Constraint-Based And Graph-Based Resolution of Ambiguities in Natural Language

---

Dissertation zur Erlangung des Grades
des Doktors der Naturwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Eingereicht von Alexander Koller
Saarbrücken, den 19.07.2004

Verfasser: Alexander Koller
Katholisch-Kirch-Straße 15
66111 Saarbrücken
koller@coli.uni-sb.de

Dekan: Prof. Dr. Jörg Eschmeier

Prüfungsausschuss: Prof. Dr. Wolfgang Wahlster
Prof. Dr. Manfred Pinkal
Prof. Dr. Gert Smolka
Prof. Dr. Michael Kohlhase
Dr. Ernst Althaus

Tag des Kolloquiums: 18.11.2004

Abstract

This thesis develops the theory of *dominance constraints*, a family of logical languages that describe trees, and applies them as a formalism for the *underspecified* description of scope ambiguities in natural language. In underspecification approaches to ambiguity resolution, all readings of a sentence at once are represented in an underspecified description, and are only enumerated by need.

On the one hand, dominance constraints allow us to model scope phenomena declaratively, within a logic formalism. On the other hand, we can view certain fragments of dominance constraints as graphs, which makes it possible to employ efficient graph algorithms to solve dominance constraints. All constraints that are used in scope underspecification fall into these fragments.

In addition, we can use extra information such as anaphoric reference and world knowledge in order to restrict the set of readings described by a dominance constraint. The constraint is strengthened to exclude unintended readings without having to enumerate them.

Kurzzusammenfassung

Diese Dissertation entwickelt die Theorie von *Dominanzconstraints*, einer Familie von Logiksprachen zur Beschreibung von Bäumen, und wendet sie als Formalismus zur *unterspezifizierten* Beschreibung von Skopusambiguitäten in natürlicher Sprache an. In der Unterspezifikation werden alle Lesarten eines Satzes auf einmal in einer unterspezifizierten Beschreibung dargestellt und nur bei Bedarf aufgezählt.

Dominanzconstraints ermöglichen einerseits die deklarative, logikbasierte Modellierung von Skopusphänomenen. Andererseits kann man bestimmte Fragmente von Dominanzconstraints als Graphen sehen, so dass effiziente Graphalgorithmen zum Lösen von Dominanzconstraints eingesetzt werden können. Alle Dominanzconstraints, die in der Skopusunterspezifikation auftreten, fallen in diese Fragmente.

Aufbauend auf diesen Ergebnissen kann man außerdem zusätzliche Informationen wie anaphorische Referenz und Weltwissen einsetzen, um die von einem Dominanzconstraint beschriebenen Lesarten einzuschränken. Dabei wird der Dominanzconstraint verstärkt, um unerwünschte Lesarten auszuschließen, ohne sie aufzählen zu müssen.

Ausführliche Zusammenfassung

Diese Arbeit entwickelt die Theorie der *Dominanzconstraints*, definiert einen effizienten, graphbasierten Algorithmus, um sie zu lösen, und wendet sie auf das Problem der Auflösung von Skopus-Mehrdeutigkeiten in natürlichen Sprachen an. Sie zeigt, dass Unterspezifikationsformalismen effizient verarbeitet werden können und eine geeignete Plattform darstellen, um unerwünschte Lesarten auszufiltern, ohne sie aufzählen zu müssen. Unterspezifikation ist ein Ansatz zur Behandlung von semantischen Mehrdeutigkeiten, bei dem alle Lesarten eines Satzes in einer unterspezifizierten Beschreibung kompakt dargestellt und nur bei Bedarf aufgezählt werden.

Dominanzconstraint-Sprachen sind Logiksprachen, die über endlichen Bäumen interpretiert werden. Die Formeln dieser Sprache sprechen über Relationen zwischen den Knoten eines Baumes, insbesondere über die Dominanzrelation (also die transitive Vorfahr-Relation). Wir definieren in einem ersten Teil der Arbeit verschiedene Dominanzconstraint-Sprachen und erweitern sie um *Bindungsconstraints*, mit denen wir Variablenbindung auf der Ebene der beschriebenen semantischen Lesarten modellieren können. Wir geben eine einfache Grammatik an, die Dominanz- und Bindungsconstraints als unterspezifizierte Beschreibungen für die semantischen Repräsentationen englischer Sätze ableitet. Dann geben wir einen Überblick über zwei verschiedene Algorithmen zum Lösen von Dominanz- und Bindungsconstraints, die auf Saturierung des Constraints sowie auf einer Übersetzung in Mengenconstraints beruhen.

Im zweiten Teil der Arbeit zeigen wir, wie bestimmte Fragmente von Dominanzconstraints als *Graphen* aufgefasst werden können. Wir geben zunächst eine Übersetzung von *normalen* Dominanzconstraints in *Dominanzgraphen* an, mit der Eigenschaft, dass der Constraint genau dann erfüllbar ist, wenn der Graph *lösbar* ist, d.h. in einen Baum konfiguriert werden kann. Wir charakterisieren Lösbarkeit von Dominanzgraphen als die Abwesenheit von *einfachen hypernormalen Zykeln* und geben einen Algorithmus an, der in quadratischer Zeit entscheidet, ob ein Dominanzgraph einen hypernormalen Zykel enthält. Damit haben wir einen quadratischen Erfüllbarkeitstest für normale Dominanzconstraints.

Allerdings sind wir in der Skopusunterspezifikation meistens nicht an beliebigen Lösungen eines Dominanzconstraints interessiert, sondern nur an *konstruktiven* Lösungen, in denen nur das vom Satz vorgegebene semantische Material vorkommt. Wir zeigen, dass erfüllbare *hypernormal verbundene* Dominanzconstraints grundsätzlich konstruktive Lösungen haben, und machen damit den

Graphlöser unmittelbar für die Skopusunterspezifikation anwendbar; er wird damit zum ersten beweisbar polynomiellen Löser für irgendeinen Unterspezifikationformalismus überhaupt. Alle Constraints, die die vorher definierte Grammatik erzeugt, sind hypernormal verbunden, und wir glauben, dass dies allgemein für alle Constraints, die in der Unterspezifikation verwendet werden, gilt.

Weiterhin zeigen wir, dass die hypernormal verbundenen Fragmente von Dominanzconstraints und von Hole Semantics, einem anderen Unterspezifikationsformalismus, äquivalent sind. Damit setzen wir erstmals zwei praktisch relevante Unterspezifikationsformalismen in Beziehung, was zum Verständnis der theoretischen Landschaft der Unterspezifikation beiträgt und uns erlaubt, Ressourcen wie Löser und Grammatiken zwischen den verschiedenen Formalismen zu übertragen.

Im dritten und letzten Teil der Dissertation wenden wir die Ergebnisse aus den ersten beiden Teilen auf das Problem an, auf der Grundlage von zusätzlichen Informationen unerwünschte Lesarten eines Satzes zu eliminieren. Wir modellieren zunächst den Zusammenhang von anaphorischen Referenzen und Skopusambiguitäten, indem wir die Zugänglichkeitsbeziehung der dynamischen Prädikatenlogik als Relationen zwischen den Knoten eines Baumes darstellen. Dann geben wir Inferenzregeln an, mit denen ein Dominanzconstraint so verstärkt werden kann, dass Lesarten herausgefiltert werden, die die Zugänglichkeitsanforderungen verletzen. In manchen Fällen können wir so alle unerwünschten Lesarten eliminieren, ohne sie aufzählen zu müssen.

Schließlich modellieren wir den Effekt von Weltwissen auf Skopusambiguitäten. Dazu geben wir ein Verfahren an, das erkennt, wenn alle Lesarten eines Constraints unerfüllbar sind, indem es nur eine Teilmenge der Lesarten mit einem Theorembeweiser überprüft. Dieses Verfahren setzt ein Rewrite-System auf den Lesarten eines Constraints ein, das wir vorher im Zusammenhang mit speziellen hypernormal verbundenen Constraints entwickelt hatten. Weil die logische Unerfüllbarkeit von Lesarten in manchen Fällen nicht die intuitiv richtigen Vorhersagen darüber macht, welche Lesarten auf Unerfüllbarkeit geprüft werden müssen, untersuchen wir, inwiefern wir Präsuppositionen der natürlichsprachlichen Sätze hinzuziehen können, um ein intuitiv korrekteres Modell zu bekommen.

Acknowledgments

If I have seen further, it is by
standing on the shoulders of giants.
– Isaac Newton

The front page of this thesis proudly displays my name, and it is true that the work I report in it has occupied the best part of my waking hours for over five years. However, virtually every result in this thesis was developed in a group effort, and the research would have been literally impossible without the contributions of my colleagues. Moreover, it took a great social support network of good friends to get through the thesis writing intact. I would like to take this opportunity to repay the help I have received over that time in a small way.

My first thanks go to my advisor, Manfred Pinkal. I find it hard to imagine a more supportive mentor, a fairer boss, and an advisor who dishes out more relevant criticism in a more constructive way. His insistence on clarity and precision in everything I wrote has made a significant direct impact on the shape of the thesis. Manfred has taught me a lot about making a coherent, clear story out of an unconnected mess of scientific ideas, and about political thinking. The large research group he manages to maintain as a result of these abilities has made for an extremely stimulating research environment that I have profited from enormously.

My other advisor, Gert Smolka, deserves no less gratitude. Gert provided a rather different perspective on my work, by taking the fundamental facts about linguistic modelling much less for granted than any other of my collaborators. The ensuing discussions have done much to help clarify my own views on these subjects. At the same time, he helped me grow up as a computer scientist by pointing out where "the rest is standard" and the gory details didn't need to be spelled out. I'm glad that he insisted that I write Chapter 3; and it is only because he brought it up again that I'm now a Doctor of Natural Sciences, rather than of Engineering.

My main collaborator in the research reported here has been Joachim Niehren. It is hard to overestimate how much I've learned from Joachim about how to do research, about scientific taste, and about paper writing. Joachim has a unique talent for identifying least publishable units that make for beautiful papers. His enthusiasm for topics and ideas that he finds interesting is infectious, and this is in fact how he first drew me into the CHORUS project seven years ago.

I owe a lot of linguistic insight to Markus Egg. Throughout these past years, it has been an enormous relief to rely on his linguistic judgments and opinions, in particular because he has a rare talent to explain them with utter clarity. He has also been a wonderful office neighbour with whom I have had many interesting and controversial discussions. Michael Kohlhase came into the thesis work at a crucial, rather late stage, and it was only due to him that I didn't throw Chapter 8 into a corner and give up in frustration. His energy and indomitable optimism are spectacular, and I hope a little of it has rubbed off on me. My three visits to Pittsburgh were intense and instructive and fun, and Michael and his family always made me feel very welcome.

The thesis research has taken place in the context of the CHORUS project within the SFB 378. This has given me the privilege of working with one of the finest research groups that I could ever hope to be part of. Beyond the people I have already mentioned, Katrin Erk, Marco Kuhlmann, Stefan Thater, Ralph Debusmann, and Denys Duchier have been both colleagues and friends, and while it seems to be impossible to work in CHORUS without being or becoming exceedingly stubborn, the general atmosphere has always been incredibly productive and supportive. In addition, the research on normal dominance constraints would have been much harder without the cooperation with the Max Planck Institute for Computer Science, most notably Kurt Mehlhorn, Sven Thiel, and Ernst Althaus. These guys really do know everything about graph algorithms, and it has been a joy to throw them half-baked problems and then watch them solve them.

The department of computational linguistics at Saarbrücken is a wonderful research environment, which I will be sad to leave when the time comes. The Saarbrücken-Edinburgh International Graduate College "Language Technology and Cognitive Systems", to which I am proud to have been associated, is a great addition to this environment and has attracted a gang of incredibly clever and cool people. I don't have enough space here to thank everybody as properly as they deserve, but I'd like to pick out a (much too small) handful specifically: Malte Gabsdil, who is one of the most "people smart" people I know, and together with Ute has become my family away from home; Kristina Striegnitz, with whom I am able to communicate at an amazing bandwidth, and who is always a pleasure to work (or do anything else) with; Geert-Jan Kruijff, my partner in crime in building talking robots; Matt

Crocker, who has managed to dissolve my intense dislike for statistical methods; and Helga Riedel, who is really the one person who holds the whole place together.

At some point in the past five years, I ended up getting a bit of a life. This amazing development was made possible by all the great people in the university rowing club and the university choir. It has been a great pleasure to sing with Note Nors, and in particular with my quartet partners Joachim Fox, Alexander Indermark, and Johannes Laubscher; I am glad that Judith Braun taught me how to sing (and a lot of related things) so I could do my part. I seem to have acquired an international network of friends, including Carlos Areces and Raffaella Bernardi through ESSLLI, Inga Hell and Meredith Patterson over the Net, and many others, and now that the thesis is finished, you can expect me to go travel and visit you all.

Especially during the last few months of writing, when I didn't allow myself the relief of working much on things that weren't thesis-related, P.H.D. comics and Language Log have been instrumental in keeping me sane.

Carlos Areces, Peter Baumgartner, Markus Egg, Geert-Jan Kruijff, Manfred Pinkal, Gert Smolka, and Stefan Thater have all proofread parts of the thesis. I am very grateful for all the comments they gave me. All remaining errors are of course my own fault.

Finally, I am privileged that Stefanie Schmetkamp-Schmidt has remained my best friend, and that her son Quentin is such a great little godson. And I am deeply grateful to my parents and family simply for being there. I'm very proud of them, and if this thesis gives them some further reason to be proud of me, then it has been worth it.

Contents

Chapter 1

Introduction

This thesis develops the theory of *dominance constraints*, defines an efficient graph-based algorithm for solving them, and applies them to the problem of ambiguity resolution in natural language. It shows that underspecification formalisms can be processed efficiently and that they can be used to eliminate unintended readings without enumerating them.

Dominance constraint languages are logical languages that are interpreted over trees. Formulas of these languages talk about relations between the nodes in a tree, most notably dominance (i.e. the transitive ancestor relation). We define the different languages of dominance constraints and present previous work on solving dominance constraints. Then we give *normal* dominance constraint formulas an interpretation as *graphs*, and translate notions such as satisfiability into graph properties. This allows us to solve dominance constraints efficiently by using graph algorithms.

In a second track of the thesis, we apply these theoretical results to the problem of resolving scope ambiguities in natural language. Ambiguity – the fact that some token of language has more than one possible analysis on some level of linguistic description – is one of the fundamental problems in computational linguistics. Scope ambiguities are a certain type of semantic ambiguity, which can be elegantly represented using *underspecification*. We show that dominance constraints can be used as underspecified descriptions of sentences with scope ambiguities, and that the graph solver can be applied to enumerate all their readings efficiently. In addition, we model the effect of contextual information (in particular, of anaphoric references and world knowledge) on the resolution of scope ambiguities, and apply the formal results to eliminate contextually impossible readings without enumerating them.

1.1 Ambiguity

Ambiguity – the phenomenon that a sentence or some of its parts have more than one linguistic analysis – is very common in natural language. The following sentences are examples which represent several different types of ambiguity on various levels of linguistic description:

(1.1) Peter went to the bank.

(1.2) Peter saw the man with the telescope.

(1.3) Every linguist speaks two languages.

(1.4) Peter met a man in the park. He carried a telescope.

The sentence (1.1) contains a *lexical ambiguity*: "Bank" could either refer to a financial institution, or to a bank of a river. (1.2) contains an *attachment ambiguity*: It is ambiguous between a reading in which Peter uses the telescope to watch the man and a reading in which the other man carries the telescope. The difference is in the syntactic structure of the sentence, i.e. whether "with the telescope" modifies (is attached to) "the man" or the verb. (1.3) contains a *scope ambiguity*: It is syntactically unambiguous, but its semantics is ambiguous between the reading where there is a single set of two languages (say, English and German) which every linguist speaks, and a reading in which every linguist can pick her own pair of languages. (1.4) contains a *referential ambiguity*: Is it Peter or the other man who carries the telescope?

Most grammars for deep linguistic processing aim at deriving fully specified syntactic or semantic representations for a sentence – e.g., a parse tree, or a formula representing the truth conditions of the sentence. As a consequence, they will assign 2^n different possible readings to a sentence that contains n ambiguities with two readings each. Because the rules and the lexicon entries of the grammar generalise over the particular properties of each single word, even harmless-looking sentences are often assigned a surprisingly large number of readings, as the following two examples illustrate:

(1.5) Covering old surfaces and other people's floors with holes and spots with dust will have great effects.

(1.6) But that would give us all day Tuesday to be there.

The sentence (1.5) contains multiple attachment ambiguities, and is predicted by the LFG ParGram grammar (Butt et al. 2002) to have 27062 syntactic readings (Anette Frank, p.c.). On the other hand, even if one specific syntactic analysis of the sentence (1.6) (which comes from the Redwoods Treebank, Oepen et al. 2002) is fixed, the English Resource Grammar (ERG, Copestake and Flickinger 2000) assigns it 64764 different semantic readings, purely due to scope ambiguity. If the two types of ambiguities combine, it is easy to imagine sentences of quite moderate length for which the grammar predicts millions of readings.

The problem of selecting the "correct" reading which was actually intended by the speaker of the sentence is called *ambiguity resolution*. Because ambiguity is such a pervasive phenomenon, it is perhaps the greatest computational challenge that deep linguistic processing is faced with today. But at the same time, human language users seem to have no problems at all in understanding sentences like (1.5) and (1.6), and indeed don't even perceive these sentences as ambiguous in any given context. They seem to apply a mixture of methods including

1. the use of information such as anaphora and world knowledge (combined with the actual meanings of the words), which allows them to reject some readings as impossible: If we continue the sentence (1.3) with the sentence "These two languages are English and German", the anaphor "these two languages" makes the reading in which each linguist can speak their own set of languages infelicitous.

2. the use of *preferential* information, such as word order or the frequency with which certain syntactic constructions are used, which allows us to favour a reading over another (still possible) reading: We have a tendency to understand a sentence like "Peter watched the man with a telescope" to mean that Peter used the telescope for watching the man, as it is such a suitable instrument for watching.

3. the ability not to resolve some ambiguities completely at all, but rather to operate with an approximation of the concrete readings: For instance, the scope distinctions that the ERG makes in (1.6) are mostly technical, the different readings are mostly equivalent, and we don't even perceive that there is an ambiguity.

The position this thesis takes towards the problem of ambiguity resolution is that we start with an unresolved representation of all the readings that a grammar predicts. Then we offer algorithms for enumerating all predicted readings, and we investigate how anaphora and world knowledge can be used to eliminate unintended (in the sense of the first method above) readings without enumerating them. The

use of preferences is not the subject of this thesis, although we briefly speculate on how to integrate preferences into our framework in the conclusion.

1.2 Scope Underspecification

The type of ambiguity we focus on here is scope ambiguity. Scope ambiguities are semantic ambiguities – i.e., a sentence can be analysed as syntactically unambiguous and still be semantically ambiguous because of scope. They are characterised by the fact that their semantic representation contains *scope-bearing elements*, such as logical quantifiers, that can take scope over each other in different ways. The two readings of the example (1.3) can be written as follows ($\exists_2$ is an ad-hoc "quantifier" expressing that its scope denotes a set of cardinality two; it can be expressed easily in standard first-order logic):

(1.7) $\forall x.\mathsf{linguist}(x) \rightarrow (\exists_2 y.\mathsf{language}(y) \wedge \mathsf{speak}(x, y))$

(1.8) $\exists_2 y.\mathsf{language}(y) \wedge (\forall x.\mathsf{linguist}(x) \rightarrow \mathsf{speak}(x, y))$

Here the two quantifier fragments $\forall x.\mathsf{linguist}(x) \rightarrow \bullet$ and $\exists_2 y.\mathsf{language}(y) \wedge \bullet$ can take scope over each other in either order, and each way of arranging them corresponds to a distinct semantic reading. These two fragments are exactly the semantic contributions of the two noun phrases "every linguist" and "two languages". Most examples of scope ambiguity that we use below are scope ambiguities between two noun phrases. But in principle, all methods developed here apply to scope phenomena in a much wider sense, such as the following:

(1.9) Max eats a cake sometimes.

(1.10) Helmut Kohl wants to visit a Chinese factory producing CFC-free fridges.

(1.11) I try to read a novel if I feel bored or I am unhappy.

Example (1.9) is ambiguous between a reading in which Max eats the same cake each time, and one in which he eats different cakes. Here one scope-bearing element is the noun phrase "a cake", and the other one is the adverb "sometimes". The scope ambiguity in the example (1.10), which translates a sentence from the NEGRA/TIGER corpus (Skut et al. 1998; Brants et al. 2002), belongs more specifically to the class of *de dicto/de re ambiguities*. It has a reading in which Kohl wants to visit a specific factory, and another one in which he would be happy with

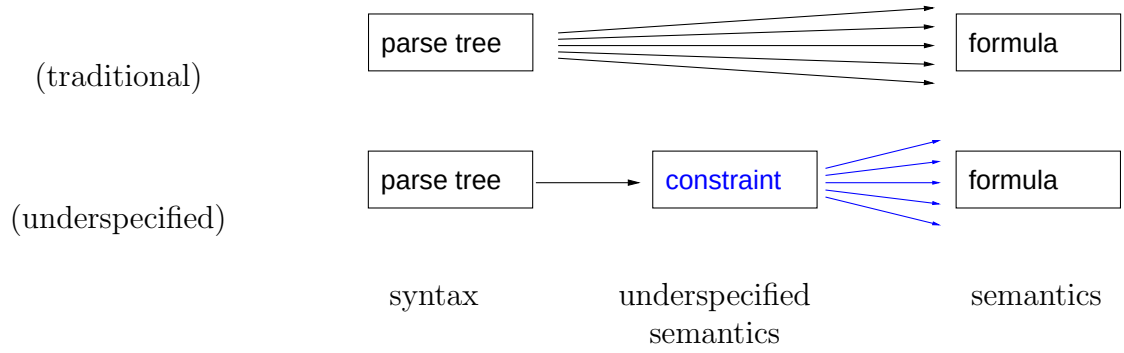


Figure 1.1: Underspecified semantics.

any factory whatsoever. One scope-bearing element here is the verb "wants". The scope-bearing elements in (1.11) are the discourse units and discourse connectives like "if" and "or". Adverbs as scope bearers in particular are rather frequent in corpora: An informal study of the first few hundred sentences of the NEGRA corpus showed that pure NP/NP scope ambiguities like (1.3) are quite rare, but about one third of all sentences contains a scope ambiguity if we also count NP/adverb or adverb/adverb scope ambiguities.

One particularly attractive approach to dealing with scope ambiguity is *underspecification* (Alshawi and Crouch 1992; Reyle 1993; Muskens 1995; van Deemter and Peters 1996; Bos 1996; Pinkal 1996; Copestake et al. 1999). The main idea behind underspecification is to delay the computation of all semantic readings for a syntactic analysis. Rather than computing them all right away, as a traditional syntax-semantics interface would, we first compute an *underspecified semantic description* (Fig. 1.1). Each syntactic analysis gives rise to only one underspecified description, and the description should be built in such a way that the semantic readings can be extracted from it efficiently if they are needed. This makes underspecification appealing from a computational point of view because the combinatorial explosion inherent in the enumeration of readings is localised into the problem of computing readings from descriptions, and thus it can be delayed until the readings are actually needed.

In addition, we can in principle use underspecified descriptions as a platform on which external information, such as about anaphoric reference and world knowledge, can be incorporated. The additional information could strengthen the underspecified description so it describes only a subset of its former readings; readings that contradict the extra information should be eliminated without ever enumerating them. This potential for modelling human strategies for ambiguity management also makes underspecification appealing from a cognitive perspective.

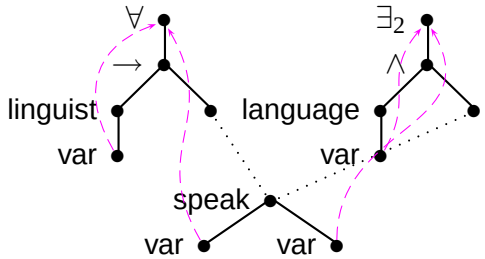


Figure 1.2: Underspecified description for (1.3) as a dominance constraint graph.

There is a range of formalisms for representing underspecified descriptions in the literature; perhaps the most well-known ones are Quasi-Logical Form (QLF, Alshawi and Crouch 1992) and Underspecified Discourse Representation Theory (UDRT, Reyle 1993) for their historical impact, Hole Semantics (Bos 1996) for its conceptual simplicity, and Minimal Recursion Semantics (MRS, Copestake et al. 1999) because it is the formalism supported by the large-scale HPSG grammars. In this thesis, we use the language family of dominance constraints as our underspecification formalism. We describe the two readings of (1.3) by the graph shown in Fig. 1.2. The dotted lines in the graph signify dominance, and indicate that both quantifier fragments must outscope the subformula $\mathsf{speak}(x, y)$. The dashed arrows indicate variable binding. Technically, we give the informal graph a formal meaning as a constraint over trees, and later translate the constraints back into graphs in order to process them efficiently with graph algorithms.

1.3 Constraint Programming

The computational framework in which we couch the algorithms developed in this thesis is *constraint programming* (Apt 2003; Saraswat et al. 1991; Smolka 1995). In its most general form, the idea behind this programming paradigm is to model a problem using *constraints* over variables, e.g. formulas from a small fragment of first-order logic. Then it aims at *solving* the constraint, either by general methods mostly involving search, or by domain-specific methods. Put this generally, constraint programming encompasses not just many forms of combinatorial problems, but also fields as large as linear programming and operations research.

To get an idea what a constraint-based model of a problem looks like, let's look at an example. Consider the following arithmetic problem, in which each letter stands for a distinct digit, and leading digits are nonzero.

```
   SEND
+  MORE
= MONEY
```

In a first step, we model this problem as a constraint satisfaction problem by reading the letters S, E, etc. as *finite domain variables*, whose range is the finite domain $\{0, \ldots, 9\}$ of integers. We can then state the digits puzzle using arithmetic formulas such as

$$\begin{array}{rcrcrcrcr} & & 1000 \cdot S & + & 100 \cdot E & + & 10 \cdot N & + & D \\ & + & 1000 \cdot M & + & 100 \cdot O & + & 10 \cdot R & + & E \\ = 10000 \cdot M & + & 1000 \cdot O & + & 100 \cdot N & + & 10 \cdot E & + & Y \end{array}$$

In a second step, we can try to *solve* this constraint – i.e., find a variable assignment that satisfies it – by applying a *search algorithm*. Such an algorithm tries various combinations of values for the variables, and reports the satisfying assignments. The most naive such algorithm could first iterate over the ten possible values for D, then within each iteration over the ten possible values for E, and so on. This *generate-and-test* algorithm clearly leads into a combinatorial explosion as it evaluates all 10^8 value combinations, and becomes impractical even for slightly larger problem instances.

Constraint programming tackles this problem by applying a *propagate-and-distribute* search algorithm instead. It classifies constraints into *simple* and *complex* constraints. Simple constraints, such as $M = 1$ and $S \geq 8$, are stored in a *constraint store*. Complex constraints are turned into *propagators*, which concurrently compute consequences of the constraint store and add them back to the store. For instance, suppose we had the complex constraint $X = M + S$. This constraint would give rise to a propagator that watches M and S for changes. As soon as the simple constraints $M = 1$ and $S \geq 8$ are in the constraint store, this propagator would contribute the simple constraint $X \geq 9$. If we also had the information $X < 10$ in the constraint store, it would add the simple constraint $S < 9$ to the constraint store, and so forth.

Once no propagator can infer additional constraints, the system must perform a *distribution* step, i.e. a case distinction. It could e.g. distinguish the cases $E = 4$ and $E \neq 4$, and create two new constraint stores that are like the old store, but with $E = 4$ added to the first copy and $E \neq 4$ added to the other copy. Then the propagators can infer new constraints on the new constraint stores. Just as in the generate-and-test case, the algorithm explores a search tree, and the search tree can still be exponential in size, but this tree is generally much smaller than that of the generate-and-test algorithm.

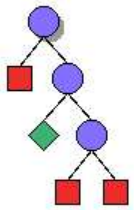


Figure 1.3: Search tree for the "send more money" puzzle using propagate and distribute.

The search tree for the digits puzzle, as explored by the solver for finite domain constraints implemented in the Mozart Programming System (Oz Development Team 1999), is shown in Fig. 1.3. Each node can be seen as a constraint store after propagation. Children are created by distribution from their parents, boxes are failed (i.e. inconsistent) constraint stores, and diamonds are solved constraint stores, in which every variable has a unique value that satisfies the constraint. Note that the search tree has only seven nodes, compared to 10^8 nodes in the generate-and-test algorithm. On the other hand, notice that there is a node that only has failed children. This happens because the propagators in the Mozart system are designed to be efficient, and this comes at the price that they can't always detect the unsatisfiability of an unsatisfiable constraint before further distribution.

Once a problem has been successfully modelled in a logic-based framework, an optional third step in the constraint programming approach is to define more efficient special-purpose algorithms for solving certain classes of constraints. In the "send more money" example, this could perhaps be a solver that processes all linear equations simultaneously with algorithms from linear algebra. This third step occupies the central chapters of this thesis, as we translate normal dominance constraints into dominance graphs, and then define graph algorithms that solve the original constraints.

1.4 What this thesis is about

Building upon the previous work in scope underspecification and constraint programming mentioned so far, this thesis defines dominance constraints, explores them theoretically, and applies them as a formalism for scope underspecification. We derive a variety of interesting theoretical results, including an efficient algorithm for solving normal dominance constraints. These results make it possible to tackle problems in ambiguity resolution that haven't been solved in this form before.

The thesis consists of three parts. The first part is an overview of the logic-based modelling of scope ambiguities based on dominance constraints. We first define the different dominance constraint languages, synthesising definitions from earlier papers (Chapter 2). This chapter also contains a novel, flexible definition of *binding constraints*, which can be used to model different forms of variable binding. We then show how we can systematically compute dominance constraints that describe the possible readings of English sentences (Chapter 3), and review algorithms for solving dominance and binding constraints based on saturation and on finite set constraints (Chapter 4).

In the second part of the thesis, we develop the graph perspective on dominance constraints. We first prove that normal dominance constraints can be seen as *dominance graphs*, in such a way that satisfiability of the constraint corresponds to the question whether the graph has a *solved form* (Chapter 5). We show that solvability of dominance graphs can be checked in quadratic time, and thus obtain a quadratic satisfiability algorithm for normal dominance constraints. The algorithm extends straightforwardly to an efficient algorithm for enumerating solutions. Because all dominance constraints used in scope underspecification are normal, this enumeration algorithm constitutes the first provably polynomial solver for any scope underspecification formalism in the literature.

However, the solved forms of the dominance graph correspond to solutions of the normal dominance constraint that may contain arbitrary additional semantic material that wasn't mentioned in the sentence. The readings of a sentence correspond to the *constructive* solutions of the constraint, and there are satisfiable constraints that have no constructive solutions. In Chapter 6, we resolve this discrepancy by proving that every solved form of a *hypernormally connected* dominance constraint has a constructive solution. Hypernormally connected constraints are normal dominance constraints whose graphs satisfy an additional connectedness property. We prove that all dominance constraints generated by the grammar from Chapter 3 are hypernormally connected, and argue that this may be more generally true for all dominance constraints that are ever needed for underspecified semantics. In addition, we prove that hypernormally connected dominance constraints are equivalent to the hypernormally connected fragment of Hole Semantics (Bos 1996), another underspecification formalism. This is the first ever formal encoding between practically useful underspecification formalisms. It helps clarify the underspecification landscape, and allows us to share resources (such as grammars and solvers) between the different formalisms.

In the third part of the thesis, we apply the results from the first two parts to the problem of resolving scope ambiguities based on extra knowledge. We first show how our methods for processing binding constraints can be applied to modelling

the interaction of scope and anaphora (Chapter 7). Using these methods, we can strengthen underspecified descriptions by propagation, and often eliminate all undesired readings without any enumeration. Then we model the interaction between scope ambiguities and world knowledge (Chapter 8). We use rewriting techniques developed in Chapter 6 in order to establish the unsatisfiability of *all* readings by checking only a *subset* of the readings for unsatisfiability with a theorem prover. Because it turns out that unsatisfiability doesn't always make the intuitively correct predictions about which readings must be checked for unsatisfiability, we explore whether *presuppositions* of the natural-language sentence can be used to obtain a more intuitive model.

The three parts are closely interlinked because many of the theoretical results have immediate consequences for computational challenges in scope underspecification. Taken together, they establish dominance constraints as a powerful underspecification formalism, which has a clear definition, is convenient to work with and prove things about, has efficient algorithms, and supports inferences that can eliminate readings by using context information. We present a syntax-semantics interface that computes dominance constraints for English directly, and we also show how to encode descriptions from a different underspecification formalism, allowing us to directly re-use resources for the other formalism.

From a more general perspective, the work reported here is relevant because solving normal dominance constraints is a *graph configuration problem*, i.e. it requires us to configure a known set of nodes into a graph while respecting additional constraints. Other problems in computational linguistics, such as dependency parsing (Duchier 2003; Debusmann 2003) and realisation with TAG (Koller and Striegnitz 2002) can be seen as graph configuration problems as well. In this context the theoretical results and efficient algorithms we develop here for dominance constraints can take on a prototype role for the development of similar results for other graph configuration problems.

1.5 Contributions

This thesis makes the following contributions:

1. A presentation of the syntax, semantics, and processing of dominance constraints, and of a syntax-semantics interface for English, together with a new generalisation of the definition and processing of binding constraints (Chapters 2, 3, and 4). Joint work with Joachim Niehren, Markus Egg, and Katrin Erk (Egg et al. 2001; Erk et al. 2003).

2. A polynomial satisfiability algorithm for normal dominance constraints (Chapter 3). Joint work with Kurt Mehlhorn, Sven Thiel, Joachim Niehren, Ernst Althaus, and Denys Duchier (Koller et al. 2000; Althaus et al. 2001; Althaus et al. 2003).

3. Definition of hypernormally connected dominance constraints, and proof that all their solved forms have constructive solutions (Chapter 6). Joint work with Stefan Thater, based on earlier work on *chains* with Joachim Niehren and Kristina Striegnitz (Koller et al. 1999; Koller et al. 2000).

4. Equivalence of hypernormally connected dominance constraints and hypernormally connected Hole Semantics descriptions (Chapter 6). Joint work with Stefan Thater (Koller et al. 2003).

5. Structural theory of the solution sets of *pure chains*: Every binary tree is the constructive solution of exactly one pure chain, and their constructive solutions can be rewritten into each other with a rewrite system (Chapter 6).

6. Resolution of scope ambiguities based on anaphoric references (Chapter 7). Joint work with Joachim Niehren (Koller and Niehren 2000).

7. Resolution of scope ambiguities based on unsatisfiability of readings (Chapter 8). Joint work with Michael Kohlhase.

Chapter 2

Dominance Constraints

This chapter sets the stage for the rest of the thesis, by introducing the different languages of dominance constraints we will use. We first present the basic intuitions behind dominance constraints (Section 2.1), and then formally define their syntax and semantics (Section 2.2). We extend dominance constraints with *binding constraints* in Section 2.3. Finally, we compare dominance constraints to some related formalisms, and show how they are embedded into the more powerful Constraint Language for Lambda Structures (CLLS) in Section 2.4.

Most of the work reported in this chapter has been published in (Egg et al. 1998; Egg et al. 2001). However, we have streamlined some definitions for this thesis, and the generalised treatment of binding in Section 2.3 is new and extends work in (Erk et al. 2003).

2.1 Elements of dominance constraints

The term "dominance constraints" stands for a family of logical languages that are interpreted over ordered labelled trees. A dominance constraint is also a formula of one of these languages; in this sense, dominance constraints are conjunctions of predicate logic atoms. The predicate symbols used in these atoms express relations between nodes in a tree, and the variables in a dominance constraint denote nodes in a tree. Different dominance constraint languages allow different sets of predicate symbols. We will define this precisely in Section 2.2.2; this expository section assumes a dominance constraint language that talks only about two node relations, namely *labelling* and *dominance*.

The idea of using a dominance constraint as an underspecified representation builds

upon the fact that semantic representations for natural language sentences, such as in (1.7) and (1.8), are typically written as logical formulas in some other logical language (the *object language* – in the example, first-order logic). Logical formulas have a natural tree structure, so we can describe a set of object language formulas as the set of trees satisfying some dominance constraint. In order to represent and describe variable binding in the object language, we can extend a dominance constraint with a *binding constraint*.

We will now first show how object language formulas can be seen as trees, if we disregard variable binding, and how a dominance constraint describes a set of such trees. Then we will add mechanisms to represent variable binding. Finally, we will illustrate that these mechanisms are indeed powerful enough to be useful for a range of different object languages that are being used for semantic representations. We will mostly choose not to write a dominance constraint as a logical formula; instead, we will use a graphical notation in order to appeal to intuitions. The connection between the graphs and the constraints will be made clear in Section 2.2.2.

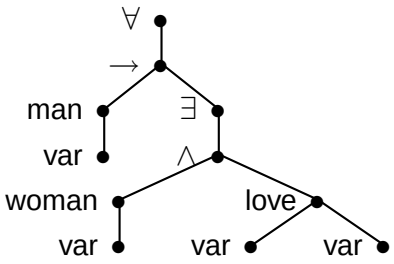
2.1.1 Trees and dominance constraints

Consider, by way of example, the following representation of one reading of the sentence "Every man loves a woman" as a formula of first-order predicate logic.

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

The main connective of this formula is a universal quantifier. Its subformula is an implication, which in turn has two subformulas, and so on recursively. That is, the formula has a natural tree structure, which we can draw as follows:

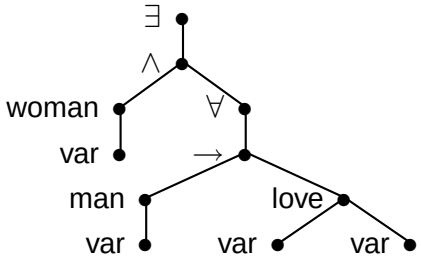
(2.2)



The examples sentence has another reading (2.3); the structural tree of this formula is (2.4).

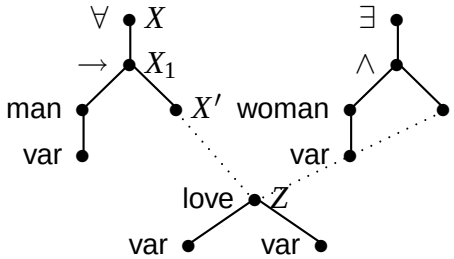
(2.3) $\exists y.\mathsf{woman}(y) \wedge \forall x.(\mathsf{man}(x) \rightarrow \mathsf{love}(x, y))$

(2.4)



If we look closely at the two trees in (2.2) and (2.4), we notice that they are composed of the same tree fragments, corresponding to the formula fragments $\forall x.\mathsf{man}(x) \rightarrow \cdot$, $\exists y.\mathsf{woman}(y) \wedge \cdot$, and $\mathsf{love}(x, y)$. But these fragments are composed in different order. We can describe both trees at once by specifying the fragments and their structural relationships. Such a description is given in (2.7):

(2.5)



Intuitively, the graph (2.5) describes all trees into which it can be embedded. Dotted edges in the graph signify *dominance*: Of the two nodes they connect, the upper one must be above the lower one in the tree structure. The graph leaves the exact relative ordering between the two quantifier fragments unspecified. But since both fragments dominate the atom $\mathsf{love}(x, y)$ and trees cannot branch in the bottom-up direction, one of the two quantifier fragments must dominate the other.

The graph can be embedded both into (2.2) and into (2.4) while respecting the dominance requirements. So we can take it as an underspecified description of these two trees (and hence, of the formulas they represent). While the original object level formulas (2.1) and (2.3) (and their structural trees) belong to the category "semantics" in Fig. 1.1, the graph belongs to the category "underspecified semantics".

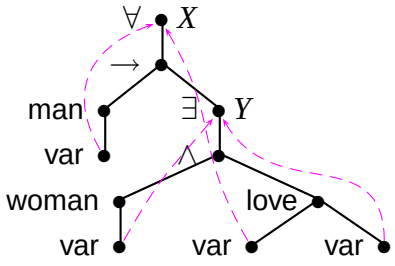
We will give pictures such as (2.5) a formal meaning as *constraint graphs* – shorthand representations of dominance constraint formulas – in Section 2.2. The formula represented by the graph in the example contains atoms such as $X' \triangleleft^* Z$ (dominance) and $X{:}\forall(X_1)$ (labelling: the node label of X is the symbol $\forall$, and its only child is X_1); X, X', etc. are node variables.

This interpretation will be consistent with the intuition of embedding the graph into the described tree. Note that there is an infinite number of trees into which a graph can be embedded: The trees can contain arbitrarily many nodes with arbitrary labels, as long as they also contain the tree fragments mentioned by the graph, and respect the dominance requirements between these fragments. This is a desired feature, because natural language semantics is subject to *reinterpretation*, which means that the semantics of a sentence may contain material that doesn't come directly from the words in that sentence (Egg 2003; Koller et al. 2000). However, we can restrict our attention to *constructive solutions* of the dominance constraint, in which each node in the tree must be denoted by a variable in the description. The trees in (2.2) and (2.4) are the only two constructive solutions of (2.5). We will prove in Chapter 6 that the addition of extra material in solving a constraint is possible, but never necessary for those constraints that actually arise in the context of underspecified semantics.

2.1.2 Lambda structures and binding constraints

One aspect of logical object languages that the trees above don't represent is that terms and formulas can contain variables which are bound e.g. by quantifiers or λ-binders. In order to represent binding, we equip the structure trees from above with *binding functions*, which map variables to binders. The binding function in the example below is drawn as curved, dashed arrows.

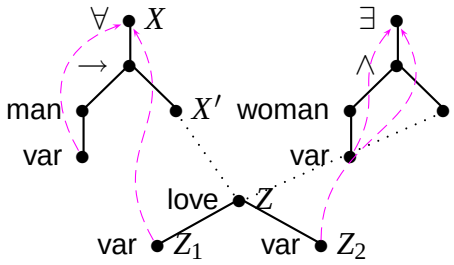
(2.6)



We call the combination of a tree with a binding function a *lambda structure*. Lambda structures speak about variable binding without using variable names. Variable names are not necessary because of the explicit binding function; we will explain in a moment why they are also not sufficient in an underspecification context, and we really need the binding function. A lambda structure represents an object-language formula uniquely, up to the choice of variable names.

In order to obtain underspecified descriptions of lambda structures, we extend dominance constraints with *binding constraints*, as in the following constraint graph.

(2.7)

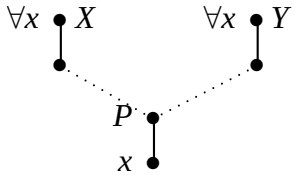


This graph is a straightforward extension of (2.5): We take it to describe all lambda structures into which it can be embedded. When we define binding constraints formally in Section 2.3, each binding arrow in (2.7) will be rendered as an atomic formula, such as $\lambda(Z_1) = X$.

2.1.3 The capturing problem

The reason why we must use explicit binding function is that variable names rely on the relative scope of the quantifiers to assign binders to variables, and this information isn't available in underspecified semantics. Consider the following attempt to specify variable binding, in which the labels $\forall x$ are intended to bind the variable x.

(2.8)



Let's assume that we also have an *inequality constraint* $X \neq Y$, which enforces that X and Y can't be mapped to the same node of an embedding lambda structure. Then we know that the lambda structure will contain two different quantifiers $\forall x$ – but which of these quantifiers actually binds the variable occurrence depends on the relative scope of the quantifiers in the lambda structure. This means that variable names are not sufficient to indicate the binding relations when the structure of the term is not fully known. The problem is a bit similar to capturing in λ-calculus, which occurs when a variable ends up bound by an unintended binder, but arises for different reasons here.

It is possible to circumvent the capturing problem in lambda structures with some additional overhead by naming all bound variables apart. But a non-constructive solution can contain binders that were not mentioned in the constraint, and these binders can capture variables that were intended to be bound by a known quantifier.

Furthermore, dominance constraints can be extended with *parallelism constraints* (Egg et al. 2001), which can force us to copy material while solving the constraint, and to unify variable names. We would need to add more and more bookkeeping mechanisms to keep variables properly named.

On the other hand, explicit binding functions provide a simple, clean, and general solution to the problem. They are flexible enough to represent a range of variable binding properties of different logical formalisms, and can be processed very generally, as we will show in Sections 2.3 and 4.2. Indeed, they are so powerful that one can even lift β-reduction to the level of underspecified descriptions of λ-terms without ever worrying about freeness conditions or variable capturing, even in cases where this would be necessary in the ordinary λ-calculus (Bodirsky et al. 2001a).

2.1.4 Object language independence

In conclusion of this exposition, we will demonstrate that lambda structures can represent not just first-order formulas, but also formulas from other languages, such as higher-order logic and dynamic predicate logic. We will also apply lambda structures to anaphoric binding.

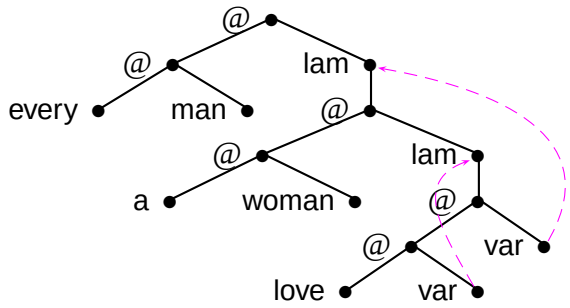
One standard formalism for semantic representations is higher-order logic, because it allows us to systematically build up the meaning representations from the meaning representations of the words (Montague 1974). A higher-order representation of "Every man loves a woman" could look as follows:

(2.9) $\mathsf{every}(\mathsf{man})(\lambda x.\mathsf{a}(\mathsf{woman})(\lambda y.\mathsf{love}(x)(y)))$

Here every and a are abbreviations for the standard semantics of these words – e.g. $\mathsf{every} = \lambda P \lambda Q. \forall x. P(x) \rightarrow Q(x)$.

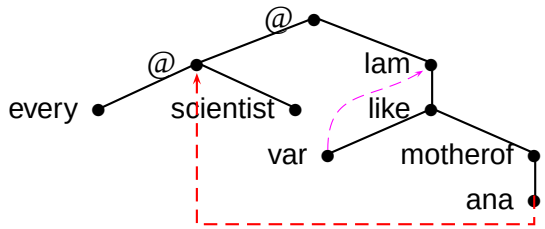
We can represent lambda terms as lambda structures by using the label lam to represent λ-abstraction, the label var for variables, the binary label @ for applications, and nullary labels for the constants. We can extend this to higher-order logic by adding labels for the logical connectives, as we did for predicate logic. The formula (2.9) looks as follows under this representation:

(2.10)



The Constraint Language for Lambda Structures (CLLS, Egg et al. 2001) uses lambda structures that represent higher-order formulas as in (2.10), but are additionally equipped with a second binding function for *anaphoric binding*. This binding function uses the label ana for anaphoric "variables", and connects them to their antecedents. For instance, the following lambda structure is a semantic representation for the sentence "Every scientist likes his mother":

(2.11)



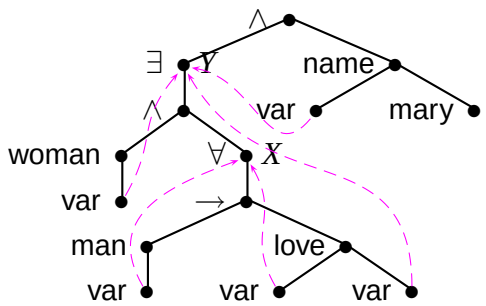
In this graph, the binding function for λ-binding is indicated by the curved arrows, and the binding function for anaphoric binding is indicated by the angular arrow.

A semantic representation formalism we will look at more closely in Chapter 7 is *Dynamic Predicate Logic* (DPL, Groenendijk and Stokhof 1991). DPL uses the same syntax as ordinary first-order logic, but gives it a different semantics that changes the way variable binding works. For instance, if we continued the reading (2.3) of our running example with "... Her name is Mary", we could get a semantic representation as follows:

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

The variable y in the right-hand conjunct is considered bound by the binder $\exists y$ in the left-hand conjunct in DPL. Thus the anaphor "her" is resolved to the antecedent "a woman". We can represent this formula as a lambda structure in a straightforward way:

(2.13)



This lambda structure looks odd at first sight because its binding function links a variable to a binder that doesn't dominate it. But this reflects the fact that a variable can be bound in DPL by a quantifier even if it isn't in the quantifier's syntactic scope.

2.2 Syntax and Semantics of Dominance Constraints

Now we will define the syntax and semantics of dominance constraints properly. We will first define tree structures, and then dominance constraints as a language to speak about them.

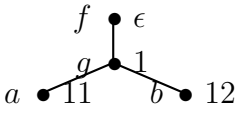
2.2.1 Tree structures

Dominance constraints are interpreted over *finite constructor trees*. We assume a ranked signature $\Sigma = \{f|_2, a|_0, \ldots\}$ of symbols f with arities $\mathsf{ar}(f)$. We assume throughout that Σ contains at least one symbol with arity at least 2, and at least one symbol with arity 0.

Definition 2.1. A finite constructor tree τ is a 4-tuple (V, E, L_V, L_E) where (V, E) is an ordered finite tree, $L_V : V \to \Sigma$ is a *labelling function*, and $L_E : E \to \mathbb{N}$ is an *edge ordering*, such that for each node $v \in V$ and each $1 \leq i \leq \mathsf{ar}(L_V(v))$, there is exactly one edge $(v, w) \in E$ with $L_E(e) = i$.

We can see finite constructor trees as ground terms over Σ. Alternatively, we will sometimes identify the nodes of a tree with their *addresses*, which are words from $\mathbb{N}^*$ (i.e., finite strings of natural numbers). The root has the address ϵ, and the i-th child of the node with address π is πi. We will switch between these three perspectives whenever it is convenient.

Consider, by way of example, the tree shown to the right. This is a finite constructor tree over the signature $\{f|_1, g|_2, a|_0, b|_0\}$. It has four nodes with the addresses ϵ, 1, 11, and 12, and it corresponds to the ground term $f(g(a, b))$.



We will be interested in talking about a variety of different node relations in a tree. The most basic relation is the *labelling relation* $u{:}f(v_1, \ldots, v_n)$ (see the picture). An $(n+1)$-tuple of nodes is in this relation if $v_1, \ldots, v_n$ are the children of u, in this order, and u has the label f. Note that we must have $n = \mathsf{ar}(f)$ because the tree structure is built from a finite constructor tree.

In addition, we define four binary node relations that we take to be primitive: *equality* ($u = v$), *strict dominance* ($u{\lhd^+}v$), *strict inverse dominance* ($u{\rhd^+}v$), and *disjointness* ($u \perp v$). As Fig. 2.1 shows, every pair of nodes in the tree is in exactly one of these four relations. Two nodes u, v are in the relation $u{\lhd^+}v$ and in the relation $v{\rhd^+}u$ if there is a path of length one or more from u to v. Two nodes are disjoint if neither is reachable from the other. In this case, there is a lowest node w which dominates both u and v. This node is called the *branching point* of u and v. We write "$u \perp v$ at w" in this case, i.e. we also consider the three-place relation that relates disjoint nodes and their branching points. If $r \in \{=, \lhd^+, \rhd^+, \perp\}$, we write r^{-1} for the *inverse relation* of r. $=$ and $\perp$ are symmetrical relations and hence their own inverse, whereas $\lhd^+$ and $\rhd^+$ are each other's inverse relations.

Finally, we will use some defined relations between nodes:

dominance	$u{\lhd^*}v$	iff $u{\lhd^+}v$ or $u = v$.
inequality	$u \neq v$	iff not $u = v$
disjointness with sets	$u \perp v$ at W	iff there is a $w \in W$ such that $u \perp v$ at w
non-intervention	$\neg(u{\lhd^*}v{\lhd^*}w)$	iff not both $u{\lhd^*}v$ and $v{\lhd^*}w$.

Every finite constructor tree $\tau = (V, E, L_V, L_E)$ induces a unique first-order model structure $\mathcal{M}_\tau$ with universe V which interprets predicate symbols $:f$, $\lhd^+$, etc. as the relations $:f$, $\lhd^+$, etc. We use the same symbols for the predicates and the relations, but there will be no danger of confusion. These structures are called *tree structures*. Conversely, we can reconstruct the tree from the tree structure because the labelling relation fixes the tree completely. We will freely switch between trees and tree structures below.

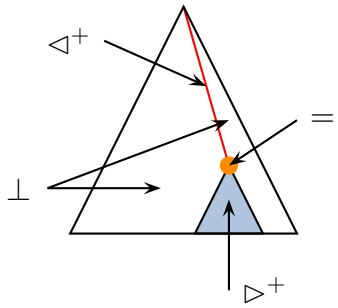


Figure 2.1: The four primitive binary relations. Each relation symbol points to the nodes that are in this relation to the marked node.

