound).

Using the programming interface, you can easily feed another sequence of statements which then computes concurrently in its own thread.

```
F=fun{$ Y} Y*Y end
```

Now, the value of F has become known. Thereby, the first thread become active again and could executed its remaining two Browse-statements.

You can also create your own threads without using the Oz-Programming-Interface. This can be done by using the command:

```
thread ... end
```

For instance, the above example can be rewritten such that the blocking application does not block the subsequent statements.

```
declare X F
thread
   X={F 2}
   {Browse 'this thread blocks ...'}
   {Browse variables(x:X f:F number:1)}
   {Browse '... but not forever'}
end
{Browse 'this thread does NOT block}
F=fun{$ Y} Y*Y end
```

This example illustrates the creation of a new thread which first blocks until the free variable F gets bound by the main thread which runs concurrently to its newly sporned thread.

Threads in Oz threads communicate over shared logic variables which play the same role such as channels in CML or PICT. In Oz, you can also consider a thread as a hand-written propagator which adds information about the value of variables to a shared constraint store.

4.2 Unification

Oz allows to compute with partial data structures, i.e partial descriptions of data structures. A partial description contains free variables, i.e. variables whose value is unspecified. We have already seen the usage of free variables for communication of concurrent threads. We will next show that a variable in Oz behaves such as a logic variables in Prolog. A logic variable can be understood as a place holder for a value which can be filled later on.

Data structures can be specified by equation systems between terms containing logic variables. Unification is the process of solving equations systems, i.e. to determine the possible values of its variables. Unification over first-order data structures such as records or tuples is built into Oz. Unification of records is known in computational linguistics under the name *feature unification*.

Suppose, for instance, that you want to unify the terms `f(X X)` and `f(g(Y Z) Y)`, where `X,Y,Z` are logic variables denoting some possibly infinite tree. In order to do so, it is sufficient to solve the equation `f(X X) = f(g(Y Z) Y)`. which can be done simply by feeding it into the Oz-emulator.

```
declare
   X Y Z
in
   f(X X) = f(g(Y Z) Y)
   {Browse [X Y Z]}
```

Equations between terms are basic constraints that can be entered directly into the constraint store without blocking their thread (the subsequent statements).

In the Browser, you can observe the result of the unification process. The variable `Z` is still free; the variables `X` and `Y` are bound to a term `g(g(g(... Z) Z) Z)` which can be solved by an infinite tree depending on the value of `Z`. Note that the equation `X.2 = X.1.2` is valid independently of the choice of `Z`.

Unification in Oz terminates even though the result can be the representation of an infinite tree. The reason is that a solved form of the equations with cycles can be stored in the Oz constraint store. This is similar to modern Prolog implementations, such as Sicstus Prolog.

4.3 Finite Domain Constraints

Oz is specifically designed for concurrent constraint programming. Now we introduce constraint programming in more detail. We consider a very popular class of constraints that are called finite domain (FD) constraints.

4.3.1 FD-Membership

Finite domain variables are variables that can denote one member of a finite set of integers. They can be used to express a simple form of disjunction. This form of disjunction is important when it comes to distribution.

A finite domain variable is a variable whose value is a natural number. Furthermore, the value of a finite domain variable can be constrained by some finite domain of natural numbers. For instance, the FD-membership constraint

```
X :: 1#5
```

is equivalent to $X \in \{1, 2, 3, 4, 5\}$ which in turn is equivalent to the disjunction:

$$X = 1 \vee X = 2 \vee X = 3 \vee X = 4 \vee X = 5$$

An FD-membership constraint such as `X :: 1#5` can be represented directely in the Oz constraint store. It is neither a propagator nor does it raise any case distinction.

4.3.2 FD-Propagators

Oz features several propagators for finite domain variables. We only present examples here and refer to the finite domain programming tutorial otherwise. The most important propagators are those for arithmetics. Propagators can be distinguished from pure evaluators by the colons like in `=:` or `=<:`.

```
3*X-Y  =: 4*Z  % linear arithmetics
3*X-Y =<: 4*Z  % inequations
```

For each FD-variable, a finite domain of possible values is maintained in the constraint store. What these propagators are doing is to restrict the upper and lower bounds of the domains of its variables; values from the interior of a finite domain are not excluded even if they contradict the logical semantics of the propagator.

Another useful propagator is the all-distinct propagator.

```
{FD.distinct [U V W X Y Z]}
```

Whenever the value of one of the variables in the list `[U V W X Y Z]` gets determined, this value is excluded from the domain of the others. The all-distinct propagator requires linear space in the number of variables it administrates, in contrast to a naive implementation which require quadratic space:

```
U\=:V   U\=:W  U\=:X   U\=:Y   U\=:Z
        V\=:W  V\=:X   V\=:Y   V\=:Z
               W\=:X   W\=:Y   W\=:Z
                       X\=:Y   X\=:Z
                               Y\=:Z
```

More on FD-propagators can be found in the tutorial on finite domain constraint programming in Oz.

4.3.3 FD-Distribution

Oz supports distribution for finite domain variables but only within encapsulated search. This is only operation which creates a choice node in a search tree.

Distributors can be created by applying the procedure `FD.distribute` to the name of a distribution strategy and a list of variables. For instance, the a distributor for the stategy first-fail (ff) picks a variable X of minimal current domain, splits this domain into two disjoint parts, each of which it considers in an independent part.

$$X \in D_1 \cup D_2 \qquad \Longrightarrow \qquad X \in D_1 \quad \vee \quad X \in D_2$$

Given that the domain $D_1 \cup D_2$ is split, encapsulated search process both possiblities $X \in D_1$ and $X \in D_2$ independently.