2.2.2 Syntax and semantics

A *dominance constraint language* is a first-order language which is interpreted over tree structures. Its formulas, which we call *dominance constraints*, are conjunctions of atoms, i.e.

$$\varphi ::= \text{Atom} \mid \varphi \wedge \varphi'.$$

Atoms are applications of predicate symbols to *variables*. The variables are taken from an infinite set $\mathsf{Vars} = \{X, X_1, Y, Z, \ldots\}$ and denote nodes in a tree structure. The atoms which are allowed in various dominance constraint languages are displayed in Fig. 2.2. They can be combined freely; so for example, the language $\mathcal{D}$ only allows atoms of the form $X{:}f(X_1, \ldots, X_n)$ and $X \triangleleft^* Y$, and the language $\mathcal{DI}$ allows these two types of atoms and also atoms of the form $X \neq Y$. The symbol $\mathcal{A}$ in the table stands for a set of variables, and the symbol R designates a set of primitive binary node relations, i.e. $R \subseteq \{=, \triangleleft^+, \triangleright^+, \perp\}$.

If φ is a dominance constraint (from any of the languages), we write $\mathsf{Var}(\varphi)$ for the set of variables occurring in φ. A *variable assignment* for the constraint φ into the tree structure $\mathcal{M}$ of the tree $\tau = (V, E, L_V, L_E)$ is a partial function $\alpha : \mathsf{Var}(\varphi) \rightsquigarrow V$.

$\mathcal{D}$	$X{:}f(X_1, \ldots, X_n)$,	$X \triangleleft^* Y$
$\mathcal{S}$	$X{:}f(X_1, \ldots, X_n)$,	$X\ R\ Y$
$\mathcal{I}$	$X \neq Y$	
$\mathcal{B}$	$X \perp Y$ at $\mathcal{A}$	
$\mathcal{N}$	$\neg(X \triangleleft^* Y \triangleleft^* Z)$	

Figure 2.2: Atoms allowed in different dominance constraint languages.

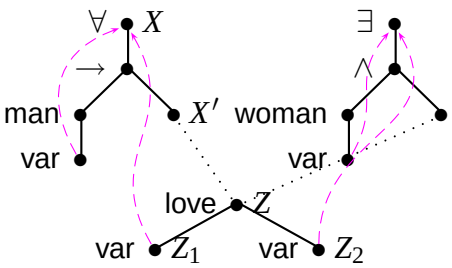
The constraint φ is *satisfied* by the tree structure $\mathcal{M}$ and the variable assignment α iff $\mathcal{M}$ makes each atom of φ true. This is defined as usual for most atoms; $X\ R\ Y$ is made true by $\mathcal{M}$ if there is a $r \in R$ such that $X\ r\ Y$ is true. We write $\mathcal{M}, \alpha \models \varphi$ in this case, and call $(\mathcal{M}, \alpha)$ a *solution* and $\mathcal{M}$ a *model* of φ. If every solution of φ is also a solution of ψ, φ *entails* ψ; we write $\varphi \models \psi$. Note that this means that every variable in ψ must occur in φ. A solution $(\mathcal{M}, \alpha)$ of φ is called *constructive* iff every node in $\mathcal{M}$ is denoted by a variable X with some $X{:}f(X_1, \ldots, X_n)$ in φ.

We will also use the following abbreviations:

$$\begin{array}{lcl} X\ \neg R\ Y & \equiv & X\ S\ Y, \text{ where } S = \{=, \lhd^+, \rhd^+, \perp\} - R \\ X\ R^{-1}\ Y & \equiv & X\{r^{-1} \mid r \in R\}Y \end{array}$$

In addition, we can embed the languages $\mathcal{D}$ and $\mathcal{I}$ into the language $\mathcal{S}$ by expressing $X \lhd^* Y$ as $X\{=, \lhd^+\}Y$ and $X \neq Y$ as $X\{\lhd^+, \rhd^+, \perp\}Y$.

$\mathcal{DI}$ is the language we will use in modelling underspecified semantics. The other languages are sometimes useful in calculi and proofs, because their atoms allow us to specify structural relations between two nodes that $\mathcal{DI}$ doesn't capture. Similarly, it is straightforward to allow more logical connectives than just conjunction, e.g. to consider the propositional or first-order languages over the permitted atoms. We will only do this in very specific circumstances; it is not necessary for the application, but can be convenient for meta-reasoning. All the above languages except $\mathcal{B}$ can be translated to the propositional language over $\mathcal{D}$. For instance, $X = Y$ can be written as $X \lhd^* Y \wedge Y \lhd^* X$, and $X \perp Y$ as $(\neg X \lhd^* Y) \wedge (\neg Y \lhd^* X)$.



The constraint graphs from Section 2.1 can be read as constraints in the "core language" $\mathcal{DI}$. Their nodes stand for variables of the dominance constraint. Node labels and solid edges in the graph represent labelling atoms, and dotted edges represent dominance atoms. In addition, we take the graph to implicitly represent an inequality atom for each pair of labelled nodes. For example, the graph (2.7), repeated here to the right, encodes among others the labelling atom $Z{:}\mathsf{love}(Z_1, Z_2)$, the dominance atom $X' \lhd^* Z$, and the inequality atom $X \neq Z$. The curved arrows indicate binding atoms, which are defined in the next section.

2.3 Binding Constraints

We will now define the binding functions introduced in Section 2.1.2; a tree structure together with one or more binding functions will be called a *lambda structure*. Next, we will extend the dominance constraint languages with *binding constraints* to talk about them. We will capture the particular requirements that a specific object language imposes on variable binding in a *binding specification*, which gives us a very general framework to talk about variable binding.

Definition 2.2. A *binding function* for a finite constructor tree $\tau = (V, E, L_V, L_E)$ is a partial function $\lambda : V \rightsquigarrow V$. A tuple $(\tau, \lambda_1, \ldots, \lambda_n)$ of a tree and some binding functions induces a *lambda structure* $\mathcal{L}_{\tau,\lambda_1,\ldots,\lambda_n}$ that extends the tree structure $\mathcal{M}_\tau$ with interpretations for the *binding atom* $\lambda_i(X) = Y$ for each $1 \leq i \leq n$. The interpretation of the binding atom is the binary relation λ_i.

If $\mathcal{A}$ is a language of dominance constraints and $\lambda_1, \ldots, \lambda_n$ are the names of binding atoms, we can define the language $\mathcal{A}\Lambda_1 \ldots \Lambda_n$ of *dominance and binding constraints*. The language consists of conjunctions of atoms from $\mathcal{A}$ and binding atoms $\lambda_i(X) = Y$. Satisfaction, entailment, etc. are defined in analogy to dominance constraints, with lambda structures instead of tree structures. Note that a language can contain binding atoms for multiple binding functions at once; this is useful e.g. in the context of CLLS, as we saw in Section 2.1.4.

So far, binding functions simply extend the tree with edges of new types. In order to make them useful for applications, they should capture the binding behaviour of a given object language. The two key features that characterise variable binding in an object language are (a) the labels of the possible binders, and (b) where a variable may occur in a term in relation to its binder. Predicate logic and lambda calculus differ on the dimension (a): Predicate logic allows the binders $\forall$ and $\exists$, whereas lambda calculus only has the binder λ. On the other hand, static and dynamic predicate logic differ on the dimension (b): Static predicate logic requires a variable to be in the syntactic scope of its binder, whereas we have seen in (2.12) that this need not be the case in DPL.

The following definition captures these two dimensions formally in the concept of a *binding specification* for an object language. We can then model well-formed formulas of an object language as lambda structures that are *admissible* with respect to this specification.

Definition 2.3. A *binding specification* for the signature Σ is a pair $\Lambda = (B, S)$ of a set $B \subseteq \Sigma$ of *binders* and a *scope specification* $S : \tau \to (\mathbb{N}^*)^2$ that maps each tree τ to the set of binder-variable pairs that can be related by a binding relation. The *variable* of Λ is a special symbol $\mathsf{var}_\Lambda|_0 \in \Sigma$.

	binders	scope
Λ_λ	$\{\mathsf{lam}\}$	$\lhd^*$
Λ_{fol}	$\{\exists, \forall\}$	$\lhd^*$
Λ_{DPL}	$\{\exists, \forall\}$	see Chapter 7
Λ_{ana}	Σ	$\neq$

Figure 2.3: Some binding specifications.

If Λ is a binding specification, the binding function λ is called Λ-*admissible* for a tree $\tau = (V, E, L_V, L_E)$ iff for all u, v such that $\lambda(u) = v$, we have $L_V(v) \in B$ and $(v, u) \in S(\tau)$. A lambda structure $\mathcal{L}_{\tau,\lambda_1,\ldots,\lambda_n}$ is called $\Lambda_1, \ldots, \Lambda_n$-*admissible* iff each λ_i is a Λ_i-admissible binding function for τ. A constraint φ is called $\Lambda_1 \ldots \Lambda_n$-*satisfiable* iff it has a $\Lambda_1 \ldots \Lambda_n$-admissible solution.

Some important binding specifications are shown in Fig. 2.3. Λ_λ encodes the binding behaviour in lambda terms: The binder has label λ and must outscope the variables it binds; we use the node label lam to mark binders, as in Section 2.1.4. Λ_{fol} encodes first-order predicate logic, and Λ_{DPL} encodes Dynamic Predicate Logic. Λ_{DPL} is defined precisely in Chapter 7. Λ_{ana} is the binding specification for anaphoric binding from CLLS (Egg et al. 2001): CLLS is an extension of $\mathcal{DI}\Lambda_\lambda\Lambda_{ana}$ in which the variables of Λ_{ana} are called ana, and the binding atom for this specification is called $\mathsf{ante}(X) = Y$.

Going back to the examples from Section 2.1, we see that the binding function in (2.6) is Λ_{fol}-admissible, as all binders dominate the bound variables and are labelled with either $\exists$ or $\forall$. On the other hand, the graph in (2.13) is not Λ_{fol}-admissible, as it contains a variable whose binder doesn't dominate it. It is still Λ_{DPL}-admissible, as we shall see in Chapter 7.

2.4 Related formalisms

Various languages of dominance constraints (without binding) have been used for a range of different applications in computational linguistics, such as incremental parsing (Marcus et al. 1983), grammar formalisms (Vijay-Shanker 1992; Rambow et al. 1995; Duchier and Thater 1999; Perrier 2000), and discourse (Gardent and Webber 1998). The two big differences between the different languages are the

logical connectives (beyond conjunction) that may be allowed, and whether immediate dominance and linear precedence of sisters are directly connected to labelling (as in our definition), or whether they can be specified separately. It is not known whether a language that allows such a separation is properly more expressive than $\mathcal{D}$, but the propositional languages are known to be equivalent (Koller 1999).

If we extend the language $\mathcal{DI}\Lambda_\lambda\Lambda_{ana}$ with *parallelism constraints*, we obtain the *Constraint Language for Lambda Structures* (CLLS, Egg et al. 1998; Egg et al. 2001). This was the original context in which dominance constraints were developed. Parallelism constraints relate entire subtrees or subcontexts of a lambda structure to each other, and state that they are structurally equal. Their primary application is to model ellipsis phenomena and their interaction with scope and anaphora (Egg et al. 2001; Erk and Koller 2001). Parallelism constraints and *context unification* (Comon 1992; Lévy 1996; Niehren et al. 1997a; Schmidt-Schauß and Schulz 1998) are equivalent (Niehren and Koller 2001; Erk et al. 2003; Niehren and Villaret 2003), and decidability of context unification is an open problem. However, the fragment of *well-nested* parallelism constraints seems to be sufficient for most cases of ellipsis, and has NP-complete satisfiability (Erk and Niehren 2003). (Non-well-nested) parallelism constraints have also been used to model beta reduction of lambda terms on the underspecified descriptions of these terms (Bodirsky et al. 2001b; Bodirsky et al. 2001a).

Dominance constraints are not the only formalism that has been used in scope underspecification. Historically the first underspecification formalism was QLF (Alshawi and Crouch 1992). Underspecified descriptions in QLF looked like augmented object-language terms. Later formalisms switched to a perspective that separated the object language from the description language more rigorously, and mostly use diagrams similar to (2.7) as underspecified descriptions; of course, the interpretation of what exactly the graph means differs. Such graphs first appeared in Underspecified DRT (Reyle 1993). Hole Semantics (Bos 1996) and Minimal Recursion Semantics (Copestake et al. 1999) are two other formalisms of this type, which are used in large-scale grammars (Copestake and Flickinger 2000; Kallmeyer and Joshi 2003). We prove relevant fragments of Hole Semantics and dominance constraints equivalent in Chapter 6, and similar results are available for MRS (Niehren and Thater 2003; Fuchss et al. 2004).

2.5 Summary

In this chapter, we have introduced languages of dominance constraints as formalisms for scope underspecification. Dominance constraint languages talk about

trees in terms of conjunctions of atoms. While the language $\mathcal{DI}$ is sufficient for the purpose of representing underspecified semantic descriptions, we will sometimes switch to more expressive languages in order to state calculi or proofs.

We extended trees with binding functions and dominance constraints with binding constraints to talk about them. Binding constraints can be parameterised by a binding specification, which represents the particular way in which variable binding works in the object language. We represented variable binding by means of an explicit binding function, rather than variable names, because variable names aren't sufficient to uniquely map variables to binders in an underspecification setting.

A key distinction that came up in this chapter was between the *object language*, in which the semantic representations of a sentence were written, and the language of dominance (and binding) constraints we used to talk about them. We consider formulas of the object language as lambda structures. The nodes of such a lambda structure may be decorated with labels that represent logical connectives from the object language; but from the perspective of a dominance constraint, these are node labels just like any other symbol, and don't carry any particular meaning. Because some object languages use the same logical symbols as the languages of dominance constraints (e.g., $\wedge$ for conjunction), we will sometimes mark the object language connectives in order to avoid confusion.

We conclude this chapter with an index of important concepts and the symbols we use to denote them.

- *Dominance constraint languages* are first-order languages. Some dominance constraint languages are $\mathcal{D}$, $\mathcal{DI}$, and $\mathcal{S}$. We sometimes use the term "dominance constraints" to refer to all dominance constraint languages collectively.

- A *dominance constraint* is a formula of a dominance constraint language, i.e. a conjunction of atoms. We use lowercase Greek letters φ, ψ, etc. to denote dominance constraints.

- A *dominance atom* $X \triangleleft^* Y$ is one of the atoms that can be used in a dominance constraint. Some other atoms are labelling and inequality atoms.

- We use uppercase letters X, X_1, Y to write *variables* in a dominance constraint. These are distinct from the variables used in the object language, which are represented by explicit binding functions, rather than with variable names, in a lambda structure.

- We write σ, τ for *finite constructor trees*, and switch freely between the perspectives of seeing a tree as a set of nodes and edges with a labelling function,

as a set of words over $\mathbb{N}$ with a labelling function, and as a ground term. We write u, v for nodes, and $\mathcal{M}$ or $\mathcal{M}_\tau$ for a *tree structure*, i.e. the first-order structure that interprets the node relations over a tree.

- A *binding function* λ over a tree can be *admissible* with respect to a *binding specification* Λ. We write $\mathcal{L}$ or $\mathcal{L}_{\tau,\lambda}$ for the *lambda structure* that interprets the relations over the tree and the binding function.

Chapter 3

Computing Dominance Constraints for English

We will now show how dominance constraints can be used in underspecified semantics. So far, we have defined their syntax and semantics, and we have seen a handful of examples, but we haven't said yet how we could construct such constraints for actual sentences. In this chapter, we will demonstrate how underspecified semantic representations for English sentences in the language $\mathcal{DI}\Lambda_\lambda$ can be systematically computed: We will define a grammar for a small (but nontrivial) fragment of English, along with a *syntax-semantics interface* that shows how underspecified semantic representations are computed from a parse tree. That is, one purpose of this chapter is to link the formalism to the actual application of underspecified semantic processing for natural language.

A second purpose of this chapter will be to demonstrate that underspecification can also contribute to keeping the syntax-semantics interface simple – in addition to its computational and cognitive advantages mentioned in the introduction. Non-underspecified approaches to semantics construction often either make questionable linguistic assumptions, or they require a rather complex semantics construction process, in order to accommodate semantic ambiguity. By contrast, underspecification-based approaches to semantics construction can get by with simple interfaces because they can map each syntactic reading into a single underspecified semantic representation and delegate the enumeration of semantic readings into a later processing step.

Finally, as we will claim later on (in Chapters 5 and 6), all dominance constraints that are actually used in underspecification belong to certain fragments of $\mathcal{DI}$. We will substantiate these claims by proving them correct for the grammar defined here.

The structure of this chapter is as follows. We will first show how semantic representations can be computed compositionally from syntactic representations (Section 3.1). This syntax-semantics interface will be restricted to a tiny grammar of English that, among many other things, doesn't analyse sentences with scope ambiguities. We will then review some attempts at extending this interface with mechanisms for dealing with scope (Section 3.2). This discussion will motivate the syntax-semantics interface for the construction of dominance constraints for the same tiny grammar we used above (Section 3.3). Finally, we will extend this interface to a larger grammar (Section 3.4) and go through an example to illustrate how it works (Section 3.5). The grammar presented in Section 3.4 is a variant of the grammars in (Egg et al. 2001; Koller et al. 2003).

3.1 Classical Semantics Construction

A classical assumption of research in natural language semantics is that the core meaning of a sentence is captured by its *truth conditions*. The idea is that a sentence makes a statement about the world, and is either true or false in any given situation. Such truth conditions can be expressed by a formula of some logical language, such as predicate logic. This is why logical formulas are often chosen as semantic representations of sentences.

Most semanticists aim at computing meaning representations *compositionally*. This means that every syntactic constituent of the sentence is assigned a semantic representation, and the semantic representations of larger constituents are computed by systematically combining the semantic representations of their parts. Compositional syntax-semantics interfaces are formally clean because they compute semantic representations simply by structural induction over the syntactic analysis, and they reflect the linguistic intuition that a syntactic constituent has an intrinsic meaning that is independent from the context in which it is used.

However, even for quite simple sentences, it is not obvious at first sight how semantic representations could be computed compositionally. Consider the following two sentences with their (first-order) meaning representations:

(3.1) *Peter* sleeps.

(3.2) $\mathsf{sleep}(\mathsf{peter})$

(3.3) *Every man* sleeps.

(3.4) $\forall x.(\mathsf{man}(x) \rightarrow \mathsf{sleep}(x))$

The two sentences are completely parallel, in that "Peter" and "every man" are the same type of syntactic constituent (a noun phrase) and appear in the same grammatical role (subject). It seems easy to compute the semantic representation (3.2) for (3.1): simply apply the meaning of the verb to the meaning of the noun. But such a simple composition algorithm doesn't work for (3.3), and indeed it is not even obvious what the semantic representation for the noun phrase "every man" should be. It is certainly not just a constant like the semantic representation of the (syntactically analogous) noun phrase "Peter".

Montague (1974) essentially started modern semantics when he showed how semantic representations in intensional logic (an extension of higher-order predicate logic; we will simply use higher-order logic here, but the interface could be straightforwardly extended to intensional logic) could be computed compositionally for sentences like (3.1) and (3.3). He analysed all noun phrases as lambda terms of type $\langle\langle e, t\rangle, t\rangle$ (i.e., sets of properties). The semantic representation of "Peter" thus became $\lambda P.P(\mathsf{peter})$ ("the set of all properties that Peter has"), and the semantic representation of "every man" became $\lambda P.\forall x.\mathsf{man}(x) \rightarrow P(x)$ ("the set of all properties that every man has"); that is, he identified an individual with the set of its properties. Then he applied the noun phrase semantics to the verb semantics, arriving at the following representations for the two examples:

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

(3.6) $(\lambda P.\forall x.\mathsf{man}(x) \rightarrow P(x))(\mathsf{sleep})$

The simpler first-order representations above can be obtained by β-reducing these higher-order formulas; the difference is irrelevant to semanticists because the reduced and unreduced formulas have the same truth conditions. Note, however, that there are sentences whose semantics can't be expressed in first-order logic, e.g. sentences like "*Most* men sleep", which use quantifiers that are inherently higher-order (Barwise and Cooper 1981).

We will now define a system that derives such semantic representations for sentences from a tiny fragment of English. The fragment is defined by the following context-free grammar, whose start symbol is S, for "sentence".

$\mathrm{S} \rightarrow \mathrm{NP\ VP}$	$\mathrm{NP} \rightarrow \mathrm{Det\ N}$
$\mathrm{VP} \rightarrow \mathrm{IV}$	$\mathrm{NP} \rightarrow \mathrm{PN}$
$\alpha \rightarrow W \quad \text{if } (W, \alpha, M) \in \mathit{Lex}$	

The grammar says that sentences can be built by combining a noun phrase and a verb phrase. In this highly simplified grammar, verb phrases can only be intransi-

$$\frac{(W, \alpha, M) \in Lex}{\alpha(W) \hookrightarrow M} \qquad \frac{np \hookrightarrow M \qquad vp \hookrightarrow N}{s(np, vp) \hookrightarrow MN}$$

$$\frac{iv \hookrightarrow M}{vp(iv) \hookrightarrow M} \qquad \frac{det \hookrightarrow M \qquad n \hookrightarrow N}{np(det, n) \hookrightarrow MN}$$

$$\frac{pn \hookrightarrow M}{np(pn) \hookrightarrow M}$$

Figure 3.1: Montague-style semantics construction rules.

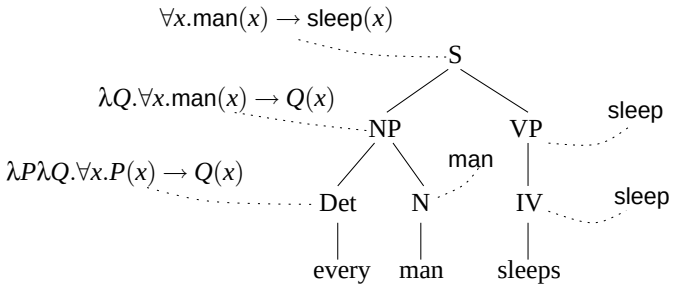


Figure 3.2: Semantics construction for the sentence "Every man sleeps."

tive verbs (such as "sleeps"). Noun phrases can either be proper names (such as "Peter"), or they can be composed from a determiner and a noun (such as "every man"). We assume a lexicon Lex, which is a set containing triples of a word W, a syntactic nonterminal symbol α, and a semantic representation M for the word. For instance, the lexical entry for the word "Peter" would be $(Peter, \mathrm{N}, \lambda P.P(\mathsf{peter}))$.

The grammar assigns a unique parse tree to each sentence that it considers grammatical. Now we can compute a semantic representation for such a sentence by applying inference rules as in Fig. 3.1 to the parse tree in a process of structural induction. The inference rules derive statements of the form $t \hookrightarrow M$, where t is a subtree of the parse tree, and M is a lambda term representing the semantics of this subtree. The preterminal nodes of the parse tree, i.e. those nodes whose only child is a node labelled with a word, derive their semantics directly from the lexicon by the top left rule. For the inner nodes of the tree, there is one semantic composition rule for each production rule of the grammar; for example, the expression $s(np, vp)$ stands for a subtree whose root is labelled with S, and whose subtrees have root labels NP and VP. These rules combine the semantic representations of the subtrees by functional application. See Fig. 3.2 for an example; some semantic representations are shown β-reduced in order to improve readability.

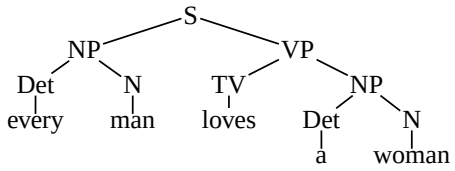


Figure 3.3: A parse tree for the sentence "Every man loves a woman" using a straightforward extension of the grammar in Section 3.1.

3.2 Scope Ambiguity

The syntax-semantics interface in Fig. 3.1 assumes that every syntactic analysis can be uniquely translated into a single semantic analysis. This assumption is violated by semantic ambiguities, such as scope ambiguities: There is no one-to-one correspondence between syntactic and semantic analyses, but a one-to-many correspondence. (Actually, it is a many-to-many correspondence, as the same semantic representation can be verbalised with different sentences; but this is not relevant here.)

We can of course extend the grammar with a rule for transitive verbs, and thus make it possible to analyse sentences such as *Every man loves a woman*, which contain scope ambiguities (Fig. 3.3). But the interface would still be systematically incapable of deriving both readings of a semantically ambiguous sentence. The problem is that our algorithm for compositional semantics construction traverses the syntax tree bottom-up and applies terms to other terms. This means that because the verb is first combined with the direct object ("a woman") syntactically, we must apply the direct object to the verb first semantically – i.e., we can only get the semantic reading in which "a woman" gets narrow scope. The other reading (in which "a woman" takes scope over "every man") seems to require us to reverse the order in which functors are applied to the arguments, compared to the syntactic analysis.

3.2.1 Quantifying In

Montague (1974) himself handled this problem by recasting the semantic ambiguity as a syntactic ambiguity through the operation of *quantifying-in*. We can reconstruct his analysis in our framework by adding lexical entries of the form (t_i, NP, x_i), which represent *traces* (empty strings) of syntactic category NP. (Montague's original analysis involves the ad hoc deletion of parts of the sentence; traces achieve the same effect more cleanly.) The semantics of the trace t_i is the variable x_i. Using these traces, the grammar can first analyse the sentence "t_1 loves t_2" as having the

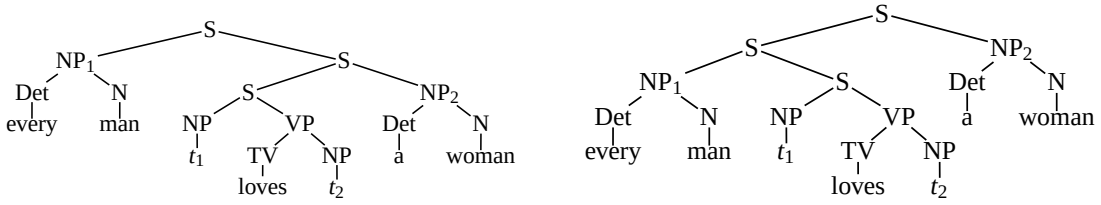


Figure 3.4: The two parse trees for "Every man loves a woman" that can be derived using the quantifying-in rule.

semantics $\mathsf{love}(x_2)(x_1)$ with the standard composition rules from above. Then this sentence is combined with the two noun phrases "every man" and "a woman" using rules S $\rightarrow$ NP$_i$ S and S $\rightarrow$ S NP$_i$; the two parse trees that can be derived using this rule are shown in Fig. 3.4. The semantic composition rules corresponding to these grammar rules are as follows:

$$\frac{s \hookrightarrow M \qquad np_i \hookrightarrow N}{s(np_i, s) \hookrightarrow N(\lambda x_i.M)} \qquad \frac{s \hookrightarrow M \qquad np_i \hookrightarrow N}{s(s, np_i) \hookrightarrow N(\lambda x_i.M)}$$

That is, we can combine the nuclear sentence (containing traces) with the two quantifiers in any order, which gives us the desired two readings:

(a) $\mathsf{every}(\mathsf{man})(\lambda x_1.\mathsf{a}(\mathsf{woman})(\lambda x_2.\mathsf{love}(x_2)(x_1)))$

(b) $\mathsf{a}(\mathsf{woman})(\lambda x_2.\mathsf{every}(\mathsf{man})(\lambda x_1.\mathsf{love}(x_2)(x_1)))$

Note that we must assume additional mechanisms for making sure that an NP$_i$ can only be combined with a sentence that still contains an unresolved trace t_i. This already points to a fundamental problem that comes with the relaxed semantic composition operation: It is no longer trivial to keep track of variable binding. Variable binding was no problem in Section 3.1, because binders and bound variables were always introduced together, in the lexicon. Now the lifecycle of many variables consists of two steps. A variable can be introduced as a free variable x_i by the lexicon entry of the trace t_i, and at some later point in the semantics construction process, its binder λx_i is added by a semantic composition rule. Strictly speaking, Montague's approach employs a controlled form of variable capturing by making a free variable bound.

While Montague's solution works (and was a huge step forward at the time), it solves the problem of the 1:n-correspondence between syntax and semantics by artificially making the syntax more ambiguous, in order to obtain an n:n-correspondence that can again be computed by a compositionally defined function.

This is computationally problematic: Such an analysis increases the number of syntactic readings by an exponential factor, which slows parsers down to an unacceptable degree. And while scope ambiguities *could* be consistently analysed as syntactic ambiguities (May 1985), there certainly doesn't seem to be a compelling reason not to analyse them as purely semantic ambiguities if we can come up with a clean formal framework for it.

3.2.2 Cooper storage

One attempt to do this is *Cooper storage* (Cooper 1983), which solves the 1:n problem by defining a nondeterministic algorithm that computes all n different semantic readings from a single syntactic analyses (as in Fig. 3.3). Cooper storage builds upon Montague's idea of separating an intermediate semantic representation from the quantifiers that still need to be applied to it. In Montague's approach, this separation takes place implicitly via the quantifying-in rule.

In Cooper storage, each node in the syntax tree is associated explicitly with a pair consisting of the semantic representation computed so far and a *quantifier store* Δ containing the unapplied quantifiers. Whenever we encounter a noun phrase in our bottom-up traversal of the syntax tree, we choose nondeterministically whether the quantifier semantics is applied immediately, or whether it is added to the store. Then whenever we reach a sentence node, we can nondeterministically pick any number of quantifiers from the store and apply them to the sentence semantics, in any order. If we change the above syntax-semantics interface from above to derive triples $t \hookrightarrow M \mid \Delta$, we can write these two operations as follows:

$$\frac{np \hookrightarrow M \mid \Delta}{np \hookrightarrow \lambda P.P(x_i) \mid \{\langle Q\rangle_i\} \cup \Delta} \; i \text{ fresh} \qquad \frac{s \hookrightarrow M \mid \{\langle Q\rangle_i\} \uplus \Delta}{s \hookrightarrow Q(\lambda x_i.M) \mid \Delta}$$

Note that unlike any of the rules above, these two rules allow us to derive multiple triples for the same node (with label S or NP). This means that the rule application now becomes nondeterministic, as we have a choice at each NP and S node whether and how often we want to apply the rules. The result is that we get many different triples for the root of a sentence, one for each semantic reading.

In our running example "Every man loves a woman", if we choose to always add the quantifiers to the store, we will compute the following triple for the root of the tree in Fig. 3.3:

$$\epsilon \hookrightarrow \mathsf{love}(x_2)(x_1) \mid \{\langle \mathsf{every}(\mathsf{man})\rangle_1, \langle \mathsf{a}(\mathsf{woman})\rangle_2\}$$

That is, the core semantics of the sentence is $\mathsf{love}(x_2)(x_1)$, the same as for "t_1 loves t_2" in Montague's analysis; and the quantifier store contains two quantifiers. These can now be retrieved from the store and applied to the core semantics in two different orders, obtaining the two correct readings of the sentence.

The original Cooper storage has problems with embedded noun phrases that cause it to predict too many semantic readings in some cases. The extra readings contain unbound variables, and come from a lack of structural constraints that are put on the quantifier retrieval process. This problem is fixed by Keller (1988), who introduces more structure into the quantifier store. But another fundamental problem that remains is that the syntax-semantics interface is made more complex, in order to remain compositional. In addition, the nondeterminism can't be eliminated: Because we must choose for each quantifier whether to apply it in place or to add it to the store, we can't compute a single "underspecified" representation for the entire sentence that collects all quantifiers and then allows us to apply them by need. So although approaches like Keller Storage correctly compute the semantic readings of a scope ambiguity in which the scope-bearing elements are noun phrases, it is still subject to a combinatorial explosion that leads to inacceptable runtimes in practice.

3.3 Semantics Construction for Dominance Constraints

The requirement that we should be able to extract a single compact description of the semantic readings of a sentence – the key requirement of underspecification – was first met by Hobbs and Shieber (1987), Alshawi (1992), and Alshawi and Crouch (1992). We could say that these are proto-underspecification approaches, in that they do allow us to complete semantics construction before we actually have to enumerate semantic readings, but they don't yet make a clear, explicit distinction between the level of underspecified descriptions and the level of proper semantic representations. This distinction was first clarified in UDRT (Reyle 1993).

Since then, underspecification has become the standard approach to dealing with scope ambiguity in large grammars (Copestake and Flickinger 2000; Butt et al. 2002; Kallmeyer and Joshi 2003), primarily because it allows us to keep the syntax-semantics interface completely functional, compositional, deterministic, and free of the burden of ambiguity management. The task enumerating readings from the underspecified description is delegated to a later processing step, and is performed only by need.

$$\frac{(W, \alpha, \varphi, X) \in \mathit{Lex}}{\alpha \hookrightarrow \varphi' \mid X'} \; X', \varphi' \text{ fresh} \qquad \frac{np \hookrightarrow \varphi \mid X \qquad vp \hookrightarrow \varphi' \mid X'}{s(np, vp) \hookrightarrow \varphi \wedge \varphi' \wedge Y{:}@(X, X') \mid Y}$$

$$\frac{iv \hookrightarrow \varphi \mid X}{vp(iv) \hookrightarrow \varphi \mid X} \qquad \frac{det \hookrightarrow \varphi \mid X \qquad n \hookrightarrow \varphi' \mid X'}{np(det, n) \hookrightarrow \varphi \wedge \varphi' \wedge Y{:}@(X, X') \mid Y}$$

$$\frac{pn \hookrightarrow \varphi \mid X}{np(pn) \hookrightarrow \varphi \mid X}$$

where $Y \notin \mathsf{Var}(\varphi) \cup \mathsf{Var}(\varphi')$ and $\mathsf{Var}(\varphi) \cap \mathsf{Var}(\varphi') = \emptyset$.

Figure 3.5: Semantics construction rules for dominance constraints.

We will now first present in detail a syntax-semantics interface that computes dominance constraints for the tiny grammar fragment introduced above. As there are no scope ambiguities in this grammar fragment, this mainly serves as an example to introduce the exact mechanism for semantics construction. Then we will present a larger grammar based on the same mechanism in the next section.

As before, we compute the (underspecified) semantic representations by applying a set of inference rules that go through the syntax tree bottom-up. These rules are shown in Fig. 3.5. They derive statements of the form $t \hookrightarrow \varphi \mid X$, which express that the semantic representations of the syntactic subtree t are described by the dominance constraint φ, and that the *interface variable* of φ is X. The inference rules use the interface variables to combine dominance constraints of siblings with each other. Thus interface variables connect different components of the underspecified description, in much the same way as the `HANDLE` feature in HPSG grammars (Copestake and Flickinger 2000).

We change the lexicon a little so it now contains 4-tuples (W, α, φ, X) of a word W, its syntactic category α, a dominance constraint φ, and its interface variable X. For instance, the lexicon entry for "Peter" could contain the constraint $X{:}\mathsf{lam}(Y) \wedge Y{:}@(Y_1, Y_2) \wedge Y_1{:}\mathsf{var} \wedge Y_2{:}\mathsf{peter} \wedge \lambda(Y_1) = X$, with the interface variable X. This is simply the dominance constraints whose (single) constructive solution corresponds to the previous semantic representation $\lambda P.P(\mathsf{peter})$. When we access the lexicon in the first rule of Fig. 3.5, we make sure to replace all variables in φ by fresh variables; X' in the rule stands for the fresh variable which was substituted for φ's interface variable X.

For instance, the Det node in Fig. 3.3 is assigned the dominance constraint shown in Fig. 3.6(a), which happens to consist only of labelling and binding atoms. This is an instance of the constraint specified in the lexicon with fresh variables $X_1, \ldots, X_{10}$.

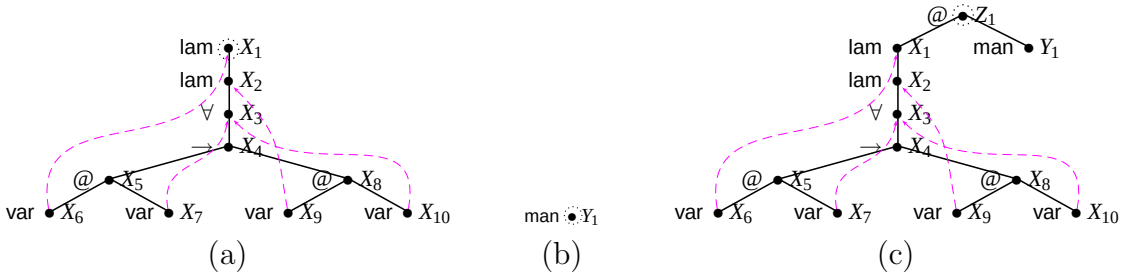


Figure 3.6: The dominance constraints associated by the construction rules in Fig. 3.5 for the Det (a), N (b), and NP (c) nodes in Fig. 3.2. The interface variables are indicated by dotted circles.

The interface variable X_1 is indicated by a dotted circle in the picture. Similarly, the constraint for the N node is shown in Fig. 3.6(b). The two constraints are plugged together by an application of the lower right construction rule in Fig. 3.5. This introduces a fresh variable Z_1 into the constraint, along with a labelling constraint $Z_1{:}@(X_1, Y_1)$ connecting Z_1 to the two previous interface variables.

The intended relation between the two systems of construction rules in Fig. 3.1 and Fig. 3.5 is that the former should derive a statement $t \hookrightarrow M$ iff the latter can derive $t \hookrightarrow \varphi \mid X$, for some variable X, and M is $\alpha\beta\eta$-equivalent to a constructive solution of φ. This relation holds for the interfaces in Fig. 3.1 and Fig. 3.5. For sentences with scope ambiguities, we still want to maintain this relation between our extended syntax-semantics interfaces for dominance constraints and the readings predicted by Montague or Keller; and indeed, this relation does hold, although we don't prove this here.

3.4 A Larger Grammar

We are now prepared to define a more serious grammar that computes dominance constraints. This grammar, a variant of the grammars in (Egg et al. 2001; Koller et al. 2003), still covers only a toy fragment of English, but it handles some nontrivial syntactic constructions and is thus a bit more realistic than the earlier one. The grammar is defined as follows:

(a1) S $\rightarrow$ NP VP
(a2) VP $\rightarrow$ IV
(a3) VP $\rightarrow$ TV NP
(a4) VP $\rightarrow$ CV NP to VP
(a5) VP $\rightarrow$ SV S
(a6) NP $\rightarrow$ PN
(a7) NP $\rightarrow$ Det $\bar{\text{N}}$
(a8) $\bar{\text{N}}$ $\rightarrow$ N
(a9) $\bar{\text{N}}$ $\rightarrow$ N RC
(a10) RC $\rightarrow$ RP S
(a11) $\bar{\text{N}}$ $\rightarrow$ N PP
(a12) PP $\rightarrow$ P NP
(a13) $\alpha \rightarrow W$ if $(W, \alpha, \varphi, X) \in Lex$

The rules (a3) to (a5) analyse sentences with verbs that are not intransitive. This includes transitive verbs (a3, see example 3.7), "object control verbs" (a4, see 3.8), and sentence-embedding verbs (a5, see 3.9).

(3.7) Every man *loves a woman.*

(3.8) Peter *wants John to kiss a woman.*

(3.9) Every man *knows some women sleep.*

Rules (a7) to (a12) analyse noun phrases that can have complex structures. A noun (N) can be combined with a determiner (Det) either on its own (a8), after modification with a relative clause (a9/a10, see 3.10), or after modification with a prepositional phrase (a11/a12, see 3.11).

(3.10) Every *man who has a mother* knows a woman.

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

A syntax-semantics interface that maps parse trees of this grammar to dominance constraints in $\mathcal{DI}\Lambda_\lambda$ is shown in Fig. 3.7. The graphs in the figure encode inference rules like in Fig. 3.5; the rules themselves would have become too unwieldy to show here. As before, each inference rule is associated with one production rule of the context-free grammar. The non-terminal symbols on both sides of the production rule are repeated in the graph, where they stand for the interface variables of the respective syntactic subtrees. For instance, the rule (b1) is the new version of the top right inference rule in Fig. 3.5. The symbols NP and VP represent the interface variables X and X' of the *np* and *vp* subtrees, and the symbol S stands for the interface variable Y of the entire *s* subtree. (The two rules are not identical. We will discuss this in a moment.) In rule (b4), VP$'$ represents the interface variable of the VP subtree that is the third child of the larger VP constituent. Whenever

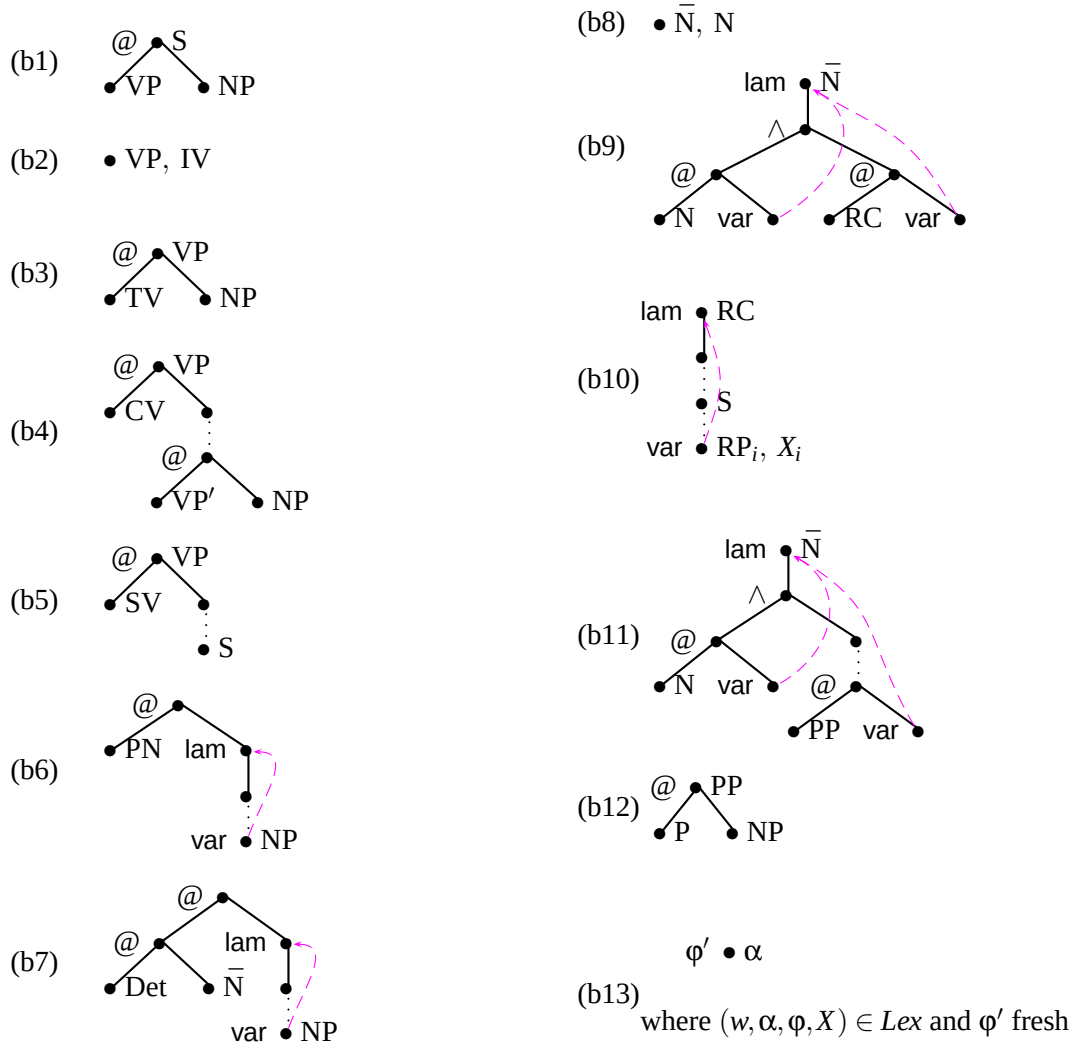


Figure 3.7: The syntax-semantics interface.

a node in one of the graphs is marked with two different variables X and Y, this represents an equality atom $X = Y$.

Relative clauses require special treatment. We make the standard assumption that a relative clause like "who Mary likes" is created from a deeper syntactic representation like "Mary likes [who]$_1$", which is an ordinary main clause, by moving the relative pronoun out of its original position and before "Mary". This means that the relative clause should really be written "[who]$_1$ Mary likes t_1", where t_1 is a trace as above – the empty string with identity 1 that was left behind when we moved the relative pronoun away. Computing such coindexations between relative pronouns and traces is beyond the capabilities of a context-free grammar, but this

is supported by all modern grammar formalisms used in computational linguistics, e.g. by the Slash feature in HPSG (Pollard and Sag 1994)

The advantage of this analysis is that the relative pronoun with index i and the trace with index i are linked by sharing the same index. We assume a special variable X_i for each index i, and a lexicon that contains lexicon entries $(t_i, \mathrm{NP}, X{:}\mathsf{var} \wedge X = X_i, X)$ for each trace t_i. Let's also say that the lexicon rule (b13) replaces only the X in this lexicon entry by a fresh variable, and leaves the X_i intact. Then the trace can be used like any other noun phrase in constructing the syntax and semantics of the S constituent to which it belongs. When we apply the rule for relative clauses (b10) to combine this S constituent with the relative pronoun with index i, the object-language variable at X_i will be bound by the lam binder introduced by (b10).

Finally, once the straightforward computation of the constraint φ according to the inference rules has finished, we add an inequality atom $X \neq Y$ for each pair of variables X and Y that occur on the left-hand side of a labelling atom in φ. If we didn't do this, the constraint for the sentence "Every man likes every man" would have the constructive solution $\forall x.\mathsf{man}(x) \rightarrow \mathsf{like}(x, x)$, as the two constraints for the first and the second "every man" would look identical and could be mapped to the same region of the satisfying tree. Adding the inequality constraints also means that the naive composition of the graph fragments in Fig. 3.7 for all constituents yields the correct constraint graph with respect to the interpretation of constraint graphs defined in Section 2.2.2. Finally, the inequality atoms have crucial implications on processing that we will explore in much more detail in Chapter 5.

It is possible, by adding additional further variables for bookkeeping to the four-place relation, to derive additional dominances for *scope islands*. Details of this construction can be found in (Egg et al. 2001).

3.5 An Example

In order to see how the larger grammar works, let's go through an example. We will compute a constraint for the following sentence, which was the example sentence (2.7) in Section 2.1.

(3.12) Every man loves a woman.

The parse tree of (3.12) according to the context-free grammar from the previous section is shown in Fig. 3.8. Now we can apply the inference rules in Fig. 3.7 in

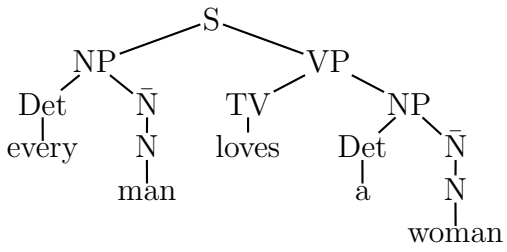
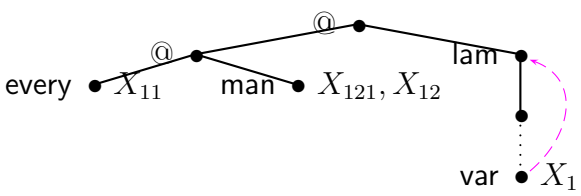


Figure 3.8: Syntax tree of (3.12).

order to compute the dominance constraint. We start with five applications of the lexicon rule (b13), one for each word in the sentence. Let's assume that we have lexicon entries $(\textit{every}, \mathrm{Det}, X{:}\mathsf{every}, X)$, $(\textit{man}, \mathrm{NP}, X{:}\mathsf{man}, X)$, and so on, i.e. every lexicon entry only introduces a labelling atom. Then the applications of (b13) gives us a conjunction of five labelling atoms, including $X_{11}{:}\mathsf{every}$ and $X_{121}{:}\mathsf{man}$. We write X_ν for the interface variable of the syntactic constituent at node address ν in the parse tree.

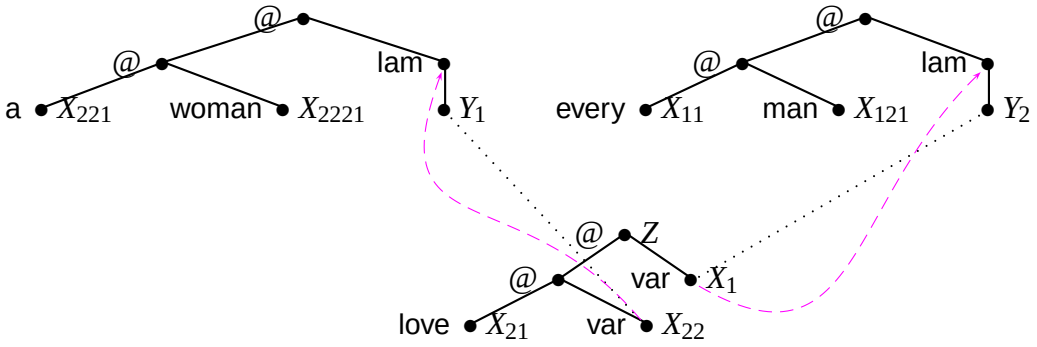
Next, we can compute a constraint for the noun phrase *every man* by combining the interface variables X_{11} and X_{12} with the inference rules (b8) and (b7). This results in the following constraint:

(3.13)



It is crucial to observe that the interface variable X_1 is not the variable at the root of the graph, but one that is embedded deeply inside the graph; we will come back to this later. We can perform the analogous construction for the noun phrase *a woman*, and then we can combine the two constraints for the noun phrases and the constraint for the transitive verb by applying the rules (b3) and then (b1). The result looks as follows:

(3.14)



As soon as we add the required inequality atoms, this is indeed the correct underspecified semantic representation (compare it with the constraint in 2.7). It has exactly the two following constructive solutions:

(3.15) $\mathsf{every}(\mathsf{man})(\lambda x.\mathsf{a}(\mathsf{woman})(\lambda y.\mathsf{love}(y)(x)))$

(3.16) $\mathsf{a}(\mathsf{woman})(\lambda y.\mathsf{every}(\mathsf{man})(\lambda x.\mathsf{love}(y)(x)))$

The main difference from (2.7), apart from the fact that the constructive solutions of (3.14) are formulas of higher-order and not first-order logic, is that (3.14) contains dominance atoms $Y_1 \triangleleft^* X_{22}$ and $Y_2 \triangleleft^* X_1$, rather than atoms like $Y_1 \triangleleft^* Z$ as in (2.7). But it is easy to verify that (3.14) *entails* $Y_1 \triangleleft^* Z \wedge Y_2 \triangleleft^* Z$ because of the additional inequality atoms.

Our syntax-semantics interface distinguishes between the quantifiers and the rest of the semantic representation in much the same way as Cooper storage. Where Cooper storage added quantifiers to the quantifier store, we add their descriptions to the constraint and add dominance atoms to control their placement. But the process of retrieving quantifiers from the store is replaced here by the constraint solving process, which takes place after the semantics construction is finished. In addition, we don't need to connect variables and quantifiers over indices i and establish binding by variable capturing, but can connect them explicitly by a binding atom which will maintain the correct variable binding regardless of the relative positions of binder and variable in the tree.

Another difference is that our interface applies the verb semantics to the variable bound by a quantifier. By comparison, Cooper storage had to give an NP a residual semantic representation $\lambda P.P(x_i)$ after pushing the quantifier semantics into the store. This was necessary in order to give NPs a uniform semantics that could be applied to the verb semantics, regardless of whether the quantifier was pushed to the store or applied directly. Because we employ dominance constraints to take care

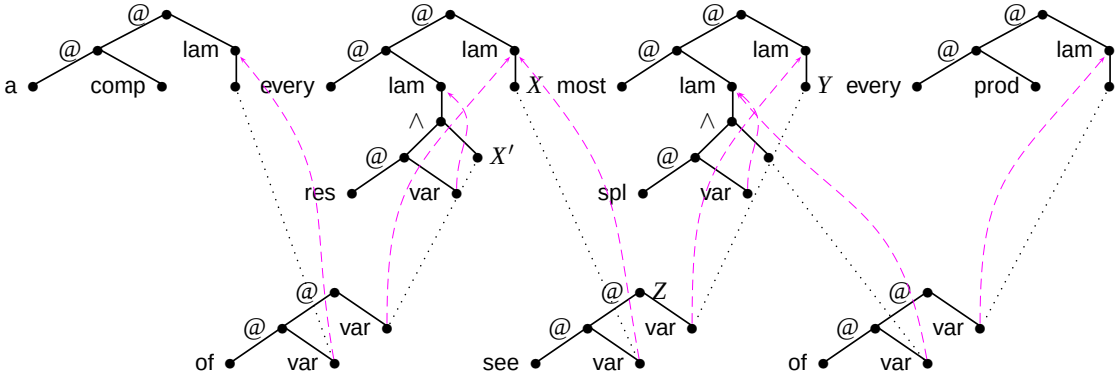


Figure 3.9: A dominance and binding constraint for "Every researcher of a company saw most samples of every product."

of the quantifier's placement, we don't need to make this distinction and replace the spurious lambda abstraction by a simple plugging together of the verb and NP semantics.