As said before, the split operation is evoked by the procedure `FD.distribute`. For instance, the domains of `X` and `Y` are split when in the following example:
⟨Distribution⟩ ≡

```
declare
proc{Problem Sol}
  X Y
in
  Sol = solution(x:X y:Y)
  X :: 1#5
  Y :: 2#3
  {FD.distribute ff [X Y]}
end
{Explore.all Problem}
```

Distribution in Oz is support during encapsulated search only (but NOT on top-level). This means that a problem has to encapsulated into a unary procedure which is then and then passed to the Oz-Explorer. Applying this procedure directely does not lead to distribution on top-level.

Note also that a distributor such as `{FD.distribute ff [X Y]}` blocks its thread (all subsequent statements) until distribution has happend (for ever on top-level). Therefor, a distributor should always be the last statement of its thread. This can be archieved either by writing it into the last line of the problem definition or by using a new thread anyway.

```
thread {FD.distribute ff [X Y]} end
```

4.4 Finite Set Constraints

Finite set constraints are also known from constraint programming but much less popular than finite domain constraints. Nevertheless, it turns out that finite set constraints are extremely useful for natural language processing.

A finite set (FS) variable denotes a finite set of integers. A finite set constraint describes the values of finite set variables based on the usual set operations. The reader should carefully note the difference between finite domain (FD) variables and finite set variables. An FD-variable denotes a single integer which can be desribed by a finite set of possibilities. A FS-variable denotes a finite set of integers which may be empty or contain more than one element.

There is two forms of basic finite set constraint which can be entered directely into the Oz-constraint-store. The upper:

```
X={FS.var.upperBound 1#6}
X={FS.var.lowerBound 2#4}
```

The former constraint states an upper bound $X \subseteq \{1,2,3,4,5,6\}$ whereas the latter requires a lower bound $\{2,3,4\} \subseteq X$. Beside of basic set constraints there are the following set propagators:

```
{FS.subset X Y}
X={FS.union Y Z}
X={FS.partition [U V W]}
{FS.include X I}
```

The declarative semantics of these constraints are rather obvious:

$$
\begin{aligned}
&X \subseteq Y \\
&X = Y \cup Z \\
&X = U \uplus V \uplus W \\
&I \in X
\end{aligned}
$$

Operationally, set propagators increase upper bounds and decrease lower bounds of set variables in the constraint store. The propagation behaviour can be tested at the following example:

```
declare
X={FS.var.upperBound 1#6}
Y={FS.var.upperBound 1#2}
Z={FS.var.upperBound 2#3}
{FS.subset X {FS.union Y Z}}
{FS.subset Y Z}
{FS.include 2 Y}

{Browse [X Y Z]}
```

There are more important set constraints in Oz that we will not present in this reader. Note also that we do not need distributors for set constraints.

4.5 Disjunctions as Propagators

There are several ways in Oz to express disjunctive information. The most convenient way are or-statements and finite domain constraints. As we will see, both of them can in an interlocked manner.

4.5.1 or-Statements

For instance, the possible gender-case-number information of the German word ‘schönen’ can be is described by the following or-statement which behaves as a disjunctive propagator.

⟨Or Statement⟩ ≡

```
or [Gen Cas Num]=[masc dat sg] then skip  % dem schönen Mann
[] [Gen Cas Num]=[masc acc sg] then skip  % den schönen Mann
[] [Gen Cas Num]=[masc nom pl] then skip  % die schönen M?er
[] [Gen Cas Num]=[masc gen pl] then skip  % der schönen M?er
[] [Gen Cas Num]=[masc dat pl] then skip  % den schönen M?ern
[] [Gen Cas Num]=[masc acc pl] then skip  % die schönen M?er
```

```
[] [Gen Cas Num]=[fem gen sg]  then skip  % der schönen Frau
[] [Gen Cas Num]=[fem dat sg]  then skip  % der schönen Frau
[] [Gen Cas Num]=[fem nom pl]  then skip  % die schönen Frauen
[] [Gen Cas Num]=[fem gen pl]  then skip  % der schönen Frauen
[] [Gen Cas Num]=[fem dat pl]  then skip  % den schönen Frauen
[] [Gen Cas Num]=[fem acc pl]  then skip  % die schönen Frauen
end
```

An or-statment consists of a set of clauses each of which has a guard and a body. For instance, the guard of the second clause above is the constraint `[Gen Cas Num]=[masc acc sg]`. The body of all clauses above are `skip`. The distinct behaviour of guards and bodies is explained in the next section.

4.5.2 Operational Semantics

An or-statement behaves as a propagator which concurrently investigates all its alternatives. Each alternative is continually monitored. The statement blocks until only one of the guards is consistent with the current constraint store; then it commits the clause according to the following rule:

```
or GUARD then BODY end   ==>  Guard Body
```

An or-statement reduces all its guards in parallel such that the constraints of the guard remain properly separated from those in the global constraint store. We say that every guard is executed in its own computation space.

As soon as a guard of a clause becomes inconsistent with the global constraint store, the clause is deleted from the or-statement. If one single clause remains then the or-statement reduces according to the rule above.

We can observe the semantics of or-statements by feeding the following pieces of code:

⟨Test the or-Statement⟩ ≡

```
declare
Gen Cas Num
⟨Or Statement⟩
{Browse 'An or-statement blocks its thread until it reduces'}
{Browse ['gender:' Gen 'case:' Cas 'number:' Num]}
/*
Cas=nom Gen=fem
*/
```

When having feeded theses lines, nothing should happen since the or-statement blocks its thread. But when feeding the constraint `Cas=nom Gen=fem` the two `Browse` statements following the or-statement should become active. Note in particular that the variable `Num` is determined to the value `pl`.

4.5.3 Choice Points versus Choice Variables

Unlike in Prolog, an Oz disjunction does not create a choice point,i.e. a case distinction. The only way to commit to one alternative is to cause all the others to become inconsistent.