The interface can assign constraints to arbitrarily complex sentences that conform to the given grammar. Fig. 3.9 shows the dominance and binding constraint the grammar computes for the sentence "Every researcher of a company saw most samples of every product"; it has fourteen constructive solutions. Notice the zig-zag shape of this constraint graph, which connects upper and lower fragments with dominance edges. This shape is characteristic of the constraints that are produced by this grammar, and we will look at it more closely in Chapter 6.

3.6 Summary

In this chapter, we have seen how dominance and binding constraints can be systematically computed as underspecified descriptions for the semantics of English sentences. Starting from the aim of compositional semantics construction, we first reproduced a Montague-style syntax-semantics interface. When it turned out that this interface couldn't be easily extended to handle scope ambiguities, we switched to an interface that directly computed dominance constraints. We used a context-free grammar to analyse the syntactic structure of an English sentence, and then we computed a dominance constraint by applying inference rules that traversed the parse tree bottom-up.

The main advantage of underspecification in semantics construction is that the syntax-semantics interface can be kept simple. Instead of artificially blowing up

the number of syntactic readings or introducing nondeterminism and complex quantifier stores into the interface, we can simply plug dominance constraints together. The problem of the 1:n-correspondence of syntactic and semantic analyses is solved by a later processing step, which enumerates semantic readings from underspecified descriptions.

A key challenge in designing syntax-semantics interfaces for scope ambiguities is to decouple the structure of the semantic representation from the variable binding, in such a way that the results still represent binding correctly. Montague's and Cooper's approaches had to assume a mechanism that kept track of traces and their corresponding quantifiers, and then used variable capturing to establish binding – and even so, the original Cooper storage allowed erroneous analyses in which not all variables were bound. By contrast, the underspecification-based interface achieved a complete separation of structure and binding; this was possible because binding atoms guarantee correct variable binding regardless of the relative positions of binder and variable. It uses the additional power of dominance and binding constraints to introduce all binding atoms rule-locally: i.e. whenever it introduces a labelling atom X:var, it introduces a binding atom $\lambda(X) = Y$ at the same time. (The only exception is the connection between the relative pronoun and its trace in the rule for relative clauses, which needs to be added separately due to the syntactic complexity of such long-distance dependencies.) This is another important factor that keeps the interface simple.

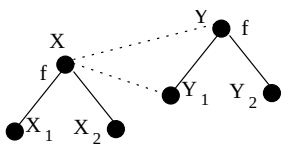
Chapter 4

Processing Dominance Constraints

This chapter gives an overview of algorithms for solving dominance and binding constraints. The two most fundamental processing questions that we can ask about a dominance constraint φ are:

1. Does φ have a solution?
2. What are the models of φ?

The first question is called the *satisfiability* problem, the second the *model enumeration* problem of dominance constraints. The satisfiability problem can be reduced to the model enumeration problem, as φ is unsatisfiable iff the set of its models is empty. Indeed, both algorithms in this chapter will be enumeration algorithms that also decide satisfiability as a special case.



The satisfiability problems of all dominance constraint languages from Chapter 2 that contain $\mathcal{D}$ are NP-complete. The fact that satisfiability of $\mathcal{SN}$ is in NP follows from the algorithm we present in Section 4.1; we could even allow arbitrary propositional connectives and still stay in NP (Duchier 2000). On the other hand, Koller et al. (2001) showed that the satisfiability problem of $\mathcal{D}$ is NP-hard. The key idea in the NP-hardness proof is that disjunction in propositional logic can be encoded into "dominance triangles", as shown to the right. Dominance triangles are constraints of the form $X{:}f(X_1, X_2) \wedge Y{:}f(Y_1, Y_2) \wedge Y \lhd^* X \wedge X \lhd^* Y_1$, i.e. without inequality atoms. In any solution of this dominance triangle, the variable X must be mapped to the same node as either Y or Y_1, and this is sufficient to encode the 3-SAT problem.

There are currently three fundamentally different classes of satisfiability/enumeration algorithms for dominance constraints:

1. *Saturation algorithms*: The constraint is successively enlarged by a system of saturation rules, until a solution has been found or false has been derived. The first sound and complete saturation algorithm for $\mathcal{D}$ was presented in 1998 by Koller et al. (2001); it was later extended to $\mathcal{S}$ by Duchier and Niehren (2000) and to binding constraints by Erk et al. (2003).

2. *Algorithms based on set constraints*: The constraint is translated into a constraint over finite sets of integers (Gervet 1994), for which efficient solvers are available (Müller and Müller 1997; Oz Development Team 1999). This line of solvers was pioneered by Duchier and Gardent (1999), extended to $\mathcal{S}$ by Duchier and Niehren (2000), and to the full propositional language by Duchier (2000).

3. *Graph algorithms*: The most efficient class of constraint solvers considers the constraint as a graph and then runs polynomial graph algorithms on it. As the satisfiability problem of dominance constraints is NP-complete, polynomial graph algorithms must assume certain restrictions on the constraints. The first graph algorithm was given for *normal* dominance constraints in (Koller et al. 2000; Althaus et al. 2001). We will take a closer look at this algorithm in Chapter 5; it has since been improved by Thiel (2004). An efficient graph algorithm for *weakly normal* dominance constraints has been proposed by Bodirsky et al. (2004).

We will present a saturation solver for $\mathcal{SN}$ in Section 4.1 and a set constraint solver for $\mathcal{SN}$ in Section 4.3. This presentation is based on (Koller et al. 2001; Duchier and Niehren 2000), although some proofs are new. Then we will present how the saturation algorithm can be generically extended to binding constraints in Section 4.2. This algorithm generalises the saturation algorithm for binding constraints in (Erk et al. 2003). We conclude the chapter by comparing runtimes of different solvers, and make some observations that will lead into the polynomial fragment in Chapter 5.

4.1 A Saturation Algorithm

The saturation algorithm is shown in Fig. 4.1. It consists of a set of *saturation rules* of the form $\varphi \to \psi_1 \vee \ldots \vee \psi_n$, where $n \geq 1$. The algorithm repeatedly applies saturation rules to an input constraint, which is extended by each rule application.

A rule $\varphi \to \psi_1 \vee \ldots \vee \psi_n$ is applied to a constraint φ' by selecting a disjunct on the right-hand side and adding that disjunct to φ'. The rule can be applied only if the following two conditions are satisfied:

1. inferences must be valid: the left-hand side φ is contained in φ';
2. inferences must add new information: no disjunct on the right-hand side is contained in φ'.

"Contained" means that each atom of some instance of the constraint is also an atom of the other constraint.

Computation terminates when no saturation rule can be applied any more. If the resulting constraint doesn't contain false, the algorithm claims that the original constraint is satisfiable. A saturated constraint that doesn't contain false is called a *solved form*, and we will see below how to extract a solution from a solved form. If all disjunctive choices lead to constraints that contain false, the algorithm claims that the original constraint is unsatisfiable.

Saturation algorithms like this fit nicely into the framework of constraint programming (Section 1.3) if we consider rules with $n = 1$ as *propagation rules* and rules with $n > 1$ as *distribution rules*. Thus the function of the propagation rules is to spell out implicit information in the constraint explicitly. We apply a single distribution rule only if no propagation rules are applicable any more. Such an application strategy can potentially reduce the runtime of the algorithm, but makes no difference for termination, soundness, or completeness in our case.

We call a saturation algorithm *sound* if unsatisfiability of a constraint φ follows from the fact that all saturated constraints derivable from φ contain false. We call the algorithm *complete* if every solved form is satisfiable. We will now prove that SN (i.e. the algorithm combining the rule sets S and N in Fig. 4.1) terminates and is sound and complete for $\mathcal{SN}$. It will follow as a special case that S is sound and complete for $\mathcal{S}$.

Proposition 4.1 (Termination of SN). *If φ is a dominance constraint with n variables, SN performs at most $O(n^2)$ saturation steps on φ.*

Proof. Each saturation step must add an atom of the form XRY that wasn't contained in the constraint before; but there are only 16 possible values for R. On the other hand, the saturation algorithm never adds new variables to the constraint. So saturation of a constraint with n variables terminates after at most $16n^2$ steps. □

Algorithm S:

Propagation rules:

(Clash)	$X \emptyset Y \quad \rightarrow \quad \mathsf{false}$
(Dom.Refl)	$\varphi \quad \rightarrow \quad X = X \qquad (X \text{ occurs in } \varphi)$
(Dom.Trans)	$X \lhd^* Y \wedge Y \lhd^* Z \quad \rightarrow \quad X \lhd^* Z$
(Lab.Decomp)	$X{:}f(X_1, \ldots, X_n) \wedge Y{:}f(Y_1, \ldots, Y_n) \wedge X = Y \quad \rightarrow \quad \bigwedge_{i=1}^{n} X_i = Y_i$
(Lab.Ineq)	$X{:}f(\ldots) \wedge Y{:}g(\ldots) \quad \rightarrow \quad X \neq Y \qquad \text{if } f \neq g$
(Lab.Disj)	$X{:}f(\ldots, X_i, \ldots, X_k, \ldots) \quad \rightarrow \quad X_i \perp X_k \qquad \text{where } 1 \leq i < k \leq n$
(Lab.Dom)	$X{:}f(\ldots, Y, \ldots) \quad \rightarrow \quad X \lhd^+ Y$
(Inter)	$X R_1 Y \wedge X R_2 Y \quad \rightarrow \quad X R Y \qquad \text{if } R_1 \cap R_2 \subseteq R$
(Inv)	$X R Y \quad \rightarrow \quad Y R^{-1} X$
(Disj)	$X \perp Y \wedge Y \lhd^* Z \quad \rightarrow \quad X \perp Z$
(NegDisj)	$X \lhd^* Z \wedge Y \lhd^* Z \quad \rightarrow \quad X \neg\perp Y$
(Child.up)	$X \lhd^* Y \wedge X{:}f(X_1, \ldots, X_n) \wedge \bigwedge_{i=1}^{n} X_i \neg\lhd^* Y \quad \rightarrow \quad Y = X$

Distribution rules:

(Distr.Child)	$X \lhd^* Y \wedge X{:}f(X_1, \ldots, X_n) \quad \rightarrow \quad X_i \lhd^* Y \vee X_i \neg\lhd^* Y \qquad (1 \leq i \leq n)$
(Distr.NegDisj)	$X \neg\perp Y \quad \rightarrow \quad X \lhd^* Y \vee X \rhd^+ Y$

Algorithm N:

Propagation rules:

(NonI1)	$\neg(X \lhd^* Y \lhd^* Z) \wedge X \lhd^* Y \quad \rightarrow \quad Y \neg\lhd^* Z$
(NonI2)	$\neg(X \lhd^* Y \lhd^* Z) \wedge Y \lhd^* Z \quad \rightarrow \quad X \neg\lhd^* Y$

Figure 4.1: Saturation rules for dominance and non-intervention constraints.

Proposition 4.2. *Every solution of a constraint φ satisfies exactly one SN-solved form of φ.*

Proof. It is clear that every solution satisfies *at most* one SN-solved form, because different SN-solved forms are created by choosing a different disjunct on the right-hand side of some distribution rule; and the disjuncts of the same rule are mutually inconsistent in SN.

For the proof that every solution satisfies *at least* one SN-solved form, observe first that the left-hand side of each saturation rule entails the right-hand side if we read choice as disjunction. This means that if a pair $(\mathcal{M}, \alpha)$ satisfies the left-hand side of any rule, it must also satisfy one of the choices on the right-hand side. On the other hand, there are no infinite sequences of saturation steps, according to Prop. 4.1. The claim follows by induction over the number of rule applications. □

Corollary 4.3 (Soundness of SN). *If all SN-saturations of the constraint φ contain* false*, then φ is unsatisfiable.*

Proof. We prove the contrapositive. If φ is satisfiable, then it has a satisfiable SN-solved form φ' according to Prop. 4.2. Because φ' is satisfiable, it can't contain false. □

As usual, the completeness proof is a bit more complex than termination and soundness. We proceed in two steps: First we show that special *simple* solved forms are satisfiable; then we show how to extend arbitrary solved forms to simple solved forms. The proof follows Section 4 of (Duchier and Niehren 2000) for the most part. The proof of Prop. 4.5 is new.

Definition 4.4. A variable X is *labelled* in the dominance constraint φ if there is a variable Y with $X = Y$ and $Y{:}f(\ldots)$ in φ. X is a *root variable* of φ if $X \lhd^* Y$ in φ for all $Y \in \mathsf{Var}(\varphi)$. A SN-solved form is called *simple* iff all its variables are labelled, and it has a root variable.

Proposition 4.5. *Every simple SN-solved form has a constructive solution. In particular, it is satisfiable.*

Proof. Let φ be a simple SN-solved form. We construct a model whose nodes are equality classes of variables in φ. This makes sense because the relation $=_\varphi := \{(X, Y) \mid X = Y \text{ in } \varphi\}$ is an equivalence relation: Reflexivity follows from saturation under (Dom.Refl), symmetry from saturation under (Inv), and transitivity from saturation under (Dom.Trans), (Inv) and (Inter). We write $\bar{X}$ for the equivalence class of X and $\mathsf{Var}(\varphi)/=_\varphi$ for the set of equivalence classes.

The solution is defined as follows:

$$\begin{array}{lcl} V & = & \mathsf{Var}(\varphi)/=_\varphi \\ E & = & \{(\bar{X},\bar{Y}) \mid X{:}f(\ldots,Y,\ldots) \text{ in } \varphi\} \\ L_V(\bar{X}) & = & f \quad \text{iff } X'{:}f(\ldots) \text{ in } \varphi, \text{ for some } X' \in \bar{X} \\ L_E(\bar{X},\bar{Y}) & = & i \quad \text{iff } X'{:}f(Z_1,\ldots,Z_{i-1},Y',\ldots) \text{ in } \varphi, \text{ for some } X' \in \bar{X},\ Y' \in \bar{Y} \\ \alpha(X) & = & \bar{X} \end{array}$$

We have to show that $\tau = (V, E, L_V, L_E)$ is well-defined, is a finite constructor tree, and that $(\mathcal{M}_\tau, \alpha)$ satisfies φ. It is then obvious that $(\mathcal{M}_\tau, \alpha)$ is also a constructive solution of φ.

1. *Well-definedness:* V is well-defined because $=_\varphi$ is an equivalence relation.

   L_V is well-defined because every variable in φ is labelled, and if there are two variables X, Y with $X = Y$, $X{:}f(\ldots)$, and $Y{:}g(\ldots)$ in φ, then $f = g$; otherwise (Lab.Ineq) would have inferred $X \neq Y$.

   Similarly, L_E is well-defined because if $X = Y$, $Z{:}f(Z_1,\ldots,Z_{i-1},X,Z_{i+1},\ldots,Z_n)$, and $W{:}f(W_1,\ldots,W_{k-1},Y,W_{k+1},\ldots,W_n)$ are in φ and $i < k$, then $W = Z$ is also in φ, as we will show in a moment. But then $X = W_i$ by (Lab.Decom), so $Y = W_i$ by (Dom.Trans) and (Inter). On the other hand, $Y \perp W_i$ by (Lab.Disj), so $Y \emptyset W_i$ by (Inter), i.e. φ contains false, a contradiction.

   The proof that $W = Z$ in φ is as follows. By (Lab.Dom), (Inter), (Trans), and (NegDisj), we know that $Z \neg\perp W$ is in φ. By (Distr.NegDisj), either $Z \lhd^* W$ or $Z \neg\lhd^* W$ are in φ. If it were $Z \neg\lhd^* W$, we can infer that $W \lhd^+ Z$ in φ by (Inter). Now we apply (Distr.Child) for each child of W. $W_i \lhd^* Z$ leads to a contradiction, for each i, because $Y_i \lhd^* X$ and $Y_i \perp X$ (Trans, Lab.Disj), so $X \perp Y$ by (Disj). $Y \lhd^* Z$ leads to a contradiction because then $X \lhd^* Z$ and $Z \lhd^+ X$ (Trans, Lab.Dom). So (Child.up) is applicable, which infers $W = Z$, contradicting $W \lhd^+ Z$. This concludes the proof that the choice in (Distr.NegDisj) couldn't be $Z \neg\lhd^* W$; it must be $Z \lhd^* W$. We can apply (Distr.NegDisj) again to $W \neg\perp Z$; by an analogous argument as above, we can infer that the choice couldn't be $W \neg\lhd^* Z$, and must have been $W \lhd^* Z$. But now we know $W = Z$ from (Inter).

2. *Finite constructor tree:* We show that every node has at most one father, the graph (V, E) is acyclic, and that it has a unique root. The fact that a node $\bar{X}$ has exactly one outgoing edge with label i for each $1 \leq i \leq \mathsf{ar}(L_V(\bar{X}))$ will then be obvious from the construction.

Let X, Y be variables with $X = Y$ in φ, and let $W{:}f(\ldots, X, \ldots)$ and $Z{:}f(\ldots, Y, \ldots)$ also be in φ. Then $W = Z$ is in φ, as we showed above, so $\bar{X}$ has the unique father $\bar{W}$.

For the acyclicity, assume that $v_0, \ldots, v_{n-1}, v_0$ is a cycle. This means that there are $X_{i1}, X_{i2} \in v_i$ with $X_{i2} \lhd^+ X_{(i+1 \bmod n)1}$ in φ for all $0 \leq i < n$, by (Lab.Dom). For each i, $X_{i1} \lhd^* X_{i2}$ is also in φ, by (Inter). Then by (Dom.Trans) and (Inter), $X_{01} \lhd^* X_{(n-1)2}$ is in φ, which derives $X_{01} \emptyset X_{(n-1)2}$ together with $X_{(n-1)2} \lhd^+ X_{01}$, a contradiction.

For the uniqueness of the root, we prove that if $X \lhd^* Y$ is in φ, then there is a path from $\bar{X}$ to $\bar{Y}$ in τ. It follows that if R is the root variable assumed in Def. 4.4, every node in V can be reached from $\bar{R}$.

So let $X, Y \in \mathsf{Var}(\varphi)$ with $X \lhd^* Y$ in φ. X is labelled, so there is an X' with $X' = X$ and some $X'{:}f(X_1, \ldots, X_n)$ in φ. Now because of saturation under (Distr.Child), either there is some i such that $X_i \lhd^* Y$ in φ, or there is no such i and $X' = Y$ in φ because of (Child.up). If we repeatedly apply this construction, starting from some arbitrary X_0 with $X_0 \lhd^* Y$, we can construct a sequence $X_0, X_1, X_2, \ldots$ of variables s.t. $X_i \lhd^* Y$ is also in φ for all i. This sequence must be finite because each $\bar{X}_i$ is a child of $\overline{X_{i-1}}$ in τ, and there are no infinite paths in τ. Hence there must be some n with $X_n = Y$ in φ; the path from $\bar{X}_0$ to $\bar{Y}$ is then $\bar{X}_0, \bar{X}_1, \ldots, \bar{X}_n$.

3. *Solution:* We let $\mathcal{M}$ be the tree structure induced by τ and show that $(\mathcal{M}, \alpha)$ indeed satisfies φ, by proving that every single atom in φ is satisfied by it. This is obvious for labelling atoms $X{:}f(X_1, \ldots, X_n)$.

   For atoms XRY with set operators, we prove the stronger claim that for any two variables $X, Y \in \mathsf{Var}(\varphi)$, there is an $r \in \{\lhd^+, \rhd^+, \perp, =\}$ such that XrY is in φ and $\mathcal{M}, \alpha \models XrY$. The claim follows trivially for $r \in R$; the case $r \notin R$ can't occur because φ would also contain $X \emptyset Y$ in this case, by (Inter).

   To prove the subclaim, we distinguish the four possible primitive relations in which $\alpha(X)$ and $\alpha(Y)$ can stand in $\mathcal{M}$.

   - If $\alpha(X) = \alpha(Y)$, we must have $X = Y$ in φ, by construction.
   - If $\alpha(X)$ is a proper ancestor of $\alpha(Y)$, then there must be a sequence $X{:}f_1(\ldots, X_1, \ldots)$, $X'_1{:}f_2(\ldots, X_2, \ldots)$, $\ldots$, $X'_{n-1}{:}f_n(\ldots, Y, \ldots)$ of atoms in φ s.t. $X_i = X'_i$ also in φ for all i. By (Lab.Dom), (Inter), and (Dom.Trans), φ also contains $X \lhd^* Y$. Now φ is also saturated under (Distr.NegDisj), which is applicable here because (Inter) derives $Y \neg \perp X$. If it contained the first disjunct, $Y \lhd^* X$, then (Trans) and (Lab.Dom) would infer $X'_{n-1} \emptyset Y$, a contradiction. Hence φ must come from the second disjunct, so $Y \neg \lhd^* X$ and hence $X \lhd^+ Y$ are in φ.

- The case where $\alpha(Y)$ is an ancestor of $\alpha(X)$ is symmetrical, and leads to $X \triangleright^+ Y$ in φ.
- If $\alpha(X)$ and $\alpha(Y)$ are disjoint, consider their infimum $\bar{Z}$ in $\mathcal{M}$. $\bar{X}$ and $\bar{Y}$ are below different children of $\bar{Z}$, so there must be a labelling atom $Z{:}f(\ldots, Z_1, \ldots, Z_2, \ldots)$ in φ s.t. $Z_1 \triangleleft^* X$ and $Z_2 \triangleleft^* Y$ (or vice versa) are also in φ. But then (Lab.Disj) and (Disj) derive $X \perp Y$.

The final type of constraint is non-intervention $\neg(X \triangleleft^* Y \triangleleft^* Z)$. Assume that $(\mathcal{M}, \alpha)$ maps Y to a node between $\bar{X}$ and $\bar{Z}$. Then $X \triangleleft^* Y$ and $Y \triangleleft^* Z$ must be in φ, by the argument we just made. But this would have derived a clash, e.g. by (NonI1) and (Inter).

□

Lemma 4.6 (Extension by labelling). *Every SN-solved form φ with an unlabelled variable X can be extended to an SN-solved form $\varphi \wedge \varphi'$ with strictly fewer unlabelled variables, and in which X is labelled.*

Proof. The detailed proof can be found in (Duchier and Niehren 2000), Lemma 2. The key idea is to determine the equality classes of variables that can denote children of X, and then to select one representative X_i from each class and extend φ with the following constraint:

$$\begin{array}{rcl} \varphi' & := & X{:}f(X_1, \ldots, X_n) \wedge \\ & & \bigwedge\{XRZ \wedge ZR^{-1}X \mid \triangleleft^+ \in R,\ X_i \triangleleft^* Z \text{ in } \varphi,\ 1 \leq i \leq n\} \wedge \\ & & \bigwedge\{YRZ \mid \perp \in R,\ X_i \triangleleft^* Y \text{ in } \varphi,\ X_j \triangleleft^* Z \text{ in } \varphi,\ 1 \leq i \neq j \leq n\} \end{array}$$

X is labelled in $\varphi \wedge \varphi'$, and all variables that were labelled before are still labelled. In addition, it can be shown that $\varphi \wedge \varphi'$ is still in SN-solved form. Note that f is some symbol of arity n. If necessary, this symbol can be simulated by using multiple occurrences of a symbol of arity 2 or more, which we assumed to exist.

Because Duchier and Niehren only considered $\mathcal{S}$, we have to check that φ' is also in SN-solved form, i.e. that (NonI1) and (NonI2) don't become applicable by the new atoms in the extension. But by construction and (Dom.Trans), φ' contains no atoms $X_1 \triangleleft^* X_2$ that weren't already contained in φ. □

Proposition 4.7 (Completeness of SN). *Every SN-solved form is satisfiable.*

Proof. Let φ be a constraint in SN-solved form. If φ doesn't contain a root variable, we can pick a fresh variable X and consider the SN-solved extension $\varphi \wedge \bigwedge\{XRY \wedge YR^{-1}X \mid \triangleleft^+ \in R,\ Y \in \mathsf{Var}(\varphi)\}$ instead. Now we can apply Lemma 4.6 and extend

this constraint further, obtaining a simple solved form φ' after a finite number of extension-by-labelling steps. φ' is satisfiable according to Prop. 4.5. Because φ' contains all atoms from φ, this solution also satisfies φ. □

Note that a constraint that is in SN-solved form but not simple does not necessarily have a constructive solution, as the "extension by labelling" process can introduce new node labels and even new nodes.

Taking all these results together, we get the following fundamental result about the satisfiability of dominance constraints:

Theorem 4.8 (Satisfiability of $\mathcal{SN}$). *The algorithm SN decides satisfiability of constraints in $\mathcal{SN}$ in nondeterministic polynomial time.*

As we have seen, it does this by enumerating all solved forms of the input constraint φ. If the input constraint is unsatisfiable, there will be no solved forms. If it *is* satisfiable, the set of solved forms will induce a partition of the solutions of φ, as we have seen in Prop. 4.2. So SN solves the model enumeration problem of $\mathcal{SN}$ as well, by computing representatives for classes of solutions that have only irrelevant differences among themselves.

4.2 Processing binding constraints

Now we will extend SN with rules to process binding constraints. The goal will be to automatically obtain a sound and complete algorithm from an axiomatisation of the semantics of an arbitrary "well-behaved" binding specification.

We will require somewhat more expressive saturation rules to deal with binding constraints. As an example, consider the rule that expresses that every binder of the lambda binding specification carries the label lam:

$$(\Lambda_\lambda.\text{Binder}) \qquad \lambda_\lambda(X) = Y \quad \rightarrow \quad \exists X'.X{:}\mathsf{lam}(X')$$

This rule doesn't follow the format of the rules in SN in that it contains an existential quantifier on the right-hand side. Intuitively, the idea is that the rule shouldn't do anything if such a labelling atom is already present in the constraint; otherwise it should pick a fresh variable in place of X' and add the resulting labelling atom.

More formally, we first define substitutions over variables of a constraint as functions from some set V of variables into the complete set Vars of variables. If V is a set of variables, φ a constraint, and $\theta : V \rightarrow \mathsf{Vars}$ a substitution, we write $\varphi\theta$ for

the result of applying θ to φ and call it a *V-variant* of φ. We call the variant *fresh* if $\theta(V) \cap \mathsf{Var}(\varphi) = \emptyset$.

Then we generalise saturation as follows. Rules are now of the form

$$\varphi \to (\exists V_1.\psi_1) \vee \ldots \vee (\exists V_n.\psi_n),$$

where $n \geq 1$ and the V_i are sets of variables such that $\mathsf{Var}(\psi_i) - \mathsf{Var}(\varphi) \subseteq V_i$ and $V_i \cap \mathsf{Var}(\varphi) = \emptyset$ for all i. Application of such a rule to a constraint φ' is possible if (a) some variant $\varphi\theta$ of the left-hand side is contained in φ', and (b) no V_i-variant of $\psi_i\theta$ belongs to φ', for any $1 \leq i \leq n$. The rule is then applied by selecting an i and a fresh V_i-variant ψ_i' of $\psi_i\theta$, and adding ψ_i' to φ'.

We can read a saturation rule as an (implicitly) universally quantified impliciation that talks about the nodes of a tree structure. For instance, we can read the rule (Trans) from Fig. 4.1 as the first-order formula

$$\forall X \forall Y \forall Z.(X \lhd^* Y \wedge Y \lhd^* Z) \to X \lhd^* Z.$$

All rules in SN are valid in every tree structure when read as such formulas: There can be no tree structure and variable assignment that satisfies the left-hand side but not the right-hand side of the same rule. Rules about binding constraints won't necessarily be satisfied by an arbitrary lambda structure $\mathcal{L}$ – unless we make sure that $\mathcal{L}$ is admissible with respect to the respective binding specification. Conversely, we can read the rules as *axioms* that enforce that $\mathcal{L}$ is actually admissible.

Definition 4.9. A set R of saturation rules is called an *axiomatisation* of a binding specification Λ if a lambda structure $\mathcal{L}$ is a model of all universal implications in R iff $\mathcal{L}$ is Λ-admissible.

An axiomatisation is *restrictive* iff no binding atoms occur on the right-hand sides of its rules. It is *equality insensitive* iff no two atoms on its left-hand side that aren't equality atoms talk about the same variables. It is *guarded* iff for each labelling atom $Z{:}f(\ldots)$ on its left-hand side, there are variables X, Y such that $\lambda(X) = Y$ and either $Z = X$ or $Z = Y$ are also on its left-hand side.

A restrictive, equality insensitive, and guarded axiomatisation is called *proper* iff it additionally contains rules of the following form:

$$\begin{array}{lll}
\text{(Func)} & \lambda(X) = Y \wedge \lambda(U) = V \wedge X = U \quad \to \quad Y = V \\
\text{(Var)} & \lambda(X) = Y \quad \to \quad X{:}\mathsf{var}_\Lambda \\
\text{(Binder)} & \lambda(X) = Y \quad \to \quad \bigvee_i \exists Y_1 \ldots \exists Y_n Y{:}f_i(Y_1, \ldots, Y_n)
\end{array}$$

Fig. 4.2 displays axiomatisations for three of the four binding specifications from Section 2.3. Each of these axiomatisations is proper. In particular, they each

(Λ_λ.Func)	$\lambda(X) = Y \wedge \lambda(U) = V \wedge X = U$	$\rightarrow$	$Y = V$
(Λ_λ.Var)	$\lambda(X) = Y$	$\rightarrow$	$X{:}\mathsf{var}$
(Λ_λ.Binder)	$\lambda(X) = Y$	$\rightarrow$	$\exists Y'.Y{:}\mathsf{lam}(Y')$
(Λ_λ.Scope)	$\lambda(X) = Y$	$\rightarrow$	$Y \lhd^* X$

(Λ_{fol}.Func)	$\lambda(X) = Y \wedge \lambda(U) = V \wedge X = U$	$\rightarrow$	$Y = V$
(Λ_{fol}.Var)	$\lambda(X) = Y$	$\rightarrow$	$X{:}\mathsf{var}$
(Λ_{fol}.Binder)	$\lambda(X) = Y$	$\rightarrow$	$\exists Y'.Y{:}\exists(Y') \vee \exists Y'.Y{:}\forall(Y')$
(Λ_{fol}.Scope)	$\lambda(X) = Y$	$\rightarrow$	$Y \lhd^* X$

(Λ_{ana}.Func)	$\mathsf{ante}(X) = Y \wedge \mathsf{ante}(U) = V \wedge X = U$	$\rightarrow$	$Y = V$
(Λ_{ana}.Var)	$\mathsf{ante}(X) = Y$	$\rightarrow$	$X{:}\mathsf{ana}$
(Λ_{ana}.Binder)	$\mathsf{ante}(X) = Y$	$\rightarrow$	$\bigvee_{f\in\Sigma} \exists\{Y_1, \ldots, Y_{\mathsf{ar}(f)}\}.Y{:}f(Y_1, \ldots, Y_{\mathsf{ar}(f)})$
(Λ_{ana}.Scope)	$\mathsf{ante}(X) = Y$	$\rightarrow$	$Y \neq X$

Figure 4.2: Saturation rules for some binding specifications.

have a (Func) rule that ensures the binding relation is a partial function, (Var) and (Binder) rules that ensure that the variables and binders are labelled appropriately, and a (Scope) rule that encodes the scope specification. The rule (Λ_{ana}.Binder) is not very useful – it states that the antecedent of the anaphor has a label, which is hopefully established by a labelling atom that came from the syntax-semantics interface –, but it is necessary for completeness, and simply encodes the binder set of Λ_{ana}.

We can now prove that if we add such rule sets to SN, we obtain a sound and complete saturation algorithm for the respective binding specifications, and that the algorithm still terminates.

Lemma 4.10. *Let $\Lambda_1, \ldots, \Lambda_n$ be binding specifications with functional, equality insensitive axiomatisations $R_1, \ldots, R_n$. Then every simple $SNR_1 \ldots R_n$-solved form is $\Lambda_1 \ldots \Lambda_n$-satisfiable.*

Proof. Let φ be a simple $SNR_1 \ldots R_n$-solved form. It is in particular a simple SN-solved form, and hence has a solution $(\mathcal{M}_\tau, \alpha)$ according to Prop. 4.5. We can extend $\mathcal{M}_\tau$ with partial binding functions $\lambda_1, \ldots, \lambda_n$ as follows:

$$\lambda_i(\bar{X}) = \begin{cases} \bar{Y} & \text{if } \lambda_i(X) = Y \text{ in } \varphi \\ \text{undefined} & \text{otherwise} \end{cases}$$

Each λ_i is a well-defined partial function because R_i is functional, and it is clear that the lambda structure $\mathcal{L} = \mathcal{L}_{\tau,\lambda_1,\ldots,\lambda_n}$ satisfies φ. It remains to show that it is admissible, by showing that each R_i is satisfied by $\mathcal{L}$ as a universal quantification.

So let $l \to r$ be a rule in some R_i, and let $\bar{X}_1, \ldots, \bar{X}_m$ be nodes in τ such that $l[\bar{X}_1, \ldots, \bar{X}_m]$ is satisfied by $\mathcal{L}$. Then there are representatives $X_1, \ldots, X_m$ of these equality classes such that $l[X_1, \ldots, X_m]$ is in φ: We can pick representatives independently for each atom in l that isn't an equality atom (because R_i is equality insensitive), and the two representatives X_k, X'_k picked for some equality atom $\bar{X}_k = \bar{X}_k$ will be present in in φ because of saturation under (Dom.Refl), (Dom.Trans), and (Inv). But φ is saturated under $l \to r$, so there is a variant of a choice in $r[X_1, \ldots, X_m]$ that is also in φ, and hence satisfied by $\mathcal{L}$. □

This lemma can be generalised to a full completeness theorem by the extension-by-labelling construction from Lemma 4.6. The main problem is that we must make sure that the extension-by-labelling process doesn't accidentally trigger a left-hand side of some binding axiomatisation, as this would destroy saturation.

Theorem 4.11 (Soundness and completeness for binding constraints). *Let $\Lambda_1, \ldots, \Lambda_n$ be binding specifications with proper axiomations $R_1, \ldots, R_n$ whose left-hand sides only contain labelling, binding, and equality atoms. Then $SNR_1 \ldots R_n$ is sound and complete for $\Lambda_1 \ldots \Lambda_n$-satisfiability of $\mathcal{SN}\Lambda_1 \ldots \Lambda_n$. The saturation algorithm terminates in nondeterministic polynomial time.*

Proof. Soundness means that every $\Lambda_1 \ldots \Lambda_n$-admissible solution of a constraint φ satisfies some $SNR_1 \ldots R_n$-solved form of φ. This is obvious, as the rules are valid as universal implications over $\Lambda_1 \ldots \Lambda_n$-admissible lambda structures, and hence won't allow us to derive false for $\Lambda_1 \ldots \Lambda_n$-satisfiable inputs.

Termination isn't trivial, because the rules in the R_i may introduce new variables through existential quantifiers. However, new variables can only be added once in the application of each rule in R_i to each binding atom. But because R_i is restrictive, no rule introduces new binding atoms, so the R_i rules only add a linear number of new variables.

For completeness, let φ be an arbitrary $SNR_1 \ldots R_n$-solved form. It's in particular an SN-solved form, so we can add a root (as in the proof of Prop. 4.7) and perform the "extension by labelling" process of Lemma 4.6 to obtain a simple SN-solved form φ' that entails φ. Now, extension by labelling will never add equality or binding atoms, and because all the R_i are guarded, the only labelling atoms that appear on the left-hand side of any rule are for variables that are equal to the X or Y in some binding atom $\lambda(X) = Y$. But no such labelling atom can be added by "extension by labelling", as X and Y are guaranteed to be labelled by the rules we

required for properness. This means that φ' is also a simple $SNR_1 \dots R_n$-solved form, so φ', and hence φ, is $\Lambda_1 \dots \Lambda_n$-satisfiable, by Lemma 4.10. □

So let's say we want to model a new object language with dominance and binding constraints. This first requires us to encode the way this object language handles variable binding in a binding specification. Then if we can define a proper axiomatisation for this new binding specification, Theorem 4.11 tells us we immediately have a sound and complete algorithm for deciding satisfiability and enumerating models that are consistent with the new binding specification. Thus the theorem is a very powerful tool, which is applicable to a wide range of binding specifications.

We conclude this discussion with some examples for why the preconditions of the theorem are all necessary, and how they can be achieved in practice.

1. *R must ensure labelling atoms for the variable and binder:* Consider a version R_1 of the system (Λ_{ana}) from Fig. 4.2 that doesn't contain the rule (Λ_{ana}.Binder), but that does contain a rule

   $$\text{(notF)} \qquad \mathsf{ante}(X) = Y \wedge Y : f(Z, W) \quad \rightarrow \quad \mathsf{false}$$

   R_1 is an axiomatisation of a binding specification Λ_1 that is like Λ_{ana}, but doesn't allow the label f for antecedents. Now imagine f is the only symbol in our signature with arity greater than 1. Then the constraint $X{:}f(Y, Z) \wedge Y \lhd^* Y' \wedge Y \lhd^* Y'' \wedge Y'{:}a \wedge Y''{:}b \wedge Z{:}\mathsf{ana} \wedge \mathsf{ante}(Z) = Y$ is Λ_1-unsatisfiable: The extension-by-labelling process must assign the label f to the node denoted by Y, but then the axiom (notF) is violated. However, Y isn't labelled in the original constraint, so the constraint is in SNR_1-solved form. Similar problems arise if an axiomatisation isn't guarded.

   Situations like these can be prevented by having the rules that are required for properness in Def. 4.9; if a rule like (Λ_{ana}.Binder) had been part of R_1, the labelling atom couldn't have been introduced by "extension by labelling". Another way to make sure such problems don't occur is by providing a sufficient number of node labels with arity 2 or more that can be used to "extend by labelling" without making a rule applicable. In the example, we could have constructed a model if we had had another symbol g of arity 2 or more.

2. *R must be equality insensitive:* Consider an axiomatisation R_2 of the binding specification Λ_2 that contains the following rule:

   $$\text{(notFG)} \qquad X{:}f(Y) \wedge Y{:}g(Z) \quad \rightarrow \quad \mathsf{false}$$

   The rule (notFG) won't be applicable to the SN-solved form of the constraint $X{:}f(Y) \wedge Z{:}g(W) \wedge Y = Z$, so this constraint is also in SNR_2-solved form.

However, Y and Z will be mapped to the same node in every solution of the constraint, which means that (notFG) won't be satisfied as a universal implication. Hence the constraint is Λ_2-unsatisfiable. The term "equality insensitive" refers to the fact that additional equality atoms in the constraint won't affect the applicability of the rule, which caused the problem in this example.

It is easy to ensure equality insensitivity by naming the variables in the rule apart, and relating them in equality atoms on the left-hand side. The rule above would have better been written as follows:

$$\text{(notFG}') \qquad X{:}f(Y) \wedge W{:}g(Z) \wedge Y = W \quad \rightarrow \quad \mathsf{false}$$

3. *R must not allow disjointness or unrestricted dominance on the left-hand sides:* Consider a weakened form R_3 of the rule system for (Λ_λ) from Fig. 4.2, in which the Scope rule has been replaced by the following rule:

$$\text{(notDisj)} \qquad \mathsf{lam}(X) = Y \wedge X \perp Y \quad \rightarrow \quad \mathsf{false}$$

Let's say R_3 is an axiomatisation of a binding specification Λ_3. Now consider the constraint $Z{\lhd^*}X \wedge Z{\lhd^*}Y \wedge X{:}\mathsf{lam}(X') \wedge X'{:}a \wedge Y{:}\mathsf{var} \wedge \lambda(Y) = X$. (notDisj) isn't applicable to the SN-normal form of the constraint, so it's also in SNR_3-normal form. But the "extension by labelling" process will add some labelling constraint $Z{:}f(X,Y)$, so X and Y *will* be disjoint in the constructed model (and in fact, *must* be disjoint in any model), so the model isn't Λ_3-admissible.

Note that this particular type of problem can be avoided by rephrasing the rule in the following way, which satisfies the application conditions of Thm. 4.11:

$$\text{(notDisj}') \qquad \mathsf{lam}(X) = Y \quad \rightarrow \quad X \neg\perp Y$$

If more than one set operator atom from the left-hand side must be moved to the right-hand side, the rule may have to be made a distribution rule. This is computationally dangerous, but can be acceptable as long as e.g. the other rules propagate so strongly that there is always only one choice left in the distribution rule.

4.3 An Algorithm Based On Set Constraints

The saturation algorithm from Section 4.1 is sound and complete, and in fact is designed in such a way that it facilitates proving soundness and completeness.

$$\begin{array}{rcl} \mathcal{B} & ::= & \mathsf{false} \mid X_1 = X_2 \mid I \in D \mid i \in S \mid i \notin S \\ \mathcal{C} & ::= & \mathcal{B} \mid S_1 \cap S_2 = \emptyset \mid S_3 \subseteq S_1 \cup S_2 \mid \mathcal{C}_1 \wedge \mathcal{C}_2 \mid \mathcal{C}_1 \textbf{ or } \mathcal{C}_2 \end{array}$$

Figure 4.3: Finite set and finite domain constraints with disjunctive propagators.

However, it is not terribly useful for solving dominance constraints in practice, as it spends a lot of time computing irrelevant consequences of a constraint in too much detail. For example, as soon as it establishes the atom $X \lhd^+ Y$, it also computes all atoms $X\ R\ Y$ with $\lhd^+ \in R$ by (Inter), all atoms $Y\ R^{-1}\ X$ by (Inv), and so on.

We can get a much more efficient way of solving dominance constraints by translating them into finite set constraints (Duchier and Gardent 1999; Duchier and Niehren 2000) and applying efficient special-purpose constraint solvers for these. These constraint solvers will provide propagation that is tailored to the problem of solving set constraints. We will now briefly introduce finite set constraints, then we will sketch the algorithm (i.e., the encoding of dominance as set constraints), and then we will go through an example.

4.3.1 Finite set constraints

Finite set (FS) constraints (Gervet 1994; Müller and Müller 1997; Oz Development Team 1999) are formulas that speak about the common relations (such as inclusion and equality) between terms denoting finite sets of integers. The terms can be either literals specifying a concrete set of integers, variables denoting sets, or terms built from these components using set operations such as intersection and union.

The abstract syntax of these formulas is specified in Fig. 4.3; S are variables denoting sets of integers, I are *finite domain* variables denoting integers (see Section 1.3), i is an integer literal such as 5, and D is a set literal such as $\{1, 2, 3\}$. The symbol X stands for a variable that can be either a finite set or a finite domain variable. We will also use further formulas that are defined as abbreviated notations for the primitive constraints, defined in Fig. 4.4. The declarative semantics of FS constraints is the usual semantics of the mathematical symbols. Ignore the A **or** B for now; we will come back to this in Section 4.3.2.

Current implementations of constraint solvers for FS constraints are quite efficient in deciding satisfiability and enumerating all satisfying variable assignments (Oz Development Team 1999). Solvers in the constraint programming framework perform simple deterministic inferences (propagation), as described in Section 1.3. Some propagation rules for FS constraints are shown in Fig. 4.5. Distribution rules

$$
\begin{array}{rll}
I \neq i & \text{for} & I \in \Delta - \{i\} \\
S_1 \parallel S_2 & \text{for} & S_1 \cap S_2 = \emptyset \\
S = D & \text{for} & \bigwedge\{i \in S \mid i \in D\} \wedge \bigwedge\{i \notin S \mid i \in \Delta - D\} \\
S_1 \subseteq S_2 & \text{for} & S_1 \subseteq S_2 \cup S_3 \wedge S_3 = \emptyset \\
S = S_1 \cup S_2 & \text{for} & S \subseteq S_1 \cup S_2 \wedge S_1 \cup S \wedge S_2 \cup S \\
S = S_1 \dot{\cup} S_2 & \text{for} & S_1 \parallel S_2 \wedge S = S_1 \cup S_2
\end{array}
$$

Figure 4.4: Some further abbreviations we will use in FS constraints.

$$
\begin{array}{lcl}
S \subseteq D \wedge i \notin D & \Rightarrow & i \notin S \\
i \notin S_1 \wedge i \notin S_2 \wedge S \subseteq S_1 \cup S_2 & \Rightarrow & i \notin S \\
S \subseteq S_1 \cup S_2 \wedge i \in S \wedge i \notin S_1 & \Rightarrow & i \in S_2 \\
i \in S_1 \wedge S_1 \subseteq S_2 & \Rightarrow & i \in S_2 \\
i \in S_1 \wedge i \in S_2 \wedge S_1 \cap S_2 = \emptyset & \Rightarrow & \mathsf{false}
\end{array}
$$

Figure 4.5: Some propagation rules for FS constraints.

for FS constraints choose a set variable whose value isn't yet completely determined, and splits its domain into two parts.

Let's look at an example to illustrate how a FS constraint solver searches for a solution; say we want to solve the constraint shown on the left-hand side of Fig. 4.6. We start by splitting the constraints into *basic* constraints (language $\mathcal{B}$ in Fig. 4.3) and *complex* constraints (language $\mathcal{C}$). The basic constraints, i.e. equality, set-membership and set-non-membership constraints, are written directly into a *constraint store*. The complex constraints are turned into *propagators*, which concurrently watch the constraint store for changes that satisfy the preconditions of some propagation rules.

In the leftmost constraint store, both propagators can contribute information. First, the store entails that $S_1 \cup S_2 \subseteq \{1, 2, 3\}$, so the first propagator, $S \subseteq S_1 \cup S_2$, can infer that $4 \notin S \wedge 5 \notin S \wedge 6 \notin S$; this information is added to the store. Second, since the store entails $1 \in S \wedge 1 \notin S_2$, the first propagator can deduce $1 \in S_1$. On the other hand, the second propagator, $S_1 \subseteq S_3$, can infer from $3 \notin S_3$ that $3 \notin S_1$; and from $1 \in S_1$, which was added by the first propagator, it can infer $1 \in S_3$. But now, the store entails $S_1 \cup S_2 \subseteq \{1, 2\}$, so the first propagator can again infer that $3 \notin S$.

The result of all these propagation steps is shown in the middle of Fig. 4.6. At this point, the two propagators have added a significant amount of information, which has reduced the number of possible values of the four variables from 1024 to 8. Now no further propagation is possible, so the constraint solver performs

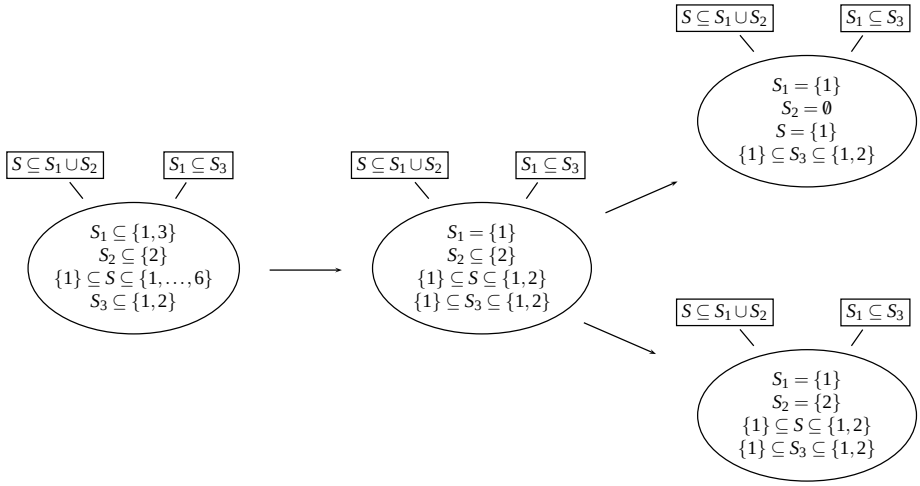


Figure 4.6: Propagation and distribution for FS constraints (example).

a single distribution. It creates two copies of the constraint store (shown on the right), and adds $2 \notin S_2$ to the first one, and $2 \in S_2$ to the second. Propagation can resume at this point, and so forth until all branches have either become inconsistent (some propagator has inferred false) or solved (the value of each variable is known completely).

4.3.2 Disjunctive propagators

Another type of constraint that we need for the dominance constraint solver are the *disjunctive propagators* A **or** B. This formula has the declarative semantics of a disjunction $A \vee B$. Operationally, the propagator waits until it can infer that either A or B is inconsistent with the current constraint store. Then it adds the other disjunct to the constraint store.

In the example of Fig. 4.6, assume that we had the disjunctive propagator $2 \in S_2$ **or** $2 \in S_3$ as an additional propagator. Then this propagator would suspend and do nothing until the constraint $2 \in S_2$ becomes inconsistent in the top right constraint in Fig. 4.6. At that point, it would commit to the choice $2 \in S_3$ and add it to the constraint store, which would fully determine the values of all variables in that store.

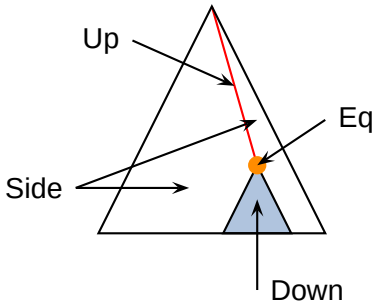


Figure 4.7: A tree, partitioned into the four node sets.

4.3.3 The solver

Now we have all the tools that we need to describe the FS-based solver for dominance constraints. What this solver does is to translate a dominance constraint into an FS constraint with disjunctive propagators, and then passes this encoding to an FS solver, e.g. within the Mozart system (Oz Development Team 1999). It was first presented by Duchier and Gardent (1999) and later extended by Duchier and Niehren (2000). We only present the key ideas here, and refer to (Duchier and Niehren 2000) for the complete details and the proof of soundness and completeness.

The key insight underlying the solver is that each node v in a tree partitions the tree into four disjoint sets of nodes (see Fig. 4.7): The node v itself, all nodes that properly dominate v, all nodes that are properly dominated by v, and all other nodes (i.e. which are disjoint from v). If we know these four node sets for each node in the tree, we know the structure of the entire tree.

In order to encode a dominance constraint φ with variables $X_1, \ldots, X_n$ as a set constraint, we introduce finite set variables $\mathsf{eq}(X_i)$, $\mathsf{down}(X_i)$, $\mathsf{up}(X_i)$, and $\mathsf{side}(X_i)$ denoting (integers from $N = \{1, \ldots, n\}$ encoding) those variables that denote nodes in those regions of the tree, relative to X_i. The fact that each node partitions the tree into the four regions is expressed by the set constraint

$$N = \mathsf{eq}(X_i) \,\dot\cup\, \mathsf{down}(X_i) \,\dot\cup\, \mathsf{up}(X_i) \,\dot\cup\, \mathsf{side}(X_i).$$

We also introduce auxiliary variables such as $\mathsf{eqdown}(X_i)$ and $\mathsf{equp}(X_i)$, which are related to the basic variables with constraints such as

$$\mathsf{eqdown}(X_i) = \mathsf{eq}(X_i) \,\dot\cup\, \mathsf{down}(X_i)$$

The other crucial component of the encoding are *choice variables*. Each choice variable $C_{X_iX_k}$ is a finite domain variable whose value must be an element of the set $\mathbf{Rel} = \{=, \lhd^+, \rhd^+, \perp\}$, encoded as the natural numbers $\{1, 2, 3, 4\}$. The value of the choice variable determines the relationship in which the nodes denoted by X_i and X_k stand in a solution.

For each pair of variables, there is a disjunctive propagator that relates the choice variable for these two variables to constraints over some FS variables encoding the semantics of the primitive tree relations, i.e. we have for each $1 \leq i, k \leq n$

$$C_{X_iX_k} \in \mathbf{Rel} \wedge \bigwedge\{Choice(X_i, r, X_k) \mid r \in \mathbf{Rel}\},$$

where

$$Choice(X, r, Y) = \mathsf{D}[\![X_i r X_k]\!] \wedge C_{X_iX_k} = r \textbf{ or } \mathsf{D}[\![X_i \neg r X_k]\!] \wedge C_{X_iX_k} \neq r.$$

The choice variables also drive the distribution: As soon as no propagator can contribute new information, the constraint solver will pick one choice variable and split its domain (rather than splitting the domain of a set variable as described above).