⟨Disjunctive Propagator⟩ ≡
```
or {Equal N M}  then skip
[] {DomPlus N M} then skip
[] {DomPlus M N} then skip
[] {Side N M}
end
```

Yet, in order to perform search, we often need to force commitment to one or the other alternative. The standard trick in constraint programming is to introduce a *choice variable*, also known as a *control variable.*

We control the alternatives by a choice variable C. C is a finite domain variable with the domain `1#4`; simply by equating it with 1, 2, 3 or 4, we can commit to the corresponding alternative.

⟨Choice Variables⟩ ≡
```
or C=1 {Equal N M}  then skip
[] C=2 {DomPlus N M} then skip
[] C=3 {DomPlus N M} then skip
[] C=4 {Side N M} then skip
end
```

By distributing the values of the finite domain contol variable C we can now created choice points on need by `{FD.distribute naive [C]}` .

4.6 Summary

- Oz supports a wide range of *values*: numbers, atoms, booleans, records, lists, procedures, etc.
- An important data type is the *record*; it's essentially the same as a feature tree in computational linguistics. A record all of whose features are numbers is called a *tuple. Lists* are a particular sort of tuples.
- The variables in Oz are *logic variables*, which can be understood as placeholders for a value which can be filled in when needed. Oz supports *unification* of terms over the class of infinite trees.
- Oz supports *concurrent threads* which communicate over logic variables. The application of a free variable blocks its thread (all subsequent statements) until another thread assigns a value to the variable.
- *Finite domain constraints* are a very important class of constraints which is supported by the Oz standard library (see the Oz-reference manual

on System Modules). They specify relations between variables denoting members of a finite set of integers. Possible values can be narrowed down by propagation, and there are standard distribution strategies for distinguishing cases if necessary.

- *Finite set constraints* are an important class of constraints which is also supported by the Oz standard library (see the Oz-reference manual on System Modules). Finite set constraints provide propagators for the usual set operations.

- A *disjunction* can be used as a propagator in Oz if it is expressed by an `or` statement. An or-statement can be turned into a distributor by using a finite domain control variables and a finite domain distributor.

Lecture 5

Solving Dominance Constraints

In this chapter, we show how to solve dominance constraints by constraint programming with sets. While we won't say anything about the details, the techniques used here can be used as a basis to build more underspecified processing mechanisms for dominance constraints. For instance, the encoding of nodes presented below lends itself very well to capturing the interaction of scope and anaphora as in *Every man loves a woman. Her name is Mary.* In the sentence, the anaphoric reference excludes one reading of the first sentence; we can make this inference purely with propagation.

5.1 Dominance Constraints

We will consider the following language of tree descriptions based on dominance constraints:

$$
\begin{array}{rcl}
\varphi & ::= & \varphi \wedge \varphi' \\
& | & X{=}Y \\
& | & X{\neq}Y \\
& | & X \lhd^* Y \\
& | & X \neg\lhd^* Y \\
& | & X{:}(Y_1, \ldots, Y_n)
\end{array}
$$

This language is a variant of the dominance constraints defined in the second lecture. The differences are as follows:

- $X{=}Y$ expresses that X and Y must denote the same node. It's an abbreviation of $X \lhd^* Y \wedge Y \lhd^* X$.
- $X \neg\lhd^* Y$ expresses that X must not dominate Y. This couldn't be expressed in the original language.

- The new language doesn't contain lambda binding constraints. This is for simplicity of presentation; it's not difficult to add binding constraints to the implementation. Note that we can now speak just about trees, instead of lambda structures, as the models of dominance constraints.
- Labeling constraints have been replaced by 'daughterhood' constraints $X{:}(Y_1, \ldots, Y_n)$; the difference is that daughterhood constraints don't specify the label of X. This, too, is for simplicity, and labels could be (and have been) added easily to the implementation.

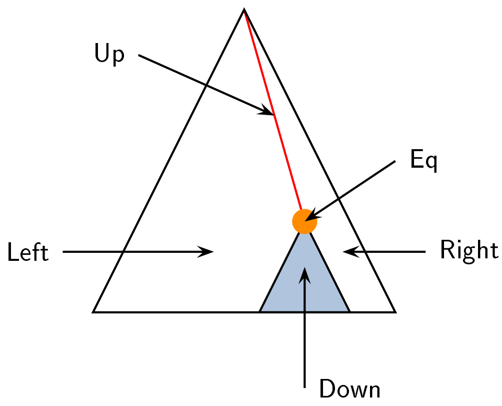
5.2 Constraint Solving as Configuration

We typically depict a dominance constraint as a (constraint) graph. A node of such a graph represents all occurrences of a variable at the same time. A graph then describes all those trees that satisfy the dominance relations required by the graph.

In the graph metaphor, solving a dominance constraint means to configure its nodes into a tree such that all required dominance relations hold. Of course, there is a naive 'generate and test' strategy for doing this: First, one can generate for each two nodes in a graph their relative positions in the tree described. A node can either be above the other node, below it, or 'to the side of it', i.e. neither above or below. In a second step, we can test which of our guesses are compatible with the dominance constraints required. This yields a non-deterministic polynomial time algorithm. In terms of complexity theory, one says that the problem of solving dominance constraints is in NP. The situation is worse than one might hope since the problem is in fact NP-complete. Thus we cannot expect any polynomial algorithm to exist. However, we can hope for an algorithm that is efficient for the applications to semantic underspecification.

5.3 Partitioning Trees

When regarded from a specific node, a tree is divided into 5 regions: (1) the node itself, (2) the nodes above, (3) the nodes below, (4) the nodes to the left, and (5) the nodes to the right.



In this chapter, we will aggregate the set of nodes to the left and to the right, and call the result the *side set*. A similar treatment can trivially be developed that retains the distinction; such a treatment would support precedence constraints.

Thus, in our treatment, any two nodes N_1 and N_2 of a tree must be in one of 4 mutually exclusive relationships:

1. $N_1{=}N_2$, they are equal
2. $N_1 \lhd^+ N_2$, N_1 strictly dominates N_2
3. $N_2 \lhd^+ N_1$, N_2 strictly dominates N_1
4. $N_1 \bot N_2$, N_1 is to the side of N_2 (i.e. none of the above).