The encodings $\mathsf{D}[\![X_i r X_k]\!]$ are set constraints relating the variable sets defined above. For instance, disjointness and non-disjointness are encoded as follows:

$$\begin{array}{lcl} \mathsf{D}[\![X \perp Y]\!] & = & \mathsf{eqdown}(X) \subseteq \mathsf{side}(Y) \wedge \mathsf{eqdown}(Y) \subseteq \mathsf{side}(X) \\ \mathsf{D}[\![X \neg\perp Y]\!] & = & \mathsf{eqdown}(X) \cap \mathsf{side}(Y) = \emptyset \wedge \mathsf{side}(X) \cap \mathsf{eq}(Y) = \emptyset \end{array}$$

The other primitive relations and their negations are encoded similarly. Labelling constraints are encoded using additional variables for the parent and daughters of a node, as well as its label, which are again related to the tree-region variables defined above. Non-intervention can be encoded as follows:

$$[\![\neg(X \lhd^* Y \lhd^* Z)]\!] \;=\; \mathsf{eq}(Y) \subseteq \mathsf{up}(X) \cup \mathsf{side}(Z) \cup \mathsf{down}(Z)$$

That is, we can encode all of $\mathcal{SN}$ into finite set constraints in such a way that the solutions of the constraints correspond one-to-one.

4.3.4 An example

An example will illustrate how the propagations on the finite set constraint correspond to the original dominance constraint. Consider the dominance constraint from Fig. 3.9 on page 44 (without the binding constraints), and let's assume that the propagation has already proceeded to the point where the dominance $X \lhd^* Z$ and the disjointness $X \perp X'$ have been made explicit as set constraints, i.e. it has derived the following information from the dominance:

$$Z \in \mathsf{eqdown}(X),$$

and it has derived from the disjointness,

$$\mathsf{eqdown}(X) \subseteq \mathsf{side}(X').$$

The solver can now propagate the information that Z and X' are also disjoint. This is done by the disjunctive propagator for disjointness of Z and X', i.e.

$$\mathsf{D}[\![Z \perp X']\!] \wedge C_{ZX'} = \perp \textbf{ or } \mathsf{D}[\![Z \neg\perp X']\!] \wedge C_{ZX'} \neq \perp$$

The first propagation rule in Fig. 4.5 can add the basic constraint $Z \in \mathsf{side}(X')$ to the constraint store. But then, the constraint store for $Z \neg\perp X'$ in the disjunctive propagator will fail, as it contains $\mathsf{eq}(Z) \cap \mathsf{side}(X') = \emptyset$, which clashes with the fact that Z is a member of both sets. This means the disjunctive propagator will commit to the branch for $Z \perp X'$, and will add the entire encoding of the disjointness atom as a set constraint.

Once all propagation is completed, the constraint solver will distribute by splitting the domain of a choice variable. In the example, the relative scope of X and Y is unknown, but propagation will have derived the information $C_{XY} \in \{=, \lhd^+, \rhd^+\}$ because they both dominate Z. Now the solver can split the domain of C_{XY}, e.g. into the cases $C_{XY} = \rhd^+$ and $C_{XY} \in \{=, \lhd^+\}$. This corresponds exactly to the behaviour of the (Distr.NegDisj) rule in Fig. 4.1.

The search tree that the set constraint solver explores looks as in Fig. 4.8. The round nodes of the search tree represent choice points at which a distribution step was performed; the diamond-shaped leaves (which are drawn green in the original) represent solutions of the set constraint, i.e. solved forms of the dominance constraint. There are fourteen green leaves because the constraint has fourteen solved forms.

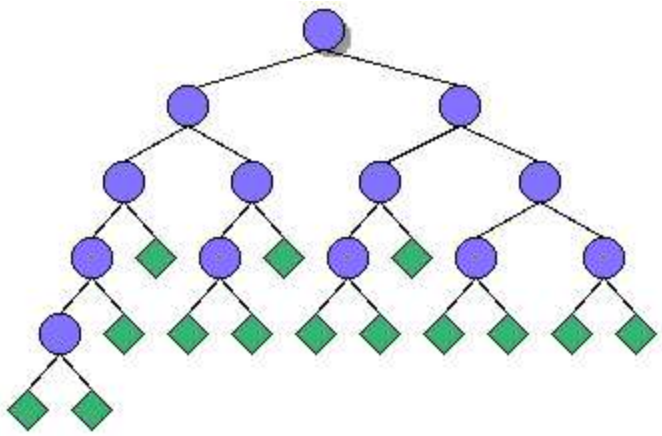


Figure 4.8: Search tree of the FS solver for the constraint in Fig. 3.9.

The most striking feature of the search tree is that there is no failure: All leaves represent solutions, and there are no inconsistent constraint stores, which would be drawn as (red) boxes. This means that the propagation was so strong that no distribution step ever made a choice where one option was unsatisfiable. Every constraint that is useful in practice (e.g., all constraints generated by the grammar in Section 3.4) seems to have failure-free search trees. This phenomenon, which we call the "miracle of the green nodes", is extremely surprising, as we have seen that satisfiability of dominance constraints is NP-complete, which means the algorithm should have to make choices that lead to failure, without being able to recognise them right away. This is a strong indication that there is a useful polynomial fragment of dominance constraints, and our constraint-based solver automatically exploits the properties of this fragment and runs in polynomial time on it.

4.4 Summary

In this chapter, we have given an overview of algorithms for solving dominance constraints. We have presented two solvers in more detail: one which saturates the dominance constraint as a logical formula, and one which encodes it into finite set constraints and runs an existing solver on the encoding. Both algorithms solve the model enumeration problem of dominance constraints, but can also serve as satisfiability algorithms. We have also shown how the saturation algorithm can be extended to a complete algorithm for certain useful classes of binding constraints.

Both solvers employed a propagate-and-distribute strategy in order to keep the size

length	solved forms	saturation	FS solver
2	2	10	10
3	5	40	30
4	14	390	150
5	42	2070	690
6	132	7300	2900
7	429	30230	10790

Figure 4.9: Runtimes of the two enumeration algorithms on the pure chains of length 2–7, in milliseconds.

of the search space manageable, and indeed it can be shown that they can simulate each other's computations (Duchier and Niehren 2000). However, the saturation solver computes every single atom entailed by the input constraint, which is useful for proofs, but makes the solver too slow for practical use. Fig. 4.9 shows the runtimes each solver takes to enumerate all solved forms of the *pure chains* of length 2–7, in milliseconds CPU time on a Pentium M at 1.6 GHz. (A chain is a constraint that connects upper and lower tree fragments in a zig-zag shape; Fig. 3.9 shows a chain of length 4. See also Section 6.4.) The FS solver outperforms the saturation solver considerably. But we will see in the next chapter that we can obtain further dramatic improvements in performance by designing a specialised graph-based solver for the polynomial fragment of *normal* dominance constraints.

While we focus mostly on dominance constraints in this thesis, binding constraints are extremely important if we want to use dominance constraints as underspecified descriptions of formulas in some object language. Under this perspective, it is very useful that the saturation algorithm for dominance constraints can be generically extended with rules for binding constraints: We get the complete algorithm for free if we can define a proper axiomatisation of the binding specification. The general theorem subsumes most object languages that are used in practice, such as (first-order or higher-order) predicate logic. One object language to which it cannot be applied directly is Dynamic Predicate Logic, which we will look at more closely in Chapter 7; but even there, we will be able to reuse the basic structure of the completeness proof.

Chapter 5

Normal Dominance Constraints

The "miracle of the green nodes" is a strong indication that dominance constraints have a useful fragment whose satisfiability problem is polynomial. In this chapter, we present such a fragment. We present an algorithm that decides satisfiability of *normal* dominance constraints in quadratic time, and can be extended to a model enumeration algorithm for normal dominance constraints.

The key insight underlying the satisfiability algorithm for normal dominance constraints is that such constraints can be seen as *dominance graphs*, and satisfiability can be represented as a property of such graphs. We have appealed to the intuition that dominance constraints can be drawn as graphs a lot so far, without stating the exact relationship between the two. Thus a second goal of this chapter is to clarify this relationship.

The constraint and graph perspectives will be linked by the concept of a *solved form*. Solved forms are the desired end results of an enumeration algorithm for constraints or graphs; in addition, the solved forms of a normal dominance constraints serve as representations of sets of mutually similar solutions. We will define solved forms of graphs and solved forms of normal dominance constraints separately, but a crucial result of this chapter will be that we can encode normal dominance constraints into dominance graphs in such a way that their solved forms correspond. Technically, we will first *compactify* normal dominance constraints, and then prove an equivalence between compact dominance constraints and dominance graphs.

The chapter is divided into two parts; its structure is illustrated in Fig. 5.1. We will first introduce dominance graphs in Section 5.1, and show intuitively how they relate to dominance constraints. Then we will properly define normal (Section 5.2) and compact (Section 5.4) dominance constraints, and prove that they are equivalent, and that their solved forms are indeed representations of sets of solutions

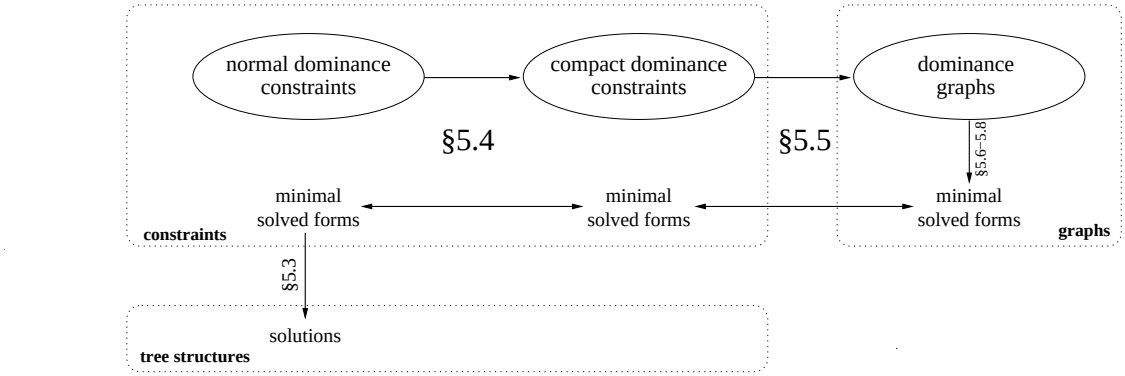


Figure 5.1: Overview of Chapter 5.

(Section 5.3). Finally we will show that compact dominance constraints correspond to dominance graphs (Section 5.5).

In a second part of this chapter, we show how solvability of dominance graphs (and hence, satisfiability of normal dominance constraints) can be decided with a polynomial graph algorithm. We will first show how we can enumerate all minimal solved forms of a dominance graph, given a solvability algorithm (Section 5.6). Then we characterise solvability as the absence of *simple hypernormal cycles* (Section 5.7) and show how to test a dominance graph for hypernormal cycles (Section 5.8). To round off the chapter, we show how the algorithms carry over to normal dominance *and binding* constraints in Section 5.9.

5.1 Dominance Graphs

In the first few chapters, we have frequently presented dominance constraints as graphs. Each node of a graph represented a variable of the constraint; we used node labels and solid edges to represent labelling atoms and dotted edges to represent dominance atoms, and we considered the graph to be an implicit representation of some inequality atoms.

Now we are going to make this intuition precise by defining *dominance graphs*. Dominance graphs are exactly like the informal graphs we have drawn so far, except that their nodes are unlabelled, and we assume that the tree fragments (i.e., the connected components over solid edges) have depth at most one.

Definition 5.1 (Dominance Graph). A *dominance graph* is a directed graph $G = (V, E \dot\cup D)$ satisfying the following two conditions:

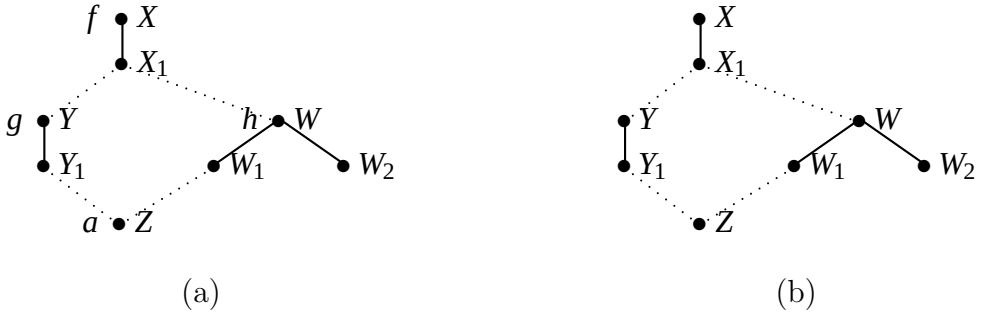


Figure 5.2: A compact dominance constraint (a) and its dominance graph (b).

1. The graph $G = (V, E)$ defines a collection T of node disjoint trees of height 0 or 1.

2. Each edge in D goes from a leaf of a tree in T with height 1 to the root of some other tree.

To bring out the similarity between dominance graphs and the earlier informal graph representations more clearly, we will call the edges in E *tree edges* and draw them as solid lines, and we will call the edges in D *dominance edges* and draw them as dotted lines. Nodes with incoming tree edges are called *holes*, and all others are called *roots*; i.e. Condition 2 in the definition can be stated as "dominance edges go from holes to roots". We will also call the elements of T the *(tree) fragments* of the dominance graph.

For example, Fig. 5.2(b) shows a dominance graph with four tree edges and four dominance edges. It contains three fragments. In the example in Fig. 5.2, X, Y, Z, and W are roots, and X_1, Y_1, W_1, and W_2 are holes. The node Z, which is not adjacent to any tree edges, counts as a root and not as a hole. Notice the similarity between the dominance graph in Fig. 5.2(b) and the graph representation in (a); the only difference between the two graphs is that (b) has no node labels.

The purpose of a dominance constraint is to serve as an underspecified description of a set of tree structures, and there is a clear intuition that the tree fragments in the graph should be *configured* into a tree. This intuition is supported, for instance, by the introductory example (2.5) and its solutions (2.2) and (2.4). In a graph setting, we can define a *solved form* of a dominance graph G directly as a tree-shaped dominance graph that contains the same nodes and tree edges as G and realises all dominances that G specifies.

For the formal definition of a solved form, we take the *reachability relation* R^*_G of a dominance graph to be the ordinary graph reachability relation of the graph, i.e.

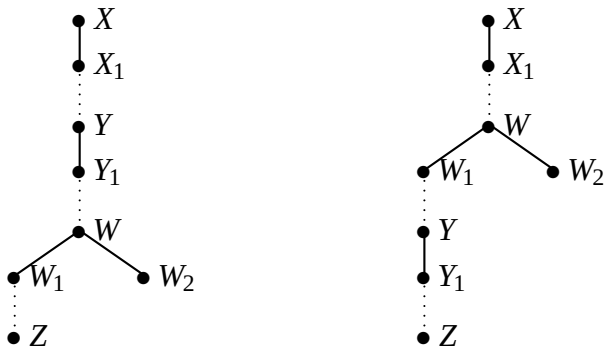
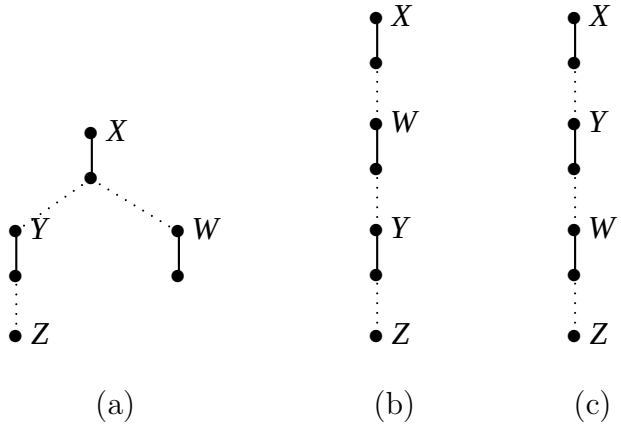


Figure 5.3: The two solved forms of the dominance graph in Fig. 5.2.



(a) (b) (c)

Figure 5.4: A minimal solved form (a), and some solved forms that properly extend it (b,c). (b) and (c) are not minimal.

$(u, v) \in R^*_G$ iff there is a (directed) path from u to v in G.

Definition 5.2 (Solved Form of a Dominance Graph). A dominance graph is *in solved form* iff it is a forest. The dominance graph $G = (V, E \dot\cup D)$ *extends* the dominance graph $G' = (V', E' \dot\cup D')$ iff $V = V$, $E = E'$, and $R^*_{G'} \subseteq R^*_G$. A dominance graph is *a solved form of* the graph G iff it is in solved form and extends G. A *minimal solved form* of G is a solved form of G that is not a proper extension of any other solved form of G. A dominance graph is *solvable* if it has a solved form.

Both solved forms in Fig. 5.3 are minimal solved forms of the graph in Fig. 5.2(b). For an example of a non-minimal solved form, consider the dominance graphs in Fig. 5.4. The graph (a) is a dominance graph in solved form, so it is its own minimal solved form. The graphs (b) and (c) are two different solved forms of (a)

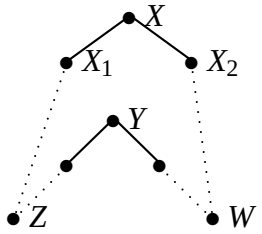


Figure 5.5: An unsolvable, acyclic dominance graph.

that are not minimal, because they are proper extensions of (a).

Not all dominance graphs are solvable. For example, dominance graphs that have a cycle can clearly not be unrolled into finite forests that realise all dominance requirements of the original graph. Another example is the dominance graph in Fig. 5.5. This graph is cycle-free, but it is still unsolvable because we can't configure X and Y so the whole graph becomes a tree. For instance, Y must not be dominated by X_1 in a solved form, because then W would be reachable both from X_1 (via Y) and from X_2; the other three cases are analogous.

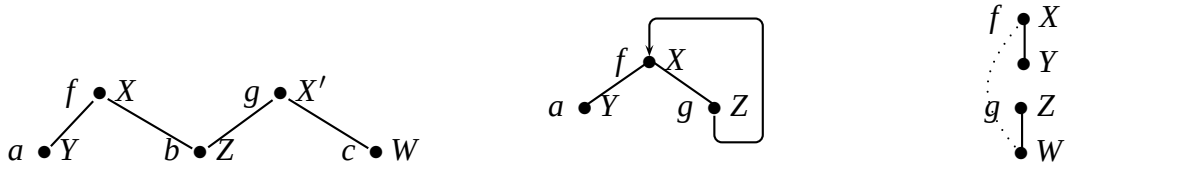
5.2 Normal Dominance Constraints

Despite the apparent similarity between the informal graph representations of dominance constraints and dominance graphs, there are a number of differences that make the relationship between dominance constraints and dominance graphs non-trivial:

1. The fragments in a dominance constraint need not be trees. For instance, there may be nodes that appear on the right-hand sides of two different labelling atoms (Fig. 5.6a), or fragments could have cycles (Fig. 5.6b).

2. Dominance graphs require us to *configure* the fragments into a forest; the fragments may not overlap each other. Without the inequality atoms, a dominance constraint as represented by the graph in Fig. 5.5 would be satisfiable because we can map X and Y to the same node in a tree structure.

3. Dominance atoms are not restricted to requiring dominances from a hole to a root. The constraint in Fig. 5.6(c) is a perfectly well-formed dominance constraint, but if we delete the node labels, it is not a dominance graph because the dominance edge goes from a root into a hole.



(a) $X{:}f(Y,Z) \wedge X'{:}g(Z,W)$ (b) $X{:}f(Y,Z) \wedge Z{:}g(X)$ (c) $X{:}f(Y) \wedge Z{:}g(W) \wedge X \lhd^* W$

Figure 5.6: Some dominance constraints that are not normal.

4. Fragments could have a depth of more than one. This was the case in most dominance constraints we have seen so far, e.g. in Fig. 3.9.

At least the first three problems arise only in dominance constraints that are not needed in practice; for example, they never come up in constraints produced by the grammar in Chapter 3. We will define *normal* dominance constraints as constraints that exclude the first three problems, and *compact* dominance constraints as normal constraints that also exclude the fourth problem.

In a first step, we introduce some definitions that allow us to use graph terminology when talking about a dominance constraint, such as roots and holes, reachability, and fragments. These notions will correspond to the respective notions in a dominance graph for normal dominance constraints.

Definition 5.3 (Graph components of dominance constraints). If $X{:}f(X_1,\ldots,X_n)$ is a labelling atom, we say that X is the *head* of the labelling atom, and $X_1,\ldots,X_n$ are its *children*.

Let φ be a dominance constraint. A variable $X \in \mathcal{V}(\varphi)$ is called a *root* iff it is not a child in any labelling atom in φ. It is called a *leaf* iff it is not the head of a labelling atom of arity one or more. In particular, it is called a *hole* iff it is not the head of any labelling atom.

The *one-step labelling reachability relation* LR_φ is the binary relation

$$LR_\varphi := \{(X,Y) \mid X{:}f(\ldots,Y,\ldots) \in \varphi\}.$$

The *one-step reachability relation* R_φ is the binary relation

$$R_\varphi := LR_\varphi \cup \{(X,Y) \mid X \lhd^* Y \in \varphi\}.$$

If R is one of the two one-step relations, we write R^+ for the transitive closure, R^* for the reflexive, transitive closure, and $R^\leftrightarrow$ for the reflexive, transitive, and symmetric closure. We call R^*_φ the *reachability relation* of φ.

A *fragment* in φ is a nonempty subset $F \subseteq \mathcal{V}(\varphi)$ such that $F^2 \subseteq LR_\varphi^\leftrightarrow$ (i.e. a variable set that is connected by labelling constraints). We call the fragment *maximal* if it has no proper superset that is also a fragment of φ.

Now we can use this terminology to define normal dominance constraints.

Definition 5.4 (Normal dominance constraints). A dominance constraint φ in the language $\mathcal{DI}$ is called *normal* iff for all variables $X, Y \in \mathcal{V}(\varphi)$:

1. There is an atom $X \neq Y$ in φ iff X and Y are different variables that occur as heads of labelling atoms *(no overlap)*.
2. Every variable appears at most once as a head and at most once as a child in a labelling atom of φ, and LR_φ^+ has no elements of the form (X, X) *(tree-shaped fragments)*.
3. If $X \triangleleft^* Y$ is in φ, then X is a hole *(dominances out of holes)*.
4. Every variable in φ occurs in a labelling atom *(no empty fragments)*.

This definition ensures that the problem cases (1) to (3) from above cannot occur. The first condition explicitly excludes situations as in the dominance triangle, where two labelled variables can be mapped to the same node in a solution. By convention, we have assumed that each of the informal graphs we have used so far encodes an inequality atom for each pair of labelled variables; so this condition is met by all constraints we have seen so far.

The second condition requires all fragments in the constraint graph to be tree-shaped by requiring each node to have a unique father, and the entire fragment to be acyclic. This excludes cases as in Fig. 5.6(a) and (b).

Condition 3 states that no node in the constraint graph has both an outgoing dominance edge and an outgoing solid edge. This excludes cases such as Fig. 5.6(c), in which the non-hole X has an outgoing dominance $X \triangleleft^* W$. Note that this condition doesn't require that a node couldn't have both an *incoming* dominance and an *incoming* solid edge: The constraint in Fig. 5.6(c) with $Y \triangleleft^* W$ instead of $X \triangleleft^* W$ is normal.

Finally, the fourth condition states that there are no "empty" fragments which don't contain any labelled nodes. This is a technicality that will simplify the relationship between constraints and graphs.

Some useful properties of normal dominance constraints are stated in the following lemma.

Lemma 5.5 (Basic properties of normal dominance constraints). *Let φ be a normal dominance constraint.*

1. *No variable in φ is both a root and a hole.*
2. *No dominance atom in φ connects variables in the same fragment.*
3. *Every fragment $\mathcal{F}$ of φ contains exactly one root, which we call $\mathcal{R}(\mathcal{F})$. All other variables in $\mathcal{F}$ are reachable from $\mathcal{R}(\mathcal{F})$.*
4. *Every variable X is a member of a unique maximal fragment $\mathcal{F}_{\max}(X)$.*

All dominance constraints that are generated by the grammar from Chapter 3 are normal. This is because we paid attention to satisfying the second to fourth condition in putting together the constraints, and then we applied an additional closure operation to the result, which added the inequalities necessary to establish the no-overlap condition. From here on, if φ is a dominance constraint that satisfies conditions 2 to 4, and that has no inequality atoms $X \neq Y$ where either X or Y is a hole, we will write $\varphi^{\neq}$ for the *normalisation*

$$\varphi^{\neq} \;:=\; \varphi \wedge \bigwedge\{X \neq Y \mid X \text{ and } Y \text{ are heads of different labelling atoms}\}.$$

It is clear that $\varphi^{\neq}$ is then always a normal dominance constraint.

Definition 5.4 differs in three details from the standard definition of normal dominance constraints in other papers, such as (Althaus et al. 2003). First, we have restricted the inequalities in Condition 1 to be *only* those that are necessary to ensure fragments don't overlap. The definition could alternatively require *at least* those inequalities, and potentially allow more. All results in this chapter remain essentially valid, but Section 6.3 requires the definition presented here. Secondly, previous versions of Condition 2 didn't require acyclicity of fragments. This makes no difference in practice because cyclic fragments are easy to detect using depth-first search, and all constraints with cyclic fragments are unsatisfiable. Finally, the present version of Condition 3 permits dominances that go into non-roots, so we can consider the constraints generated by the grammar in Chapter 3 as normal. Earlier definitions required dominances to go only into roots, and made the constraints from the grammar normal by an additional postprocessing step. This postprocessing step exploited the fact that the other three conditions ensure that the normalisation of a constraint like $X{:}f(Y) \wedge Z \lhd^* Y$ entails $Z \lhd^* X$, i.e. we can always move the dominance edges up to the roots.

5.3 Solved Forms

When we solve dominance constraints, it is generally impossible to enumerate all solutions of the constraint. This is because every satisfiable dominance constraint has an infinite number of solutions, which differ from each other only in additional material that is filled into dominance edges, or around the topmost node denoted by a variable. If we want to compute the readings represented by an underspecified description, we are not interested in such additional material – we are only interested in the way that the existing fragments are configured.

So as we did in Chapter 4 before, we don't require that a enumeration algorithm should enumerate all solutions, but only the *solved forms* of a constraint. Solved forms for normal dominance constraints are defined quite differently from the SN-solved forms from the previous chapter, but they serve the same purpose: as representations of classes of solutions that differ only in "irrelevant details". Every satisfiable normal dominance constraint has a finite number of solved forms, and every solution of the constraint satisfies exactly one of its minimal solved forms.

The defininition of a solved form of a normal dominance constraint is designed in analogy to the definition of solved forms of a dominance graph in Section 5.1.

Definition 5.6 (Solved Form). A normal dominance constraint φ is in *solved form* if it satisfies the following three properties for all variables X, Y, Z in $\mathcal{V}(\varphi)$:

1. Dominance does not branch upwards: if X and Y are distinct then not both $X \triangleleft^* Z$ in φ and $Y \triangleleft^* Z$ in φ.
2. Dominances go from holes to roots: there are no variables X, Y, Z such that both $X \triangleleft^* Z$ and some $Y{:}f(\ldots, Z, \ldots)$ are in φ.
3. The graph of φ is acyclic: $(X, X) \notin R_\varphi^+$.

We say that a constraint φ' is an *extension* of the constraint φ if φ and φ' contain the same labelling and inequality atoms, and $R_\varphi^* \subseteq R_{\varphi'}^*$. A *solved form of a normal constraint* φ is a normal constraint φ' that is in solved form and extends φ. A solved form of φ which is not a proper extension of another solved form of φ is called *minimal*.

A normal dominance constraint is *solvable* iff it has a solved form.

For example, the normal dominance constraint shown in Fig. 5.7(a) is not in solved form because it does branch upwards, i.e. it contains the dominance atoms $X' \triangleleft^* Z$ and $Y' \triangleleft^* Z$. It has exactly two solved forms, which are shown as (b) and (c).

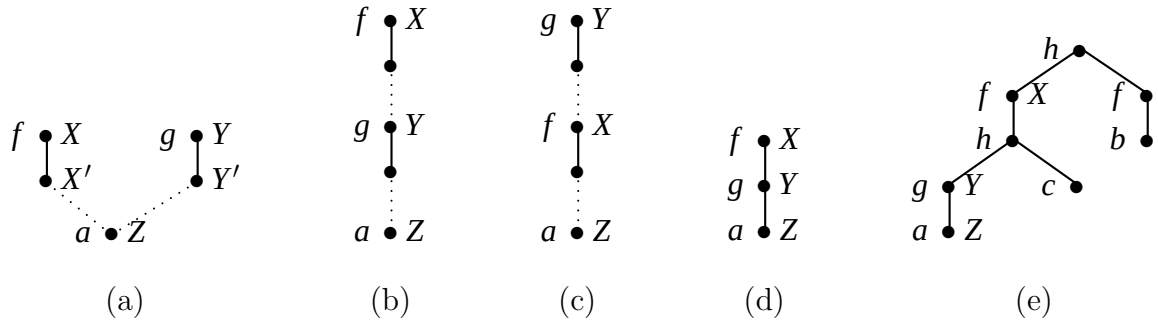


Figure 5.7: A normal dominance constraint (a) and its two solved forms (b and c), along with two solutions of the first solved form (c and d).

Each solved form is a normal dominance constraint that represents an entire set of solutions of (a). For example, two different solutions of (b) are shown in (d) and (e). While (d) is the "intuitive" solution which contains only the node labels that were mentioned in the constraint (a), there is also an infinite number of solutions like (e), in which there is more material above and around X, or additional material between the variables in a dominance atom (e.g. between X and Y). Note that the constraint $(X{:}f(Y) \wedge Y{:}g(Z) \wedge Z{:}a)^{\neq}$, which could be considered a solved form of (a), isn't a solved form of (a) because its labelling atoms contain different variables (e.g. $X{:}f(Y)$ rather than $X{:}f(X')$. In this we deviate from the definition of Althaus et al. (2003); we have changed it for this presentation to simplify the connection to solved forms of dominance graphs.

The constraints (b) and (c) are both minimal solved forms of (a), because they have incomparable reachability relations. But there are also solved forms that are non-minimal, in complete analogy to the solved forms of dominance graphs (see Fig. 5.4). Minimal solved forms are solved forms, i.e. they are guaranteed to be satisfiable (as we will show below); but they make the least possible commitments beyond this. That is, they are representatives of maximal sets of solutions. In addition, the two (minimal) solved forms in Fig. 5.7 correspond to the two readings of the natural language sentence, i.e. to the two constructive solutions of the dominance constraint. We will clarify this connection in Chapter 6.

The intuition that minimal solved forms represent classes of solutions is captured by the following lemmas: Every solution of a constraint satisfies one of its minimal solved forms (Lemma 5.9), and every solved form does have a solution (Lemma 5.8).

Lemma 5.7. *Every normal dominance constraint that contains no dominance atoms is satisfiable.*

Proof. This is obvious for a constraint whose graph is connected. Now let's say that $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_n$, where each φ_i is connected, and $\mathcal{V}(\varphi_i) \cap \mathcal{V}(\varphi_n) = \emptyset$ for any $i \neq k$; and let s_i be the ground term corresponding to a model φ_i, for each i. Let f be a symbol of arity $m \geq 2$, and let a be a nullary symbol (which we assumed to exist). Now the tree structure corresponding to the ground term $f(s_1, \ldots, s_{m-1}, f(s_m, \ldots, s_{2m-2}, f(s_{2m-1}, \ldots, s_n, a, \ldots, a)))$ is a model of φ. □

Proposition 5.8. *Every normal dominance constraint in solved form has a solution.*

Proof. Let's say that φ is a normal dominance constraint in solved form; we have to find a solution for φ. We will do this by applying Lemma 5.7. In order to do so, we successively replace all dominance atoms in the constraint by labelling atoms, in much the same way as in the proofs of Lemma 5.7 and Lemma 4.6.

The construction by which we eliminate the dominance atoms from φ is illustrated in Fig. 5.8. Let X be a hole that appears on the left-hand side of a dominance atom in φ, and let $Y_1, \ldots, Y_n$ be all roots with $X \lhd^* Y_i$ in φ. Further, let $f|_m$ be a symbol with arity $m \geq 2$ (which we assumed to exist), and let a be a nullary symbol. Now let φ_0 be the constraint obtained from φ by removing all dominance atoms out of X. We distinguish two cases:

- If $n > m$, we introduce a new variable Z and repeat the iteration with the constraint
$$\left(\varphi_0 \wedge X{:}f(Y_1, \ldots, Y_{m-1}, Z) \wedge \bigwedge_{i=m}^{n} Z \lhd^* Y_i\right)^{\neq}.$$
- If $n \leq m$, we introduce new variables $Y_{n+1}, \ldots, Y_m$ and repeat the iteration with the constraint
$$\left(\varphi_0 \wedge X{:}f(Y_1, \ldots, Y_m) \wedge \bigwedge_{i=n+1}^{m} Y_i{:}a\right)^{\neq}.$$

The result of each step in the iteration is a normal dominance constraint that has fewer dominance atoms than the constraint from the previous iteration and entails it (and hence, inductively, φ). Eventually the iteration terminates with a normal dominance constraint that entails φ and contains no more dominance atoms. This constraint has a solution according to Lemma 5.7, and because of the entailment, this is also a solution of φ. □

Proposition 5.9. *Every solution of φ also satisfies some minimal solved form of φ.*

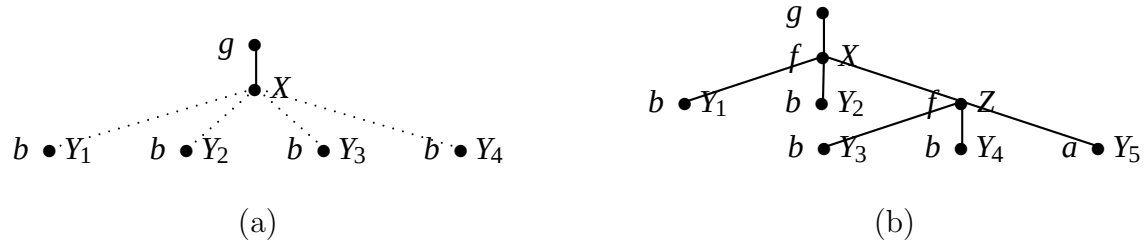


Figure 5.8: Elimination of dominance atoms from a solved form.

Proof. Let $(\mathcal{M}, \alpha)$ be a solution of φ. We construct a constraint φ' which is a solved form of φ and which is satisfied by $(\mathcal{M}, \alpha)$. (It follows that there is also some minimal solved form of φ that is satisfied by $(\mathcal{M}, \alpha)$, because every solved form extends and therefore entails some minimal solved form.) The general idea is to define a partial function hole that maps each root to the lowest hole which dominates it, according to α. Then we add dominance atoms connecting this hole and the root to φ.

Consider a root Y of φ. The partial function $\mathsf{hole} : \mathcal{V}(\varphi) \rightsquigarrow \mathcal{V}(\varphi)$ is defined on Y iff there is a hole X with $\alpha(X) \lhd^* \alpha(Y)$. We define $\mathsf{hole}(Y)$ to be the lowest such hole, i.e. the unique hole for which $\alpha(\mathsf{hole}(Y)) \lhd^* \alpha(Y)$ and for any other hole X with $\alpha(X) \lhd^* \alpha(Y)$, we have $\alpha(X) \lhd^* \alpha(\mathsf{hole}(Y))$.

Now we make φ more specific by putting the dominances between every root Y and its $\mathsf{hole}(Y)$ explicitly into the constraint. Let φ_l be the conjunction of all labelling and inequality (but not dominance) atoms in φ.

$$\varphi' = \varphi_l \wedge \bigwedge \{\mathsf{hole}(Y) \lhd^* Y \mid Y \text{ is a root for which } \mathsf{hole} \text{ is defined}\}.$$

φ' is satisfied by $(\mathcal{M}, \alpha)$, and it is a normal dominance constraint in solved form. It remains to show that it is a solved form of φ, i.e. that $R^*_\varphi \subseteq R^*_{\varphi'}$. We will do this by proving that if X is a hole and Y is a root with $\alpha(X) \lhd^* \alpha(Y)$, then $(X, Y) \in R^*_{\varphi'}$. This establishes the claim because φ and φ' have the same labelling atoms, and we know for any atom $X \lhd^* Y$ in φ that $(X, Y) \in R^*_{\varphi'}$.

We proceed by induction over the number n of holes on the path from $\alpha(X)$ to $\alpha(Y)$. This path must contain $\alpha(\mathsf{hole}(Y))$, as by the definition of hole, we know that $\alpha(X) \lhd^* \alpha(\mathsf{hole}(Y)) \lhd^* \alpha(Y)$. So if the number of holes on the path is 1, we know that $X = \mathsf{hole}(Y)$, and we are done. Otherwise, let R be the root of the maximal fragment that contains $\mathsf{hole}(Y)$. We know that $\alpha(X) \lhd^* \alpha(R)$, and the number of holes on this path is smaller than n (it doesn't contain $\mathsf{hole}(Y)$). So by induction hypothesis, $(X, R) \in R^*_{\varphi'}$. But of course $(R, \mathsf{hole}(Y))$ and $(\mathsf{hole}(Y), Y)$ are also in $R^*_{\varphi'}$, which proves the claim. □

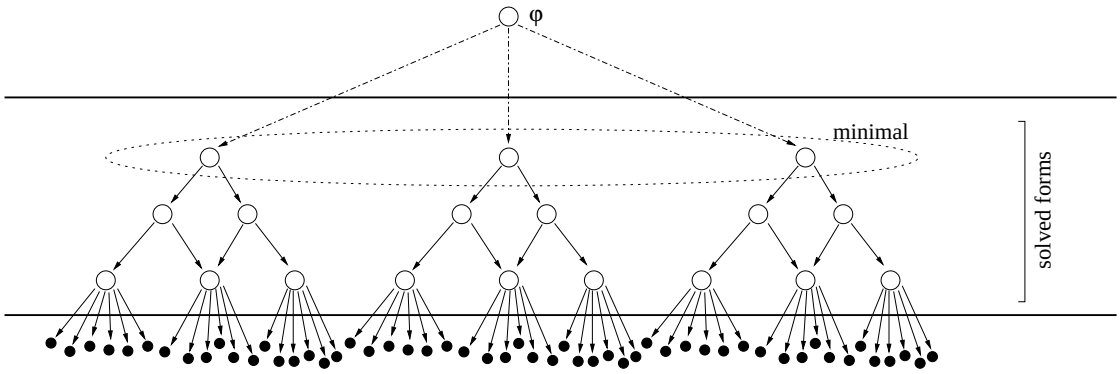


Figure 5.9: Solved forms and solutions of one constraint φ. Empty circles indicate constraints; filled circles indicate solutions.

In fact, every solution of φ satisfies *exactly one* minimal solved form of φ. We don't have the tools yet to prove this conveniently, but we will show it in Corollary 5.25. This clarifies the structure of the solved forms and solutions of a normal dominance constraint as shown in Fig. 5.9. The (satisfiable) constraint φ has a finite number of minimal solved forms. They are characterised as the minimal elements of the extension order among all solved forms, but have the additional property that every solved form extends exactly one minimal solved form. We have nothing specific to say about the structure of the extension order, except that it has a set of maximal elements, which is typically exponentially larger than the set of minimal solved forms. Each maximal solved form is still satisfiable, and has an infinite number of solutions. Thus the solved forms successively split up the solution space of a normal constraint, and the minimal solved forms have the special status of splitting even the set of solved forms into disjoint parts.

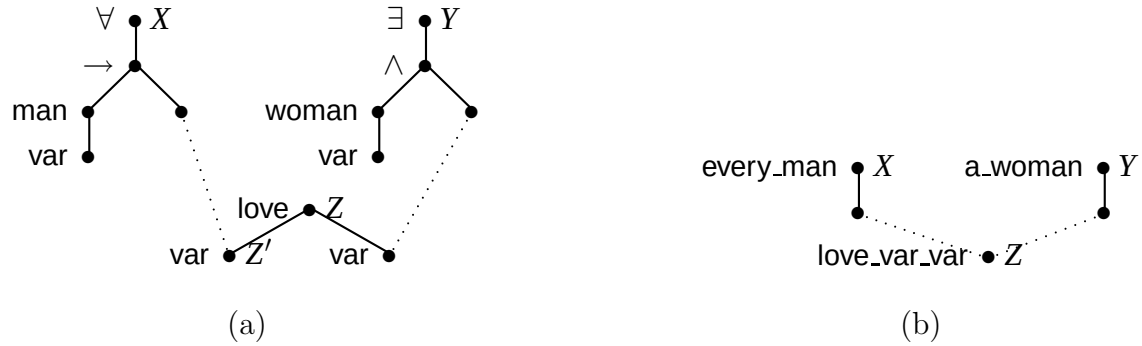
The dotted arrows pointing from φ to the minimal solved forms stand for some mechanism of enumerating the minimal solved forms of a constraint. We will fill in this part of the picture in Fig. 5.13.

Proposition 5.10. *A normal dominance constraint is satisfiable iff it is solvable.*

Proof. Follows immediately from Lemmas 5.8 and 5.9. □

5.4 Compact Dominance Constraints

While normal dominance constraints are the class of constraints that we want to use in applications (e.g. the underspecified semantic descriptions our grammar



(a) (b)

Figure 5.10: A normal dominance constraint which is not compact (a) and its compactification (b).

computed), we will only state the encoding of constraints into graphs for *compact* dominance constraints. This is essentially the class in which every fragment has depth one, and every variable is therefore either a hole or a root. We will show below that every normal dominance constraint can be *compactified* into a compact dominance constraint with the same solutions, so by this detour we will still connect normal constraints and dominance graphs.

Definition 5.11 (Compact Dominance Constraints). A *compact* dominance constraint is a normal dominance constraint in which every variable occurs in exactly one labelling atom, and for every dominance atom $X \triangleleft^* Y$, Y is a root.

For instance, the normal dominance constraint shown in Fig. 5.7 is compact, as all of its fragments have depth one or less. The constraint shown in Fig. 5.10(a) is not compact. But it is easy to see that if we replace each fragment with a single labelling atom and move the dominances up to the roots (as in Fig. 5.10(b)), the constraint becomes compact, and the solutions stay essentially the same – in other words, because we have overlap-freeness, the internal structure of the fragments is irrelevant, and we can just use fragments of depth zero or one instead. (Note that the fragment at Z had depth one before, but e.g. the variable Z' was involved in two labelling constraints.) This is what it means to *compactify* a normal dominance constraint.

Definition 5.12 (Compactification). Let φ be a normal dominance constraint. We construct the *compactification* $\mathsf{Comp}(\varphi)$ of φ as follows:

1. Let $F_1, \ldots, F_n$ be the maximal fragments of φ.
2. For each $1 \leq i \leq n$, let a_i be the number of holes in F_i, and let f_i be a new constructor with arity a_i.

3. For each $1 \leq i \leq n$, let X_i be the root of F_i and $X_{i1}, \ldots, X_{ia_i}$ be the holes of F_i. Let $V_c = \{X_1, \ldots, X_n, X_{11}, \ldots, X_{na_n}\}$.

4. The compactification is

$$\begin{array}{rcl} \mathsf{Comp}(\varphi) & := & \bigwedge\{X \lhd^* \mathcal{R}(\mathcal{F}_{\max}(Y)) \mid X \lhd^* Y \text{ in } \varphi\} \\ & \wedge & \bigwedge\{X \neq Y \mid X \neq Y \text{ in } \varphi,\ X, Y \in V_c\} \\ & \wedge & \bigwedge_{1 \leq i \leq n} X_i{:}f_i(X_{i1}, \ldots, X_{ia_i}) \end{array}$$

Lemma 5.13 (Compactifications are compact). $\mathsf{Comp}(\varphi)$ *is a compact dominance constraint that can be computed in linear time from* φ*.*

Proof. Condition 1 of Def. 5.4 carries over literally, as the two constraints have the same roots. Conditions 2 and 4 are entailed by the stronger property of compact constraints that every variable occurs in exactly one labelling atom, which is guaranteed by the construction. Dominances go from holes to roots, which establishes the other compactness condition and Condition 3 for normal constraints. □

Proposition 5.14 (Compactification preserves solutions). *There is a one-to-one correspondence between the solutions of* φ *and the solutions of* $\mathsf{Comp}(\varphi)$*. In particular,* φ *and* $\mathsf{Comp}(\varphi)$ *are satisfiability equivalent, and have corresponding solved forms.*

Proof. We prove the first, stronger claim. Consider first a solution $(\mathcal{M}, \alpha)$ of φ. Let's say that $\tau = (V, E, L_V, L_E)$ is the tree underlying $\mathcal{M}$. We can construct a solution $(\mathcal{M}', \alpha')$ for $\mathsf{Comp}(\varphi)$ by specifying the tree $\tau' = (V', E', L'_V, L'_E)$ underlying $\mathcal{M}'$ and the assignment α' as follows:

$$\begin{array}{rcl} V' & = & V - \{\alpha(X) \mid X \in \mathcal{V}(\varphi) - V_c\} \\ E' & = & \{(v, w) \in E \mid v, w \in V'\} \\ & & \cup\, \{(\alpha(X_i), \alpha(X_{ik})) \mid 1 \leq i \leq n, 1 \leq k \leq a_i\} \\ L'_V(v) & = & \begin{cases} f_i & \text{if } v = \alpha(X_i) \\ L_V(v) & \text{otherwise} \end{cases} \\ L'_E(v, w) & = & \begin{cases} k & \text{if } v = \alpha(X_i) \text{ and } w = \alpha(X_{ik}) \\ L_E(v, w) & \text{otherwise} \end{cases} \\ \alpha' & = & \alpha|V_c \end{array}$$

τ' is a finite constructor tree: Every node except for one has exactly one father (as the only nodes which lost their fathers are the the holes, and they are now children of the roots), the old root is still there (as it is not denoted by an internal variable of a fragment), and the labelling functions are still total and respect arities (this is

vacuously true for unchanged nodes, and true by construction for the roots). Furthermore, $(\mathcal{M}', \alpha')$ satisfies $\mathsf{Comp}(\varphi)$ because it satisfies every single atom. Notice that $\mathcal{M}$ can still contain nodes that are not α-denoted by any variable; these nodes are taken over into $\mathcal{M}'$ unchanged.

Conversely, consider a solution $(\mathcal{M}', \alpha')$ of $\mathsf{Comp}(\varphi)$. We can construct a solution for φ by reversing the construction above: For each labelling constraint $X_i{:}f_i(X_{i1}, \ldots, X_{ia_i})$ we introduce nodes for the internal nodes of F_i, and replace all labels by the labels required by φ. Then we extend α' by mapping the extra variables in φ to the new nodes. □

Note that all solved forms of a compact dominance constraint are compact, as they contain the same labelling atoms.

5.5 Constraints as Graphs

We can now complete the first part of this chapter by making the connection between dominance graphs and compact dominance constraints precise. Intuitively, we can obtain a dominance graph by drawing the informal graph for a compact constraint and deleting all node labels. The crucial result that connects the constraint and graph perspectives is Prop. 5.17, which says that a constraint and its encoding have corresponding solved forms.

Definition 5.15 (Dominance Graph of a Constraint). Let φ be a compact dominance constraint. The *dominance graph of* φ is the graph $G(\varphi) := (\mathcal{V}(\varphi), E \dot\cup D)$ with

$$\begin{aligned} E &= \{(X, X_i) \mid X{:}f(X_1, \ldots, X_n) \text{ in } \varphi,\ 1 \leq i \leq n\} \\ D &= \{(X, Y) \mid X \lhd^* Y \text{ in } \varphi\} \end{aligned}$$

Lemma 5.16. *Let φ be a compact dominance constraint.*

1. *$G(\varphi)$ is a dominance graph.*
2. *$G(\varphi)$ and φ have the same roots and holes.*
3. *The relations R^*_φ and $R^*_{G(\varphi)}$ are equal.*
4. *φ is in solved form iff $G(\varphi)$ is.*
5. *φ' is a solved form of φ iff $G(\varphi')$ is a solved form of $G(\varphi)$.*

Proof. 1. The edge sets E and D are disjoint because the head of each labelling atom in φ is a root, and the variable on the left-hand side of a dominance atom is a hole. The graph (V, E) is a forest whose trees are the maximal fragments of φ (Lemma 5.5). The height of each tree in this forest is at most one because φ is compact.

2. We just showed this.

3. The two relations are transitive closures over the same set.

4. Let φ be in solved form. All nodes in $G(\varphi)$ have at most one incoming edge: This is by definition for holes, and it is true for roots because φ is in solved form. $G(\varphi)$ is also acyclic because $R^+_\varphi = R^+_{G(\varphi)}$, and the former contains no elements of the form (X, X) . The converse direction is obvious.

5. This is an immediate consequence of the previous statements and the definitions of solved forms. Note that $G(\varphi')$ is well-defined because solved forms of compact constraints are compact.

□

We can use these basic properties to prove that a compact dominance constraints and its corresponding dominance graph always have the same solved forms.

Proposition 5.17. *Let φ be a compact dominance constraint. Then there is a one-to-one correspondence between the solved forms of φ and the solved forms of $G(\varphi)$, which is given by the mapping $\varphi' \mapsto G(\varphi')$.*

Proof. One direction follows immediately from Lemma 5.16 (5): If φ' is a solved form of φ, then $G(\varphi')$ is a solved form of $G(\varphi)$.

Conversely, let G' be some solved form of $G(\varphi)$. G' contains the same trees as $G(\varphi)$, each of which encodes a labelling atom in φ, plus some dominance edges. So if we take φ' to be the conjunction of all labelling and inequality atoms from φ and the dominance atoms $X \lhd^* Y$ for each dominance edge (X, Y) in G', we know that $G' = G(\varphi')$. Applying Lemma 5.16 (5) in the other direction, it follows that φ' must be a solved form of φ. □

Note that we have to assume here that different solved forms are encoded as different dominance graphs. This is true if we consider the graphs themselves, and not their isomorphy classes, i.e. solved forms such as (b) and (c) in Fig. 5.4 are still different because W is above Y in one graph and below in the other.

Corollary 5.18. *The solvability problems of compact dominance constraints and dominance graphs are linear time equivalent.*

Proof. We just reduced solvability of compact dominance constraints to solvability of dominance graphs.

For the reverse direction, let G be an arbitrary dominance graph whose solvability we want to check. We can pick a ranked signature Σ for dominance constraints such that if G contains a root with outdegree k, Σ contains a symbol with arity k. By assigning symbols with appropriate arities to the roots, we can reconstruct a compact dominance constraint φ with $G = G(\varphi)$. But then G is solvable iff φ is. □

5.6 Enumeration of Minimal Solved Forms

Let's take stock at this point where we are within the story of this chapter. We have solved the first problem we set out to solve: We can encode every normal dominance constraint as a dominance graph, in such a way that their solved forms correspond. This encoding consists of two smaller steps: We first compactify the normal constraint into a compact constraint (which preserves solutions and solved forms), and then we translate the compact constraint into a dominance graph (which preserves solved forms).

Taking all this together, we know that satisfiability of normal dominance constraints is equivalent to solvability of dominance graphs, and that the solved forms of a normal dominance constraint and its encoding as a dominance graph correspond. This means that if we knew how to compute the solved forms of a dominance graph efficiently, we would know how to do the same for a normal dominance constraint; and if we had a polynomial algorithm for checking solvability of a dominance graph, we could use it as a polynomial satisfiability test for normal dominance constraints.

We will start with an enumeration algorithm which computes the minimal solved forms of a dominance graph, given an subroutine which tests dominance graphs for unsolvability. For the initial presentation, we will use a naive unsolvability procedure, which simply checks the dominance graph for directed cycles. This procedure is sound (because all dominance graphs with cycles are unsolvable), but not complete (there are dominance graphs without directed cycles that are still unsolvable). However, this is sufficient to prove correctness of the enumeration algorithm. In the next section, we will then replace the cycle test by a complete unsolvability procedure, which will speed up the enumeration.

1. Make the dominance graph G irredundant, i.e. remove all dominance edges that are implied by transitivity.
2. If $\mathsf{unsolvable}(G)$, then report failure.
3. If the Choice rule is applicable to G, apply it and call the algorithm recursively for the two extended graphs.
4. Otherwise, report G as a solved form.

Figure 5.11: The enumeration algorithm.

The enumeration algorithm, shown in Fig. 5.11, is a search algorithm that successively arranges fragments which dominate a common root (see Fig. 5.12). It is initially called with the graph G whose solved forms we want to compute, and calls itself recursively with extensions of G. Whenever the algorithm reaches Line 4, it reports the current graph as a minimal solved form of G. If no recursive call ever reaches Line 4, the graph is reported to be unsolvable.