We say that any 2 nodes N_1 and N_2 must satisfy the *treeness condition* expressed as the following disjunction:

$$\text{(A1)} \qquad N_1{=}N_2 \vee N_1 \lhd^+ N_2 \vee N_2 \lhd^+ N_1 \vee N_1 \bot N_2$$

In fact, we can reflect the 4 mutually exclusive possibilities above and associate, with a node N, 4 sets of variables:

1. N.eq, the set of variables of whose interpretation is equal N
2. N.up, the set of variables whose interpretations are strictly above N,
3. N.down, the set of variables whose interpretations are strictly below N,
4. N.side, the set of variables whose interpretations are to the side of N

The whole idea of our approach resides here: for each node, to characterize its position in a tree model in terms of these four sets of variables. These sets are disjoint and form a partition of the set V of variables in the input description:

$$\text{(A2)} \qquad V = N.eq \uplus N.up \uplus N.down \uplus N.side$$

5.4 Dominance Constraints as Set Constraints

In this section, we provide an implementation of the dominance constraint solver based on finite set constraints in Oz. We collect the functionality provided by the solver in a record called `DC` for dominance constraint.

5.4.1 Representation of Dominance Constraints

We encode a dominance constraint as a functional procedure taking as argument a list `[N1 N2 ... Nk]` of nodes, one for each variable of the description formula. This procedure then constrains theses nodes as required by the dominance constraint using procedures for atomic constraints that we are going to make available. Consider the dominance constraint which is typical for a scope ambiguity with two quantifiers.

$$X_1 : (X_2) \wedge X_2 \lhd^* X_5 \wedge X_3 : (X_4) \wedge X_4 \lhd^* X_5$$

We are interested in all solutions of this constraints where no variables are identified. This reflects that quantifiers should not be identified. It is slightly stronger than saying that nodes with distinct labels should not be identified.

$$\begin{array}{l} X1 \neq X2 \wedge X1 \neq X3 \wedge X1 \neq X4 \wedge X1 \neq X5 \wedge \\ X2 \neq X3 \wedge X2 \neq X4 \wedge X2 \neq X5 \wedge \\ X3 \neq X4 \wedge X3 \neq X5 \wedge \\ X4 \neq X5 \end{array}$$

Using the `DC` module, it would be expressed as a record which contains the number of variables and a procedure which inputs a list of nodes and creates set constraints for these nodes and the dominance constraint.

⟨DomConExample⟩ ≡
```
local
   proc {DomCon [N1 N2 N3 N4 N5]}
      {DC.daughters N1 [N2]}
      {DC.dominates N2 N5}
      {DC.daughters N3 [N4]}
      {DC.dominates N4 N5}
      {ForAll [N1#N2 N1#N3 N1#N4 N1#N5
               N2#N3 N2#N4 N2#N5
               N3#N4 N3#N5
               N4#N5]
       proc{$ N#M}
          {DC.notEqual N M}
       end}
   end
in
   DomConExample = 'unit'(domCon:DomCon
                          vars:5)
end
```

5.4.2 The Solver as a Module

We proved the dominance constraint solver as a record DC which provides all functionality required for solving dominance constraints. In a more serious implementation, modules could be defined by functors which can be made available as applets on the Internet.

⟨DC: dominance constraint solver⟩ ≡
```
local
   ⟨DC: daughters⟩
   ⟨DC: dominates⟩
   ⟨DC: not equal⟩
   local
      ⟨DC: equal⟩
      ⟨DC: strictly dominates⟩
      ⟨DC: side⟩
      ⟨DC: make node⟩
   in
      ⟨DC: make predicate⟩
   end
in
   DC=dom(makePredicate:MakePredicate
          daughters:Daughters
          dominates:Dominates
          notEqual:NotEqual)
end
```

In particular, the record DC exports the procedure MakePredicate which turns a dominance constraint into a predicate appropriate as input to encapsulated search as provided by e.g. Explorer.all or Search.all. For example, we could now use the Explorer[1] to search for all possible (constructive) models of DomConExample:

⟨DC.oz⟩ ≡
```
declare
  ⟨DC: dominance constraint solver⟩
  ⟨DomConExample⟩
in
   {Explorer.all {DC.makePredicate DomConExample}}
```

Here, the number 4 indicates the number of variables in the dominance constraint DomConExample.

5.4.3 Node Representation

A node is represented by a record. It contains an entry for each of the sets Eq, Down, Up and Side explained above, plus for the auxiliary sets EqDown (resp.

[1] http://www.mozart-oz.org/documentation/explorer/

EqUp), which are the unions of Eq and Down (resp. Up). Finally, the record has a feature daughters which will contain the set of daughter nodes, and a feature user, which is reserved for application-specific data. In the code below, I is the integer representing the variable. VDom is [1#N], where N is the number of variables in the description.

The constraints after the in specify that Eq, Down, Up and Side must form a partition of the set of variables in the description. Furthermore, the variable (encoded as integer I) that is interpreted by this node must be in the Eq set of the node.

⟨DC: make node⟩ ≡

```
fun {MakeNode I VDom}
   Eq     = {FS.var.upperBound VDom}
   Down   = {FS.var.upperBound VDom}
   Up     = {FS.var.upperBound VDom}
   Side   = {FS.var.upperBound VDom}
   EqDown = {FS.union Eq Down}
   EqUp   = {FS.union Eq Up}
in
   {FS.partition [Eq Down Up Side] {FS.value.make VDom}}
   {FS.include I Eq}
   node(
      eq        : Eq
      down      : Down
      up        : Up
      side      : Side
      eqdown    : EqDown
      equp      : EqUp
      daughters : _)
end
```

5.4.4 Translation to Set Constraints

If N1 dominates N2, then everything that is (weakly) below N2 must be (weakly) below N1, everything that is (weakly) above N1 must be (weakly) above N2, and everything that is beside N1 is also beside N2. Note however that there can be nodes beside N2 that are below N1.

⟨DC: dominates⟩ ≡

```
proc {Dominates N1 N2}
   {FS.subset N2.eqdown N1.eqdown}
   {FS.subset N1.equp   N2.equp  }
   {FS.subset N1.side   N2.side  }
end
```

The equality constraint is simply implemented by unification:

⟨DC: equal⟩ ≡

```
proc {Equal N1 N2} N1=N2 end
```

The disequality constraint states that the Eq sets of N1 and N2 must be disjoint:

⟨DC: not equal⟩ ≡

```
proc {NotEqual N1 N2}
   {FS.disjoint N1.eq N2.eq}
end
```

N1 strictly dominates N2 iff it dominates N2 and is not equal to N2:

⟨DC: strictly dominates⟩ ≡

```
proc {StrictlyDominates N1 N2}
   {Dominates N1 N2}
   {NotEqual  N1 N2}
end
```

If N1 is to the side of N2 (and reciprocally), then N1 and everything below it is to the side of N2 (and resp.):

⟨DC: side⟩ ≡

```
proc {Side N1 N2}
   {FS.subset N1.eqdown N2.side}
   {FS.subset N2.eqdown N1.side}
end
```

Finally, here is the constraint that deals with immediate dominance by specifying explicitly the daughters of a node `N` as a list `Nodes` of nodes. The set of nodes that are weakly below each of the daughters form a partition of the set of nodes that are strictly below the mother. Furthermore, the set of nodes that are strictly above each daughter is precisely the set of nodes that are weakly above the mother.

⟨DC: daughters⟩ ≡

```
proc {Daughters N L}
   N.daughters = L
   {FS.partition {Map L fun {$ D} D.eqdown end} N.down}
   {ForAll L proc {$ D} D.up=N.equp end}
end
```

5.4.5 Solution Predicate

`MakePredicate` is given the arguments `N`, the number of variables in the dominance constraint, and `P`, a procedure which takes a list of nodes corresponding to these variables and imposes the set constraints for the given dominance constraint. `MakePredicate` returns a unary predicate appropriate as an argument to e.g. `Search.all` or `Explorer.all`.

A search predicate always has the same form: it is a unary predicate whose argument denotes a solution. First it posts all constraints on the solution, then it specifies a search/distribution strategy:

⟨DC: make predicate⟩ ≡
```
fun {MakePredicate 'unit'(domCon:DomCon vars:N)}
   proc {$ Nodes}
      ⟨DC: create nodes⟩
      ⟨DC: translation to set constraints⟩
      ⟨DC: impose treeness⟩
   in
      ⟨DC: distribute⟩
   end
end
```

The solution `Nodes` must be a list of `N` nodes. Each variable is represented by a distinct integer between 1 and `N`. Thus sets of variables can be represented by sets of integers. (We store the specification of the finite domain from 1 to `N` in the variable `VDom`.) For each variable, `MakeNode` creates a term representing the node that is the interpretation of this variable.

⟨DC: create nodes⟩ ≡
```
VDom = [1#N]
{List.make N Nodes}   % constrains Nodes to a list
                      % [_ ... _] of length N
{List.forAllInd Nodes
 proc {$ I N} {MakeNode I VDom N} end}
```

Then we constrain these nodes using the procedure `DomCon` that implements the dominance constraint. After this we execute `choice skip end` whose only effect is to wait for stability; i.e. until constraint propagation has inferred as much as it could. Typically the dominance constraint `DomCon` provides very strong constraints and it is a good idea to impose them first and wait until they have achieved full effect before going on with the quadratic number of expensive treeness constraints.

⟨DC: translation to set constraints⟩ ≡
```
{DomCon Nodes}
% waits for stability
local H in H::1#1 {FD.distribute naive [H]} end
```

Now we impose the treeness constraint between every pair of nodes `Ni` and `Nj`. For every such pair we impose a choice which is controlled by its own choice variables with domain [1..4]. We collect the quadratic number of choice variables within the list `ChoiceVariables`.

⟨DC: impose treeness⟩ ≡
```
ChoiceVariables =
{List.foldRTail Nodes
 fun {$ Ni|Ns Cs}
    {List.foldR Ns
     fun {$ Nj Cs}
        ⟨DC: treeness condition between Ni and Nj⟩
```

```
            C|Cs
         end Cs}
     end nil}
```

Finally, we specify the distribution strategy: here we use *First Fail* on the choice variables. Each choice variable is a finite domain variable in [1..4]. First fail is a strategy which attempts to minimize the branching factor in the search tree: it picks a (non-determined) variable with the minimum number of remaining possible assignments.

⟨DC: distribute⟩ ≡

```
{FD.distribute ff ChoiceVariables}
```

5.4.6 Treeness Condition

The treeness condition that must hold between `Ni` and `Nj` is realized by four concurrent disjunctions and is controlled by choice variable `Cij`. The latter is a finite domain variable taking a value in [1..4].

⟨DC: treeness condition between Ni and Nj⟩ ≡

```
C in C::1#4
thread
   or C = 1 {Equal Ni Nj}
   [] C = 2 {StrictlyDominates Ni Nj}
   [] C = 3 {StrictlyDominates Nj Ni}
   [] C = 4 {Side Nj Ni}
   end
end
```

The `thread ... end` statements in the code fragment cause the computation to create four new concurrent threads, one for each choice variable. This is necessary because the `or` statements within the new threads block until only one of their guards can be satisfiable, and we don't want this to block our entire computation.

5.4.7 Better Propagation

A better implementation of the treeness condition can be obtained when providing propagators for further relations between nodes. This can be observed at the example given. The search tree of the more naive solver contains a failure node and two solution nodes. The smart solver contain no failure node any more and still the two solution nodes.