A dominance edge $d = (v, w)$ is *redundant* if there is a path from v to w in $G - d$. A dominance graph is *irredundant* if it contains no redundant edges.

The algorithm maintains the invariant that the graph in Line 3 is acyclic and *irredundant*. We call a dominance edge $d = (v, w)$ in a dominance graph redundant if there is a path from v to w in $G - d$, and we call the entire graph G irredundant if it contains no redundant edges. In order to ensure the invariant, the algorithm first eliminates all redundant edges, using the following simplification rule:

Simplification Rule 5.1 (Redundancy Elimination). *Remove all redundant dominance edges from a dominance graph.*

We can eliminate redundancy using a standard algorithm for the transitive reduction of directed graphs (Goralcikova and Koubek 1979; Simon 1988), using time $O(mn)$ for a graph with n nodes and m edges.

If the resulting graph is reported as unsolvable by the unsolvability procedure, the recursive instance of the algorithm terminates without reporting a solved form (Line 2). If the graph is reported as solvable (in particular, it has no cycles) and contains no nodes with two incoming dominance edges, it is a forest, and hence reported as a solved form. Otherwise, the algorithm picks some node with two (or more) incoming dominance edges, and arranges the two fragments dominating this node (Fig. 5.12). This *Choice rule* is the driving force behind the enumeration

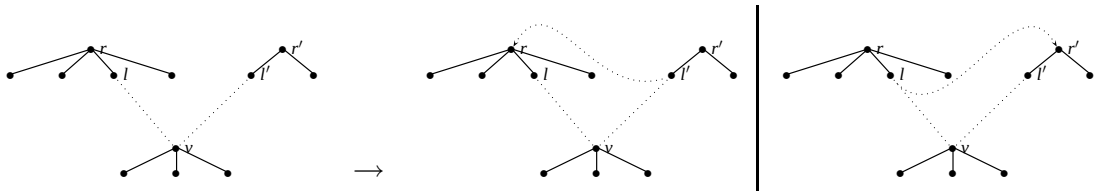


Figure 5.12: The Choice rule.

process. It can be seen as a combination of the rules (NegDisj) and (Distr.NegDisj) from Algorithm S in Chapter 4, stated in terms of dominance graphs.

Simplification Rule 5.2 (Choice). *Let v be a root with at least two incoming dominance edges (l, v) and (l', v), and let r and r' be the roots of the trees containing l and l', respectively. Generate two new graphs by adding either (l, r') or (l', r) to the dominance graph as a dominance edge.*

In order to prove the correctness of the enumeration algorithm, we must of course make some minimal assumptions about the correctness of the unsolvability procedure.

Definition 5.19 (Unsolvability Procedure). A function unsolvable is a *sound* unsolvability procedure iff whenever it claims a dominance graph G is unsolvable, G is indeed unsolvable. unsolvable is a *cyclicity complete* unsolvability procedure iff it claims that G is unsolvable at least whenever G has a cycle.

Note that we don't require that unsolvable is complete in the sense that whenever G is unsolvable, unsolvable will report it as unsolvable. It is sufficient to detect the trivial case in which G is unsolvable because it has a directed cycle: This test is complete on graphs to which the Choice rule can't be applied, so it will detect unsolvability eventually.

The correctness proof uses the concept of a *choice tree*, which is the tree of recursive call instances of the enumeration algorithm for a given graph and a given unsolvability procedure. We will also use choice trees in the proof of our main Theorem 5.29 below.

Definition 5.20 (Choice Tree). Let G be a dominance graph, and let U be an unsolvability procedure. The *choice tree* ct_G^U is the call tree of the enumeration algorithm parametrised with U on input G. We write $\mathsf{ct}_G^U(v)$ for the irredundant graph considered in node v of the choice tree.

Each node in the choice tree corresponds to a recursive call instance, and is decorated with the irredundant version of the graph which was passed to it as an

argument; in particular, the root of ct_G^U is decorated with G. The leaves of the choice tree can either be successful (i.e. the graph argument was reported as a solved form), or failed (i.e. the unsolvability procedure claimed the graph was unsolvable). Every internal node has two children, which correspond to the two graphs generated by the Choice rule.

Lemma 5.21. *Let G be a dominance graph, U any unsolvability test, and v, v_1, v_2 arbitrary nodes in ct_G^U.*

1. *$\mathsf{ct}_G^U(v_1)$ extends $\mathsf{ct}_G^U(v_2)$ if v_2 dominates v_1 in ct_G^U; otherwise the two graphs have no solved forms in common.*
2. *$\mathsf{ct}_G^U(v)$ extends G.*
3. *If v_1 and v_2 are different leaves of ct_G^U, then $\mathsf{ct}_G^U(v_1)$ and $\mathsf{ct}_G^U(v_2)$ have no solved forms in common.*

Proof. We only prove (1), as the other two statements follow trivially from it.

Applications of the Choice rule don't affect nodes and tree edges, and strictly increase the reachability relation. Redundancy elimination changes neither nodes, tree edges, nor reachability. This means that if v_2 dominates v_1, $\mathsf{ct}_G^U(v_1)$ extends $\mathsf{ct}_G^U(v_2)$.

If v_1 and v_2 are in disjoint positions of the choice tree, there must be a lowest node v that dominates both v_1 and v_2 and different children v_1' and v_2' of v such that v_1' dominates v_1, and v_2' dominates v_2. $\mathsf{ct}_G^U(v_1')$ and $\mathsf{ct}_G^U(v_2')$ were created by the same application of the Choice rule, and therefore there are two leaves l and l' for which one of the two graphs claims that l dominates l', and the other graph claims that l' dominates l, i.e. they make contradictory claims about the reachability relation. Because $\mathsf{ct}_G^U(v_1)$ and $\mathsf{ct}_G^U(v_2)$ make these same contradictory claims, they can have no solved forms in common. □

Lemma 5.22. *Let G be a dominance graph, U a* sound *unsolvability test, and S an arbitrary solved form of G. Then there is a successful leaf v of ct_G^U of which S is an extension.*

Proof. Induction over the height of ct_G^U. If the height is 0, G decorates a leaf. It is either failed, in which case G is unsolvable (due to the soundness of U) and has no solved forms; or it is a successful leaf, in which case G is in solved form and therefore S is an extension of G.

If the height n is positive, then the two children H and H' of G in the choice tree are extensions of G. They are the results of an application of Choice to G, so S is

a solved form either of H or of H'. Let's say S is a solved form of H. The choice tree for H has depth $n-1$, so it has a successful leaf v of which S is an extension, by induction hypothesis. But v is of course also a leaf in the choice tree of G. □

Lemma 5.23. *Let G be a dominance graph and U an unsolvability test which is* cyclicity complete.

1. *For every successful leaf v of ct^U_G, $\mathsf{ct}^U_G(v)$ is a graph in solved form.*
2. *The length of every path in ct^U_G is finite and in $O(|V_G|^2)$.*

Proof. 1. The graph decorating a successful leaf of ct^U_G has no node with two incoming edges (because Choice isn't applicable), and it is cycle-free (because U doesn't report it as unsolvable). This means it is a forest.

2. Because the graphs are made irredundant before the Choice rule is applied, $r_G = |R^*_G|$ increases strictly from parent to child in the choice tree. An acyclic graph G with n nodes can have at most n^2 pairs in its reachability relation, i.e. $r_G \leq n^2$. So every path in ct^U_G can contain at most n^2 acyclic graphs. But if a node v on a path is decorated with a cyclic graph, U would report v as a failure and terminate the computation on this path.

□

Proposition 5.24 (Correctness of the enumeration algorithm). *The algorithm in Fig. 5.11, parameterised with a sound and cyclicity complete unsolvability test, terminates and computes exactly the minimal solved forms of a dominance graph.*

Proof. Termination is clear from the fact that the choice tree is finite (Lemma 5.23).

Now let G be a dominance graph, and let U be a sound and cyclicity complete unsolvability test. If we pick any minimal solved form G' of G, we can show that G' decorates some successful leaf of ct^U_G. This is because G' extends $\mathsf{ct}^U_G(u)$ for some successful leaf u (Lemma 5.22), and $\mathsf{ct}^U_G(u)$ is itself in solved form (Lemma 5.23(1)). Because G' is minimal this means that the two graphs must have the same reachability relations, and therefore must be equal.

On the other hand, let v be an arbitrary successful leaf of ct^U_G. $\mathsf{ct}^U_G(v)$ is a solved form of G (Lemma 5.23(1); Lemma 5.21(2)). It is minimal, for assume it weren't, and let S' be a minimal solved form of which $\mathsf{ct}^U_G(v)$ is an extension. S' also decorates a leaf v' of ct^U_G, and v and v' are disjoint nodes. But then $\mathsf{ct}^U_G(v)$ and S' have no solved forms in common (Lemma 5.21(3)), which contradicts the fact that $\mathsf{ct}^U_G(v)$ solves both of them. □

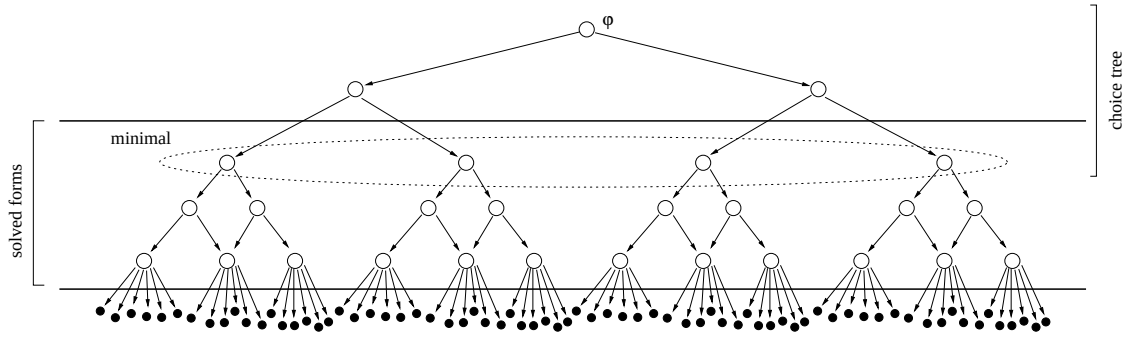


Figure 5.13: Getting from a normal dominance constraint to its minimal solved forms, and then on to the solutions.

As a further application of Lemma 5.21, we can now give the promised proof of the fact that every solution of a normal dominance constraint satisfies exactly one minimal solved form.

Corollary 5.25. *Every solution of a normal dominance constraint φ satisfies exactly one minimal solved form of φ.*

Proof. We proved existence of a minimal solved form in Lemma 5.9. For the uniqueness, consider the fact that any two different minimal solved forms of φ correspond to different minimal solved forms of $G = G(\mathsf{Comp}(\varphi))$, and therefore to different leaves of ct^U_G. As we argued in the proof of Lemma 5.21(1), different leaves of the choice tree make contradictory claims about the relative positions of encodings of two variables X and Y in φ; i.e. the two minimal solved forms must have disjoint solution sets. □

This allows us to complete the picture in Fig. 5.9 by filling in the gap between the constraint φ and the minimal solved forms. We can read the empty circles either as normal dominance constraints or as dominance graphs. The choice tree is shown in the upper part of the picture: It has the original constraint at its root, and its leaves are the minimal solved forms. Each minimal solved form is extended by a number of other solved forms. In the case of constraints (rather than graphs), each solved form is then satisfied by an infinite number of solutions, in such a way that each solution satisfies exactly one minimal solved form.

Thus we have a complete correspondence between the constraint world and the graph world, and we can apply the graph-based enumeration algorithm to compute minimal solved forms of the constraints.

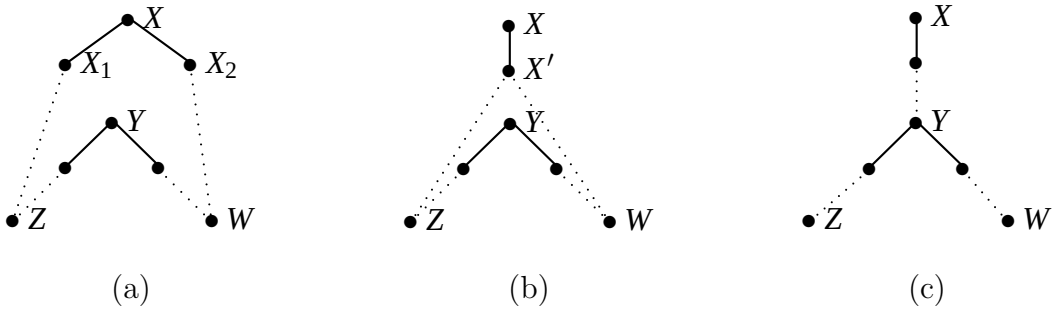


Figure 5.14: The dominance graph in (a) has a simple hypernormal cycle and is thus unsolvable. The graph in (b) has an undirected cycle which is not hypernormal; it has the solved form (c).

5.7 Hypernormal Cycles

Now we are prepared to answer the second question above, i.e. how to check a dominance graph for solvability in polynomial time. This polynomial algorithm can take the role of a very strong propagator in the context of the enumeration algorithm. Choice trees using this algorithm will be smaller than choice trees using a simple cyclicity test, because the latter can contain entire subtrees that have only failed leaves. A consequence in terms of runtime is that we can use the polynomial solvability algorithm to compute a minimal solved form in polynomial time; but of course the number of *all* minimal solved forms can still be exponential in the size of the graph.

The polynomial solvability algorithm will test whether a dominance graph contains a certain type of undirected cycle: a *simple hypernormal cycle*. We will define hypernormal cycles and prove their relation to graph solvability in this section, and then we will show how to check for the presence of a hypernormal cycle efficiently in Section 5.8.

Let's first try to get an intuition for what sort of dominance graphs is unsolvable. The simplest example is a dominance graph with a cycle: It is clear that cyclic dominance requirements can never be realised by a tree-shaped solved form. But an unsolvable graph need not contain a directed cycle. An example is the graph shown in Fig. 5.14(a). This graph is unsolvable because the fragments at X and Y cannot be arranged with respect to each other without overlap. It is acyclic, but its *undirected* version has a cycle. On the other hand, the graph in Fig. 5.14(b) is solvable – the solved form is shown in (c) –, and its undirected graph has a cycle, too. The crucial difference is that the cycle in (b) visits a fragment via a dominance edge into the hole X' and leaves it by a dominance edge, without going through

a tree edge first. The cycle in (a) enters the fragment of X by the hole X_1 and leaves it by the hole X_2, going across tree edges in between. Undirected cycles as in (a) are called *hypernormal*, and they are the type of cycle we will check for to establish unsolvability.

Definition 5.26. If G is a dominance graph, we write G_u for the undirected graph obtained by deleting the direction of all edges in G. The nodes of G_u are still partitioned into roots and holes, and the edges are still partitioned into dominance and tree edges.

A *bend* in a (directed or undirected) graph is a triple $\langle e, v, f\rangle$ of two edges e, f and a node v such that $e \circ f$ is a path whose middle node is v. *Cycles* in undirected graphs are defined as usual. A bend is *on a cycle* if its two edges appear in adjacent positions on the cycle.

A bend $\langle e, v, f\rangle$ is called *hypernormal* if either v is a root, or one of e and f is a tree edge. A *hypernormal path* is a path in an undirected dominance graph whose bends are all hypernormal. A *hypernormal cycle* is a hypernormal path which is a cycle.

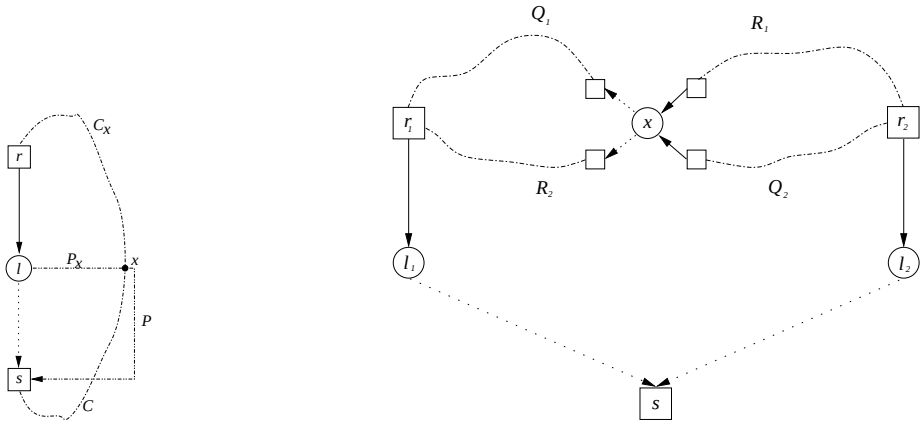
The cycle in Fig. 5.14(a) is hypernormal, because all of its bends are hypernormal: for instance, the bends at X and X_1 each use a tree edge, and the bend at Z is at a root. On the other hand, the cycle in (b) is *not* hypernormal because the bend at X' is not hypernormal.

We will now prove that a dominance graph is unsolvable iff its undirected version has a simple hypernormal cycle, i.e. a hypernormal cycle that visits no node twice. This proof will go by induction over the structure of the choice tree using the naive cyclicity test. So I will first show that the two simplification rules used in the enumeration algorithm from Section 5.6 preserve the presence of hypernormal cycles, and then we will put the pieces together in the main theorem. The structure of this proof follows (Thiel 2004), and simplifies the original proof from (Althaus et al. 2003).

Lemma 5.27. *Let the dominance graph G' be the result of applying Simplification Rule 5.1 (Redundancy Elimination) to G. Then G_u has a simple hypernormal cycle iff G'_u has one.*

Proof. If G'_u has a simple hypernormal cycle, then G_u also has one, as it contains all edges that are in G'_u.

Conversely, let's say that G_u has a simple hypernormal cycle C, and that redundancy elimination has removed the edge $e = (l, s)$ from G. Because C is hypernormal, the bend at the leaf l is also hypernormal, which means that C also uses the



(Lemma 5.27) (Lemma 5.28)

Figure 5.15: Construction of the hypernormal cycles in the proofs of Lemmas 5.27 and 5.28. Squares indicate roots, circles indicate holes.

tree edge $\{r, l\}$ into l. So let's say that C starts with $\{r, l\}$ and uses $\{l, s\}$ as its second edge.

Since e is redundant in G, there is a simple directed path P from l to s in G'. Let x be the last node on C which is also visited by P; we know $x \neq l$, but x may be equal to s. Let P_x be the prefix of P from l to x. If $x = r$, then $P_x \circ (r, l)$ is a simple directed cycle in G', which translates to a simple hypernormal cycle in G'_u.

So assume $x \neq r$; this means that the suffix C_x of C from x to r is not empty (see Fig. 5.15). Identifying P_x with the corresponding undirected path in G'_u, we obtain the simple cycle $C' = \{r, l\} \circ P_x \circ C_x$. Since $x \neq l$, we can conclude that C' (and in particular, C_x) does not use the edge $\{l, s\}$, and hence it is a cycle in G'_u.

It is obvious that the bends at r and l and all bends on C_x are hypernormal. All bends on P_x must also be hypernormal, because P_x is a directed path. Furthermore, the bend at x is also hypernormal, as it is either a root, or the directed path P_x enters it via a tree edge. This means that C' is also a hypernormal cycle, which concludes the proof. □

Lemma 5.28. *Let H and H' be the two dominance graphs computed by Simplification Rule 5.2 (Choice) from G. Then G_u has a simple hypernormal cycle iff both H_u and H'_u have simple hypernormal cycles.*

Proof. Again, it is clear that if G_u has a simple hypernormal cycle, then both H_u and H'_u have a (the same) simple hypernormal cycle too.

For the converse direction, let's assume that G contains the dominance edges (l_1, s) and (l_2, s), and that we created H_u and H'_u by adding the edges (l_1, r_2) and (l_2, r_1), respectively, where each r_i is the root of the fragment containing l_i. Let's say further that both H_u and H'_u have simple hypernormal cycles, C_1 and C_2. We can assume that the two cycles use the new edges, otherwise the proof is trivial. We can decompose the cycles into $C_1 = \{r_2, l_1\} \circ \{l_1, r_1\} \circ P_1$ (in H_u) and $C_2 = \{r_1, l_2\} \circ \{l_2, r_2\} \circ P_2$ (in H'_u). Because C_1 and C_2 are simple, we know that P_1 avoids l_1, and P_2 avoids l_2.

Now we distinguish the following cases.

1. P_1 or P_2 visits s. Then we can construct a simple hypernormal cycle in G_u as follows. Suppose $P_1 = P' \circ P''$ such that P' ends in s. Then

   $$C = P' \circ \{s, l_1\} \circ \{l_1, r_1\}$$

   is a hypernormal cycle because all bends in P' are hypernormal, s is a root, and we leave l_1 through a tree edge. C is simple because P_1 avoids l_1. The analogous argument applies if P_2 visits s.

2. Neither P_1 nor P_2 visits s. In this case, let x be the first node on P_1 different from r_1 that also lies on P_2. If $x = r_2$, i.e. P_1 and P_2 have no inner node in common, then $P_1 \circ P_2$ is a simple cycle. This cycle is hypernormal because all bends on P_1 and P_2 are, and the two paths are joined at the roots r_1 and r_2, so these bends are hypernormal too.

3. The final case is that neither P_1 nor P_2 visits s, and $x \neq r_2$. In this case we construct a simple hypernormal cycle by first decomposing P_1 and P_2 into $P_i = Q_i \circ R_i$, where Q_i ends at x, and R_i starts at x. Q_1 is a path from r_1 to x, and R_2 is a path from x to r_1, so $Q_1 \circ R_2$ is a simple cycle.

   If $Q_1 \circ R_2$ is also hypernormal, we are done.

   Otherwise, the bend at x must be not hypernormal, i.e. x is a leaf, Q_1 ends with a dominance edge, and R_2 starts with a dominance edge. This situation is shown in Fig. 5.15. Now we consider the cycle

   $$C = Q_2 \circ {Q_1}^{\text{rev}} \circ \{r_1, l_1\} \circ \{l_1, s\} \circ \{s, l_2\} \circ \{l_2, r_2\}.$$

   Let's prove simplicity of C first. Q_1 and Q_2 have no nodes in common, except for x. We know that Q_1 avoids l_1 and s and that Q_2 avoids l_2 and s, because each Q_i is a prefix of the respective P_i. It remains to show that Q_1 avoids l_2 and that Q_2 avoids l_1; we show the case of Q_1 and l_2, the other case follows by analogy. So let's assume that even P_1 visits l_2. Because the bend of P_1

at l_2 is hypernormal, P_1 must either enter or leave l_2 through the tree edge $\{r_2, l_2\}$. But P_1 is a simple path that ends in r_2, so this must be the last edge on P_1. Q_1 is the prefix of P_1 that ends in x. Now $x \neq r_2$ as per our assumption for the third case above; and on the other hand, $x \neq l_2$ because it x is on Q_2, and Q_2 does not visit l_2. This means that l_2 is an inner node of R_1, and hence not visited by Q_1.

As for hypernormality of C, we know that all bends on Q_1 and Q_2 are hypernormal. The bend at x is hypernormal because P_2 is hypernormal and R_2 starts with a dominance edge, so Q_2 must end with a tree edge. The bends at the r_i and l_i all involve a tree edge, and s is a root. Therefore all bends on C are hypernormal, which concludes the proof.

□

Theorem 5.29 (Solvability of dominance graphs). *A dominance graph G is unsolvable iff G_u has a simple hypernormal cycle.*

Proof. Assume first that G_u has a simple hypernormal cycle. Because of the Lemmata 5.27 and 5.28, we know that every dominance graph that decorates any node of ct_G^{cyc}, where *cyc* is the cyclicity test, has a simple hypernormal cycle. Now let v be any leaf of the choice tree. $\mathsf{ct}_G^{cyc}(v)$ contains no node with two incoming dominance edges, which means that the simple hypernormal cycle which its undirected version contains can never change directions. Hence, it contains a directed cycle, and is thus reported as a failed leaf by the enumeration algorithm. But because of Prop. 5.24, we know that a graph whose choice tree has only failed leaves is unsolvable.

Conversely, assume that G_u doesn't have a simple hypernormal cycle. Consider again the choice tree ct_G^{cyc}. Because the two simplification rules also preserve the *absence* of a hypernormal cycle, we know that at least one child of each node in ct_G^{cyc} that doesn't have a simple hypernormal cycle doesn't have a simple hypernormal cycle either. The choice tree has only finite paths, so it must have some leaf v whose graph G' doesn't have a simple hypernormal cycle, and therefore no directed cycle either. But this means that G' is in solved form, and therefore solves G. □

The simplicity condition in the theorem and the lemmas is essential. This is illustrated by the example in Fig. 5.16. The dominance graph (a) has a non-simple hypernormal cycle, which starts at A_1, traverses the left-hand cycle via B, F, and C, then moves via A_1 and A to A_2, then traverses the right-hand cycle via D, G, and E, and then returns to A_1 via A_2 and A. The cycle is not simple because

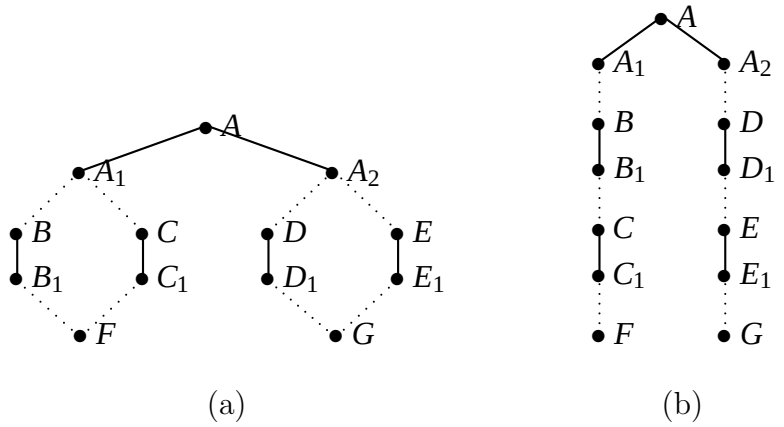


Figure 5.16: A graph which has a non-simple hypernormal cycle, but no simple one (a), and one of its solved forms (b).

it visits the nodes A, A_1, and A_2 twice. Indeed, the graph does not contain any simple hypernormal cycle, and it does have a solved form, shown in (b).

The definition of hypernormal cycles in (Althaus et al. 2003) is subtly different in that it requires that a hypernormal cycle may contain no two dominance edges that emanate from the same hole; these two dominance edges can be anywhere in the cycle, and need not be in adjacent positions, i.e. the cycle in Fig. 5.16 would not count as hypernormal. Let's call this definition *graph hypernormality*, and the one from Def. 5.26, *path hypernormality*. The two notions coincide in the case of simple cycles. Graph hypernormal cycles always have simple graph hypernormal sub-cycles, which is why Althaus et al. (2003) could state Theorem 5.29 for *arbitrary* and not just simple cycles. From this perspective, the example in Fig. 5.16 illustrates that there are path hypernormal cycles that have no simple path hypernormal sub-cycles.

We have still chosen to state the theorem in terms of path hypernormal cycles because it simplifies the definitions and proofs, and is more directly tested by the hypernormality algorithm in the next section. However, note the following lemma, which can be used to relax the application condition of Theorem 5.29. An edge-simple path in a graph is one that uses each *edge* only once.

Lemma 5.30. *If an undirected dominance graph has an edge-simple hypernormal cycle, it also has a simple hypernormal cycle.*

Proof. Let C be an edge-simple hypernormal cycle. We can assume that C contains no roots twice, as we can cut out subcycles that join the main cycle at a root without destroying the hypernormality. But such a cycle cannot contain a hole twice either,

for assume that v is a hole that appears twice on C. At most one tree edge on C can be incident to v (as only one tree edge is incident to v in G, and C is edge-simple). This means that all other (at least three) edges on C that are incident to v must be dominance edges, which contradicts the assumption that C is hypernormal. □

5.8 Testing for Hypernormal Cycles

The keystone of this chapter is a quadratic algorithm that tests a dominance graph for hypernormal cycles, which we will define now. We reduce the problem of detecting hypernormal cycles to a matching problem in an auxiliary graph. Recall the following basic definitions from matching theory.

Definition 5.31. A *matching* M in a graph G is a set of edges of G such that every node of G is incident to at most one edge in M. M is a *perfect* matching iff every node is incident to exactly one edge in M. We call the edges in M *matching edges* and the other edges *non-matching edges*. An *alternating path* with respect to M is a simple path which alternately uses a matching edge and a non-matching edge. An *alternating cycle* is a cycle which is an alternating path.

We construct the auxiliary graph $A(G_u)$ from the undirected dominance graph G_u as follows. For every edge $e = \{v, w\}$ in G_u, $A(G_u)$ contains the nodes $e_v = (e, v)$ and $e_w = (e, w)$. $A(G_u)$ contains edges of two different types:

(a) For every edge $e = \{v, w\}$ in G_u, $A(G_u)$ contains the edge $a(e) = \{e_v, e_w\}$.

(b) For every hypernormal bend $\langle e, v, f\rangle$ in G_u, $A(G_u)$ contains the edge $b(\langle e, v, f\rangle) = \{e_v, f_v\}$.

The edges of type (a) form a perfect matching M in $A(G_u)$, and connect the two nodes corresponding to each edge. These edges are drawn as solid lines in the example graph in Fig. 5.17. The edges of type (b) encode hypernormal bends, and are drawn as dashed lines in Fig. 5.17. Note that all bends in the example are hypernormal, except for the bend at c – which is why the nodes $\{c, d\}_c$ and $\{c, f\}_c$ are not connected by an edge of type (b) in $A(G_u)$.

Now compare the simple undirected cycles in G in Fig. 5.17. The cycle (a, b, d, c, a), which is hypernormal, translates to an alternating cycle in $A(G_u)$ with respect to M. However, the cycle (c, d, e, h, g, f, c), which is not hypernormal, does not correspond to an alternating cycle in $A(G_u)$. The crucial edge of type (b) at the exclamation mark is missing to close the cycle. We can make a detour (d, b, a, c)

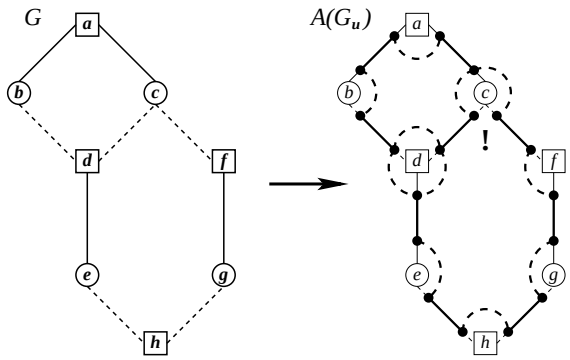


Figure 5.17: Construction of the auxiliary graph $A(G_u)$.

in $A(G_u)$ to obtain an alternating cycle again, but this again corresponds to a (different) simple hypernormal cycle in G. This suggests the following proposition.

Proposition 5.32. *The undirected dominance graph G_u contains a simple hypernormal cycle iff the auxiliary graph $A(G_u)$ contains an alternating cycle with respect to the matching M.*

Proof. Suppose first that G_u contains a simple hypernormal cycle C. Then every pair of consecutive edges on C forms a hypernormal bend. Suppose $C = e_0 \circ e_1 \circ \ldots \circ e_{n-1}$, where $e_i = \{v_i, v_{i+1}\}$ for $i = 0, \ldots, n-1$. Then $C' = a(e_0) \circ b(\langle e_0, v_1, e_1 \rangle) \circ a(e_1) \circ \cdots \circ a(e_{n-1}) \circ b(\langle e_{n-1}, v_0, e_0 \rangle)$ is an alternating cycle in $A(G_u)$.

Conversely, suppose that $A(G_u)$ contains an alternating cycle $C' = a(e_0) \circ b(\langle e'_0, v'_1, e'_1 \rangle) \circ a(e_1) \circ \ldots \circ b(\langle e'_{n-1}, v'_0, e'_0 \rangle)$. By construction of $A(G_u)$, we know that for each $0 \leq i \leq n-1$, either $e_i = e'_i$ and $e_{(i+1) \bmod n} = e'_{(i+1) \bmod n}$, or $e_i = e'_{(i+1) \bmod n}$ and $e_{(i+1) \bmod n} = e'_i$; i.e., both e_i and $e_{(i+1) \bmod n}$ are adjacent to $v'_{(i+1) \bmod n}$. This means that the sequence $C = e_0 \circ \ldots \circ e_{n-1}$ is a cycle in G_u, which is edge-simple because $A(G_u)$ is simple. Every bend in C is hypernormal because any two consecutive edges came from a type (b) edge in $A(G_u)$, which encoded hypernormal bends; so C is hypernormal. Combined with Lemma 5.30, this means that G_u has a simple hypernormal cycle. □

It remains to show how the alternating cycle can be found in polynomial time.

Proposition 5.33. *An irredundant dominance graph G_u with n nodes and m edges can be tested for the presence of hypernormal cycles in time $O(m')$, where $m' = O(m + \sum_{v \in V} \deg(v)^2) = O(nm)$.*

Proof. We first bound the size of the auxiliary graph $A(G_u)$; then we determine the runtimes of algorithms on a graph of that size.

The graph $A(G_u)$ has $n' = 2m$ nodes – two for each edge in G_u. It has m edges of type (a). For the edges of type (b), we count the number of hypernormal bends at a node v. For a leaf, we get $\mathrm{outdeg}(v)$ hypernormal bends (one bend for the incoming tree edge and each outgoing dominance edge), so the total number of type (b) edges around leaves is $O(m)$. For a root, we get $\binom{\deg(v)}{2}$ (one bend for each pair of edges incident to v). Thus the number of edges of $A(G_u)$ is

$$m' = O(m + m + \sum_{v \text{ is a root}} \binom{\deg(v)}{2}) = O(m + \sum_{v \text{ is a root}} \deg(v)^2).$$

Now we bound the sum over the square degrees of the roots. We can assume that each root has outdegree 2 or less; otherwise we replace each root of higher arity with a small binary tree, which does not affect hypernormal cycles and increases the number of nodes and edges only by a constant factor. On the other hand, any root with indegree greater than n must have two incoming dominance edges from different leaves of the same fragment because G_u is irredundant. This situation can be recognised in time $O(m)$, and we do not need to consider it further, as the graph has an obvious hypernormal cycle in this case. So if we have $r \leq n$ roots, and the indegree of the i-th root is d_i, we have $d_i \leq n$ and $\sum_{i=1}^{r} d_i \leq m$.

The sum $S = \sum_{i=1}^{r} d_i^2$ is maximised if we set $d_1, \ldots, d_{m/n} = n$ and $d_{m/n+1}, \ldots, d_r = 0$. For consider otherwise; then the maximising assignment has indices i, k with $0 < d_i \leq d_k < n$. But then if we increase d_k and decrease d_i by 1, we still have a valid assignment for which the sum of the squares is strictly greater than S, a contradiction. Thus $S = O(m/n \cdot n^2) = O(nm)$, and hence

$$\sum_{v \text{ is a root}} \deg(v)^2 = O(nm),$$

i.e. $m' = O(mn)$.

Now we can apply an algorithm by Gabow et al. (2001) which decides whether $A(G_u)$ has an alternating cycle with respect to M in time $O(m')$. This proves the claim that G_u can be tested for hypernormal cycles in time $O(mn)$. □

Corollary 5.34 (Solvability of dominance graphs). *Solvability of a dominance graph with n nodes and m edges can be decided in time $O(mn)$. A minimal solved form can be computed in time $O(mn^3)$.*

Proof. The paths in the choice tree have a depth of at most n^2. In each node of the choice tree, we must make the graph irredundant (i.e. transitive reduction) and test it for hypernormal cycles. This takes time $O(mn)$ each. □

Corollary 5.35 (Satisfiability of normal dominance constraints). *Satisfiability of a normal dominance constraint with m atoms can be decided in time $O(m^2)$. A solved form can be computed in time $O(m^4)$.*

Proof. The dominance graph of a constraint with m atoms and n variables has n nodes and $O(m)$ edges, and of course $n = O(m)$. □

5.9 Normal Dominance and Binding Constraints

To conclude this chapter, we will show how satisfiability of certain dominance and binding constraints can also be decided in polynomial time. The crucial property of these constraints will be that the dominance part is normal, and strong enough to represent the relevant structural implications of the binding part. This will allow us to simply delete the binding atoms while preserving solutions, and run the satisfiability algorithm for normal dominance constraints.

Definition 5.36. A dominance and binding constraint φ is called *normal* iff its dominance part is normal and it satisfies, for any binding specification Λ, the following additional conditions:

5. If X is a variable in φ, then there is at most one variable Y in φ s.t. $\lambda_\Lambda(X) = Y$.

6. If $\lambda_\Lambda(X) = Y$ is in φ, then the labelling atoms $X{:}\mathsf{var}_\Lambda$ and $Y{:}b(Y_1, \ldots, Y_n)$ are in φ, where b is a binder of Λ.

If Λ is a binding specification whose scope specification is just dominance (i.e. $S = \lhd^*$), a normal dominance and binding constraint φ is called *elaborated* with respect to Λ iff for any $\lambda_\Lambda(X) = Y$ in φ, $(X, Y) \in R^*_\varphi$.

All the dominance and binding constraints generated by the grammar in Chapter 3 are normal and elaborated.

Proposition 5.37 (Deleting Binders). *Let φ be an elaborated normal dominance and binding constraint with respect to the binding specification Λ, and let φ_0 be the result of deleting all binding atoms $\lambda_\Lambda(X) = Y$ from φ. Then φ is satisfiable iff φ_0 is satisfiable. In fact, there is a one-to-one correspondence between the solutions of φ and the solutions of φ_0.*

Proof. We prove the second, stronger claim. Let $(\mathcal{L}_{\tau,\lambda}, \alpha)$ be a Λ-admissible solution of φ. Then it is clear that $(\mathcal{M}_\tau, \alpha)$ is a solution of φ_0, because it must satisfy all labelling, dominance, and inequality atoms.

Conversely, let $(\mathcal{M}_\tau, \alpha)$ be a solution of φ_0, and let

$$\lambda(v) = \begin{cases} \alpha(Y) & \text{if } \lambda(X) = Y \text{ in } \varphi \text{ and } v = \alpha(X) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

This function is well-defined because of Condition 5 in Def. 5.36. The labelling requirements of each binding atom in φ are met because of Condition 6. Each binder/variable pair is in the scope relation for S (namely, dominance) because φ is elaborated. Hence $(\mathcal{L}_{\tau,\lambda}, \alpha)$ is a Λ-admissible solution of φ. □

The proposition generalises Theorem 4.3 from (Erk et al. 2003), which said the same thing for the binding specifications Λ_λ and Λ_{ana} (see Fig. 2.3 on page 25). It could itself be further generalised to cover certain cases where Condition 5 and the elaboratedness condition aren't satisfied. This would, however, complicate the proofs, while being not very useful in practice.

5.10 Summary

In this chapter, we have shown that the satisfiability problem of *normal* dominance constraints can be decided in polynomial (quadratic) time, and a minimal solved form can be computed in polynomial (biquadratic) time as well. This result is highly relevant, as all practically interesting dominance constraints seem to be normal; this is certainly true for all constraints generated by the grammar in Chapter 3.

In defining the algorithms, it was instrumental to establish the equivalence of normal dominance constraints and dominance *graphs*, as this made it possible to bring tools from graph theory (e.g., cycles and matchings) to bear on the problem. In many cases, the graph metaphor also made it much easier to speak about properties of a constraint. We also defined the fragment of *compact* dominance constraints as a stepping stone in the encoding of normal constraints as dominance graphs.

The algorithm reported here was historically the first polynomial satisfiability algorithm for a useful fragment of dominance constraints (or, indeed, any underspecification formalism). As the runtimes in Fig. 5.18 illustrate, it is dramatically more efficient than the previous algorithms. Building upon the work reported here, even

length	solved forms	saturation	FS solver	graph solver
2	2	10	10	<10
3	5	40	30	<10
4	14	390	150	10
5	42	2070	690	30
6	132	7300	2900	110
7	429	30230	10790	400

Figure 5.18: Runtimes of the different enumeration algorithms on the pure chains of length 2–7, in milliseconds.

more efficient satisfiability algorithms have been developed. Thiel (2004) gives an insightful algorithm based on depth-first search that tests for the presence of hypernormal cycles in time $O(n + m)$, where n is the number of nodes and m the number of edges of the dominance graph. This translates to a satisfiability test for normal dominance constraints that runs in time linear in the number of atoms in the constraint. Thiel also defines an incremental algorithm for redundancy elimination and an optimised version of the enumeration algorithm in Fig. 5.11 whose recursion depth is bounded by n. This allows him to compute a minimal solved form of a graph in time $O(mn)$, i.e. in time $O(m^2)$ for a constraint.

In addition, Bodirsky et al. (2004) have defined *weakly normal dominance constraints*, which generalise normal dominance constraints by allowing that a variable can appear as the head both of a labelling and of a dominance atom. They present an enumeration algorithm which also gets by with runtime $O(m^2)$ to compute one minimal solved form for this more powerful class of constraints. Their algorithm is also a graph algorithm, but operates fundamentally differently than the one presented here.

On a conceptual level, the polynomial satisfiability algorithm explains the "miracle of the green nodes" from the previous chapter. We can modify the enumeration algorithm from Fig. 5.11 so it computes all possible results of a Choice rule application to a certain node, but then checks each resulting graph for solvability and only calls itself recursively on solvable graphs. If we read the search tree in Fig. 4.8 as a choice tree spanned by the modified enumeration algorithm, it is clear that each node and in particular each leaf corresponds to a satisfiable dominance constraint.

Chapter 6

Hypernormal Connections

The algorithm from the previous chapter is very efficient at checking whether solved forms exist, and at enumerating all minimal solved forms. We know that solved forms are satisfiable, so from a purely computational perspective, the satisfiability probem of normal dominance constraints is solved. However, from the perspective of modelling scope ambiguities, we still have one problem left: The standard assumption is that the grammar already specifies all the material that should go into a semantic representation, so we are interested in *constructive* solutions. There are linguistic theories which allow us to deliberately add semantic material in a process of *reinterpretation* (Egg 2003; Koller et al. 2000), but even these theories introduce material in a controlled, linguistically motivated fashion, and not just to make a solved form satisfied.

Not all normal dominance constraints in solved form have constructive solutions. An example is the constraint in Fig. 5.8, in whose graph the node X has four different outgoing dominance edges; these edges have to be realised by additional branching labels f in the solution. However, there are constraints in which a node has multiple outgoing dominances, and which still have constructive solutions. An example is the constraint in Fig. 5.14(b), whose solved form in Fig. 5.14(c) clearly has a constructive solution. This discrepancy begs the question: Is there a class of normal dominance constraints for which we can guarantee that every solved form has a constructive solution, and which is large enough to encompass all constraints that are needed in practice?

In this chapter, we give an affirmative answer to this question by defining the class of *hypernormally connected* dominance constraints. We prove that all solved forms of a hypernormally connected dominance constraints are *simple*, and that every simple solved form has a constructive solution. In addition, we prove that all constraints

generated by the grammar in Chapter 3 are hypernormally connected; together with further empirical evidence which we will cite, this supports our conjecture that all dominance constraints that arise in underspecification are indeed hypernormally connected.

By connecting the graph-based concept of hypernormal connectedness and the constraint-based concept of constructive satisfiability, these results further strengthen the relationship between the graph perspective and the constraint perspective. In addition, we can also use them to build a bridge between normal dominance constraints and other underspecification formalisms. We prove that the hypernormally connected fragments of dominance constraints and *Hole Semantics* (Bos 1996; Bos 2002) are equivalent; the crucial point is that the pluggings of Hole Semantics correspond to constructive solutions of dominance constraints, and we can then exploit the constructive satisfiability of hypernormally connected constraints to e.g. apply the solver from Chapter 5 to Hole Semantics.

A third class of results in this chapter concerns *chains*, a special class of hypernormally connected constraints which impose a left-to-right order on tree fragments and therefore allow us to derive additional structural properties. As a consequence of these structural properties, every binary tree is the constructive solution of exactly one *pure chain*. This induces an equivalence relation on the set of all binary trees. We derive an alternative characterisation of this relation's equivalence classes as the equality classes of a rewrite system that rotates binary trees. These results clarify the structure of a chain's solution space, and will be useful in Chapter 8.

The structure of the chapter is as follows. We will first define hypernormally connected graphs (and constraints), prove that all of their solved forms are simple, and show that simple solved forms have constructive solutions (Section 6.1). Then we will show that all constraints generated by the grammar are hypernormally connected (Section 6.2). We will then prove the equivalence of the hypernormally connected fragments of dominance constraints and Hole Semantics (Section 6.3), and conclude with the investigation of chains (Section 6.4).

6.1 Simple Solved Forms

If we look back at the examples in Fig. 5.8 and Fig. 5.14 more closely, we can observe that the deeper reason why the constraint in Fig. 5.14(b) has a constructive solution and the constraint in Fig. 5.8 doesn't is that Fig. 5.14(b) has a solved form (c) in which all holes have exactly one outgoing dominance edge. In such a case, we can obtain a constructive solution by mapping the ends of each dominance edge to the same node.

It is easy to guarantee that every hole in every solved form of a constraint φ has *at least* one outgoing dominance edge: We only need to require that every hole in φ has at least one. The property that every hole in a solved form has *at most* one outgoing dominance is a little harder to establish. The following definition captures this property in terms of dominance graphs.

Definition 6.1. A dominance graph G is called a *simple solved form* iff it is a tree and every hole has at most one outgoing dominance edge.

We will now first prove that all hypernormally connected dominance graphs (and hence, constraints) have only simple solved forms; then we will prove that simple solved forms without *open holes* have constructive solutions.

6.1.1 Solved forms are simple

One way in which we can guarantee that all solved forms of a graph are simple is to find a property that is invariant under the Choice and Redundancy Elimination rules from Chapter 5 and that will ensure that every solved form with this property is simple. If a graph has this property, every graph in its choice tree will have it too, and in particular all minimal solved forms will have it and thus be simple. Simple solved forms can't be extended (in the sense of Definition 5.2), so it follows that all solved forms are minimal and thus simple.

A property that satisfies these two conditions is that every pair of nodes is connected by a simple hypernormal path:

Definition 6.2. A dominance graph G is called *hypernormally connected* iff every pair of nodes is connected by a simple hypernormal path in G_u. A normal dominance constraint is hypernormally connected iff its graph is.

On the one hand, we have already seen that Choice and Redundancy Elimination preserve the presence of hypernormal *cycles*, and we can adapt the proofs to hypernormal paths. On the other hand, a solved form that is not simple cannot be hypernormally connected: Either the solved form has more than one connected component (i.e. it is a forest that is not a tree); or there must be two nodes that are children of the same node over dominance edges, and then these two nodes can't be connected by a simple hypernormal path either. Consider the solved form in Fig. 5.8 for illustration; no two of its leaves are hypernormally connected.

Below we make this intuitive argument precise.

Lemma 6.3. *Let G be a dominance graph, and let G' be the result of applying Redundancy Elimination (Simplification Rule 5.1) to G. Then G is hypernormally connected iff G' is hypernormally connected.*

Proof. Analogous to Lemma 5.27, except that we may have to split up the directed path P that replaces the redundant edge twice in order to make it simple – once for the prefix of the hypernormal path before it enters l, and once for the suffix of the hypernormal path after it leaves s. □

Lemma 6.4. *Let G be a dominance graph, and let G' be a result of applying Choice (Simplification Rule 5.2) to G. If G is hypernormally connected, then G' is also hypernormally connected.*

Proof. The Choice rule adds a new edge, so all hypernormal paths are still present in G'. □

Unlike the analogous lemma in the previous chapter, Lemma 6.4 only goes in one direction: If G is connected, then G' is connected. The converse direction required a substantial amount of cycle manipulation in the proof of Lemma 5.28; in particular, we extracted simple subcycles from hypernormal cycles in some cases, and the same construction wouldn't be applicable here because the start and end node of the path must remain fixed.

However, even as they are stated here, the two lemmas are sufficient to prove the following proposition.

Proposition 6.5. *All solved forms of a hypernormally connected dominance graph are simple.*

Proof. Let G be some hypernormally connected dominance graph. We prove that all *minimal* solved forms of G are simple. It follows that *all* solved forms are simple, because simple solved forms can't be extended, and therefore all solved forms are minimal.

So let $G_1, \ldots, G_n$ be the minimal solved forms of G. We know from Prop. 5.24 that each G_i can be reached from G by a sequence of Choice and Redundancy Elimination steps. By Lemma 6.3 and Lemma 6.4, we know that each G_i is hypernormally connected. But hypernormally connected solved forms are simple. □

We conjecture that the converse of Proposition 6.5 is also true, i.e. that all dominance graphs that have only simple solved forms must be hypernormally connected. This would give us a complete characterisation of those dominance graphs that have

only simple solved forms, i.e. we could establish that hypernormal connectedness is not just a sufficient, but also a necessary condition for this. The crucial question is whether the converse of Lemma 6.4 is true; if yes, the proof of Prop. 6.5 could be straightforwardly extended for the following conjecture.

Conjecture 1. *A solvable dominance graph is hypernormally connected if and only if all its solved forms are simple.*

6.1.2 Simple solved forms have constructive solutions

We turn to the proof that all constraints in simple solved form that have no *open holes* have constructive solutions.

Definition 6.6. A hole in a dominance graph is called *open* iff it has no outgoing dominance edge.

The two conditions together imply that every hole in the graph of the constraint has exactly one outgoing dominance edge. Because simple solved forms are trees, this means that we can obtain a constructive solution by identifying the endpoints of each dominance edge.

Proposition 6.7. *Every simple solved form without open holes has a constructive solution.*

Proof. Let φ be such a simple solved form. We "read off" a constructive solution for φ. Define first a partial function $p : \mathsf{Var}(\varphi) \rightsquigarrow \mathsf{Var}(\varphi)$ that maps X to Y iff $X \lhd^* Y$ is in φ. p is functional because φ is simple, and therefore no variable in φ has two outgoing dominance edges. The domain of p is the set of all holes of φ.

Let $L(\varphi)$ be the set of all variables in $\mathsf{Var}(\varphi)$ that occur on the left-hand side of a labelling atom. Now we can define a finite constructor tree $\tau = (V, E, L_V, L_E)$ and a variable assignment α as follows:

$$\begin{aligned} V &= L(\varphi) \\ E &= \{(\alpha(X), \alpha(Y)) \mid X{:}f(\ldots, Y, \ldots) \text{ in } \varphi\} \\ L_V(X) &= f \text{ if } X{:}f(\ldots) \text{ in } \varphi \\ L_E(X, Y) &= i \text{ if there is a } Z \text{ with } \alpha(Z) = Y \text{ and } Z \text{ is } i\text{-th child in } X{:}f(\ldots) \end{aligned}$$

$$\alpha(X) = \begin{cases} X & \text{if } X \in L(\varphi) \\ p(X) & \text{otherwise.} \end{cases}$$

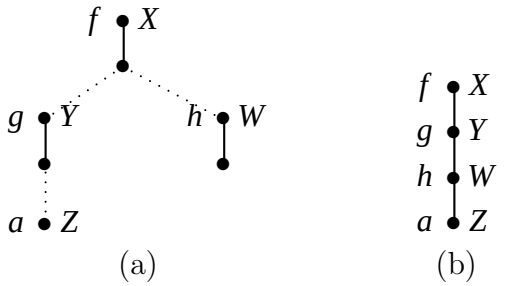


Figure 6.1: Solved forms that are not simple (a) can still have constructive solutions (b).

$\alpha : \mathsf{Var}(\varphi) \to V$ is a total function because every variable is either the head of a labelling atom (i.e. a member of $L(\varphi)$), or it is a hole and therefore occurs on the left-hand side of a dominance atom.

Because the graph of φ is a tree, every node in (V, E) has at most one incoming edge, and there are no cycles. Every hole has exactly one outgoing dominance edge, so there is exactly one root that has no incoming dominance edge; this root is the root of (V, E). Hence, (V, E) is a tree. By construction, τ also respects the arities of the labelling atoms, and so τ is a finite constructor tree.

It remains to prove that $(\mathcal{M}_\tau, \alpha)$ is a constructive solution of φ. All labelling and inequality atoms are obviously satisfied. The dominance atoms are satisfied as equalities: If $X \triangleleft^* Y$ is a dominance atom in φ, then $\alpha(X) = p(X) = Y = \alpha(Y)$. The solution is constructive because $\mathcal{M}_\tau$ only contains nodes that are denoted by labelled variables. □

The converse of Proposition 6.7 does not necessarily hold, i.e. there are non-simple solved forms with open holes that do have constructive solutions. An example is the constraint in Fig. 6.1(a), which has the constructive solution shown in Fig. 6.1(b). The general problem of deciding whether a constraint in solved form has a constructive solution is NP-complete (Althaus et al. 2003), although we just showed that the special case of simple solved forms is trivial.