If N1 does not strictly dominate N2, then N1 is not strictly above N2 nor is N2 strictly below N1:

⟨DC smart: not strictly dominates⟩ ≡

```
proc {NotStrictlyDominates N1 N2}
   {FS.disjoint N1.eq N2.up  }
   {FS.disjoint N2.eq N1.down}
end
```

The fact that neither N1 nor N2 is on the side of the other can be expressed by:

⟨DC smart: not side⟩ ≡

```
proc {NotSide N1 N2}
   {FS.disjoint N1.eq N2.side}
   {FS.disjoint N2.eq N1.side}
end
```

We can now state the treeness condition in a smarter way.

⟨DC smart: treeness condition between Ni and Nj⟩ ≡

```
C in C::1#4
thread or C = 1 {    Equal Ni Nj}
       [] C\=:1 {NotEqual Ni Nj}
       end
end
thread or C = 2 {    StrictlyDominates Ni Nj}
       [] C\=:2 {NotStrictlyDominates Ni Nj}
       end
end
thread or C = 3 {    StrictlyDominates Nj Ni}
       [] C\=:3 {NotStrictlyDominates Nj Ni}
       end
end
thread or C = 4 {    Side Ni Nj}
       [] C\=:4 {NotSide Ni Nj}
       end
end
```

Note that this code is equivalent to an `or` of four alternatives as above but the code shown here leads to fewer better propagation and thus less failure.

Here comes the rest of the code for a smarter dominance constraint solver which is based on the smarter treeness condition. Apart from the smarter treeness condition there is nothing else new here.

⟨DC smart: impose treeness⟩ ≡

```
ChoiceVariables =
{List.foldRTail Nodes
 fun {$ Ni|Ns Cs}
    {List.foldR Ns
     fun {$ Nj Cs}
        ⟨DC smart: treeness condition between Ni and Nj⟩
        C|Cs
     end Cs}
 end nil}
```

⟨DC smart: make predicate⟩ ≡

```
fun {MakePredicate 'unit'(domCon:DomCon vars:N)}
   proc {$ Nodes}
```

```
         ⟨DC: create nodes⟩
         ⟨DC: translation to set constraints⟩
         ⟨DC smart: impose treeness⟩
      in
         ⟨DC: distribute⟩
      end
   end
```

⟨DC smart: dominance constraint solver⟩ ≡

```
   local
      ⟨DC: daughters⟩
      ⟨DC: dominates⟩
      ⟨DC: not equal⟩
      local
         ⟨DC: equal⟩
         ⟨DC: strictly dominates⟩
         ⟨DC: side⟩
         ⟨DC: make node⟩
         ⟨DC smart: not strictly dominates⟩
         ⟨DC smart: not side⟩
       in
         ⟨DC smart: make predicate⟩
      end
   in
      DC=dom(makePredicate:MakePredicate
     daughters:Daughters
     dominates:Dominates
     notEqual:NotEqual)
   end
```

⟨DCSmart.oz⟩ ≡

```
   declare
      ⟨DC smart: dominance constraint solver⟩
      ⟨DomConExample⟩
   in
      {Explorer.all {DC.makePredicate DomConExample}}
```

5.5 Full Code of the Dominance Constraint Solvers

The code below is available from the file DC.oz[2].

```
   declare
     local
        proc {Daughters N L}
           N.daughters = L
```

[2] code/DC.oz

```
   {FS.partition {Map L fun {$ D} D.eqdown end} N.down}
   {ForAll L proc {$ D} D.up=N.equp end}
end
proc {Dominates N1 N2}
   {FS.subset N2.eqdown N1.eqdown}
   {FS.subset N1.equp   N2.equp  }
   {FS.subset N1.side   N2.side  }
end
proc {NotEqual N1 N2}
   {FS.disjoint N1.eq N2.eq}
end
local
   proc {Equal N1 N2} N1=N2 end
   proc {StrictlyDominates N1 N2}
      {Dominates N1 N2}
      {NotEqual  N1 N2}
   end
   proc {Side N1 N2}
      {FS.subset N1.eqdown N2.side}
      {FS.subset N2.eqdown N1.side}
   end
   fun {MakeNode I VDom}
      Eq     = {FS.var.upperBound VDom}
      Down   = {FS.var.upperBound VDom}
      Up     = {FS.var.upperBound VDom}
      Side   = {FS.var.upperBound VDom}
      EqDown = {FS.union Eq Down}
      EqUp   = {FS.union Eq Up}
   in
      {FS.partition [Eq Down Up Side] {FS.value.make VDom}}
      {FS.include I Eq}
      node(
         eq        : Eq
         down      : Down
         up        : Up
         side      : Side
         eqdown    : EqDown
         equp      : EqUp
         daughters : _)
   end
in
   fun {MakePredicate 'unit'(domCon:DomCon vars:N)}
      proc {$ Nodes}
        VDom = [1#N]
        {List.make N Nodes}   % constrains Nodes to a list
                              % [_ ... _] of length N
```

```
            {List.forAllInd Nodes
             proc {$ I N} {MakeNode I VDom N} end}
            {DomCon Nodes}
            % waits for stability
            local H in H::1#1 {FD.distribute naive [H]} end
            ChoiceVariables =
            {List.foldRTail Nodes
             fun {$ Ni|Ns Cs}
                {List.foldR Ns
                 fun {$ Nj Cs}
                    C in C::1#4
                    thread
                       or C = 1 {Equal Ni Nj}
                       [] C = 2 {StrictlyDominates Ni Nj}
                       [] C = 3 {StrictlyDominates Nj Ni}
                       [] C = 4 {Side Nj Ni}
                       end
                    end
                    C|Cs
                 end Cs}
             end nil}
          in
            {FD.distribute ff ChoiceVariables}
          end
       end
    end
in
   DC=dom(makePredicate:MakePredicate
          daughters:Daughters
          dominates:Dominates
          notEqual:NotEqual)
end
local
   proc {DomCon [N1 N2 N3 N4 N5]}
      {DC.daughters N1 [N2]}
      {DC.dominates N2 N5}
      {DC.daughters N3 [N4]}
      {DC.dominates N4 N5}
      {ForAll [N1#N2 N1#N3 N1#N4 N1#N5
               N2#N3 N2#N4 N2#N5
               N3#N4 N3#N5
               N4#N5]
       proc{$ N#M}
          {DC.notEqual N M}
       end}
   end
```

```
   in
      DomConExample = 'unit'(domCon:DomCon
                             vars:5)
   end
in
   {Explorer.all {DC.makePredicate DomConExample}}
```