We can put the propositions together in the following theorem.

Theorem 6.8. *Every solved form of a hypernormally connected normal dominance constraint without open holes has a constructive solution.*

Proof. Let φ be a hypernormally connected normal dominance constraints without open holes, and let φ' be one of its solved forms. We know from Prop. 6.5 that φ' is simple. Because *all* solved forms are simple, we also know that φ' is minimal,

so we can obtain φ' from φ by a sequence of Choice and Redundancy Elimination steps. Neither of these two operations adds new open holes, so because φ doesn't have open holes, φ' doesn't have any either. But then we can apply Prop. 6.7 to obtain a constructive solution for φ'. □

6.2 Grammars and Hypernormal Connections

The second half of the story about hypernormally connected constraints is that we claim that all dominance constraints that are actually used for scope underspecification are hypernormally connected. We will prove this for all constraints that can be generated by the grammar in Chapter 3. Then we will present evidence that supports the claim on a broader scale. Finally, we will relate our claim to an earlier, much less well-founded claim that was used to speed up a solver for context unification for the purpose of scope underspecification (Koller 1998).

Lemma 6.9. *Let $\varphi_0, \varphi_1, \ldots, \varphi_n$ be hypernormally connected constraints such that*

1. $\mathsf{Var}(\varphi_i) \cap \mathsf{Var}(\varphi_j) = \emptyset$, *for* $1 \leq i < j \leq n$,
2. $\mathsf{Var}(\varphi_0) \cap \mathsf{Var}(\varphi_i) = \{X_i\}$, *for* $1 \leq i \leq n$,

where $X_1, \ldots, X_n$ are open holes in φ_0. Then the constraint $\varphi_0 \wedge \cdots \wedge \varphi_n$ is hypernormally connected.

Proof. We prove this by induction over n. The case $n = 0$ is trivial. For the induction step, let's pick a variable X in $\psi = \varphi_0 \wedge \ldots \varphi_{n-1}$ and a variable Y in φ_n. By induction hypothesis, there is a simple hypernormal path π_1 from X to X_n in $G_u(\psi)$ and a simple hypernormal path π_2 from X_n to Y in $G_u(\varphi_n)$. The concatenation $\pi_1 \circ \pi_2$ is a path in $G_u(\psi \wedge \varphi_n)$. This path is simple because $\mathsf{Var}(\psi) \cap \mathsf{Var}(\varphi_n) = \{X_n\}$. It is hypernormal because π_1 enters X_n over a tree edge by assumption. □

The X_i will generally not be open holes (or even holes at all) in $\varphi_0 \wedge \ldots \wedge \varphi_n$; they only need to be open holes in φ_0.

Proposition 6.10. *Every constraint generated by the grammar in Fig. 3.7 is hypernormally connected if all constraints in the lexicon entries are hypernormally connected.*

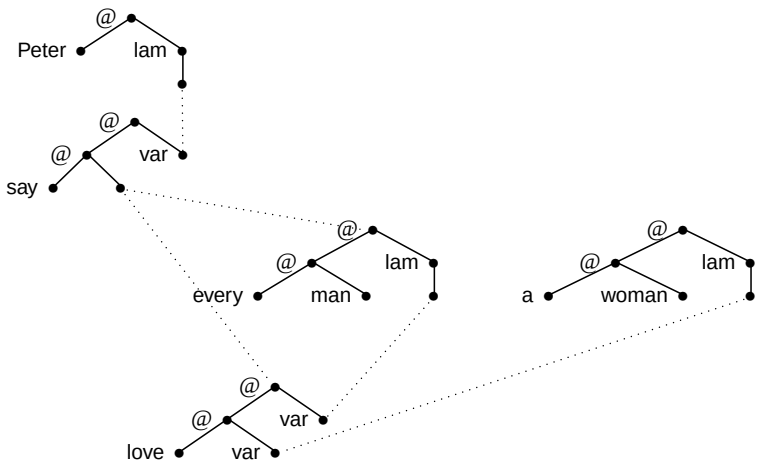


Figure 6.2: The constraint for the sentence "Peter says every man loves a woman" is not a chain, but it is hypernormally connected.

Proof. We prove this by structural induction over the syntax tree. The constraint for each leaf comes from the lexicon, so it is hypernormally connected by assumption. Now all semantics construction rules except for (b5) and (b10) introduce a new hypernormally connected constraint φ_0 and plug the constraints for the syntactic daughters into it, so they maintain hypernormal connectedness by Lemma 6.9. A similar argument works for (b5) if we consider the fact that the only way in which the S child can be expanded is via the rule (b1), and the conjunction of the constraints introduced by (b1) and (b5) is again hypernormally connected. Finally, (b10) maintains hypernormal connectedness because the interface variable of the RP_i child will always be in the same fragment as the interface variable of the S child because it will be identified with the variable X_i introduced by the trace t_i. □

As the constraints in the lexicon are generally in simple solved form in practice (and often consist of a single labelling atom), the precondition of the proposition is satisfied. Note that we said in Section 3.4, after introducing the grammar, that a more serious version of this grammar, e.g. the one from (Egg et al. 2001), would introduce additional dominance atoms to enforce island constraints. For instance, such a grammar produces the constraint in Fig. 6.2 for the sentence "Peter says every man loves a woman." The dominance edge from "say" to "every man" represents the scope island created by the sentence-embedding verb, from which the universal noun phrase may not escape. Such constraints, too, are hypernormally connected, as additional dominance atoms can never break the hypernormal connectivity. However, Fig. 6.2 is an example of a hypernormally connected constraint that doesn't fall into the more specific class of *chains*, which we will look at in Section 6.4.

For larger grammars, it is probably unfeasible to prove formally that they only generate hypernormally connected constraints. However, we still conjecture that this is the case:

Conjecture 2. *Every dominance constraint that is needed for underspecified semantics is hypernormally connected.*

Fuchss et al. (2004) evaluate the claim on all constraints that the English Resource Grammar (Copestake and Flickinger 2000), a large-scale HPSG grammar, produces for the sentences from the Redwoods Treebank (Oepen et al. 2002). It turns out that over 80% of the computed constraints are indeed hypernormally connected. The remaining constraints seem to be systematically incomplete, i.e. they represent too many semantic readings; when they are restricted to the correct set of readings, it seems they generally become hypernormally connected. So while there is no conclusive evidence at the time of writing, there is at least considerable support for the claim that the conjecture is true.

The realisation that all constraints that are needed in scope underspecification are hypernormally connected also helps justify a restriction made in (Koller 1998). The aim of this paper was to speed up a solver for *context unification*. Context unification (CU) is a restricted form of second-order unification that can also be used for scope underspecification (Niehren et al. 1997b); it is known that dominance constraints can be encoded into CU (Niehren and Koller 2001; Erk et al. 2003). There are sound and complete (but not necessarily terminating) solution procedures for context unification (Niehren et al. 1997a), but they span huge search spaces and are very slow even for rather small examples.

We obtained a dramatically faster solver (which was, however, still much slower than even the saturation algorithm from Chapter 4 on encodings of dominance constraints) by removing the following rule from the complete solution procedure:

$$\text{(Flex-Flex2)}\quad C(t)=C'(t') \longrightarrow \mathsf{true} \quad | \quad \begin{array}{l} C \mapsto \lambda Y.C_1(f(\,\pi(\overline{X}, C_2(Y), C_3(t'))\,)), \\ C' \mapsto \lambda Z.C_1(f(\,\pi(\overline{X}, C_2(t), C_3(Z))\,)) \\ \text{where } \pi \text{ is a permutation} \end{array}$$

The main effect of this rule is that it can assign a solution to unification problems such as $X = C_1(a) \wedge X = C_2(b)$ – solutions such as $X = f(a, b)$, which contain a symbol f that was not mentioned in the unification problem. Removing this rule makes the algorithm incomplete in general, but even then, there was a clear intuition that it should never be necessary to add more semantic material when resolving scope ambiguities.

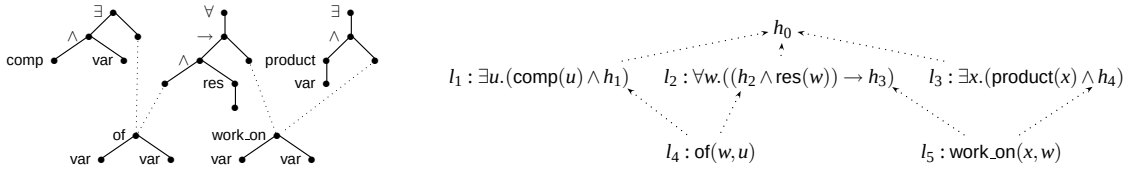


Figure 6.3: The normal dominance constraint (left) and the Hole Semantics description (right) for "Every researcher of a company works on a product."

Conjecture 2 allows us to phrase this intuition in more concrete terms. The dominance constraints that are needed in scope underspecification are hypernormally connected, so their solved forms are simple and therefore have hypernormal solutions. Hence, the relevant solutions of the CU encodings of these constraints can be computed purely by applying rules that use the present semantic material, rather than inventing new symbols.

6.3 Hole Semantics as Dominance Constraints

A second application of Theorem 6.8 is that it allows us to establish the equivalence of the hypernormally connected fragments of dominance constraints and of Hole Semantics (Koller et al. 2003). Hole Semantics (HS, Bos 1996; Bos 2002) is an alternative formalism for scope underspecification in which formulas of an object language (e.g. predicate logic or DRT) can have *holes* that can be *plugged* by other formulas.

The similarity between underspecified descriptions based on dominance constraints and on Hole Semantics is apparent (Fig. 6.3). Both formalisms specify the material of which the actual semantic representation should consist (in one case, as tree fragments; in the other, as formulas with holes), as well as constraints on the way in which this material can be combined. So it is a natural question whether this similarity is only superficial, or whether the two formalisms are related more deeply.

The answer we give here is that it is straightforward to encode HS descriptions into normal dominance constraints, and vice versa, in such a way that the solutions of the HS description correspond to the constructive solutions of the normal dominance constraint: HS requires us to plug each hole with exactly one formula, which means that e.g. constraints as in Fig. 5.8(a) are unsatisfiable. This means that the solver from Chapter 5, which computes solved forms, overgenerates in the sense of Hole Semantics; and indeed it means that the general solvability problem of Hole Semantics is NP-complete and cannot possibly be solved by a polynomial solver

(unless P=NP). By restricting ourselves to the hypernormally connected fragments, we can exploit the correspondence between solved forms and constructive solutions so the solver becomes correct for Hole Semantics.

6.3.1 Hole Semantics

Let's first define Hole Semantics. The definition we give here follows (Bos 2002), as the definition given there is closer to dominance constraints, and repairs some flaws in the definition of admissible pluggings in (Bos 1996).

Hole Semantics configures formulas of an object language (such as FOL or DRT) that have *holes* into which other formulas can be plugged. Formally, a formula with n holes is a complex function symbol of arity n, just like in the definition of dominance constraints. We assume a signature Σ of ranked function symbols, and disjoint infinite sets of *labels* and *holes*. The equivalent of a dominance constraint is an *underspecified representation* (USR).

Definition 6.11 (Underspecified Representation). An USR $U = (L_U, C_U)$ consists of a finite set L_U of labelled formulas $l{:}F(h_1, \ldots, h_n)$, where l is a label and F is an object-language formula with holes $h_1, \ldots, h_n$, and a finite set C_U of constraints. Constraints are of the form $l \leq h$, where l is a label and h a hole; this constraint means that h *outscopes* l.

As we have seen in Fig. 6.3, there is a natural way of writing USRs as graphs, which is very similar to the way we write dominance constraints as graphs. The nodes of this graph are the labelled formulas with holes, and the edges indicate the outscoping constraints.

There is again a class of underspecified representations that are considered particularly well-behaved, in the same sense that normal dominance constraints also exclude pathological cases that are never needed for underspecified semantics.

Definition 6.12 (Proper USR). An USR U is called *proper* if it has the following properties:

(P1) U has a unique *top element*, from which all other nodes in the graph can be reached.

(P2) The graph of U is acyclic.

(P3) Every label and every hole except for the top hole occurs exactly once in L_U.[1]

For example, the USR shown in Fig. 6.3 is proper; its top element is h_0.

The solutions of underspecified representations are called *admissible pluggings*. A *plugging* is a bijection from the holes to the labels of an USR. Intuitively, we "plug" every hole with exactly one formula (named by its label), and a plugging is admissible if it respects the constraints.

Definition 6.13 (P-domination). Let k, k' be holes or labels of some underspecified representation U, and P a plugging on U. Then k *P-dominates* k' iff one of the following conditions holds:

1. $k{:}F \in L_U$ and k' occurs in F, or
2. k is a hole and $P(k) = k'$, or
3. there is a hole or label k'' such that k P-dominates k'' and k'' P-dominates k'.

Definition 6.14 (Admissible Plugging). A plugging P is *admissible* for a proper USR U iff $k \leq k' \in C_U$ implies that k' P-dominates k.

6.3.2 Hole Semantics as Dominance Constraints

Now we can make the intuitive similarity between Hole Semantics and dominance constraints precise. We define encodings from Hole Semantics to normal dominance constraints and back, and show that the admissible pluggings of the USR and the constructive solutions of the dominance constraint correspond.

To keep things simple, we will present the encoding only for compact dominance constraints. This has the advantage that every variable is either a root or a hole. It is no restriction because every normal constraint can be compactified in a way that preserves constructive solutions (Prop. 5.14).

From Hole Semantics to Dominance Constraints. Assume $U = (L_U, C_U)$ is a proper USR. To obtain a compact dominance constraint φ_U that encodes the same information, we first encode every labelled formula $l{:}F(h_1, \ldots, h_n)$ as the labelling atom $l{:}F(h_1, \ldots, h_n)$. We encode every constraint $l \leq h$ in C_U as a dominance

[1] The restriction on hole occurrences is missing in (Bos 2002), but is necessary to rule out counterintuitive USRs.

constraint $h \triangleleft^* l$ – except if h is the unique top hole and does not occur as a hole in a labelled formula. Finally, we add a constraint $l \neq l'$ for every pair of different labels l, l'.

This encoding maps labels and holes to variables; labels end up as roots, and holes become holes. Because of (P3), every variable appears in exactly one labelling atom (in particular, Conditions 2 and 4 of Def. 5.4 are satisfied). Outscoping constraints go from holes to labels, so Condition 3 and the two additional conditions from Def. 5.11 are also satisfied. Finally, Condition 1 is satisfied by construction. Hence φ_U is compact.

From Dominance Constraints to Hole Semantics. Assume φ is a compact dominance constraint whose graph is acyclic. To obtain a proper USR U_φ encoding the same information, we first split the variables $\mathcal{V}(\varphi)$ into holes and labels: roots become labels, and holes become holes. This can be done in a unique way, because every variable in φ occurs exactly once in a labelling constraint, either on the left or on the right hand side.

Then we encode every labelling atom $X{:}f(X_1, \ldots, X_n)$ as the labelled formula $X{:}f(X_1, \ldots, X_n)$, and we encode every dominance constraint $X \triangleleft^* Y$ as the constraint $Y \leq X$. Finally, we add a new top hole h_0 and a constraint $l \leq h_0$ for every label l in U_φ.

It follows from the compactness of φ that U_φ is indeed an USR which satisfies the axiom (P3). (P1) is clear from the construction (h_0 is the unique top hole), and (P2) follows from acyclicity of $G(\varphi)$. Therefore U_φ is a proper USR.

This back-and-forth encoding has the following property:

Theorem 6.15. *Normal dominance constraints φ with acyclic graphs and proper USRs U can be encoded into each other, in such a way that the pluggings of U and the constructive solutions of φ correspond.*

Proof. We only show that the solutions of an USR U and its encoding φ_U correspond. The other direction is analogous, after compactifying φ if necessary.

Assume first that we have an admissible plugging P of U. If we consider the constraint $\varphi' = \varphi_U \wedge \bigwedge\{h \triangleleft^* l \mid P(h) = l\}$ and make it irredundant by applications of Simplification Rule 5.1, we get a simple solved form of φ_U without open holes, which has a constructive solution, by Prop. 6.7.

Conversely, assume we have a constructive solution $(\mathcal{M}, \alpha)$ of φ_U. We know from Lemma 5.9 that $\mathcal{M}$ satisfies some simple solved form φ' of φ_U, because $\mathsf{hole}(X)$ in the proof of the lemma is the unique root Y with $\alpha(X) = \alpha(Y)$. But then the plugging P with $P(h) = l$ iff $h \triangleleft^* l$ in φ' is an admissible plugging for U. □

Definition 6.16. We call an USR U hypernormally connected iff φ_U is chain-connected and has no open holes. We say that U has no open holes if every hole h of U appears on the right-hand side of an outscoping constraint $l \leq h$.

Corollary 6.17. *If U or φ are hypernormally connected and have no open holes, the admissible pluggings of the USR and the solved forms of the dominance constraint correspond. In particular, solvability of hypernormally connected USRs with no open holes can be decided in polynomial time.*

Proof. Follows immediately from Thm. 6.15 together with Thm. 6.8 and Corollary 5.35. □

6.4 Pure Chains

In the last part of this chapter, we will focus our view on a subclass of hypernormally connected constraints called *pure chains*. Pure chains are special cases of *chains*, which were the first known class of hypernormally connected constraints (Koller et al. 2000). The fragments in a chain are arranged in a left-to-right order which allows us to make very strong claims about the structure of their solutions.

We will first define pure chains and prove some of these basic structure lemmas. Then we will prove that every binary tree is the constructive solution of exactly one pure chain. This means that pure chains induce an equivalence relation on the set of all binary trees. Next, we will prove that subtrees and subchains correspond, and finally we will apply these results to give an alternative characterisation of the equivalence classes as the equality classes of a rewrite system.

6.4.1 Chains and their basic Properties

A chain is intuitively a normal dominance constraint that looks as in Fig. 6.4. Each triangle in the picture stands for a fragment of the constraint, and the bullets at its lower side represent holes of the fragment. The chain consists of *upper fragments* and *lower fragments*, in such a way that each lower fragment is linked to two upper fragments with dominances, and each upper fragment is linked to at most two lower fragments, by dominances going out of different holes.

Chains occur very frequently in dominance constraints that are used for underspecified semantics. This is not terribly surprising because it can be shown that every pair of variables in a hypernormally connected constraint is also connected

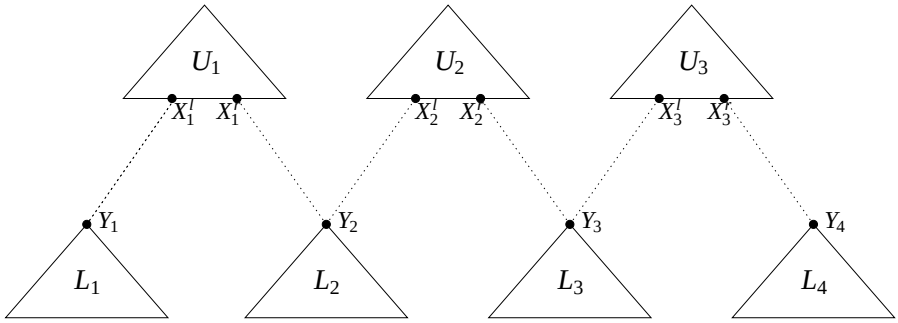


Figure 6.4: A schematic picture of a pure chain.

by a chain. But many interesting constraints even consist of a single chain; the constraint in Fig. 3.9 is an example.

Here we will focus on the special case of *pure chains*; see e.g. (Koller et al. 2000) for more details about chains in general. Pure chains are constraints that look exactly as in Fig. 6.4, i.e. the upper fragments all have exactly two holes, and they are linked to the lower fragments by dominance atoms.

Definition 6.18. A *pure chain* $\mathcal{C}$ of length n is a dominance constraint of the form

$$\left(\bigwedge_{i=1}^{n} X_i{:}f_i(X_i^l, X_i^r) \wedge \bigwedge_{i=1}^{n+1} Y_i{:}a_i \wedge \bigwedge_{i=1}^{n} (X_i^l \triangleleft^* Y_i \wedge X_i^r \triangleleft^* Y_{i+1}) \right)^{\neq}$$

The *upper fragments* of $\mathcal{C}$ are the variable sets $U_i = \{X_i, X_i^l, X_i^r\}$ for $1 \leq i \leq n$, and the *lower fragments* of $\mathcal{C}$ are the variable sets $L_i = \{Y_i\}$ for $1 \leq i \leq n+1$. We write $\mathcal{C} = (L_1, U_1, L_2, \ldots, U_n, L_{n+1})$.

It is clear that every pure chain is compact and hypernormally connected, and that the upper and lower fragments are fragments in the sense of Definition 5.3. The following lemma holds about the structure of a pure chain.

Lemma 6.19 (Structural Relations in Chains). *Let φ be a pure chain, and let $1 \leq i, k \leq n$ with $i \neq k$.*

1. *If U_i is an upper fragment and $1 < i < k$, then*

$$\varphi \models \neg\, X_i^l \triangleleft^* X_k.$$

If $n > i > k$, then

$$\varphi \models \neg\, X_i^r \triangleleft^* X_k.$$

2. The following entailment holds:

$$\varphi \wedge X_i \perp X_k \;\models\; X_i \perp X_k \text{ at } \mathcal{U}_{i,k},$$

where $\mathcal{U}_{i,k}$ is the set of all variables in the upper fragments $\{U_{i+1}, \ldots, U_{k-1}\}$ that are not holes.

Proof. 1. We prove the case for $1 < i < k$; the other direction is analogous. So let $\varphi' = \mathsf{Comp}(\varphi \wedge X_i^l \lhd^* X_k)$. Because φ is a pure chain, there must be a simple hypernormal path π from X_i^l to X_k in $G_u(\varphi)$. Hence, $\pi \circ \{X_k, X_i^l\}$ is a simple cycle in $G_u(\varphi')$. It is hypernormal because π leaves X_i^l via a tree edge. So φ' is unsatisfiable, i.e. the claimed entailment holds.

2. Let's assume w.l.o.g. that $i < k$, and let $(\mathcal{M}, \alpha)$ be an arbitrary solution of $\varphi \wedge X_i \perp X_k$. We know (Lemma 5.9) that there is some minimal solved form φ' of φ that is also satisfied by $(\mathcal{M}, \alpha)$; this solved form is simple (Prop. 6.5). Because this means that $G(\varphi')$ is a tree in which each hole has only one outgoing dominance edge, there must be some labelling atom $X_j{:}f(X_j^l, X_j^r)$ for which X_j^l and X_j^r dominate X_i and X_k in $(\mathcal{M}, \alpha)$. Because of (1), we know that these two dominances must be $\alpha(X_j^l) \lhd^* \alpha(X_i)$ and $\alpha(X_j^r) \lhd^* \alpha(X_k)$, and that $i < j < k$. This proves the claim.

□

That is, we can make rather strong predictions about the structure of a solution of a pure chain. We know that all the variables in lower fragments end up as (disjoint) leaves in a satisfying tree structure. In addition, we have just shown that upper fragments may dominate each other or they may be mapped to disjoint nodes – but if they dominate each other, they must dominate each other usingenerag the hole that faces the dominated fragment, and if they are disjoint, their branching point must be denoted by the root of one of the upper fragments between them.

We have proved Lemma 6.19 only for pure chains, but it remains true for general chains. The proof is more complex, but it hinges on the same idea of closing a hypernormal path to a hypernormal cycle to establish unsatisfiability.

6.4.2 Pure chains and binary trees

In fact, a pure chain fixes the structure of its solutions so strictly that we can turn the structure lemma around to prove that every binary tree is the constructive solution of *only one* pure chain: Every other pure chain would impose a different ordering on the nodes in the tree.

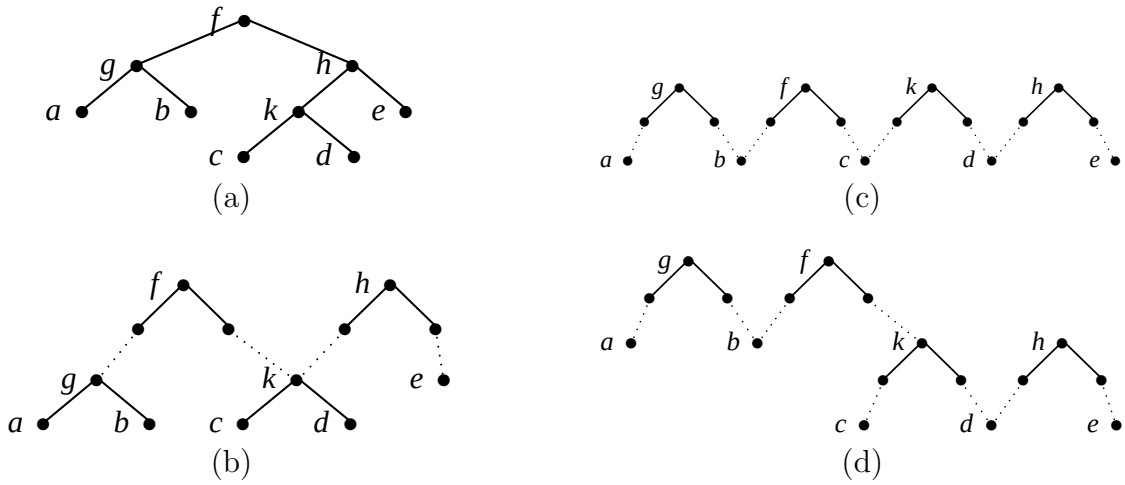


Figure 6.5: A binary tree (a) and some hypernormally connected chains that it satisfies constructively (b,c,d). Only (c) is a pure chain.

Such a claim is not true for hypernormally connected constraints in general. For instance, the binary tree in Fig. 6.5(a) is a constructive solution of the hypernormally connected constraints shown in Fig. 6.5(b), (c), and (d). Even if we require that the constraint should be compact – which excludes (b) and fixes the set of fragments –, a hypernormally connected constraints could still require dominances between upper fragments (d). In this sense, pure chains are maximally weak among all hypernormally connected constraints: They impose no structure on the set of fragments beyond the dominances that are needed to make the constraint connected.

Proposition 6.20. *Every finite binary tree is the constructive solution of exactly one pure chain (up to renaming of variables).*

Proof. Let τ be a binary tree with leaves $v_1, \ldots, v_{n+1}$ and non-leaves $u_1, \ldots, u_n$; we know that there is one more leaf than there are non-leaves because the tree is binary. Now we assign each node in τ an *index*, i.e. a position number between 1 and $n+1$. The leaves are numbered in order of their linear precedence, and the index of a leaf is its position in this order. The index of a non-leaf is the maximum index of any leaf that its left-hand child dominates. The binary tree from Fig. 6.5, together with its node indices, is shown in Fig. 6.6. It is easy to see that both each leaf and each non-leaf is assigned an index that is unique among all leaves and all non-leaves, respectively, and that the leaves get indices 1 to $n+1$ and the non-leaves get indices 1 to n.

Now let's assume w.l.o.g. that the $u_1, \ldots, u_n$ and the $v_1, \ldots, v_{n+1}$ are ordered with

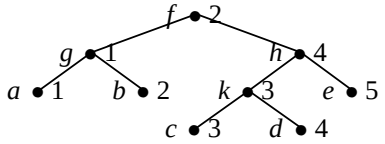


Figure 6.6: A binary tree with indices, as used in the proof of Prop. 6.20.

respect to ascending indices, i.e. if $i < j$, the index of u_i (of v_i) is smaller than the index of u_j (of v_j). Then we can define the following pure chain; I write $\tau(u)$ for the label of the node u in the tree τ.

$$\varphi := \left(\bigwedge_i^n X_i{:}\tau(u_i)(X_i^l, X_i^r) \wedge \bigwedge_{i=1}^{n+1} Y_i{:}\tau(v_i) \wedge \bigwedge_{i=1}^{n} (X_i^l \lhd^* Y_i \wedge X_i^r \lhd^* Y_{i+1}) \right)^{\neq}$$

The tree structure of τ, together with the obvious variable assignment, satisfies φ, as v_i is dominated in τ by the left-hand child of u_i by construction, and v_{i+1} is the successor of v_i in the precedence relation and therefore dominated by the right-hand child of u_i. It remains to prove that φ is the *only* pure chain (up to renaming of variables) that is constructively satisfied by τ.

It is clear that the labelling and inequality atoms in φ are unique up to variable renaming. What we need to show is that the dominance atoms that are required to connect the fragments into a pure chain are exactly the atoms of the forms $X_i^l \lhd^* Y_i$ and $X_i^r \lhd^* Y_{i+1}$. Because φ is a pure chain, there must be exactly one dominance atom $X_i^r \lhd^* Y_k$ for some k. We can conclude $k > i$ from Lemma 6.19. Now assume that we had $k > i + 1$; let i be minimal with this property. For all $s > i$ we know that $X_s^r \lhd^* Y_t$ for some $t > s$. This means that there is no j with $X_j^r \lhd^* Y_{i+1}$ in φ, as all the X_j^r to the left of X_i^r dominate Y_{j+1}, and all the X_j^r to the right of X_i^r dominate Y_l with $l > i + 1$. This is a contradiction, because there must be some dominance atom with Y_{i+1} on its right-hand side. Hence we have $X_i^r \lhd^* Y_{i+1}$ in φ, for each i. The analogous argument shows that we also have the $X_i^l \lhd^* Y_i$ in φ. □

This means that we can now say that the relation

$$\mathcal{PC} = \{(\sigma, \tau) \mid \mathcal{M}_\sigma \text{ and } \mathcal{M}_\tau \text{ constructively satisfy the same pure chain}\}$$

over the finite binary trees is an equivalence relation. Before we had Proposition 6.20, it was not clear that the relation was transitive. We can furthermore say that there is a bijection between the set of all pure chains and the equivalence classes of $\mathcal{PC}$. We will derive an alternative characterisations of the equivalence classes as equality classes of a rewrite system in Section 6.4.4.

6.4.3 Subchains and subtrees

But first, let's take a closer look at the relationship between subchains of a compact chain and the subtrees of its constructive solutions. It is relatively easy to see that if $\mathcal{M}_\tau$ is a constructive solution of the pure chain φ, and τ' is a subtree of τ, then $\mathcal{M}_{\tau'}$ is a constructive solution of some subchain of φ. Much less obvious is the converse result which we will prove next: It says that the constructive solutions of subchains are always subtrees of the constructive solutions of the complete chain. This is interesting by itself, but it will also be crucial for the proofs in Section 6.4.4.

Proposition 6.21. *Let φ be a pure chain with fragments $(F_1, \ldots, F_n)$, let $\mathcal{M}_\tau$ be a constructive solution of φ, and let τ' be a subtree of τ. Then there are indices $1 \leq i \leq k \leq n$ such that τ' is a constructive solution of φ restricted to the variables in the subchain $(F_i, \ldots, F_k)$.*

Proof. The important point of the proof is to show that the sequence of fragments denoting nodes in τ' is contiguous, i.e. if F_r and F_s belong to the sequence, then all F_j for j between r and s also belong to it. In the proof, we will again write X_i for the root of F_i and X_i^l and X_i^r for its left and right hand connecting hole, respectively.

So let $(\mathcal{M}_\tau, \alpha)$ be a constructive solution of φ, let τ' be a subtree of τ, let F_r be the upper fragment that is mapped to the root of τ', and let F_s be an arbitrary upper fragment that is mapped into τ', such that $r \neq s$. Now let F_j be a third upper fragment with j between r and s; let's assume $s < j < r$, the other case is analogous. We will argue that X_j must be dominated by X_r in $(\mathcal{M}_\tau, \alpha)$. If it weren't, then either $\alpha(X_j^r) \lhd^* \alpha(X_r)$, from which we can obtain $\alpha(X_j^r) \lhd^* \alpha(X_s)$, in contradiction to Lemma 6.19. Or $\alpha(X_j) \perp \alpha(X_r)$, in which case the branching point v of $\alpha(X_j)$ and $\alpha(X_r)$ is in a fragment between F_j and F_r. But because $\alpha(X_r) \lhd^* \alpha(X_s)$, v is also the branching point of $\alpha(X_s)$ and $\alpha(X_j)$, which must be between F_s and F_j by Lemma 6.19, a contradiction.

So we know that there are numbers $i' \leq k'$ such that the set of upper fragments denoting nodes in τ' is the set of upper fragments between $F_{i'}$ and $F_{k'}$. All lower fragments between $F_{i'-1}$ and $F_{k'+1}$ are also mapped into τ'. It remains to show that no lower fragments outside of this range are mapped into τ'. Assume there is an index $1 \leq j < i' - 1$ such that F_j is a lower fragment and $\alpha(X_r^l) \lhd^* \alpha(X_j)$. (The case $j > k' + 1$ is analogous.) Then X_{j+1}, the root of the adjacent upper fragment, must dominate X_r, as it is not itself dominated by X_r (by assumption), and cannot be disjoint because X_j is below X_r. But then $\alpha(X_{j+1}^r) \lhd^* \alpha(X_r)$ and $\alpha(X_{j+1}^l) \lhd^* \alpha(X_j)$, so $\alpha(X_r^l)$ and $\alpha(X_j)$ are disjoint, a contradiction. □

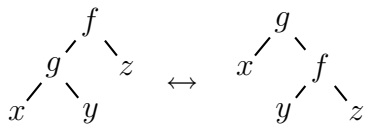


Figure 6.7: Schematic picture of the rotation rules.

6.4.4 A rewrite system for chains

We can use results on subchains and subtrees to characterise the equivalence classes of $\mathcal{PC}$ as the equality classes of a rewrite system. This rewrite system is of independent interest as a tool for rewriting readings of a chain into each other, and we will use it this way in Chapter 8.

The rewrite system we will use is defined as follows; Fig. 6.7 illustrates the definition. We switch freely between tree structures, trees, and ground terms here to keep the presentation transparent.

Definition 6.22. The *right rotations* and *left rotations* over the signature Σ are defined as follows:

$$\begin{array}{lll} f(g(x,y),z) & \to_{rrot} & g(x,f(y,z)) \\ g(x,f(y,z)) & \to_{lrot} & f(g(x,y),z), \end{array}$$

for all $f|_2, g|_2 \in \Sigma$.

The union of the two rewrite systems $\to_{rot} = \to_{rrot} \cup \to_{lrot}$ is called the rewrite system of *rotations*.

Proposition 6.23. *Let φ be a pure chain. If σ is a constructive solution of φ, and if $\sigma \to_{rot} \tau$, then τ is also a constructive solution of φ.*

Proof. We prove the direction $\sigma \to_{rrot} \tau$; the other direction is symmetrical. Let $\sigma = C[\sigma']$ and $\tau = C[\tau']$, where C is the context under which the rule $\to_{rrot}$ is applied. According to Lemma 6.21, there is a subconstraint φ' of φ that is also a pure chain and that is satisfied by σ'. Let's say that the root of σ' is denoted by X_f, and its left child is denoted by X_g, and that we have labelling atoms $X_f{:}f(X_f^l, X_f^r)$ and $X_g{:}g(X_g^l, X_g^r)$ in φ, for some f and g. Finally, let's say that the roots of the subterms x, y and z are denoted by the variables R_x, R_y, and R_z, respectively.

The fragments F_f and F_g containing X_f and X_g must be upper fragments in the chain, because they couldn't dominate R_y otherwise. It is sufficient to prove that τ' is also a constructive solution of φ', as all other atoms in φ remain satisfied.

τ' satisfies all labelling atoms in φ', and it satisfies all dominance atoms that σ' satisfies, except possibly for the following atoms:

- $X_g^r \triangleleft^* R_z$: but F_g must appear to the left of F_f in the chain (Lemma 6.19), so such a dominance atom can't be entailed by the chain;
- $X_f^l \triangleleft^* X_g$: but a pure chain never entails a dominance between upper fragments.

This means that τ' is also a constructive solution of φ'. □

Prop. 6.23 is not generally true for compact chains; a counterexample is the compact chain in Fig. 6.5(d), which has a solution which contains the subtree $f(b, k(c, d))$, but no constructive solution which contains the subtree $k(f(b, c), d)$. The problem here is that this chain *does* entail a dominance between the variables with labels f and k.

Proposition 6.24. *Let σ, τ be constructive solutions of the pure chain φ. Then there is a sequence of rewrite steps $\sigma \to_{rot}^* \tau$.*

Proof. Induction over the length n of the chain φ. The base case is trivial, for $n = 1$ means that φ has only one constructive solution.

Now let φ be of arbitrary length n, and let F_i be the fragment at the root of τ. There is a (possibly empty) sequence $\sigma \to_{rot} \ldots \to_{rot} \sigma'$ that moves F_i to the top of the tree $\sigma' = f_i(\sigma'_1, \sigma'_2)$. Let's say that $\tau = f_i(\tau_1, \tau_2)$.

σ' is still a constructive solution of φ by Prop. 6.23, so by Lemma 6.21, the two subtrees σ'_1 and σ'_2 are constructive solutions of two disjoint pure subchains φ_1 and φ_2 of φ. Each σ'_j contains the same fragments as the respective τ_j, namely the ones to the left of F_i and the ones to the right of F_i, respectively. So τ_1 is also a constructive solution of φ_1, and τ_2 is a constructive solution of φ_2. But φ_1 and φ_2 both have length strictly less than n, so by the induction hypothesis, $\sigma'_1 \to_{rot}^* \tau_1$ and $\sigma'_2 \to_{rot}^* \tau_2$. □

So we can characterise the set of constructive solutions of a pure chain in terms of the rewrite system. This leads immediately to the following corollary, which tells us what, exactly, the equivalence classes that pure chains induce on the set of finite binary trees look like.

Corollary 6.25. *The equivalence classes of $\mathcal{PC}$ and the equality classes of $\to_{rot}$ are identical.*

6.5 Summary

In this chapter, we have investigated hypernormally connected constraints. The key property of hypernormally connected constraints is that all their solved forms have constructive solutions. This closes the gap between the linguistic modelling of scope ambiguities, which focuses on constructive solutions, and the efficient algorithm from the previous chapter, which computes solved forms. In addition, the hypernormally connected fragments of dominance constraints and of Hole Semantics are equivalent. Finally we investigated pure chains, a very specific kind of hypernormally connected constraints that impose a left-to-right order on their fragments. Every binary tree is the constructive solution of exactly one pure chain, and the equivalence classes thus induced can also be characterised as the equality classes of a certain rewrite system.

The work on hypernormally connected constraints originally stems from research on chains, which were first introduced to make structural statements about the solutions of a constraint; the paper (Koller et al. 2000) contains a structure lemma very similar to Lemma 6.19. Chains were generalised to *chain-connected* dominance constraints in (Koller et al. 2003), which proved that the chain-connected fragments of dominance constraints and Hole Semantics are equivalent. This result is reported here as Corollary 6.17, but the proof presented here is much simpler. A normal dominance constraint is chain-connected iff it is hypernormally connected, but they are defined in rather different terms, and the notion of hypernormal connectedness is not restricted to normal constraints. Finally, Niehren and Thater (2003) introduced the notion of *nets* to establish an encoding from MRS (Copestake et al. 1999) to dominance constraints. Nets are special weakly normal dominance constraints; we strongly believe that all nets are hypernormally connected, but have no detailed proof for this yet.

The research in this chapter is relevant first of all because it closes the gap between modelling and computation. The intuition that the discrepancy between solved forms and constructive solutions should never cause problems in practice had been around for quite a while; the simultaneous proof that hypernormal connectedness guarantees constructive satisfiability and is satisfied by the constraints generated from grammars makes this intuition concrete. The solved forms still have nonconstructive solutions, so the mechanisms for reinterpration developed in (Egg 2003; Koller et al. 2000) can still be applied; but the addition of more material just to satisfy the solved form is never *necessary*.

At the same time, this discrepancy is the key difference between dominance constraints and other similar underspecification formalisms, most notably Hole Semantics and MRS. Because the discrepancy can be resolved for hypernormally

connected underspecified descriptions, we have built the first ever bridge between practically useful underspecification formalisms. An immediate benefit of the encoding between the formalisms is that we can share resources. For instance, there are large-scale HPSG grammars that can generate MRS descriptions (Copestake and Flickinger 2000); on the other hand, users of Hole Semantics and MRS can now use the constraint solvers for normal dominance constraints. A comparison by Fuchss et al. (2004) shows that dominance constraint solvers can outperform a native solver for MRS by several orders of magnitude.

It will probably remain unfeasible to give a formal proof that all constraints generated by a grammar are hypernormally connected for grammars that are larger than the small one from Chapter 3. However, an empirical study by Fuchss et al. (2004) for the large-scale English Resource Grammar (Copestake and Flickinger 2000) provides substantial evidence that all correct underspecified descriptions should indeed be hypernormally connected (or, as they literally claim, nets). The evidence is so persuasive that one might even consider hypernormal connectedness as a well-formedness condition on the semantic outputs of a grammar: A sentence to which the grammar assigns an underspecified semantic description that is not hypernormally connected could be interpreted as a warning sign for an error in the semantics construction component. A further evaluation of the hypernormal connectedness hypothesis is an exciting issue for further research.

Finally, the results on pure chains are the first ever theorems about the structure of the solution set of a dominance constraint. One consequence is that we can now make claims about what sets of trees can be described as constructive solutions of one constraint, in a spirit that is similar but more application-oriented than the perspective taken by Ebert (2003). But another important detail is that any two pure chains have disjoint solution sets. Because it is also easy to enumerate all their solved forms, this makes them a candidate for being the outputs of a new kind of dominance constraint solver. Such a solver might only enumerate a set of chains that partition the solution set of a (perhaps hypernormally connected) constraints, rather than all minimal solved forms. Whether this new type of solver would be faster than the existing ones is an open question, but certainly one that merits further research.

Chapter 7

Resolving Scope Ambiguities Using Anaphora

At this point in the thesis, we have developed the formalism of dominance constraints into a powerful tool for scope underspecification. We have seen how to compute constraints as underspecified semantic representations of English sentences, we have seen how to enumerate all their minimal solved forms efficiently, and we have seen that the minimal solved forms correspond to the constructive solutions, and hence to the readings of the ambiguous sentence.

A formalism that allows us to represent all readings of a sentence compactly, and to enumerate all readings by need, is certainly useful because it saves memory, gives us control over when we want to spend time on enumeration, and makes semantics construction simpler. However, this is only half the story of underspecification. When we motivated it in the introduction, we claimed that underspecified representations could be enriched with external information, so as to eliminate readings that are theoretically possible but not intended in the specific context in which the sentence was uttered.

In the final two chapters of the thesis, we will model two sources of such information – anaphora and world knowledge – that can be used to disambiguate scope ambiguities. Both models reduce the set of constructive solutions of a constraint to the set of those constructive solutions that are compatible with the external information. We will then apply the results from the earlier chapters to show how the set of compatible constructive solutions can be *computed*. Applied to constraints, our algorithms will be able to propagate away some (but not necessarily all) of the incompatible readings. Applied to solved forms, they will be complete in the sense that they detect whether the (unique) constructive solution is compatible or not.

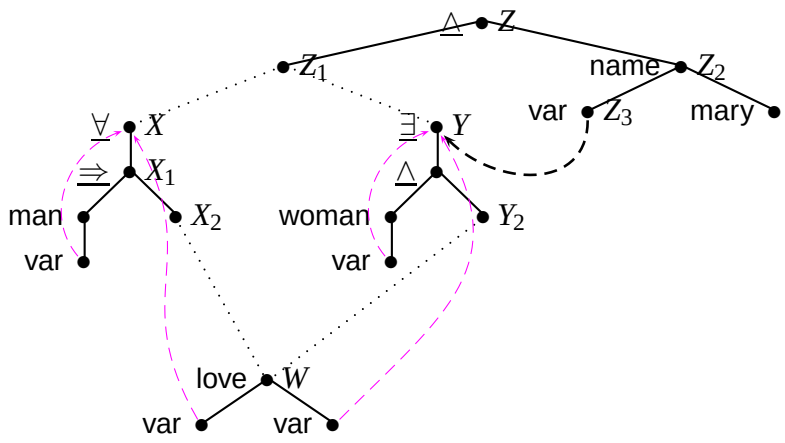


Figure 7.1: Constraint graph for (7.1).

In this chapter, we start with the effect of anaphoric references on the disambiguation of scope ambiguities. Consider the following example.

(7.1) Every man loves a woman. Her name is Mary.

The first sentence on its own is ambiguous; but taken together with the second sentence, the reading which assigns the universal quantifier wide scope is no longer felicitous, and the ambiguity goes away. That is, the scope ambiguity in the first sentence gets resolved once we take the anaphoric reference into account.

The infelicity of the second reading can be explained using a theory of *dynamic semantics* (Kamp and Reyle 1993; Groenendijk and Stokhof 1991) which determines the indefinite "a woman" to be *inaccessible* for the anaphoric reference with the pronoun "her". We will model the felicitous readings as those in which all antecedents of pronominal anaphora are accessible in the theory of *Dynamic Predicate Logic* (DPL), one particular formalism of dynamic semantics. Then we will recast accessibility as admissibility of a lambda structure according to a certain binding specification Λ_{DPL}. Finally we will use the methods from Section 4.2 to obtain a sound and complete algorithm for this binding specification.

The structure of the chapter is as follows. We will first give a brief introduction to DPL in Section 7.1. Then we will define Λ_{DPL}, the binding specification which represents the binding behaviour of DPL, in Section 7.2. In Section 7.3, we will extend the inference system $\mathcal{SN}$ from Fig. 4.1 with rules that are sound and complete for Λ_{DPL}. Finally, we will go through some examples in Section 7.4 in order to illustrate that the new rules for Λ_{DPL} can indeed resolve ambiguities without enumerating solutions, and explore the limits of our system.

7.1 Dynamic Predicate Logic in A Nutshell

Dynamic Predicate Logic (DPL, Groenendijk and Stokhof 1991) belongs to a family of logics developed in the eighties known as *dynamic logics*. The primary challenge in natural language semantics that dynamic logics address is to capture the peculiarities of anaphoric reference across clause boundaries, as in the examples (7.2) and (7.3) below. The indices 1 and 2 indicate which anaphora refer to which antecedents.

(7.2) [A man]$_1$ walks in the park. He$_1$ whistles.

(7.3) If [a farmer]$_1$ owns [a donkey]$_2$, he$_1$ beats it$_2$.

It seems like an attractive idea to analyse pronouns simply as bound variables in a first-order representation of the semantics of a sentence. Such an approach works well for sentences such as (7.4), whose semantics can be represented by the formula (7.5):[1]

(7.4) [Every man]$_1$ loves his$_1$ mother.

(7.5) $\underline{\forall} x.(\mathsf{man}(x) \underline{\Rightarrow} \mathsf{love}(x, \mathsf{mother\_of}(x)))$

But if we try to do this for the examples (7.2) and (7.3), we get semantic representations with free variables:

(7.6) $(\underline{\exists} x.\mathsf{man}(x) \mathbin{\underline{\wedge}} \mathsf{walk\_in\_park}(x)) \mathbin{\underline{\wedge}} \mathsf{whistle}(x)$

(7.7) $(\underline{\exists} x.\mathsf{farmer}(x) \mathbin{\underline{\wedge}} \underline{\exists} y.\mathsf{donkey}(y) \mathbin{\underline{\wedge}} \mathsf{own}(x, y)) \underline{\Rightarrow} \mathsf{beat}(x, y)$

The basic idea of dynamic semantics, pioneered in theories such as Discourse Representation Theory (DRT, Kamp and Reyle 1993) and File Change Semantics (Heim 1983a), was to view each clause as changing the hearer's context of interpretation, rather than just as transporting truth conditions as we assumed in Chapter 3. Such theories assume that when a hearer processes the indefinite "a man", a new *discourse referent* x for an individual who is a man is inserted into his context of interpretation. Other clauses can then refer back to this discourse referent by means of anaphora. There are linguistically motivated restrictions on which anaphora can refer to which antecedents, but the discourse referent x can generally be used outside of the scope of the quantifier that introduces it.

[1] We write the logical connectives of the object language (predicate logic or DPL) as $\underline{\wedge}$, $\underline{\exists}$, etc., in order to distinguish them from the respective connectives in dominance constraints.

$$
\begin{aligned}
[\![P(t_1,\ldots,t_n)]\!]_{\mathcal{M}} &= \{(g,g) \mid \mathcal{M}, g \models P(t_1,\ldots,t_n)\} \\
[\![\underline{\neg} A]\!]_{\mathcal{M}} &= \{(g,g) \mid \text{exists no } h \text{ s.t. } (g,h) \in [\![A]\!]_{\mathcal{M}}\} \\
[\![A \mathbin{\underline{\wedge}} B]\!]_{\mathcal{M}} &= \{(g,h) \mid \text{exists } k \text{ s.t. } (g,k) \in [\![A]\!]_{\mathcal{M}} \text{ and } (k,h) \in [\![B]\!]_{\mathcal{M}}\} \\
[\![A \mathbin{\underline{\vee}} B]\!]_{\mathcal{M}} &= \{(g,g) \mid \text{exists } h \text{ s.t. } (g,h) \in [\![A]\!]_{\mathcal{M}} \text{ or } (g,h) \in [\![B]\!]_{\mathcal{M}}\} \\
[\![A \mathbin{\underline{\Rightarrow}} B]\!]_{\mathcal{M}} &= \{(g,g) \mid \text{for all } h \text{ s.t. } (g,h) \in [\![A]\!]_{\mathcal{M}}, \text{ ex. } k \text{ s.t. } (h,k) \in [\![B]\!]_{\mathcal{M}}\} \\
[\![\underline{\exists} x.A]\!]_{\mathcal{M}} &= \{(g,h) \mid \text{exists } k \text{ s.t. } g[x]k \text{ and } (k,h) \in [\![A]\!]_{\mathcal{M}}\} \\
[\![\underline{\forall} x.A]\!]_{\mathcal{M}} &= \{(g,g) \mid \text{for all } h \text{ s.t. } g[x]h, \text{ there is a } k \text{ s.t. } (h,k) \in [\![A]\!]_{\mathcal{M}}\}
\end{aligned}
$$

Figure 7.2: Semantics of Dynamic Predicate Logic.

Dynamic Predicate Logic achieves the goal of allowing a quantifier to bind variables outside of its scope by using ordinary formulas of first-order predicate logic, but giving them a different semantics, shown in Fig. 7.2. It uses ordinary first-order variables as discourse referents, and models the current context of interpretation (which specifies how each discourse referent is interpreted) as a variable assignment. The denotation of each formula is now a binary relation between variable assignments, which represents how each possible "input" variable assignment is changed into an "output" variable assignment when the hearer interprets the clause.

To understand the definition in Fig. 7.2, let's take a closer look at the interpretation of the existential quantifier and the conjunction. Starting from some input variable assignment g, the existential quantifier in a formula $\underline{\exists} x.A$ allows us to make an arbitrary change to the value of x; the notation $g[x]k$ indicates that g and k agree on the value of each variable except possibly for x. Then the formula A is interpreted with k as its input variable assignment. A might make some changes to k, resulting in the assignment h, and then h is passed on as the output variable assignment of $\underline{\exists} x.A$. Because the formula $\underline{\exists} x.A$ is capable of changing the input variable assignment, we call the connective $\underline{\exists}$ *externally dynamic*. By contrast, connectives such as $\underline{\vee}$ for which the input and output assignments must be equal are called *externally static*.

Now conjunction $A \mathbin{\underline{\wedge}} B$ starts with some input variable assignment g and interprets A with respect to g. A might be externally dynamic, i.e. it might change g into a different output assignment k. Then B is interpreted with the input assignment k, and its output assignment h is passed on as the output assignment of the conjunction. Thus conjunction is clearly externally dynamic, but it is also *internally dynamic*: It is a binary connective that passes the output assignment of its left-hand subformula on to the interpretation of its right-hand subformula. Again, disjunction $A \mathbin{\underline{\vee}} B$ is an example of an *internally static* connective: Even if A changes a variable assignment, B is interpreted with respect to the original

input assignment g and not with respect to the changed assignment.