The code of the smart solver is available from the file `DCSmart.oz`[3].

```
declare
   local
      proc {Daughters N L}
         N.daughters = L
         {FS.partition {Map L fun {$ D} D.eqdown end} N.down}
         {ForAll L proc {$ D} D.up=N.equp end}
      end
      proc {Dominates N1 N2}
         {FS.subset N2.eqdown N1.eqdown}
         {FS.subset N1.equp   N2.equp  }
         {FS.subset N1.side   N2.side  }
      end
      proc {NotEqual N1 N2}
         {FS.disjoint N1.eq N2.eq}
      end
      local
         proc {Equal N1 N2} N1=N2 end
         proc {StrictlyDominates N1 N2}
            {Dominates N1 N2}
            {NotEqual  N1 N2}
         end
         proc {Side N1 N2}
            {FS.subset N1.eqdown N2.side}
            {FS.subset N2.eqdown N1.side}
         end
         fun {MakeNode I VDom}
            Eq     = {FS.var.upperBound VDom}
            Down   = {FS.var.upperBound VDom}
            Up     = {FS.var.upperBound VDom}
            Side   = {FS.var.upperBound VDom}
            EqDown = {FS.union Eq Down}
            EqUp   = {FS.union Eq Up}
         in
            {FS.partition [Eq Down Up Side] {FS.value.make VDom}}
            {FS.include I Eq}
            node(
```

[3] `code/DC.oz`

```
            eq        : Eq
            down      : Down
            up        : Up
            side      : Side
            eqdown    : EqDown
            equp      : EqUp
            daughters : _)
      end
      proc {NotStrictlyDominates N1 N2}
         {FS.disjoint N1.eq N2.up  }
         {FS.disjoint N2.eq N1.down}
      end
      proc {NotSide N1 N2}
         {FS.disjoint N1.eq N2.side}
         {FS.disjoint N2.eq N1.side}
      end
   in
      fun {MakePredicate 'unit'(domCon:DomCon vars:N)}
         proc {$ Nodes}
            VDom = [1#N]
            {List.make N Nodes}   % constrains Nodes to a list
                                  % [_ ... _] of length N
            {List.forAllInd Nodes
             proc {$ I N} {MakeNode I VDom N} end}
            {DomCon Nodes}
            % waits for stability
            local H in H::1#1 {FD.distribute naive [H]} end
            ChoiceVariables =
            {List.foldRTail Nodes
             fun {$ Ni|Ns Cs}
                {List.foldR Ns
                 fun {$ Nj Cs}
                    C in C::1#4
                    thread or C = 1 {   Equal Ni Nj}
                           [] C\=:1 {NotEqual Ni Nj}
                           end
                    end
                    thread or C = 2 {   StrictlyDominates Ni Nj}
                           [] C\=:2 {NotStrictlyDominates Ni Nj}
                           end
                    end
                    thread or C = 3 {   StrictlyDominates Nj Ni}
                           [] C\=:3 {NotStrictlyDominates Nj Ni}
                           end
                    end
                    thread or C = 4 {   Side Ni Nj}
```

```
                              [] C\=:4 {NotSide Ni Nj}
                              end
                       end
                       C|Cs
                    end Cs}
                end nil}
            in
               {FD.distribute ff ChoiceVariables}
            end
         end
      end
   in
      DC=dom(makePredicate:MakePredicate
             daughters:Daughters
             dominates:Dominates
             notEqual:NotEqual)
   end
   local
      proc {DomCon [N1 N2 N3 N4 N5]}
         {DC.daughters N1 [N2]}
         {DC.dominates N2 N5}
         {DC.daughters N3 [N4]}
         {DC.dominates N4 N5}
         {ForAll [N1#N2 N1#N3 N1#N4 N1#N5
                  N2#N3 N2#N4 N2#N5
                  N3#N4 N3#N5
                  N4#N5]
          proc{$ N#M}
             {DC.notEqual N M}
          end}
      end
   in
      DomConExample = 'unit'(domCon:DomCon
                             vars:5)
   end
in
   {Explorer.all {DC.makePredicate DomConExample}}
```

5.6 Summary

- Concurrent Constraint Programming allows a very intuitive implementation of a solver for dominance constraints.
- Every variable is associated with *four sets of nodes*: the sets of variables equal, strictly above, strictly below, and to the side of it.

- *Finite set constraints* can be used to axiomatize the problem; they can be taken over in Mozart with only syntactic variations.

- The dominance constraint solver based on finite set constraints has been integrated into the *CHORUS demo system* and runs efficiently on dominance constraints from underspecified semantics.

Bibliography

Alshawi, H., D. Carter, R. Crouch, S. Pulman, M. Rayner, and A. Smith (1992). CLARE: A contextual reasoning and cooperative response framework for the Core Language Engine. Technical Report CRC-028, SRI International, Cambridge, England. `http://www.cam.sri.com/tr/crc028/paper.ps.Z`.

Alshawi, H. and R. Crouch (1992). Monotonic semantic interpretation. In *Proceedings of the 30th ACL*, Kyoto, 32–39.

Backofen, R., J. Rogers, and K. Vijay-Shanker (1995). A first-order axiomatization of the theory of finite trees. *Journal of Logic, Language, and Information 4*, 5–39.

Blackburn, P. and J. Bos (1999). Representation and inference for natural language: A first course in computational semantics. Lecture notes, `http://www.coli.uni-sb.de/~bos/comsem`.

Bodirsky, M., M. Egg, A. Koller, J. Niehren, K. Striegnitz, and S. Thater (1999). Chorus demo system. `http://www.coli.uni-sb.de/cl/projects/chorus/software.html`.

Bos, J. (1996). Predicate logic unplugged. In *Proceedings of the 10th Amsterdam Colloquium*, 133–143.

Cooper, R. (1975). *Montague's semantic theory and transformational syntax.* Ph. D. thesis, University of Massachusetts, Amherst.

Cooper, R. (1983). *Quantification and Syntactic Theory.* Dordrecht: Reidel.

Copestake, A., D. Flickinger, and I. Sag (1997). Minimal Recursion Semantics. An Introduction. Manuscript, available at `ftp://csli-ftp.stanford.edu/linguistics/sag/mrs.ps.gz`.

Dalrymple, M., J. Lamping, F. Pereira, and V. Saraswat (1997). Quantifiers, anaphora, and intensionality. *Journal of Logic, Language, and Information 6*, 219–273.

Duchier, D. and C. Gardent (1999). A constraint-based treatment of descriptions. In *Proceedings of IWCS-3*, Tilburg.

Duchier, D., C. Gardent, and J. Niehren (1999). Concurrent constraint programming in Oz for natural language processing. Lecture notes, `http://www.ps.uni-sb.de/~niehren/oz-natural-language-script.html`.

Duchier, D. and J. Niehren (1999). Solving dominance constraints with finite set constraint programming. Submitted. `http://www.ps.uni-sb.de/Papers/abstracts/DomCP99.html`.

Egg, M., J. Niehren, P. Ruhrberg, and F. Xu (1998). Constraints over Lambda-Structures in Semantic Underspecification. In *Proceedings COLING/ACL'98*, Montreal.

Gamut, L. T. F. (1991). *Logic, Language, and Meaning.* Chicago and London: University of Chicago Press.

Gardent, C. and B. Webber (1998). Describing discourse semantics. In *Proceedings of the 4th TAG+ Workshop*, Philadelphia. University of Pennsylvania.

Hirschbühler, P. (1982). VP deletion and across the board quantifier scope. In J. Pustejovsky and P. Sells (eds), *NELS 12*, Univ. of Massachusetts.

Hobbs, J. (1983). An improper treatment of quantification in ordinary english. In *Proceedings of the 21st ACL*, Cambridge, Massachusetts, 57–63.

Hobbs, J. and S. Shieber (1987). An algorithm for generating quantifier scoping. *Computational Linguistics 13*, 47–63.

Keller, W. (1988). Nested Cooper storage: The proper treatment of quantification in ordinary noun phrases. In U. Reyle and C. Rohrer (eds), *Natural Language Parsing and Linguistic Theory.* Dordrecht: Reidel.

Koller, A. (1999). Constraint languages for semantic underspecification. Diplom thesis, Universität des Saarlandes, Saarbrücken, Germany. `http://www.coli.uni-sb.de/~koller/papers/da.html`.

Koller, A., J. Niehren, and K. Striegnitz (1999). Relaxing underspecified semantic representations for reinterpretation. In *Proceedings of the Sixth Meeting on Mathematics of Language (MOL6)*, Orlando, Florida. `http://www.coli.uni-sb.de/~koller/papers/reint.html`.

Koller, A., J. Niehren, and R. Treinen (1998). Dominance constraints: Algorithms and complexity. In *Proceedings of the Third Conference on Logical Aspects of Computational Linguistics*, Grenoble.

Marcus, M. P., D. Hindle, and M. M. Fleck (1983). D-theory: Talking about talking about trees. In *Proceedings of the 21st ACL*, 129–136.

Montague, R. (1974). The proper treatment of quantification in ordinary English. In R. Thomason (ed.), *Formal Philosophy. Selected Papers of Richard Montague*. New Haven: Yale University Press.

Muskens, R. (1995). Order-Independence and Underspecification. In J. Groenendijk (ed.), *Ellipsis, Underspecification, Events and More in Dynamic Semantics*. DYANA Deliverable R.2.2.C. `http//www.ims.uni-stuttgart.de/ftp/pub/papers/DYANA2/95copy/R2.2.C/Musk%ens.ps.gz`.

Oz Development Team (1999). The Mozart Programming System web pages. `http://www.mozart-oz.org/`.

Partee, B. H. and H. L. W. Hendriks (1997). Montague grammar. In J. van Benthem and A. ter Meulen (eds), *Handbook of Logic and Language*, Chapter 1, 5–91. Amsterdam: Elsevier.

Reyle, U. (1993). Dealing with ambiguities by underspecification: construction, representation, and deduction. *Journal of Semantics 10*, 123–179.

Saraswat, V. A., M. Rinard, and P. Panangaden (1991). Semantic foundations of concurrent constraint programming. In *ACM Symposium on Principles of Programming Languages*, 333–352. ACM Press, New York.

Schiehlen, M. (1997). Disambiguation of underspecified discourse repesentation structures under anaphoric constraints. In *Proceedings of IWCS-2*, Tilburg.

Smolka, G. (1994). A foundation for concurrent constraint programming. In *Constraints in Computational Logics*, 50–72. Springer-Verlag, Berlin.

Smolka, G. (1995). The Oz Programming Model. In J. van Leeuwen (ed.), *Computer Science Today*, 324–343. Springer-Verlag, Berlin.

Vijay-Shanker, K. (1992). Using descriptions of trees in a tree adjoining grammar. *Computational Linguistics 18*, 481–518.