It is this behaviour of the connectives $\underline{\exists}$, $\underline{\wedge}$, and $\underline{\Rightarrow}$ (which is internally but not externally dynamic) that gives (7.6) and (7.7) meaningful interpretations in DPL. The existential quantifier in each formula changes the original input variable assignment by giving x some random value. The changed assignment is directly available for the atoms in the scope of the quantifier. Atoms act as tests, i.e. their denotations include only pairs (g, g), and they contain a pair (g, g) only if $\mathcal{M}, g$ satisfies the atom in the ordinary sense. So the output assignment of the formula $\underline{\exists}x.\mathsf{man}(x) \underline{\wedge} \mathsf{walk\_in\_park}(x)$ in (7.6) may have a changed value for x, but x must denote a man who walks in the park. Because conjunction is internally dynamic, this changed variable assignment is used in the interpretation of the formula $\mathsf{whistle}(x)$ – that is, the variable x has a well-defined value even outside the syntactic scope of $\underline{\exists}x$.

We still say that the occurrence of x is *bound* by the quantifier occurrence $\underline{\exists}x$, as it is this quantifier occurrence that assigned x the value that is used when interpreting the variable occurrence. A variable occurrence no longer has to be in the scope of a quantifier occurrence to be bound by it; it only has to be *accessible*. Some position in a formula counts as accessible from another position if a quantifier occurrence in the first position will bind a variable occurrence in the second position. As usual, a formula is called *closed* if all of its variables are bound. DPL predicts that a semantic representation of a sentence can be felicitous only if it is a closed formula, i.e. all of its anaphora have accessible antecedents.

Now let's apply DPL to the motivating example (7.1) above. The two readings of the sentence are shown below; (7.8) is the felicitous reading, whereas (7.9) is infelicitous.

(7.8) $(\underline{\exists}y.\mathsf{woman}(y) \underline{\wedge} (\underline{\forall}x.\mathsf{man}(x) \underline{\Rightarrow} \mathsf{love}(x, y))) \ \underline{\wedge} \ \mathsf{name}(y, \mathsf{mary})$

(7.9) $(\underline{\forall}x.\mathsf{man}(x) \underline{\Rightarrow} (\underline{\exists}y.\mathsf{woman}(y) \underline{\wedge} \mathsf{love}(x, y))) \ \underline{\wedge} \ \mathsf{name}(y, \mathsf{mary})$

The variable y in the conjunct $\mathsf{name}(y, \mathsf{mary})$ represents the anaphor "her" referring to "a woman". It is bound by the existential quantifier in (7.8), because the quantifier changes its output variable assignment so as to give y a value, and this changed variable assignment is passed on by the conjunctions because they are internally and externally dynamic. In (7.9), however, the existential quantifier is in the scope of a universal quantifier. Although it assigns y a value in its output assignment, the universal quantifier is externally static, and therefore doesn't pass this change on to the outside, so the existential quantifier is accessible from the variable occurrence in (7.8), but not in (7.9). That is, DPL can explain the difference in felicity by the fact that (7.8) is closed and (7.9) isn't.

7.2 DPL in Dominance Constraints

If we see DPL formulas as lambda structures, we can recast the binding behaviour of DPL as a binding specification Λ_{DPL} (see Definition 2.3). We will now define Λ_{DPL}, in such a way that a lambda structure representing a DPL formula is Λ_{DPL}-admissible iff the formula is closed.

First, let's assume a signature Σ that is partitioned into three sets: *connectives* $\Sigma_{con} = \{\underline{\forall}|_1, \underline{\wedge}|_2, \underline{\neg}|_1, \ldots\}$, *predicate symbols* $\Sigma_{pred} = \{\mathsf{man}|_1, \mathsf{likes}|_2, \ldots\}$, and *term symbols* $\Sigma_{term} = \{\mathsf{var}|_0, \mathsf{peter}|_0, \mathsf{mother\_of}|_1, \ldots\}$. Term symbols are the variable symbol var, constant symbols such as peter, and function symbols such as $\mathsf{mother\_of}$. Two lambda structures based on Σ, representing the DPL formulas (7.8) and (7.9), are shown in Fig. 7.3. Among the connectives, we further distinguish the internally dynamic connectives $\Sigma^{dyn}_{con} = \{\underline{\wedge}, \underline{\Rightarrow}\}$ and the externally static connectives $\Sigma^{stat}_{con} = \{\underline{\neg}, \underline{\forall}, \underline{\Rightarrow}, \underline{\vee}\}$.

Now the binding specification $\Lambda_{DPL} = (B_{DPL}, S_{DPL})$ consists of the binder set $B_{DPL} = \{\underline{\forall}, \underline{\exists}\}$ and the scope specification S_{DPL} for which $(v, w) \in S_{DPL}(\mathcal{M})$ iff

1. $v \triangleleft^* w$; or
2. there is a node u such that $L_{\mathcal{M}}(v) = \underline{\exists}$, $v \perp w$ at u, $u1 \triangleleft^* v$, $u2 \triangleleft^* w$, $L_{\mathcal{M}}(u) \in \Sigma^{dyn}_{con}$, and no node between $u1$ and v, inclusively, is labelled with an externally static connective from Σ^{stat}_{con}.

The first case reflects the standard situation from ordinary predicate logic where the variable is in the syntactic scope of the (existential or universal) quantifier. The second case is the situation where the quantifier is accessible, but the variable is not in its scope. Here the quantifier must be existential (i.e. it changes the output assignment of the quantified subformula so the variable has a defined value), all connectives up to the branching point u must be externally dynamic (i.e. pass the changed variable assignment along), and the label of the branching point itself must be internally dynamic (i.e. pass the changed variable assignment on to its right-hand child). It is easy to check that this structural notion of accessibility indeed coincides with the one that follows from the semantics in Fig. 7.2.

Lemma 7.1. *A lambda structure whose underlying tree represents a first-order formula is Λ_{DPL}-admissible iff the formula is a closed DPL-formula.*

The two readings of the running example are shown as lambda structures in Fig. 7.3. If we apply our new binding specification Λ_{DPL} to them, we see that the left-hand lambda structure (representing the felicitous reading) is indeed Λ_{DPL}-admissible.

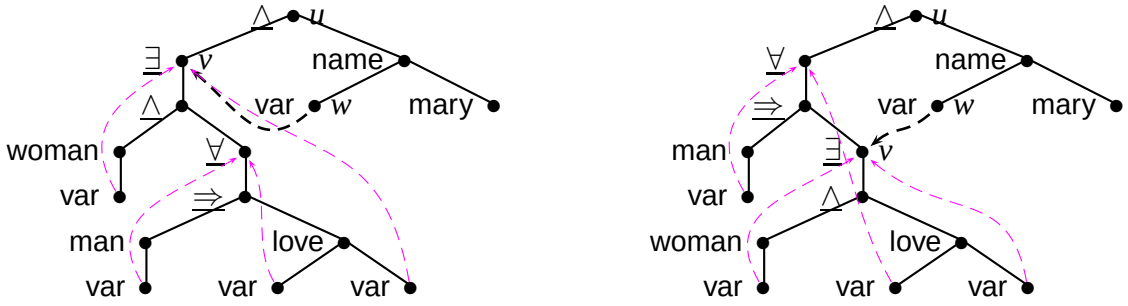


Figure 7.3: Lambda structures with binding arrows representing the two readings of (7.1).

All variable-binder pairs are in a dominance relation, except for the pair (v, w) marked in the picture by a darker arrow. The branching point u of v and w is the root of the tree, which is labelled by $\underline{\wedge}$ – an internally dynamic connective. v is below the left-hand child $u1$ of u, w is below the right-hand $u2$ child of u. $u1 = v$ is labelled with $\underline{\exists}$, which is externally dynamic. So all conditions for the second clause of the scope specification are satisfied.

On the other hand, the lambda structure shown on the right, representing the infelicitous reading, is not Λ_{DPL}-admissible. The branching point u of v and w is again the root, but now the path from $u1$ to v contains a node with the label $\underline{\forall}$, which is externally static.

Syntactically, the binding specification comes with a binding atom $\lambda_{DPL}(X) = Y$. We will use the language $\mathcal{SN}\Lambda_{DPL}$ to speak about dynamic binding within a dominance and binding constraint.

7.3 The Inference Procedure

Now let's turn to the problem of computing the Λ_{DPL}-admissible solutions of a dominance and binding constraint. One possible solution for this problem would be to run the sound and complete enumeration algorithm SN for dominance constraints from Section 4.1, ignoring all binding constraints. Then we could extract constructive solutions from the solved forms of the algorithm, and check each of them for Λ_{DPL}-admissibility. This algorithm would enumerate both lambda structures shown in Fig. 7.3 for the input constraint in Fig. 7.1, and would then filter out the inadmissible one.

The obvious disadvantage of this procedure is that it potentially requires us to

$$
\begin{array}{ll}
(\Lambda_{\mathrm{DPL}}.\mathrm{Func}) & \lambda_{DPL}(X) = Y \wedge \lambda_{DPL}(U) = V \wedge X = U \quad \rightarrow \quad Y = V \\
(\Lambda_{\mathrm{DPL}}.\mathrm{Var}) & \lambda_{DPL}(X) = Y \quad \rightarrow \quad X{:}\mathsf{var} \\
(\Lambda_{\mathrm{DPL}}.\mathrm{Binder}) & \lambda_{DPL}(X) = Y \quad \rightarrow \quad \exists Y'.(Y{:}\underline{\forall}(Y') \wedge Y' \lhd^* X) \vee \\
 & \qquad \exists Y'.(Y{:}\underline{\exists}(Y') \wedge Y' \lhd^* X) \vee \\
 & \qquad \bigvee_{f \in \Sigma^{dyn}_{con}} \exists Y' Z Z_1 Z_2.(Y{:}\underline{\exists}(Y') \wedge Z{:}f(Z_1, Z_2) \wedge Z_1 \lhd^* Y \wedge Z_2 \lhd^* X) \\
(\Lambda_{\mathrm{DPL}}.\mathrm{NonInter}) & \lambda_{DPL}(X) = Y \wedge Z{:}f(Z_1, Z_2) \wedge Z_1 \lhd^* Y \wedge Z_2 \lhd^* X \wedge W{:}g(W_1, \ldots, W_n) \\
 & \qquad \rightarrow \quad \neg(Z_1 \lhd^* W \lhd^* Y) \qquad\qquad (f \in \Sigma^{dyn}_{con}, g|_n \in \Sigma^{stat}_{con})
\end{array}
$$

Figure 7.4: An axiomatisation of Λ_{DPL}.

enumerate many solutions that turn out not to be admissible later. So we will once again use propagation rules to detect inadmissibility early, which we will add to the rule system SN from Chapter 4. In many cases, these rules will either allow us to derive a contradiction when all solutions of a constraint are inadmissible, or they will even allow us to derive explicit dominance atoms that are satisfied in all remaining admissible solutions.

Fig. 7.4 displays an axiomatisation of Λ_{DPL}. The first two rules are the standard functionality and variable-labelling rules that were also present in the axiomatisations in Section 4.2. The function of the (Binder) and (Scope) rules from the earlier axiomatisations is performed here by the rules (Binder) and (NonInter) together. (Λ_{DPL}.Binder) states that if Y is a binder, it must be labelled with $\underline{\forall}$ or $\underline{\exists}$, can either dominate the variable X, or can be connected to X via an internally dynamic branching point. Thus, it contains positive information about the label of the binder and about the relative positions of the binder and the variable. The negative information that if X and Y are linked by an internally dynamic branching point, there must be no externally static connective blocking the binding, is expressed by (Λ_{DPL}.NonInter). This rule uses the non-intervention constraint $\neg(Z_1 \lhd^* W \lhd^* Y)$ from the language $\mathcal{N}$, which expresses that W must not be between Z_1 and Y in a solution.

Lemma 7.2. *The rules in Fig. 7.4 are an axiomatisation of the binding specification Λ_{DPL}.*

Because they are an axiomatisation of Λ_{DPL}, it is clear that all the rules are sound over Λ_{DPL}-admissible lambda structures, i.e. whenever the left-hand side is Λ_{DPL}-satisfied, the right-hand side is also Λ_{DPL}-satisfied. As for completeness, Theorem 4.11 showed that an axiomatisation could be read immediately as a complete set of saturation rules, if it met certain prerequisites. Unfortunately, not all of these

prerequisites are satisfied in the case of Λ_{DPL}. In particular, (Λ_{DPL}.NonInter) is not guarded, and its left-hand side contains dominance atoms. This means that problems similar to the counterexample (1) in Section 4.2 (on page 59) could occur.

By arguing a little more carefully, we can nevertheless prove the following completeness result.

Proposition 7.3. *Every solved form of $SN\Lambda_{DPL}$ is Λ_{DPL}-satisfiable.*

Proof. We know that every *simple* $SN\Lambda_{DPL}$-solved form is $SN\Lambda_{DPL}$-satisfiable, by Lemma 4.10: (Λ_{DPL}.NonInter) is not equality insensitive, but this is no problem here because SN derives $X \lhd^* Z$ from $X = Y \wedge Y \lhd^* Z$.

It remains to show that the "extension by labelling" process from the proof of Prop. 4.7 can be done in such a way that (Λ_{DPL}.NonInter) doesn't become applicable – we get this for free for the three other rules, because they satisfy the preconditions of Theorem 4.11. So let's assume that we choose the connective $\underline{\wedge} \notin \Sigma^{stat}_{con}$ as the connecting symbol f in the extension by labelling process. Thus this rule will never be triggered either. □

Note that it is indeed necessary to choose the connecting symbol f carefully in this proposition. Consider a variant of the constraint in Fig. 7.1 without the bottom fragment (whose root is the variable W). This constraint is in solved form but not simple, so the "extension by labelling" procedure adds a labelling atom for Z_1 to make it simple. If the label f in this labelling atom is externally static, we obtain an extended solved form that has only inadmissible solutions – a fact which would be detected by the (Λ_{DPL}.NonInter) rule if we could still apply it to the extended solved form. On the other hand, if we choose $\underline{\wedge}$ as our label f, the extended solved form is a simple solved form that is $SN\Lambda_{DPL}$-saturated, and therefore has a Λ_{DPL}-admissible solution.

7.4 Examples

To illustrate how $SN\Lambda_{DPL}$ excludes inadmissible solutions, we will now go through two examples. we will first consider our running example (7.1), and then a more complex example which demonstrates the limits of the propagation rules.

Consider the constraint graph in Fig. 7.1. This is a normal dominance and binding constraint from $\mathcal{DI}\Lambda_{DPL}$, i.e. it contains an inequality atom for each pair of variables that are heads of labelling atoms. We want to infer the information that the existential quantifier must have wide scope, i.e. $Y_2 \lhd^* X$.

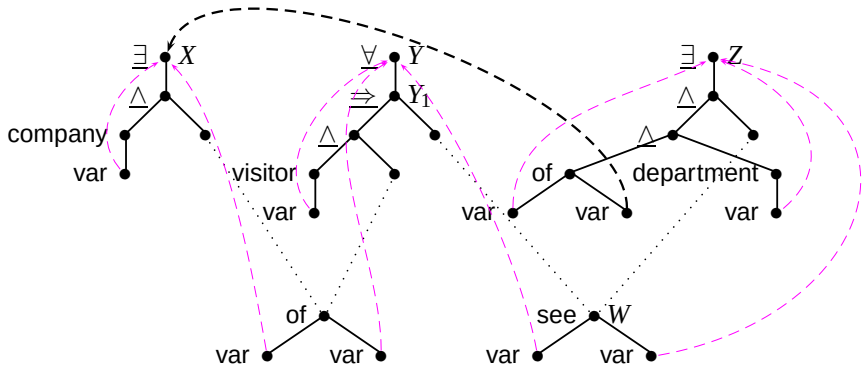


Figure 7.5: Constraint graph for (7.10).

In the first step, we want to apply the rule (Λ_{DPL}.NonInter) to the binding atom $\lambda_{DPL}(Z_3) = Y$ in order to infer a non-intervention atom. We first establish the dominance preconditions by some applications of (Lab.Dom) and (Inter). Then we can apply the following rule instance:

$$(\Lambda_{\mathrm{DPL}}.\mathrm{NonInter}) \quad \lambda_{DPL}(Z_3) = Y \wedge Z{:}\underline{\wedge}\,(Z_1, Z_2) \wedge Z_1 \lhd^* Y \wedge Z_2 \lhd^* Z_3 \wedge X{:}\underline{\forall}(X_1) \rightarrow \neg(Z_1 \lhd^* X \lhd^* Y)$$

The rule was applicable because $\underline{\forall}$ is an externally static connective. Now we can convert the non-intervention information into a positive dominance atom $Y \lhd^* X$ by exploiting the non-disjointness atom we can get from the two dominances into W, with the following SN derivation:

$$\begin{array}{ll} (\mathrm{NonI1}) & \neg(Z_1 \lhd^* X \lhd^* Y) \wedge Z_1 \lhd^* X \rightarrow X \neg \lhd^* Y \\ (\mathrm{NegDisj}) & X \lhd^* W \wedge Y \lhd^* W \rightarrow X \neg \perp Y \\ (\mathrm{Inter}) & X \neg \lhd^* Y \wedge X \neg \perp Y \rightarrow Y \lhd^* X \end{array}$$

Because the constraint contains an inequality atom for each pair of labelled variables, further applications of the rules (Child.down), (Lab.Disj), (NegDisj), and (NegDom) will derive $Y_2 \lhd^* X$. This means that we have resolved the scope ambiguity purely by propagation based on the anaphoric reference: We started with an implicit description of the scope ambiguity, and then strengthened it until only one scope reading remained.

Now let us consider a more complicated example, which illustrates the limitations of the algorithm. Fig. 7.5 is the underspecified description of the semantics of the following sentence:

(7.10) Every visitor of [a company]$_1$ saw one of [its]$_1$ departments.

This dominance and binding constraint from $\mathcal{DI}\Lambda_{DPL}$ has five constructive solutions, but only three of them are Λ_{DPL}-admissible: the two where X takes wide scope and Z is either between X and Y or below the right-hand hole of Y, plus the one where X is below the left-hand hole of Y and Z is below the right-hand hole. These three solutions can be characterised as satisfying the additional dominance constraint $Z \neg \lhd^* X$.

Unfortunately, this information cannot be inferred just by using the propagation rules in $SN\Lambda_{DPL}$. The two choices $X \lhd^* Y$ and $Y \lhd^* X$ give rise to very different structural constraints on Z. The case $X \lhd^* Y$ Λ_{DPL}-entails $X \lhd^* Z$: If this dominance did not hold, (Λ_{DPL}.Binder) would tell us that X and Z must be disjoint (with an internally dynamic label at a branching point that is not denoted by any variable), which contradicts the fact that both X and Z dominate W. But the atom $X \lhd^* Z$ is not inferred by $SN\Lambda_{DPL}$. In the case of $Y \lhd^* X$, the algorithm will infer a new labelling atom for $\underline{\Rightarrow}$; it can't infer without additional distribution that the head of this labelling atom must be Y_1.

This means that the propagation rules in Fig. 7.4 are incomplete in the sense that they do not compute *all* atoms that are Λ_{DPL}-entailed by the constraint. This is unfortunate, but as the rules are still guaranteed to detect Λ_{DPL}-unsatisfiability on constraints to which no more distribution rules can be applied, it is a bearable limitation, and it is still an improvement over a naive generate-and-test algorithm. In addition, complete propagation rules would probably require some form of hypothetical reasoning such as the disjunctive propagators we used in the set constraint based solver in Section 4.3. This would make the propagation itself more computationally expensive, and it is unclear whether such a step would actually make the whole system more efficient.

7.5 Summary

In this chapter, we have shown how to resolve scope ambiguities by taking anaphoric information into account. We first recast the accessibility conditions of DPL as a binding specification Λ_{DPL} for dominance and binding constraints. Then we defined an axiomatisation for this binding specification, which can be used in a sound and complete saturation algorithm. This algorithm is guaranteed to enumerate exactly the Λ_{DPL}-admissible solutions of the constraint – corresponding to the felicitous readings of the sentence –, and is capable of resolving some (but not all) cases of scope ambiguity by propagation, i.e. without enumerating the infelicitous readings at all.

The chapter looks deceptively simple: There was no deep logic involved, and all

our claims were so obviously true that we didn't prove most of them explicitly, but relied on an informal explanation. However, it is important to note that we owe this simplicity to the general theory of dominance constraints developed in the first two parts of the thesis. It was essential that we could build upon the sound and complete algorithm SN for dominance constraints, as well as on the earlier results on axiomatisations of binding specifications. It is thus rather a testament to the power of dominance constraints how easily they could be adapted to an object language with a highly nonstandard notion of variable binding. The pioneer work in combining underspecification and dynamic semantics (Schiehlen 1997) still had to develop quite a formidable formal apparatus specifically to compute the admissible solutions. By comparison, our formal tools are straightforward extensions to the basic theory, and in addition we are now able to eliminate inadmissible solutions by propagation.

It is interesting to compare the relationship between formula structure and variable binding in DPL and in lambda structures. In DPL, an occurrence of a quantifier $\underline{\forall}x$ or $\underline{\exists}x$ defines a set of positions in the formula at which it can bind occurrences of x. This means that the relative positions of the quantifier and the variable determine the binding relation. By contrast, the primitive concept in a lambda structure is the binding function mapping variable occurrences to quantifier occurrences. This binding function, together with the binding specification, determines the possible relative positions in which the quantifier and the variable can stand in an admissible lambda structure. In other words, in DPL structure determines binding, and in lambda structures binding restricts structure. This reversal of the relationship between binding and structure was instrumental in our algorithm, as it made it much easier to infer structural information (such as dominance atoms) from binding atoms. Defining a similar algorithm based purely on DPL (or any other formalism that establishes binding based on variable names) would have been much more difficult.

Chapter 8

Resolving Scope Ambiguities Using World Knowledge

The second source of information that we will use to resolve scope ambiguities is world knowledge. Consider the following examples.

(8.1) *Every boy ate a cookie.*

(8.2) *So we put a wooden elephant into every package.*
(British National Corpus, AJ9-135)

These sentences will be analysed as ambiguous by a standard grammar, such as the one from Chapter 3, because they contain two quantified noun phrases that are arguments of the same verb. However, one reading of each sentence is highly implausible: cookies are typically not eaten more than once, and an elephant won't be inside multiple packages. A human listener will automatically discard the implausible readings, thus resolving the scope ambiguity.

In this chapter, we model this kind of implausibility as inconsistency with world knowledge. We assume that our world knowledge contains axioms that talk about the eating of cookies and the location of physical objects, and that both our semantic representations and the axioms are formulas of ordinary first-order predicate logic. We will consider only readings that are logically consistent with the world knowledge as "compatible" in the sense of the introduction of Chapter 7. Our aim in this chapter will be to model the incompatible readings, and to provide computational mechanisms for strengthening an underspecified descriptions so incompatible readings are eliminated.

Because world knowledge is much more complex than anaphoric reference, we will approach this problem differently than in the previous chapter. We will develop a procedure for checking whether all constructive solutions of a constraint are unsatisfiable. Then we will strengthen a constraint φ by performing a single distribution step on it; let's say that the two possible results are φ_1 and φ_2. If φ_1 has only unsatisfiable constructive solutions, we commit to φ_2; this constraint has fewer constructive solutions than φ, but contains the same satisfiable constructive solutions. Such a procedure fits straightforwardly into the enumeration algorithm from Chapter 5, where it could be used as a propagator that removes entire subtrees from the search tree, if all of its constructive solutions are unsatisfiable.

The main challenge in this approach is to check efficiently that all constructive solutions of a constraint are unsatisfiable. We define *unsatisfiability criteria* as a general framework for tackling this problem (Section 8.1), and show how an unsatisfiability criterion can be obtained based on a rewrite system that makes a constructive solution weaker (Section 8.2). Then we present a rewriting-based unsatisfiability criterion for classical unsatisfiability. This criterion makes it possible to establish the unsatisfiability of *all* constructive solutions of a constraint by checking only *some* constructive solutions for unsatisfiability, and it is cheap to determine which solutions need to be checked.

However, it will turn out at this point that classical unsatisfiability doesn't reflect the linguistic intuition about the perceived logical strength of a reading, because it doesn't take *presuppositions* into account – in our case, this is most apparent for the existential presuppositions of universally quantified noun phrases. We will investigate several candidate definitions for these presuppositions (Section 8.3), but none of them supports a working unsatisfiability criterion. Nevertheless, this part of the chapter lays a foundation of concepts and methods upon which later research can build.

8.1 Unsatisfiability Criteria

The key component of the architecture we have just described is a procedure that checks, for any given normal dominance constraint φ, whether all of its readings are inconsistent with our world knowledge. For now, we assume that the world knowledge is represented as a set Γ of first-order formulas, and that a formula A is "inconsistent with our world knowledge" if the formula set $\Gamma \cup \{A\}$ has no common model. We write capital Latin letters A, B, etc. for first-order formulas; as before, we use underlined connectives, such as $\underline{\wedge}$ and $\underline{\forall}$, for the connectives in the object language.

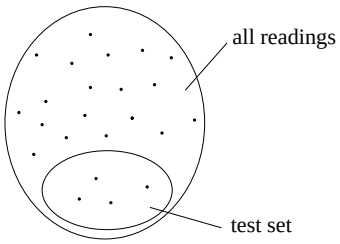


Figure 8.1: An unsatisfiability criterion.

It is conceptually straightforward to check a formula set $\Gamma \cup \{A\}$ for mutual inconsistency – all we need to do is send the conjunction over all formulas to a theorem prover and have it test the conjunction for unsatisfiability. But of course, if we have a sentence with N readings, where N is exponential in the size of the underspecified representation, this approach is computationally unfeasible.

Fortunately, the different constructive solutions of a normal dominance constraint are not an arbitrary set of formulas. We know that they are built up from the same fragments, and there are restrictions on the way that these fragments can be configured. Thus it may be possible to establish that *all* constructive solutions of a constraint are unsatisfiable by selecting an appropriate subset (the "test set") and proving that the constructive solutions in this *subset* are unsatisfiable. We formalise this idea in the following definition.

Definition 8.1. A triple $C = (U, T, P)$ is called an *unsatisfiability criterion* iff

1. U is a property of first-order formulas (the *unsatisfiability condition*);
2. P is another property of first-order formulas (the *test condition*);
3. for any dominance constraint φ, $T(\varphi)$ is a subset of the constructive solutions of φ (the *test set*); and
4. if all elements of $T(\varphi)$ have the property P, then all constructive solutions of φ have the property U.

The naive strategy for unsatisfiability checking that we just described fits easily into this framework, as follows:

$$\begin{array}{rcl} C_1 = (U_1, T_1, P_1)\text{: check all readings} & & \\ \hline U_1 & = & \{A \mid \Gamma \underline{\wedge} A \text{ is unsatisfiable}\} \\ P_1 & = & U_1 \\ T_1(\varphi) & = & \text{all constructive solutions of } \varphi \end{array}$$

It is trivial that C_1 is indeed an unsatisfiability criterion, but as we have already argued, it is computationally unfeasible because the test set can be very big.

We can obtain an unsatisfiability criterion C_2 with a much smaller test set if we consider that first-order entailment is a pre-order (i.e. a reflexive and transitive relation) on the set of all first-order formulas. Thus if we could prove that those readings of a constraint that are *minimal* with respect to this order (i.e., that don't properly entail any other constructive solution) are unsatisfiable, we would know that *all* constructive solutions are unsatisfiable.

$C_2 = (U_2, T_2, P_2)$: check minimal readings

$$\begin{aligned} U_2 &= \{A \mid \Gamma \underline{\wedge} A \text{ is unsatisfiable}\} \\ P_2 &= U_2 \\ T_2(\varphi) &= \text{the } \models\text{-minimal elements of the constructive solutions of } \varphi \end{aligned}$$

The set of constructive solutions will generally have just a few minimal elements, so the number of theorem prover calls that we need to establish unsatisfiability of the test set is much smaller than for C_1. However, it is not obvious how the set of minimal elements can be computed in general without computing the entire entailment pre-order. If φ has N constructive solutions, computing the entire pre-order will require $O(N^2)$ theorem prover calls. Even if we exploit equivalences and transitivity to reduce the number of necessary proofs somewhat (Gabsdil and Striegnitz 2000), this model is computationally even more expensive than C_1. So we will somehow have to strike a balance between computing a small test set on the one hand and computing this test set efficiently.

Note that the minimal elements of the entailment order are very close in spirit to the notion of *weakest readings* that is sometimes considered in the underspecification literature, see e.g. (Monz and de Rijke 1998). If a formula A entails a formula B, B is "weaker" in that it contains less information than A. One hope in underspecification is that there are unique weakest readings, which could then be taken as representing the safe information that is common to all readings of an ambiguous sentence. Unfortunately, there are sentences with more than one minimal reading. But the idea of looking for minimal constructive solutions is a generalisation of looking for weakest readings that is still applicable in this case.

8.2 Rewriting-Based Unsatisfiability Criteria

We will now show how a balance between computing a small test set and computing a test set cheaply can be achieved by defining unsatisfiability criteria based on

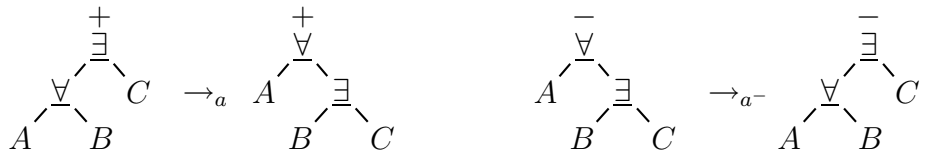


Figure 8.2: Some instances of the rotation rewrite system. The rule (a) weakens a formula; (a^-) strengthens it.

rewriting. We will achieve this by defining a rewrite system that transforms first-order formulas into other first-order formulas, such that each rewrite step makes the formula weaker. Essentially, every entailment-minimal constructive solution of a constraint will be in normal form with respect to such a rewrite system, because it can't be made weaker by any rewrite step. So we can take those constructive solutions of a constraint that are normal forms of the rewrite system as our test set. The test set will typically be smaller than the whole set of constructive solutions, and it can be computed efficiently, as we only need to check the N constructive solutions for applicability of the rewrite rules.

To make these ideas more concrete, let's consider the following example.

(8.3) A researcher of every company is paid well.

(8.4) $(\underline{\forall} x : \mathsf{comp}(x))[(\underline{\exists} y : \mathsf{res}(y) \mathbin{\underline{\wedge}} \mathsf{of}(x, y))[\mathsf{paidwell}(y)]]$

(8.5) $(\underline{\exists} y : (\underline{\forall} x : \mathsf{comp}(x))[\mathsf{res}(y) \mathbin{\underline{\wedge}} \mathsf{of}(x, y)])[\mathsf{paidwell}(y)]$

We represent the two readings of the sentence – (8.4) and (8.5) – in a variant of first-order predicate logic with *binary* quantifiers $(\underline{\forall} x : R)[S]$ and $(\underline{\exists} x : R)[S]$. This is to preserve the distinction between the linguistic *scope* and the *restriction* of the quantifier. In this case, the restriction of the indefinite is "researcher of every company" and the scope is "is paid well"; the restriction of the universal quantifier is "company" and the nuclear scope is "of". The formulas can be straightforwardly translated to ordinary predicate logic by replacing every occurrence of $(\underline{\forall} x : R)[S]$ by $\underline{\forall} x . R \mathbin{\underline{\Rightarrow}} S$ and every occurrence of $(\underline{\exists} x : R)[S]$ by $\underline{\exists} x . R \mathbin{\underline{\wedge}} S$.

The second reading of the sentence is intuitively stronger: If there is a single researcher who works for every company and who is paid well, it is of course true that every company employs *some* well-paid researcher. We can map the stronger reading into the weaker one by the rewrite step (a) in Fig. 8.2. The example motivates a general intuition that switching the positions of an existential and a universal quantifier in this way should make the semantic representation of a

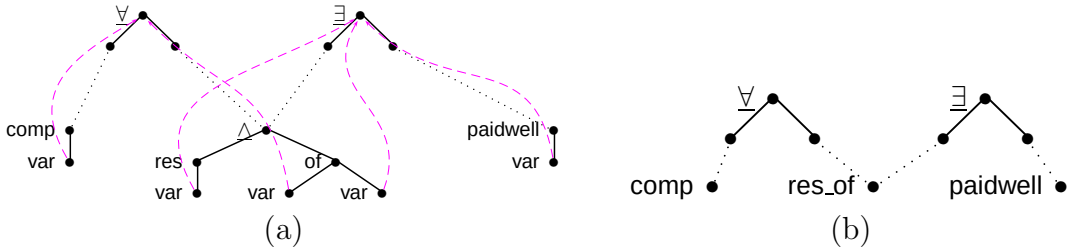


Figure 8.3: Constraint for (8.3), as an elaborated normal dominance constraint (a) and as the compactified pure chain (b).

sentence weaker – if the redex occurs in a positive context. If the redex occurs in a negative context ("It is not true that a researcher of every company is paid well"), the same rewrite step would make the formula *stronger* (shown as (a^-) in Fig. 8.2). Indeed, this intuition is supported by the fact that the left-hand side of (a) logically entails its right-hand side, but not vice versa, and the converse claim holds for (a^-).

Both rules in Fig. 8.2 are instances of the rotation rewrite system from Def. 6.22, with particular symbols substituted for f and g. For constraints that are pure chains, we know that instances of these rules only map solutions to solutions (Prop. 6.23). The lambda structures corresponding to the formulas (8.4) and (8.5) don't satisfy any pure chain, because they contain unary symbols. But they both satisfy the *alternating chain* in Fig. 8.3(a). Alternating chains (which we will define in a moment) are constraints whose compactifications are pure chains: The compactification of the example constraint is shown in Fig. 8.3(b). Because of this correspondence, the relevant rewriting lemmas from Section 6.4.4 still hold for alternating chains. Many interesting sentences have underspecified semantic representations that can be written as alternating chains, but there are sentences that don't; we will investigate their exact expressive power below.

8.2.1 Alternating chains

Before we define how to obtain unsatisfiability criteria by way of rewrite systems, let's now first have a closer look at alternating chains and at the class of formulas that they can describe.

Definition 8.2. An *alternating chain* is a normal dominance and binding constraint of the form

$$\left(\bigwedge_{i=1}^{n} X_i{:}f_i(X_i^l, X_i^r) \wedge \bigwedge_{i=1}^{n+1} \varphi_i^L\right)^{\neq} \wedge \bigwedge_{i=1}^{n}(X_i^l \lhd^* Y_i \wedge X_i^r \lhd^* Y_{i+1}) \wedge \bigwedge_{i=1}^{n}(\lambda(Y_i^{vl}) = X_i \wedge \lambda(Y_{i+1}^{vr}) = X_i),$$

where (a) $f_i \in \{\underline{\forall}, \underline{\exists}\}$ for each i, (b) each φ_i^L is a description of a first-order formula that consists only of labelling atoms for symbols that are neither $\underline{\forall}$ and $\underline{\exists}$, (c) the root of φ_i^L (as a fragment) is Y_i, (d) no two φ_i^L have any variables in common, and no X_i, X_i^l, or X_i^r occurs in any φ_k^L, (e) $Y_i^{vl}, Y_i^{vr} \in \mathcal{V}(\varphi_i^L)$.

Alternating chains are elaborated (in the sense of Def. 5.36), so we can remove the binding atoms and compactify an alternating chain φ and obtain a compact dominance constraint φ' whose constructive solutions correspond one-to-one to the constructive solutions of φ (Prop. 5.37, Prop. 5.14). This constraint φ' will be a pure chain with the same upper fragments as φ. The name "alternating chain" derives from the fact that as we follow the dominance edges of the dominance graph, we alternate between holes representing the scope and the restriction of the respective quantifier.

Now consider all instances of the rotation rules from Def. 6.22 in which each symbol f and g can be either $\underline{\forall}$ or $\underline{\exists}$. No such rule can rewrite a redex inside the denotation of a lower fragment, as we have assumed that the φ_i^L don't contain the symbols $\underline{\forall}$ or $\underline{\exists}$. So it follows immediately that Propositions 6.23 and 6.24 carry over to alternating chains.

We can characterise the sets of formulas that can occur as constructive solutions of an alternating chain as the formulas in the language $\mathcal{F}$.

Definition 8.3. Let V be some set of (object-language) variables, and let $\bullet \notin V$ be another symbol. Then $\mathcal{F}$ is defined as follows:

1. $A \in \mathcal{F}_{x,y}$ if A is a quantifier-free first-order formula whose free variables are x and y;
2. $A \in \mathcal{F}_{\bullet,x}$ and $A \in \mathcal{F}_{x,\bullet}$ if A is a quantifier-free first-order formula whose only free variable is x;
3. if $A \in \mathcal{F}_{x,y}$ and $B \in \mathcal{F}_{y,z}$ for some $x, z \in V \cup \{\bullet\}$ and $y \in V$, then $(\underline{\forall} y : A)[B]$ and $(\underline{\exists} y : A)[B]$ are in $\mathcal{F}_{x,z}$.
4. $\mathcal{F} := \bigcup_{x,y \in V \cup \{\bullet\}} \mathcal{F}_{x,y}$.

Intuitively, the subscript of $\mathcal{F}_{x,y}$ indicates the free variables of a formula; $\bullet$ is a placeholder to indicate that the formula has less than two free variables. It is easy to show that a formula in $\mathcal{F}$ is closed iff it is in $\mathcal{F}_{\bullet,\bullet}$. The order of the two variables in the subscript determines how the formula can be combined with others: A quantifier (as per clause 3 of the definition) must bind the *right-hand* variable of its restriction and the *left-hand* variable of its scope.

Many interesting natural-language sentences have semantic representations that belong to $\mathcal{F}$. For instance, both formulas (8.4) and (8.5) are members of $\mathcal{F}_{\bullet,\bullet}$. $\mathcal{F}$ contains all semantic representations that the grammar in Chapter 3 assigns to sentences whose main verb is intransitive, and whose subject is an arbitrarily complex structure built from noun phrases (NP) and prepositional phrases (PP).

However, not all sentences have semantic representations in $\mathcal{F}$. Our familiar running example, "Every man loves a woman", one of whose readings is repeated here as (8.6), is one such sentence.

(8.6) $(\underline{\forall}x : \mathsf{man}(x))[(\underline{\exists}y : \mathsf{woman}(y))[\mathsf{love}(x, y)]]$

The reason for this is that the subformula $\mathsf{love}(x, y)$ is in the scope of both quantifiers, rather than in the scope of one and the restriction of another. In order to combine the subformulas $\mathsf{woman}(y)$ and $\mathsf{love}(x, y)$ via the quantifier $\underline{\exists}y$, we must consider the former as a formula in $\mathcal{F}_{\bullet,y}$ and the latter as a formula in $\mathcal{F}_{y,x}$; so the entire existential formula is in $\mathcal{F}_{\bullet,x}$. But this means it cannot be the right-hand subformula of a quantifier $\underline{\forall}x$, as the x appears as the right-hand variable.

It is straightforward to see that alternating chains and $\mathcal{F}$ belong together in the same ways as pure chains and binary trees.

Proposition 8.4. *A formula in $\mathcal{F}$ is the constructive solution of exactly one alternating chain, and all constructive solutions of an alternating chain are in $\mathcal{F}$.*

Proof. The first claim follows essentially like the proof that a binary tree satisfies exactly one pure chain (Prop. 6.20). For the second point, observe that the constructive solution that satisfies $X_i^r \lhd^* X_{i+1}$ for all i is in $\mathcal{F}$. The claim follows if we combine this with Prop. 6.24. □

8.2.2 Rewrite systems with polarities

Now we are prepared to connect rewriting to unsatisfiability criteria. We will equip rewrite systems with a notion of polarity, so as to be able to distinguish positive and negative contexts of a redex. Then we will show how a rewrite system with polarities can be used to define an unsatisfiability criterion.

First, let's equip rewrite systems with a mechanism for keeping track of polarities. We take the notion of a term and a ground term (i.e., a variable-free term) for granted. A substitution θ is a mapping from variables to terms; we write $s\theta$ for the result of the application of the substitution θ to the term s.

Definition 8.5. A *term with polarities* A is a pair (t, p) of a term t and a function p that maps the nodes of t to polarities in $\{+, -\}$. A *context* is a term with polarities that has exactly one hole. The context is *positive* if the polarity of the root and the hole are equal, and *negative* otherwise.

A *rewrite system with polarities* R is a set of triples (s, t, ϵ) of terms s, t (which may include variables) and a polarity $\epsilon \in \{+, -\}$. The rule $r = (s, t, \epsilon)$ can be applied to the term A with polarities iff $A = C[s']$, where C is a context, $s' = s\theta$ is an instance of s, and the polarity of C's hole is ϵ. The result of the rule application is $C[t\theta]$. We write $A \to_{r,C} B$ in this case. If R is a rewrite system with polarities, we write $A \to_{R,C} B$ if there is some $r \in R$ such that $A \to_{r,C} B$. We write $A \to_R B$ (or just $A \to B$) if there is a context C such that $A \to_{R,C} B$.

The definition of rewrite systems with polarities is a straightforward extension of ordinary rewrite systems which only allows the application of rewrite rules to positions of appropriate polarity. We can read formulas from $\mathcal{F}$ as terms with polarities in the obvious way: The root of the whole formula is positive; the two children of a node v with label $\underline{\exists}$ and the right-hand child of a node with label $\underline{\forall}$ have the same polarity as v; and the left-hand child of a node with label $\underline{\forall}$ has the other polarity. An example for a rewrite system that operates on $\mathcal{F}$ is the two-rule system in Fig. 8.2; note that each rule is marked with the polarity in which it is applicable.

Definition 8.6. A triple $C_R = (U, R, P)$ is called a *rewriting-based unsatisfiability criterion* iff

1. U and P are properties of first-order formulas (the unsatisfiability and test conditions as in Def. 8.1);
2. R is a weakly normalising rewrite system with polarities;
3. if A and B are constructive solutions of the same dominance constraint, $A \to_R^* B$, B is in R-normal form, and $P(B)$, then $U(A)$ holds.

The idea is that because the rewrite system normalises weakly, every constructive solution A of a constraint φ can be rewritten into some normal form B in a finite number of steps. Intuitively, each rewrite step makes the formula weaker. Now assume that we know $P(B)$ for all normal forms. Then the third clause will allow us to conclude that we must have $U(A)$ for any constructive solution A. This means that a rewriting-based unsatisfiability criterion induces an ordinary unsatisfiability criterion (in the sense of Def. 8.1) with the same P and U, and whose test set is the set of R-normal forms among the constructive solutions of a constraint.

Lemma 8.7. *If $C_R = (U, R, P)$ is a rewriting-based unsatisfiability criterion, then $C = (U, T_R, P)$, where $T_R(\varphi)$ is the set of constructive solutions of φ that are normal forms of R, is an unsatisfiability criterion.*

So one way to create an unsatisfiability criterion is as follows:

1. Decide upon an unsatisfiability condition U that we are interested in.
2. Define a weakly normalising rewrite system with polarities R such that each rewrite rule makes the formula it is applied to "weaker" with respect to U.
3. Find a suitable test condition P such that (U, R, P) becomes a rewriting-based unsatisfiability criterion.
4. Apply Lemma 8.7 in order to get the induced unsatisfiability criterion.

The main problem with this recipe is to prove that (U, R, P) is a rewriting-based unsatisfiability criterion. More specifically, it is hard to establish the third condition of Definition 8.6. This condition makes a very strong claim about the connection of "semantic" notions such as U and P and "syntactic" notions such as the rewrite system R, and it isn't trivial to prove such a connection.

We can solve this problem by assuming a binary relation E between terms; if U and P are intuitively unsatisfiability, E is intuitively entailment. On the one hand, we will require that $P(B)$ and $E(A, B)$ together imply $U(A)$. On the other hand, we will use the following lemma to prove that $A \to^* B$ implies $E(A, B)$ by looking only at single rewrite steps. We will say that R *weakens formulas* with respect to E if it satisfies the latter condition.

Importantly, we assume that R is an *antisymmetric* rewrite system with polarities (ARSP). If we consider the two rules in Fig. 8.2 again, it is obvious that if we reverse the direction of (a), the rule becomes a strengthening rule. If we apply the reversed rule in a negative context, it becomes a weakening rule again. The defining characteristic of an ARSP is that if we reverse a weakening rule and apply it in the wrong polarity, we get a weakening rule again.

Definition 8.8. If $r = (s, t, \epsilon)$ is a rewrite rule with polarities , we call $r^- = (t, s, -\epsilon)$ the *antisymmetric rule* of r. An entire rewrite system with polarities R is called antisymmetric iff $\{r^- \mid r \in R\} \subseteq R$. The *antisymmetric closure* $R^{\leftrightarrow}$ of a rewrite system R with polarities is the system $R \cup \{r^- \mid r \in R\}$.

Lemma 8.9. *Let R be an ARSP and E a binary relation over $\mathcal{F}$. Then $A \to^* B$ implies $E(A, B)$ if*

1. *E is reflexive.*

2. *If $A \to_\bullet B$, then $E(A, B)$.*[1] (base case)

3. *If $E(A, B)$ and C is a positive context, then $E(C[A], C[B])$. If C is a negative context, then $E(C[B], C[A])$.* (insertion into contexts)

4. *If $E(A, B)$ and $B \to C$, then $E(A, C)$.* (transitivity)

Proof. Because of the reflexivity and transitivity conditions, it is sufficient to show that $A \to B$ implies $E(A, B)$. To do this, we first let R^+ be the positive polarity rules in R as an ordinary rewrite system, i.e. $R^+ = \{(s, t) \mid (s, t, +) \in R\}$. Then we prove by structural induction over the context C of the redex that $A \to_{R^+,C} B$ implies $E(A, B)$ if C is positive, and $E(B, A)$ if C is negative.

Assume we had this, and assume that $A \to_{R,C} B$. If C is positive, then the rewrite step was also a rewrite step according to R^+, and we have $E(A, B)$ directly. If C is negative, it was an application of a negative rewrite rule $r = (s, t, -)$. Because R is antisymmetric, it also contains the rule $r^- = (t, s, +)$. Now let $R^{-1} = \{(s, t, \epsilon) \mid (t, s, \epsilon) \in R\}$. Then we know that $B \to_{(R^{-1})^+,C} A$, by the version of r^- without polarities. Because R^{-1} is an ARSP that simply contains the reversed rules of R, the lemma is applicable with all the arguments of E reversed. If we apply the above result of the structural induction, we find that $B \to_{(R^{-1})^+,C} A$ implies $E(A, B)$, which concludes the proof.

Now for the structural induction. Because $A \in \mathcal{F}$, we have to consider only the following two cases.

$C = \bullet$ This is the base case (condition 2). The empty context is positive.

$C = D(C')$ where C' is a smaller context, and D is either $f(A', \bullet)$ or $f(\bullet, A')$, where $f \in \{\underline{\forall}, \underline{\exists}\}$ and $A' \in \mathcal{F}$. Let's say that C' and D are both negative contexts; the proof is analogous for the other three cases. Then by induction hypothesis, we have $E(C'[B], C'[A])$. By condition 3, we then have $E(D[C'[A]], D[C'[B]])$.

□

[1] $\bullet$ is the empty context.

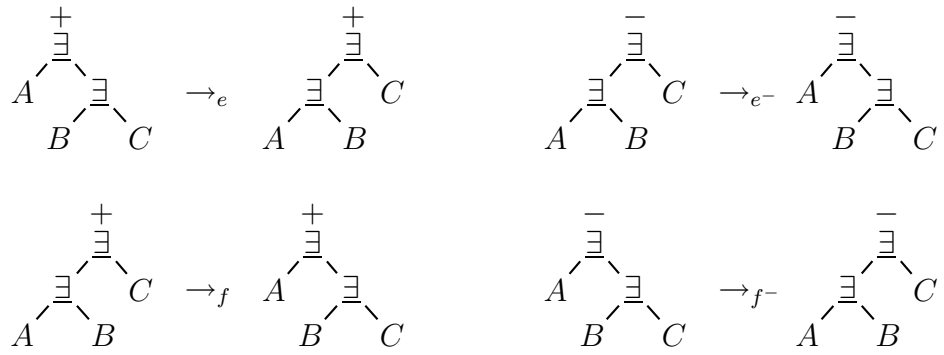


Figure 8.4: The equivalence rules e and f.

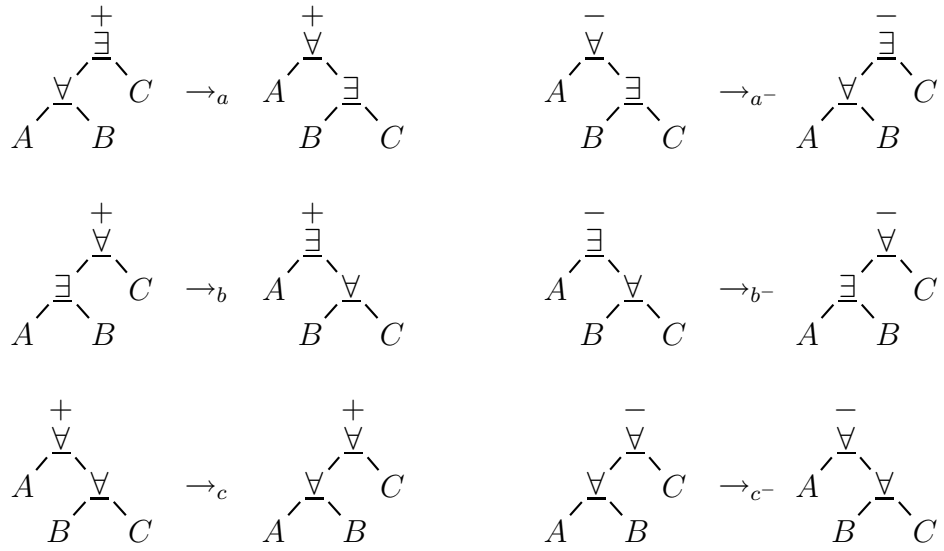


Figure 8.5: The weakening rules a, b, and c.

8.2.3 Classical unsatisfiability

Now let's apply our recipe for rewriting-based unsatisfiability criteria to our original problem of checking whether all constructive solutions of a constraint are unsatisfiable.

Using the rewrite rules from Fig. 8.4 and Fig. 8.5, we can define two different rewriting-based unsatisfiability criteria as follows:

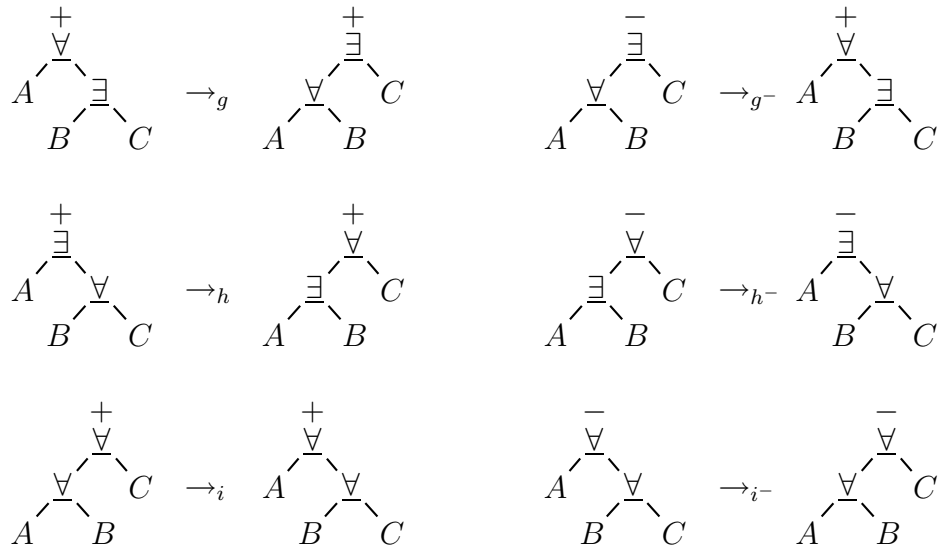


Figure 8.6: The strengthening rules g, h, and i.

$$\begin{array}{rcl} \multicolumn{3}{c}{C_3 = (U_3, R_3, P_3)\text{: rewriting-based, with rule } e} \\ \hline U_3 & = & U_1 = \{A \mid \Gamma \mathbin{\underline{\wedge}} A \text{ is unsatisfiable}\} \\ P_3 & = & U_3 \\ R_3 & = & \{a, e\}^{\leftrightarrow} = \{a, e, a^-, e^-\} \end{array}$$

$$\begin{array}{rcl} \multicolumn{3}{c}{C'_3 = (U_3, R'_3, P_3)\text{: rewriting-based, with rule } f} \\ \hline U_3 & = & U_1 = \{A \mid \Gamma \mathbin{\underline{\wedge}} A \text{ is unsatisfiable}\} \\ P_3 & = & U_3 \\ R'_3 & = & \{a, f\}^{\leftrightarrow} = \{a, f, a^-, f^-\} \end{array}$$

The two criteria differ only in the choice of adding either e of f to the rule system. A criterion can't contain both rules because this would destroy weak normalisation. An example for a formula that has no normal form is $(\underline{\exists} x : A(x))[(\underline{\exists} y : B(x, y))[C(y)]]$.

Lemma 8.10. *Both C_3 and C'_3 are rewrite-based unsatisfiability criteria.*

Proof. We need to establish Conditions 2 and 3 of Definition 8.6. Let's start with Condition 3, and let's use the following relation as our stepping stone to prove the connection between U_3, P_3, and R_3 (or R'_3):

$$E_3(A, B) \quad :\Leftrightarrow \quad A \models B.$$

Note that A and B may be subformulas of the complete constructive solution, and may therefore contain free variables. We deal with these free variables as usual, i.e. $A \models B$ is equivalent to $\models \underline{\forall} x_1 \ldots \underline{\forall} x_n (A \underline{\Rightarrow} B)$.

It is clear that $P_3(B)$ and $E_3(A, B)$ imply $U_3(A)$. Now we need to check the applicability conditions of Lemma 8.9. All conditions except Condition 2 are obvious. Condition 2 can be verified automatically, after substituting formulas $A(x, y)$, $B(y, z)$, and $C(z, w)$ for the three subterms.

In fact, the left-hand side of the rule (a) *strictly* entails the right-hand side, and the right-hand side of (a^-) strictly entails the left-hand side. This means that these rule can be applied only a finite number of times. The rules (e) and (e^-) can only be applied finitely often either because each rotation leaves the polarity of every subterm intact. So R_3 normalises, even strongly. The same argument goes through for R_3'. □

Thus we have achieved the goal we have set for ourselves in the beginning of the chapter. We have defined an unsatisfiability criterion with which we can establish the unsatisfiability of all constructive solutions without sending every single constructive solution to a theorem prover: Unsatisfiability of the reducible solutions follows from the unsatisfiability of the normal forms. On the other hand, it is relatively cheap to decide which constructive solutions are in normal form and which aren't.

However, the practical efficiency gain may not be as high as we hoped. The rewrite rules (e) and (f) allow us to save on theorem prover calls for obviously equivalent readings, and beyond that we have essentially a single rewrite rule (a) that actually moves us from stronger readings to weaker ones. Because the rules are so specific, most readings are in normal form.

8.3 Existential Presuppositions

But there is a more fundamental problem with unsatisfiability criteria for classical unsatisfiability in general: Classical unsatisfiability doesn't reflect the linguistic intuitions on the relative strength of readings. This becomes apparent when we consider the rules (b) and (c) from Fig. 8.5. The following example sentences illustrate the kind of inference that these two rules represent:

Rule (b): *Every researcher of a company is paid well.*

The left-hand reading says that every researcher who works for any company

at all is paid well. We can conclude from this that if we pick one specific company, all the researchers it employs must be paid well.

Rule (c): *Every researcher of every company is paid well.*

The left-hand reading says that it is true for every company that every researcher who works for it is paid well. But if that is the case, then surely every researcher who works for *all* companies is paid well too.

So intuitively, the right-hand sides seem to follow from the left-hand sides, and we would like to be able to use the rules (b) and (c) in our weakening rewrite system. However, neither side of these rules entails the other side logically. This means that no unsatisfiability criterion based on the rotations can contain these rules if we want to model classical unsatisfiability, because they might rewrite a satisfiable reading into an unsatisfiable one. Conversely, if we take any rewriting-based unsatisfiability criterion that weakens with respect to classical entailment, the criterion will use test sets that contain intuitively non-minimal readings. Clearly there is some part of the meaning of a sentence which we haven't represented yet.

8.3.1 Existential presuppositions

The reason why the semantics of the example sentences seems to be incompletely represented is that they contain universal noun phrases. A universally quantified formula $(\underline{\forall}x : R(x))[S(x)]$ in predicate logic is vacuously satisfied by a model in which the extension of R is empty. Under this interpretation of the universal quantifier, the readings of the example sentences are indeed incomparable. In the case of (b), if there is no company (and therefore, no researchers who work for a company), then the first reading is true, and the second is false.

A human listener who is not a trained logician, on the other hand, feels distinctly uneasy if forced to accept that the first reading is true if there are no companies; indeed, there is some psycholinguistic evidence that human listeners routinely conclude statements of the form "some R S" from the information "every R S" (Geurts 2003b). This is because universal noun phrases like "every R" and "all R" are *strong noun phrases* (Milsark 1977), and strong noun phrases are commonly accepted to carry a *presupposition* that the restriction R is not empty (Lasersohn 1993; Lappin and Reinhart 1988; Geurts 2003a; de Jong and Verkuyl 1984).

Presuppositions (Beaver 1997) are conditions that must be satisfied for an utterance to be interpretable. The classical example is that of definite noun phrases. Compare the following examples:

(8.7) The president of the United States is bald.

(8.8) The king of France is bald.

At the time of writing, (8.7) is false. However, (8.8) is neither true nor false, because there is no king of France. The definite noun phrase "the king of France" *presupposes*, roughly, that there is exactly one king of France; because this presupposition is violated, the sentence can't be interpreted, i.e. assigned a truth value. There are many other lexical expressions and syntactic constructions beyond definite noun phrases that trigger presuppositions; strong noun phrases are one of them.

Presuppositions behave differently than the ordinary assertional content of a sentence that we have tried to capture by its truth conditions in Chapter 3. For example, the negation of a sentence usually has the same presuppositions as the original sentence ("The king of France is *not* bald"), and the same is true for questions ("Is the king of France bald?"). When human listeners are confronted with an utterance that triggers a presupposition, they try to reconcile the presupposition with their world knowledge. If they didn't know that the presupposition is true, they will typically try to *accommodate* it, i.e. integrate it into their world knowledge. Presuppositions can be *cancelled* (or *filtered*): For instance, the sentence "If France has a king, then the king of France is bald" doesn't presuppose that France has a king. The problem of determining those presuppositions that are triggered but not cancelled is called the *projection problem* of presuppositions.

For the present chapter, we are interested in the presuppositions of strong noun phrases, and whether they can help us understand why the rules (b) and (c) appear to weaken readings. Unfortunately, while it seems to be widely accepted in the literature that strong noun phrases trigger existential presuppositions, it appears that the only time these presuppositions have been worked out in detail was (Bergmann 1981). Bergmann defines a four-valued variant of first-order logic in which a sentence is on the one hand true or false, and on the other hand, either *secure* or insecure; roughly, a formula is secure in a model if its presuppositions are satisfied by the model. However, multi-valued accounts of presuppositions seem to have fundamental difficulties with the projection problem. In addition, Bergmann only defines the *truth conditions* of the existential presuppositions as part of the semantics of her four-valued logic. We would like to have access to the presuppositions as formulas that we can then flexibly combine with the assertional content of a sentence.

So we have to rely on our linguistic intuitions, especially with respect to the example sentences above, to define the precise presuppositions of a universal noun phrase. This is easy in the case of (b): The noun phrase "every researcher of a com-

pany" presupposes that there is a researcher of a company, i.e. $(\underline{\forall}x : R(x))[S(x)]$ presupposes $\underline{\exists}x.R(x)$. But the situation is less clear if multiple universal noun phrases are nested within each other, as happens for the example sentence for (c). This sentence has the following two readings.[2]

(8.9) $(\underline{\forall}x : \mathsf{company}(x))[(\underline{\forall}y : \mathsf{researcher}(y) \underline{\wedge} \mathsf{workfor}(y, x))[\mathsf{paidwell}(y)]]$

(8.10) $(\underline{\forall}y : \mathsf{researcher}(y) \underline{\wedge} (\underline{\forall}x : \mathsf{company}(x))[\mathsf{workfor}(y, x)])[\mathsf{paidwell}(y)]$

Both readings have the obvious presuppositions that there is a researcher, and that there is a company. Beyond this, our intuition is that the second reading presupposes that there is a researcher who works for every company, and the first reading presupposes that every company (that we are talking about) employs some researcher, although the latter presupposition in particular is debatable.

Armed with these intuitions, we will now spend the rest of the chapter in the pursuit of a satisfactory definition of the existential presuppositions. Each definition of the existential presupposition will give rise to an alternative notion of "entailment" (technically, a relation E as we used above for the rewriting-based unsatisfiability criteria). Our main goal will be to find a plausible notion of presupposition and entailment such that all three rules (a, b, and c) weaken with respect to E and become a rewriting-based unsatisfiability criterion. We won't be able to reach this goal completely. But the different alternatives we consider should provide a starting point for future investigations that we hope will eventually lead to an unsatisfiability criterion that reflects the intuitions about the relative strength of readings, and gives rise to much smaller test sets than the criterion we developed earlier.

8.3.2 Presuppositions as separate formulas

As a first approximation, let's say that every universal quantifier (representing a universally quantified noun phrase) in a formula A triggers a separate presupposition formula $\pi(A)$, and that we add $\pi(A)$ to A conjunctively in order to strengthen the reading with its presuppositional content. We have seen in the discussion of (8.9) that it can be necessary to carry along quantifiers in whose scope the triggering universal quantifier is, so all free variables can be bound. This gives rise to the following definition.

[2]There is at least a third, *cumulative*, reading of the sentence that only expresses that every researcher of *some* company is paid well, and presupposes that there is at least one researcher-company pair, but this reading can't be represented using the scope mechanisms we use here, and thus we don't go into it any further.

$$\begin{aligned}
\pi(A) &= \bigwedge \pi_l(A) \\
\\
\pi_l((\underline{\forall} x : R)[S]) &= \pi_l(R) \cup (\underline{\forall} x : R)[\pi_l(S)] \cup \{\underline{\exists} x.R\} \\
\pi_l((\underline{\exists} x : R)[S]) &= \pi_l(R) \cup (\underline{\exists} x : R)[\pi_l(S)] \\
\pi_l(A(x,y)) &= \emptyset
\end{aligned}$$

The cases for the two quantifiers are similar in that they pass the presuppositions of their restrictions along unchanged, and that they pass the presuppositions of their scopes along in the original context of the scope. (The notation $(\underline{\forall} x : R)[\pi_l(S)]$ is shorthand for $\{(\underline{\forall} x : R)[S'] \mid S' \in \pi_l(S)\}$). The universal quantifier adds to this the statement that its restriction is not empty. If $A \in \mathcal{F}$, $\pi(A)$ has at most one free variable; this can be shown along the lines of the proof that any subformula of A has at most two free variables (Proposition 8.4). In particular, if A is closed, then $\pi(A)$ is also closed.

Two natural candidates for a rewriting-based unsatisfiability criterion based on π are the following:

$C_4 = (U_4, R_4, P_4)$: separate presuppositions, with rule e

$$\begin{aligned}
U_4 &= \{A \in \mathcal{F} \mid \Gamma \cup \{A, \pi(A)\} \text{ is unsatisfiable}\} \\
P_4 &= \{B \in \mathcal{F} \mid \Gamma \cup \{\pi(B) \underline{\Rightarrow} B\} \text{ is unsatisfiable}\} \\
R_4 &= \{a, b, c, e\}^{\leftrightarrow} \\
E_4 &= \{(A, B) \mid A \underline{\wedge} \pi(A) \underline{\wedge} \pi(B) \models B\}
\end{aligned}$$

$C'_4 = (U_4, R'_4, P_4)$: separate presuppositions, with rule f

$$\begin{aligned}
U_4 &= \{A \in \mathcal{F} \mid \Gamma \cup \{A, \pi(A)\} \text{ is unsatisfiable}\} \\
P_4 &= \{B \in \mathcal{F} \mid \Gamma \cup \{\pi(B) \underline{\Rightarrow} B\} \text{ is unsatisfiable}\} \\
R'_4 &= \{a, b, c, f\}^{\leftrightarrow} \\
E_4 &= \{(A, B) \mid A \underline{\wedge} \pi(A) \underline{\wedge} \pi(B) \models B\}
\end{aligned}$$

All rules in R_4 and R'_4 weaken readings with respect to E_4. This is possible because E_4 is a proper subset of the ordinary entailment relation. It doesn't require that A entails B, but only that A together with the presuppositions of A and of B entails B. This is plausible because if we were asked to decide whether the sentence "If A, then B" (a kind of natural-language deduction theorem) is true, we would accommodate the presuppositions of both A and B if we didn't know them already.

In addition, it is clear that $P_4(B)$ and $E_4(A, B)$ do imply $U_4(A)$. Unfortunately, C_4 and C'_4 are still not unsatisfiability criteria, as we can see when we check the remaining application conditions of Lemma 8.9. The culprit is Condition 3 in Lemma 8.9 (insertion into contexts): This condition fails for the rules (b) and (c),

both in the context $(\underline{\exists}x : R(x))[\bullet]$ and in the context $(\underline{\forall}x : \bullet)[S(x)]$. To see this more clearly, let's consider the case for the rule (b) and the existential context, $C_{\underline{\exists}} = (\underline{\exists}x : R(x))[\bullet]$; let L_b be the left-hand side and R_b the right-hand side of the rule (b). Condition 3 requires us to prove the following claim; we omit the additional precondition $C_{\underline{\exists}}[L_b]$, which makes no difference to the argument.

$$\begin{array}{ll} (\underline{\exists}x : R(x))[(\underline{\exists}z : A(x,z))[\underline{\exists}y.B(z,y)]] & (\pi(\underline{\exists}[R_b])) \\ (\underline{\exists}x : R(x))[(\underline{\forall}y : (\underline{\exists}z : A(x,z))[B(z,y)])[C(y)]] & (\underline{\exists}[L_b]) \\ \hline \therefore\ (\underline{\exists}x : R(x))[(\underline{\exists}z : A(x,z))[(\underline{\forall}y : B(z,y))[C(y)]]] & (\underline{\exists}[R_b]) \end{array}$$

If a theorem prover (say, based on a tableau calculus) attempted to prove this entailment, it would start with the first two formulas as they are and the negation of the third, and it would then try to derive a contradiction. Eliminating the quantifier $\underline{\exists}x$ on the first two lines would give us formulas like $R(c) \mathbin{\underline{\wedge}} \pi(R_b)[c/x]$ and $R(d) \mathbin{\underline{\wedge}} L_b[d/x]$, where c and d are different constants. The third line would be expanded to a formula of the form $\underline{\forall}x.R(x) \mathbin{\underline{\Rightarrow}} \underline{\neg} \ldots$. In order to proceed with the proof, the variable x in this formula must be instantiated either with c or with d. But the c instance can only be combined with the c instance of the first line afterwards, and the d instance can only be combined with the d instance of the second line, and this is not enough to derive a contradiction.

The key problem here is that when we computed $\pi(C_{\underline{\exists}}[R_b])$, we *copied* the existential quantifier that occurs in $C_{\underline{\exists}}$, and when the calculus started expanding these formulas, it was forced to choose a different witness for each copy. This made it impossible to combine information from different lines after a certain point in the proof. This is essentially the same problem that Heim (1983b) criticised in Karttunen and Peters's (1979) account of presuppositions, which performed a similar separation of the assertional content of a sentence and its presupposition into two different formulas. It is most clearly visible in examples like the following:

(8.11) A student managed to become a professor.

This sentence presupposes that *the same* (and not just an arbitrary) student tried to become a professor.

8.3.3 Presuppositions in place

So let's now investigate a different definition, which never copies quantifiers from the original formula, but instead "folds" presuppositions into the original formula at the location where they are triggered.

If A is a formula, we'll define A^+ as the result of enriching A with its presuppositions in such a way that if A^+ occurs positively, a calculus will unpack the presuppositions to a positive polarity as well. A^- is A, enriched in such a way that if it occurs negatively, the presuppositions will be unpacked to a positive polarity. The idea is similar to $A \mathbin{\underline{\wedge}} \pi(A)$ and $\pi(B) \mathbin{\underline{\Rightarrow}} B$ above, but with the difference that the presuppositions are now folded into the original formula, reusing its quantifiers rather than copying them.

A^+ and A^- are defined as follows:

$$\begin{array}{rcl}
((\underline{\forall}x : R)[S])^+ & = & (\underline{\exists}x.R) \mathbin{\underline{\wedge}} ((\underline{\forall}x : R^-)[S^+]) \\
((\underline{\forall}x : R)[S])^- & = & (\underline{\exists}x.R) \mathbin{\underline{\Rightarrow}} ((\underline{\forall}x : R^+)[S^-]) \\
((\underline{\exists}x : R)[S])^\epsilon & = & (\underline{\exists}x : R^\epsilon)[S^\epsilon] \\
A(x,y)^\epsilon & = & A(x,y),
\end{array}$$

where $\epsilon \in \{+,-\}$. Indeed, A^+ is stronger than A, and A^- is weaker than A, and hence its negation is stronger than the negation of A, as the following lemma shows.

Lemma 8.11. *For any formula $A \in \mathcal{F}$,*

$$A^+ \models A \models A^-.$$

Proof. Structural induction over A. The induction steps for $(\underline{\forall}x : R)[S]$ and $(\underline{\exists}x : R)[S]$ each require a half-page tableau proof. □

The obvious candidates for an unsatisfiability criterion based on this definition are the following.

$C_5 = (U_5, R_5, P_5)$: presuppositions in place, with rule e

$$\begin{array}{rcl}
U_5 & = & \{A \in \mathcal{F} \mid \Gamma \cup \{A^+\} \text{ is unsatisfiable}\} \\
P_5 & = & \{B \in \mathcal{F} \mid \Gamma \cup \{B^-\} \text{ is unsatisfiable}\} \\
R_5 & = & \{a,b,c,e\}^\leftrightarrow \\
E_5 & = & \{(A,B) \mid A^+ \models B^-\}
\end{array}$$

$C'_5 = (U_5, R'_5, P_5)$: presuppositions in place, with rule f

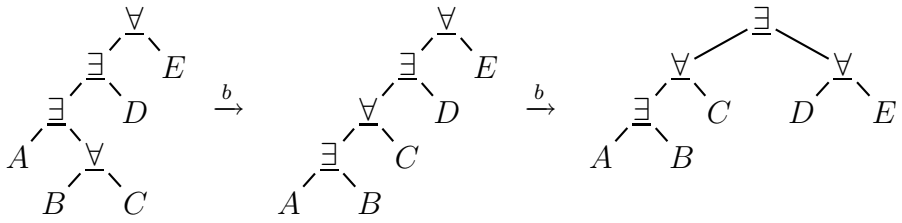
$$\begin{array}{rcl}
U_5 & = & \{A \in \mathcal{F} \mid \Gamma \cup \{A^+\} \text{ is unsatisfiable}\} \\
P_5 & = & \{B \in \mathcal{F} \mid \Gamma \cup \{B^-\} \text{ is unsatisfiable}\} \\
R_5 & = & \{a,b,c,f\}^\leftrightarrow \\
E_5 & = & \{(A,B) \mid A^+ \models B^-\}
\end{array}$$

Indeed, both C_5 and C'_5 satisfy the "insertion into context" condition that failed for the separate presuppositions; this can be proved by structural induction over

the context around the redex. In addition, E_5 still makes all three rewrite rules (a, b, and c) weakening.

However, the two systems are still not unsatisfiability criteria because they violate the transitivity condition. This is not surprising, as transitivity would mean that $A^+ \models B^-$ and $B^+ \models C^-$ would have to imply $A^+ \models C^-$, for any sequence $A \rightarrow B \rightarrow C$ of rewrite steps. Roughly, B together with the presuppositions of A would have to be strong enough to derive the presuppositions of B, which is not necessarily true. The following sequence of rewrite steps is a counterexample:

(8.12)



Because (b) weakens formulas with respect to E_5, each adjacent pair of formulas in this sequence is in the relation E_5, but the first and third formula are not.

One possible way of repairing the transitivity condition is to replace E_5 by relations such as $A^+ \models B^+$ or $A^- \models B^-$, along with their associated test and unsatisfiability conditions. These relations are clearly transitive, but now the first relation doesn't make the rule (c) weakening, and the second relation doesn't make (b) weakening, and more importantly, neither supports insertion into contexts. Again, this is not surprising: Each relation requires us to prove the presuppositions of a formula on the right-hand side of a rule using the left-hand side and its presuppositions. This is not given in our rules, and it also contradicts the intuition that the presuppositions of *both* sides should be available as axioms in a proof.

8.3.4 A binding-theory account

We can obtain one more candidate for a presupposition-based unsatisfiability criterion by looking at *binding theories* of presupposition (van der Sandt 1999; Geurts 1999). These theories assume that a presupposition is triggered at some location in the semantic representation of the sentence, and then it can float upwards within the formula, until it is cancelled or accommodated. The positions in the formula at which the presupposition can be placed is subject to a number of constraints, the most central of which is that all variables in the presupposition must still be bound (hence the name "binding theories").

Our final candidate for an unsatisfiability criterion attempts to place the presupposition at the highest position in the formula such that (a) this position dominates the presupposition trigger, and (b) all variables in the presupposition are still bound as they would be if the presupposition had been placed at the trigger. We define formulas $A^{p,+}$ and $A^{p,-}$, with the intention that the former represents the formula $A \in \mathcal{F}$ together with its presuppositions, and the latter represents A conditioned upon its presuppositions, in the same spirit as above. These formulas will be inductively built up from intermediate results $A^{\text{ass},\epsilon}$. We carry along the set of presuppositions that have been triggered but not yet accommodated in the expressions A^{pre}, and write A^{pre}_x for the set of those formulas in A^{pre} that have x as a free variable.

$$
\begin{array}{rcl}
A^{p,+} & = & (\bigwedge A^{\text{pre}}) \mathbin{\underline{\triangle}} A^{\text{ass},+} \\
A^{p,-} & = & (\bigwedge A^{\text{pre}}) \mathbin{\underline{\Rightarrow}} A^{\text{ass},-}
\end{array}
$$

$$
\begin{array}{rcl}
((\underline{\forall} x : R)[S])^{\text{ass},+} & = & \underline{\forall} x.((\bigwedge R^{\text{pre}}_x) \mathbin{\underline{\Rightarrow}} R^{\text{ass},-}) \mathbin{\underline{\Rightarrow}} ((\bigwedge S^{\text{pre}}_x) \mathbin{\underline{\triangle}} S^{\text{ass},+}) \\
((\underline{\forall} x : R)[S])^{\text{ass},-} & = & \underline{\forall} x.((\bigwedge R^{\text{pre}}_x) \mathbin{\underline{\triangle}} R^{\text{ass},+}) \mathbin{\underline{\Rightarrow}} ((\bigwedge S^{\text{pre}}_x) \mathbin{\underline{\Rightarrow}} S^{\text{ass},-}) \\
((\underline{\forall} x : R)[S])^{\text{pre}} & = & \{\underline{\exists} x.R\} \cup (R^{\text{pre}} - R^{\text{pre}}_x) \cup (S^{\text{pre}} - S^{\text{pre}}_x)
\end{array}
$$

$$
\begin{array}{rcl}
((\underline{\exists} x : R)[S])^{\text{ass},+} & = & \underline{\exists} x.(\bigwedge R^{\text{pre}}_x \cup S^{\text{pre}}_x) \mathbin{\underline{\triangle}} (R^{\text{ass},+} \mathbin{\underline{\triangle}} S^{\text{ass},+}) \\
((\underline{\exists} x : R)[S])^{\text{ass},-} & = & \underline{\exists} x.(\bigwedge R^{\text{pre}}_x \cup S^{\text{pre}}_x) \mathbin{\underline{\Rightarrow}} (R^{\text{ass},-} \mathbin{\underline{\triangle}} S^{\text{ass},-}) \\
((\underline{\exists} x : R)[S])^{\text{pre}} & = & (R^{\text{pre}} - R^{\text{pre}}_x) \cup (S^{\text{pre}} - S^{\text{pre}}_x)
\end{array}
$$

$$
\begin{array}{rcl}
A(x,y)^{\text{ass},\epsilon} & = & A(x,y) \\
A(x,y)^{\text{pre}} & = & \emptyset
\end{array}
$$

Presuppositions are triggered in the rule for universal quantifiers, on the fifth line. Then the rules for a quantifier for variable x extract all of the presuppositions accumulated so far that have x as a free variable, and adds these presuppositions to the formula; at the same time, they are removed from the set of open presuppositions. This is done in a way that guarantees that all occurrences of presuppositions in a formula $A^{p,+}$ will be positive, and all occurrences in the formula $A^{p,-}$ will be negative.

The definition gives rise to two final candidates for unsatisfiability criteria.

$C_6 = (U_6, R_6, P_6)$: binding-theoretic presuppositions, with rule e

$$
\begin{array}{rcl}
U_6 & = & \{A \in \mathcal{F} \mid \Gamma \cup \{A^{p,+}\} \text{ is unsatisfiable}\} \\
P_6 & = & \{B \in \mathcal{F} \mid \Gamma \cup \{B^{p,-}\} \text{ is unsatisfiable}\} \\
R_6 & = & \{a, b, c, e\}^{\leftrightarrow} \\
E_6 & = & \{(A, B) \mid A^{p,+} \models B^{p,-}\}
\end{array}
$$

$$
\begin{array}{rcl}
\multicolumn{3}{l}{C_6' = (U_6, R_6', P_6)\text{: binding-theoretic presuppositions, with rule } f} \\
\hline
U_6 & = & \{A \in \mathcal{F} \mid \Gamma \cup \{A^{p,+}\} \text{ is unsatisfiable}\} \\
P_6 & = & \{B \in \mathcal{F} \mid \Gamma \cup \{B^{p,-}\} \text{ is unsatisfiable}\} \\
R_6 & = & \{a, b, c, f\}^{\leftrightarrow} \\
E_6 & = & \{(A, B) \mid A^{p,+} \models B^{p,-}\}
\end{array}
$$

We conjecture that C_6 and C_6' might indeed be unsatisfiability criteria, but this is hard to prove because the definition of $A^{p,\epsilon}$ is somewhat unwieldy: Because the location in which a presupposition is accommodated depends on where its free variables are bound, the structure of $A^{p,\epsilon}$ is difficult to predict from the structure of A, which makes structural induction difficult to apply.

One more technical conjecture that would help with the proof is that for all formulas $A \in \mathcal{F}$, we have $A^{p,+} \models A^{+}$ and $A^{-} \models A^{p,-}$. This conjecture is motivated by the fact that presuppositions are generally accommodated higher up in $A^{p,\epsilon}$ than in A^{ϵ}, which makes them depend on fewer conditions. If the conjecture were true, we could conclude easily that for any rewrite step $A \to B$ we have $A^{p,+} \models B^{p,-}$. It would remain to verify transitivity and insertion into contexts. So far, we have not found any counterexamples; for instance, the first and third formula in the earlier transitivity counterexample (8.12) are in the relation E_6.

8.4 Summary

In this chapter, we set out to solve the problem of strengthening a dominance constraint in order to eliminate constructive solutions that are unsatisfiable or inconsistent with world knowledge. To this end, we investigated the problem of recognising whether all N constructive solutions of an alternating chain are unsatisfiable, while using less than N theorem prover calls. We defined *unsatisfiability criteria* as a general framework for approaching this problem, and we proposed to use rewrite systems to specify unsatisfiability criteria conveniently. Based on this, we presented a rewrite system whose (fewer than N) normal forms can serve as a test set for ordinary unsatisfiability, and thus solved our original problem.

However, we realised at this point that ordinary entailment is more cautious in declaring one reading of a sentence stronger or weaker than another than our intuitive judgments; this has the consequence that test sets for ordinary unsatisfiability are generally much larger than they should intuitively be. The discrepancy is caused by the fact that universally quantified noun phrases trigger existential presuppositions. We investigated various linguistically plausible definitions of these presuppositions. While we didn't arrive at a completely satisfactory solution, we

made some progress in understanding the problems, and thus prepared the issue for future research.

There are a number of further directions along which the work in this chapter could be extended. Most obviously, the fragment of dominance constraints to which we can apply rewriting-based unsatisfiability criteria needs to be expanded. On the one hand, alternating chains can't describe the semantic representations of sentences with transitive verbs. On the other hand, we intend to use unsatisfiability criteria in the context of the enumeration algorithm from Section 5.6, but alternating chains are not invariant under the Choice rule applied there. Linguistically, all proposed definition candidates remain a bit naive because none of them addresses the projection problem. There are also a number of technical annoyances, such as the fact that we can't use the equivalence rules (e) and (f) in the same rewrite system. This problem could perhaps be solved by considering rewriting modulo ef-equivalence.

The problem of eliminating unsatisfiable readings from an underspecified description has been considered before (Schiehlen 1999), but (as far as we are aware) only for specific rules of object-language inference, and not with the ambition of capturing interactions with arbitrary world knowledge. It differs from the literature on *direct deduction* (van Deemter 1996; König and Reyle 1996; Monz and de Rijke 1998; Jaspars and Koller 1999) in that these approaches try to derive from an underspecified description φ another underspecified description φ' whose readings follow logically from the readings of φ. All such approaches suffer from the problem that it is very difficult to come up with a satisfactory definition of "underspecified entailment", i.e. the relation between underspecified descriptions that direct deduction should model. Perhaps most closely related to the work reported here in its perspective on using the meaning of the readings to disambiguate the description is (Chaves 2003), who aims at eliminating readings that are equivalent to some other reading.

Chapter 9

Conclusion

This chapter summarises the thesis and presents ideas for future work.

9.1 Summary

In this thesis, we have shown how constraint programming and graph algorithms can be used to resolve natural language ambiguities. The larger part of the thesis was concerned with the investigation of dominance constraints; then we applied them to the problem of resolving scope ambiguities.

The thesis was roughly split in three parts. Its first part (Chapters 2–4) was an introduction to scope underspecification with dominance constraints. We motivated and defined dominance constraints as a language of tree descriptions in Chapter 2, and extended them with an account of object-language variable binding based on explicit binding functions. Then we showed how dominance and binding constraints could be systematically computed as underspecified semantic representations of English sentences (Chapter 3). It turned out that underspecification allows us to keep the syntax-semantics interface simple, and that this is especially true for dominance constraints because we can introduce all variable binding rule-locally. Third, we reviewed basic algorithms for solving dominance constraints (Chapter 4). We defined a saturation algorithm for the dominance constraint language $\mathcal{SN}$ and proved its correctness. Then we explored a generic extension of the saturation algorithm to binding constraints, and finally presented a different kind of dominance constraint solver based on finite set constraints.

In the second part of the thesis (Chapters 5 and 6), we applied *graph-based methods* to various problems related to dominance constraints. Chapter 5 showed that nor-

mal dominance constraints could be translated into dominance graphs, in such a way that their solved forms correspond. By characterising solvability of dominance graphs as the absence of simple hypernormal cycles, and reducing the detection of hypernormal cycles to a matching problem, we were able to give a polynomial algorithm for solvability of dominance graphs, and hence of normal dominance constraints. Because we are specifically interested in *constructive* solutions of a dominance constraint in underspecification, and not all solved forms have constructive solutions, we explored hypernormally connected dominance constraints in Chapter 6, and proved that all their solved forms are simple and thus have constructive solutions. Hypernormally connected constraints were again defined in terms of dominance graphs, and carried over to constraints by way of the constraint/graph correspondence from Chapter 5. We also proved that hypernormally connected dominance constraints and hypernormally connected underspecified descriptions from Hole Semantics are equivalent, thus building a bridge between two previously unrelated underspecification formalisms. These results are practically relevant because all constraints that can be generated by the grammar from Chapter 3 are hypernormally connected, and we conjecture more generally that all constraints that are needed in underspecification are.

In the third and final part of the thesis (Chapters 7 and 8), we applied the methods from the first two parts to the problem of *ambiguity resolution*. We showed in Chapter 7 how the effect of anaphoric references on the resolution of scope ambiguities can be modelled, by recasting the anaphoric accessibility conditions of Dynamic Predicate Logic as a binding specification. Going back to the purely constraint-based view, we then extended the saturation algorithm from Chapter 4 to solve dominance and binding constraints for this new binding specification, and illustrated that this algorithm is indeed able to strengthen constraints so as to eliminate readings that violated the accessibility conditions. In Chapter 8, on the other hand, we defined an inference procedure that could recognise that all constructive solutions of an alternating chain contradicted given world knowledge. Such a procedure can be used to strengthen a constraint by making a local case distinction and committing to one choice if all constructive solutions of the other one are unsatisfiable. We also explored how presuppositions of strong noun phrases made the human intuition about the strength of readings different from ordinary first-order entailment, and analysed various candidates for a definition of such presuppositions.

From a perspective of efficient processing, the first two parts of the thesis represent a movement from a general formalism (dominance constraints) to increasingly specific fragments. It was on the one hand essential to define dominance constraints as a declarative, logic-based formalism with a well-understood semantics. On the other hand, the discovery that normal dominance constraints can be read as graphs

enabled us to bring the power of graph algorithms to bear on the problem of solving dominance constraints – thus the restriction of dominance constraints to normal dominance constraints reduced the complexity of the satisfiability problem from NP-complete to $O(n^2)$ (and with cleverer algorithms, $O(n)$). At the same time, we took great pains to ensure that the normal and the hypernormally connected fragments of dominance constraints contained all the constraints that we needed in underspecification. We believe that there is a general lesson in this. It is important that a formalism is expressive enough to allow a linguist the convenient modelling of natural language phenomena. But often the full expressivity isn't used in reasonable models, and we can look for fragments that are unrestrictive for the purpose of modelling, and can still be processed efficiently.

This thesis has also exhibited a number of advantages that dominance constraints have over other underspecification formalisms. The most obvious one is of course that solvers for dominance constraints are unparalleled in their efficiency; Fuchss et al. (2004) report an improvement of several orders of magnitude over the LKB solver for Minimal Recursion Semantics (Copestake and Flickinger 2000), which is itself quite efficient. The efficiency of our solvers is fueled by the availability of a clear definition and well-understood formal properties of the formalism. It is this formal clarity that made it possible to use logic-based and graph-based methods and e.g. establish the correspondence between dominance constraints and Hole Semantics. In addition, as far as we are aware, the ability of dominance and binding constraints to link specific nodes with a binding function is unique. This improves over other underspecification formalisms in that the syntax-semantics interface can be simpler because no variable naming mechanism is required. It also makes it possible for the underspecification formalism to take over some of the burden that is usually carried by dynamic logics such as DPL and DRT. These logics specify binary node relations for anaphoric accessibility, but they must encode them into semantic interpretation rules. We have seen in Chapter 7 that such relations can sometimes be specified much more straightforwardly in a logical language that is designed to talk about trees.

9.2 Outlook

There are many technical aspects of the work reported in this thesis that can still be improved in future research. We have listed many of them in the chapter summaries, along with other research that has already continued the work reported here. To conclude the thesis, we present some more high-level lines of research that spin off from our results.

XDG. One of the dominance constraint solvers in Chapter 4 was based on finite set constraints. This algorithm was more efficient than the saturation algorithm, and it already exhibited the "miracle of the green nodes" that was an indicator that dominance constraints have a useful polynomial fragment. But it was later superseded by the graph-based solver, which was again much more efficient.

Almost the same system of finite set constraints (with many extensions) is also at the heart of the current parser for *Extensible Dependency Grammar* (XDG, Duchier 2003; Debusmann 2003). XDG is a dependency grammar formalism that allows us to analyse the syntactic structure of a sentence, and supports e.g. the description of languages with free word order in a particularly elegant way. It is an interesting question whether the ideas behind the graph-based solvers we developed here can also be applied to the efficient parsing of XDG.

In addition, XDG can be used to describe the syntax and semantics of a sentence at the same time. This allows the grammar writer to specify a *relational* syntax-semantics interface, in which neither syntax nor semantics is more primitive than the other (Debusmann et al. 2004). The XDG framework supports the simultaneous representation of partial syntactic and semantic information, and in this way, it can be seen as a natural continuation of semantic underspecification. It will be interesting to see whether this elegant setup can also buy us computational efficiency, or whether it makes sense to keep the levels of linguistic representation apart.

Graph Configuration Problems. The reason why the finite set solvers for dominance constraints and XDG are so similar is that the processing problems are closely related: Both require us to *configure* a known set of nodes into a tree, while respecting various structural constraints (such as dominance or, in the case of XDG, valency).

It is interesting to investigate whether XDG solving and dominance constraint solving can be seen as instances of a more general *graph configuration problem*. We could follow the same approach as in this thesis, namely to develop a logic-based modelling language for graph configuration problems, and then to investigate its formal and computational properties, with the aim of eventually defining efficient graph algorithms to process them.

Preferences. We have argued in the introduction that safe inferences as presented in the Chapters 7 and 8 are only one of a number of mechanisms that human language users apply to deal with ambiguity. A second mechanism that warrants further research is to draw default inferences based on *preferences* – sta-

tistical information about how some ambiguity is *typically* resolved. In principle, preferences can be integrated into an architecture based on constraints and search as we have developed it here, by feeding into a search strategy that specifies how to traverse the search tree. An interface between constraint solving and preferences has been specified (but not yet evaluated) for XDG (Dienes et al. 2003), but not yet for dominance constraints.

Expressivity. In Chapter 6, we have put up the claim that all dominance constraints that are used in underspecification are hypernormally connected. This conjecture is corroborated by empirical evidence (Fuchss et al. 2004), but it still needs to be verified more carefully in practice. If it is true, it is a claim about the expressivity that is really needed in an underspecification formalism. Similar investigations about the logical primitives that a satisfactory formalism needs can be useful too: For instance, Fuchss et al. (2004) also show that the difference between the qeq relation in MRS and the dominance relation in dominance constraints doesn't matter in practice.

Presuppositions. One conclusion that we draw from the modelling of anaphoric accessibility as a binding specification in Chapter 7 is that one core function of a dynamic logic is to define structural relations between positions in a formula – and this is of course something that dominance constraints excel at. This aim comes out most clearly in the binding-theory approaches to presupposition, as e.g. presented by van der Sandt (1999). It is an interesting question whether the possible accommodation sites of a presupposition can be modelled and processed based on dominance and binding constraints. This would also make it possible to model the interaction of scope and presupposition in the spirit of Blackburn et al. (2001). But rather than using the presuppositions to filter the fully resolved scope readings as they do, we could again try to use propagation techniques as in Chapters 7 and 8 to eliminate undesired readings without enumeration.

Bibliography

Alshawi, H. (ed.) (1992). *The Core Language Engine*. Cambridge/London: MIT Press.

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

Althaus, E., D. Duchier, A. Koller, K. Mehlhorn, J. Niehren, and S. Thiel (2001). An efficient algorithm for the configuration problem of dominance graphs. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*, 815–824.

Althaus, E., D. Duchier, A. Koller, K. Mehlhorn, J. Niehren, and S. Thiel (2003). An efficient graph algorithm for dominance constraints. *Journal of Algorithms 48*(1), 194–219.

Apt, K. R. (2003). *Principles of Constraint Programming*. Cambridge University Press.

Barwise, J. and R. Cooper (1981). Generalized quantifiers and natural language. *Linguistics and Philosophy 4*, 159–219.

Beaver, D. (1997). Presupposition. In J. van Benthem and A. ter Meulen (eds), *Handbook of Logic and Language*, 939–1008. Elsevier.

Bergmann, M. (1981). Presupposition and two-dimensional logic. *Journal of Philosophical Logic 10*, 27–54.

Blackburn, P., J. Bos, M. Kohlhase, and H. de Nivelle (2001). Inference and computational semantics. In H. Bunt, R. Muskens, and E. Thijsse (eds), *Computing Meaning, Volume 2*, 11–28. Kluwer Academic Publishers.

Bodirsky, M., D. Duchier, J. Niehren, and S. Miele (2004). A new algorithm for normal dominance constraints. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

Bodirsky, M., K. Erk, A. Koller, and J. Niehren (2001a). Beta reduction constraints. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications*, Utrecht.

Bodirsky, M., K. Erk, A. Koller, and J. Niehren (2001b). Underspecified beta reduction. In *Proceedings of the 39th ACL*, Toulouse.

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

Bos, J. (2002). *Underspecification and resolution in discourse semantics*. Ph. D. thesis, Saarland University.

Brants, S., S. Dipper, S. Hansen, W. Lezius, and G. Smith (2002). The TIGER Treebank. In *Proceedings of the Workshop on Treebanks and Linguistic Theories*, Sozopol.

Butt, M., H. Dyvik, T. H. King, H. Masuichi, and C. Rohrer (2002). The parallel grammar project. In *Proceedings of COLING-2002 Workshop on Grammar Engineering and Evaluation*, 1–7.

Chaves, R. P. (2003). Non-redundant scope disambiguation in underspecified semantics. In B. ten Cate (ed.), *Proceedings of the Eighth ESSLLI Student Session*, Vienna, 47–58.

Comon, H. (1992). Completion of rewrite systems with membership constraints. In *Coll. on Automata, Languages, and Programming*, Volume 623 of *LNCS*. Springer.

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

Copestake, A. and D. Flickinger (2000). An open-source grammar development environment and broad-coverage english grammar using HPSG. In *Conference on Language Resources and Evaluation*.

Copestake, A., D. Flickinger, and I. Sag (1999). Minimal Recursion Semantics. An Introduction. Unpublished manuscript, available at `http://www.cl.cam.ac.uk/~aac10/papers/newmrs.pdf`.

Debusmann, R. (2003). Dependency grammar as graph description. In D. Duchier (ed.), *Prospects and Advances in the Syntax/Semantics Interface*, Lorraine-Saarland Workshop Series, Nancy/FRA, 79–84. LORIA.

Debusmann, R., D. Duchier, A. Koller, M. Kuhlmann, G. Smolka, and S. Thater (2004). A relational syntax-semantics interface based on dependency grammar. In *Proceedings of the 20th COLING*, Geneva.

van Deemter, K. (1996). Towards a logic of ambiguous expressions. In *(van Deemter and Peters 1996)*. Stanford: CSLI Publications.

van Deemter, K. and S. Peters (1996). *Semantic Ambiguity and Underspecification*. Stanford: CSLI.

Dienes, P., A. Koller, and M. Kuhlmann (2003). Statistical A* dependency parsing. In D. Duchier and G.-J. Kruijff (eds), *Proceedings of the Lorraine-Saarland Workshop on Prospects and Advances in the Syntax-Semantics Interface*, Nancy.

Duchier, D. (2000). A model-eliminative treatment of quantifier-free tree descriptions. In D. Heylen, A. Nijholt, and G. Scollo (eds), *Algebraic Methods in Language Processing, AMILP 2000, TWLT 16*, Twente Workshop on Language Technology (2nd AMAST Workshop on Language Processing), Iowa City, USA, 55–66. Universiteit Twente, Faculteit Informatica.

Duchier, D. (2003). Configuration of labeled trees under lexicalized constraints and principles. *Research on Language and Computation*. To appear.

Duchier, D. and C. Gardent (1999). A constraint-based treatment of descriptions. In H. Bunt and E. Thijsse (eds), *Third International Workshop on Computational Semantics (IWCS-3)*, Tilburg, NL, 71–85.

Duchier, D. and J. Niehren (2000). Dominance constraints with set operators. In *Proceedings of the First International Conference on Computational Logic*, Number 1861 in Lecture Notes in Computer Science, 326–341. Springer-Verlag.

Duchier, D. and S. Thater (1999). Parsing with tree descriptions: a constraint-based approach. In *Sixth International Workshop on Natural Language Understanding and Logic Programming*, 17–32.

Ebert, C. (2003). On the expressive completeness of underspecified representations. In *Proceedings of the 14th Amsterdam Colloquium*, Amsterdam.

Egg, M. (2003). Beginning novels and finishing hamburgers – remarks on the semantics of *to begin*. *Journal of Semantics 20*, 163–191.

Egg, M., A. Koller, and J. Niehren (2001). The constraint language for lambda structures. *Journal of Logic, Language, and Information 10*, 457–485.

Egg, M., J. Niehren, P. Ruhrberg, and F. Xu (1998). Constraints over lambda-structures in semantic underspecification. In *Proceedings of the Joint 17th COLING and 36th ACL*, 353–359.

Erk, K. and A. Koller (2001). VP ellipsis by tree surgery. In *Proceedings of the 13th Amsterdam Colloquium*, Amsterdam.

Erk, K., A. Koller, and J. Niehren (2003). Processing underspecified semantic representations in the constraint language for lambda structures. *Research on Language and Computation 1*, 127–169.

Erk, K. and J. Niehren (2003). Well-nested parallelism constraints for ellipsis resolution. In *11th Conference of the European Chapter of the Association of Computational Linguistics*, 115–122.

Fuchss, R., A. Koller, J. Niehren, and S. Thater (2004). Minimal recursion semantics as dominance constraints: Translation, evaluation, and analysis. In *Proceedings of the 42nd ACL*, Barcelona.

Gabow, H., H. Kaplan, and R. Tarjan (2001). Unique maximum matching algorithms. *Journal of Algorithms 40*, 159–183.

Gabsdil, M. and K. Striegnitz (2000). Classifying scope ambiguities. *Journal of Language and Computation 1*(2), 307–313.

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

Gervet, C. (1994). Conjunto: Constraint logic programming with finite set domains. In *Proceedings of the International Logic Programming Symposium (ILPS '94)*, 339–358.

Geurts, B. (1999). *Presuppositions and Pronouns*. Oxford: Elsevier.

Geurts, B. (2003a). Existential import. In E. Comorovski and K. von Heusinger (eds), *Existence: Syntax and Semantics*. Kluwer. To appear.

Geurts, B. (2003b). Reasoning with quantifiers. *Cognition 86*, 223–251.

Goralcikova, A. and V. Koubek (1979). A reduct-and-closure algorithm for graphs. In *Proceedings of the 8th Symposium on Mathematical Foundations of Computer Science*, Volume 74 of *LNCS*, 301–307. Springer.

Groenendijk, J. and M. Stokhof (1991). Dynamic predicate logic. *Linguistics & Philosophy 14*, 39–100.

Heim, I. (1983a). File change semantics and the familarity theory of definiteness. In R. Bauerle, C. Schwarze, and A. von Stechow (eds), *Meaning, use, and interpretation of language*. De Gruyter.

Heim, I. (1983b). On the projection problem for presuppositions. In M. Barlow, D. Flickinger, and M. Westcoat (eds), *Second Annual West Coast Conference on Formal Linguistics*, Stanford, 114–126.

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

Jaspars, J. and A. Koller (1999). A calculus for direct deduction with dominance constraints. In *Proceedings of the Twelfth Amsterdam Colloquium*, Amsterdam.

de Jong, F. and H. Verkuyl (1984). Generalized quantifiers: The properness of their strength. In J. van Benthem and A. ter Meulen (eds), *Generalized Quantifiers in Natural Language*, 21–43. Dordrecht: Foris Publications.

Kallmeyer, L. and A. K. Joshi (2003). Factoring predicate argument and scope semantics: Underspecified semantics with LTAG. *Research on Language and Computation 1*(1–2), 3–58.

Kamp, H. and U. Reyle (1993). *From Discourse to Logic*. Dordrecht: Kluwer.

Karttunen, L. and S. Peters (1979). Conventional implicature. In C.-K. Oh and D. A. Dinneen (eds), *Syntax and Semantics*, Volume 11: Presupposition, 1–56. New York: Academic Press.

Keller, W. R. (1988). Nested Cooper Storage: The proper treatment of quantification in ordinary noun phrases. In U. Reyle and C. Rohrer (eds), *Natural Language and Linguistic Theories*, Volume 35 of *Studies in Linguistics and Philosophy*, 432–447. Reidel.

Koller, A. (1998). Evaluating context unification for semantic underspecification. In I. Kruijff-Korbayová (ed.), *Proceedings of the Third ESSLLI Student Session*, Saarbrücken, Germany, 188–199.

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., K. Mehlhorn, and J. Niehren (2000). A polynomial-time fragment of dominance constraints. In *Proceedings of the 38th Annual Meeting of the Association of Computational Linguistics*, 368–375.

Koller, A. and J. Niehren (2000). On underspecified processing of dynamic semantics. In *Proceedings of COLING-2000*, Saarbrücken.

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.

Koller, A., J. Niehren, and K. Striegnitz (2000). Relaxing underspecified semantic representations for reinterpretation. *Grammars 3*(2-3).

Koller, A., J. Niehren, and S. Thater (2003). Bridging the gap between underspecification formalisms: Hole semantics as dominance constraints. In *Proceedings of the 11th EACL*, Budapest.

Koller, A., J. Niehren, and R. Treinen (2001). Dominance constraints: Algorithms and complexity. In *Proceedings of the Third International Conference on Logical Aspects of Computational Linguistics (Dec. 1998)*, Volume 2014 of *Lecture Note in Artificial Intelligence*, 106–125. Springer-Verlag.

Koller, A. and K. Striegnitz (2002). Generation as dependency parsing. In *Proceedings of the 40th ACL*, Philadelphia.

König, E. and U. Reyle (1996). A general reasoning scheme for underspecified representations. In H. J. Ohlbach and U. Reyle (eds), *Logic and its applications. Festschrift for Dov Gabbay. Part I.* Kluwer.

Lappin, S. and T. Reinhart (1988). Presuppositional effects of strong determiners: a processing account. *Linguistics 26*, 1021–1037.

Lasersohn, P. (1993). Existence presuppositions and background knowledge. *Journal of Semantics 10*, 113–122.

Lévy, J. (1996). Linear second order unification. In *International Conference on Rewriting Techniques and Applications*. Springer-Verlag.

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

May, R. (1985). *Logical Form. Its structure and derivation*. Cambridge: MIT Press.

Milsark, G. (1977). Towards an explanation of certain peculiarities in the existential construction in English. *Linguistic Analysis 3*, 1–30.

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.

Monz, C. and M. de Rijke (1998). A tableaux calculus for ambiguous quantification. In H. de Swart (ed.), *Proceedings of TABLEAUX 98*, Number 1397 in LNAI, 232–246. Springer Verlag.

Müller, T. and M. Müller (1997). Finite set constraints in Oz. In F. Bry, B. Freitag, and D. Seipel (eds), *13. Workshop Logische Programmierung*, Technische Universität München, 104–115.

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.

Niehren, J. and A. Koller (2001). Dominance Constraints in Context Unification. In M. Moortgat (ed.), *Proceedings of the Third Conference on Logical Aspects of Computational Linguistics (Dec. 1998)*, Volume 2014 of *Lecture Note in Artificial Intelligence*, Grenoble. Springer-Verlag.

Niehren, J., M. Pinkal, and P. Ruhrberg (1997a). On equality up-to constraints over finite trees, context unification, and one-step rewriting. In *Proceedings 14th CADE*. Townsville: Springer-Verlag.

Niehren, J., M. Pinkal, and P. Ruhrberg (1997b). A uniform approach to underspecification and parallelism. In *Proceedings ACL'97*, Madrid, 410–417.

Niehren, J. and S. Thater (2003). Bridging the gap between underspecification formalisms: Minimal recursion semantics as dominance constraints. In *41st Meeting of the Association of Computational Linguistics*, 367–374.

Niehren, J. and M. Villaret (2003). Describing lambda-terms in context unification. In P. Blackburn and J. Bos (eds), *Proceedings of the Fourth International Workshop on Inference in Computational Semantics (ICOS-4)*, Nancy.

Oepen, S., K. Toutanova, S. Shieber, C. Manning, D. Flickinger, and T. Brants (2002). The LinGO Redwoods treebank: Motivation and preliminary applications. In *Proceedings of the 19th International Conference on Computational Linguistics (COLING'02)*, 1253–1257.

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

Perrier, G. (2000). From intuitionistic proof nets to interaction grammars. In *Proceedings of the 5th TAG+ Workshop*, Paris.

Pinkal, M. (1996). Radical underspecification. In *Proceedings of the 10th Amsterdam Colloquium*, 587–606.

Pollard, C. and I. Sag (1994). *Head-driven Phrase Structure Grammar*. CSLI and University of Chicago Press.

Rambow, O., K. Vijay-Shanker, and D. Weir (1995). D-Tree grammars. In *Proceedings of the 33rd annual meeting of the Association of Computational Linguistics*, 151–158.

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

van der Sandt, R. (1999). Presupposition projection as anaphora resolution. *Journal of Semantics 9*, 333–377.

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.

Schiehlen, M. (1999). *Semantikkonstruktion*. Ph. D. thesis, Universität Stuttgart.

Schmidt-Schauß M. and K. Schulz (1998). On the exponent of periodicity of minimal solutions of context equations. In T. Nipkow (ed.), *International Conference on Rewriting Techniques and Applications*, LNCS.

Simon, K. (1988). An improved algorithm for transitive closure on acyclic digraphs. *Theoretical Computer Science 58*(1–3), 325–346.

Skut, W., T. Brants, B. Krenn, and H. Uszkoreit (1998). A linguistically interpreted corpus of German newspaper text. In *Proceedings of LREC-98*, Granada, 705–711.

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

Thiel, S. (2004). *Efficient Algorithms for Constraint Propagation and for Processing Tree Descriptions*. Ph. D. thesis, Department of Computer Science, Saarland University.

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

Index