Fachbereich 14 Informatik
Universität des Saarlandes

Constraint Languages for Semantic Underspecification

Diplomarbeit

Angefertigt unter der Leitung von
Prof. Dr. Manfred Pinkal
und
Prof. Dr. Gert Smolka

Alexander Koller

5. Februar 1999

Hiermit erkläre ich, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Saarbrücken, den 5. Februar 1999

Alexander Koller

Abstract

At all levels of linguistic analysis, natural language can be ambiguous. The numbers of readings of different ambiguous components of a sentence or discourse multiply over all these components, yielding a number of readings that can be exponential in the number of ambiguities. Both from a computational and a cognitive point of view, it seems necessary to find small representations for ambiguities that describe all readings in a compact way. This approach is called underspecification, and it has received increasing attention in the past few years.

Lately, two particularly elegant formalisms for the underspecified treatment of scope ambiguities in semantics have been proposed: Context Unification (Niehren et al. 1997b) and the Constraint Language for Lambda Structures, CLLS (Egg et al. 1998). Common to both is that they regard the term representing the semantics of a sentence as a tree and describe it by imposing tree constraints. Furthermore, both offer the expressive power to describe simple ellipses and their interaction with scope ambiguities.

This thesis investigates some formal properties of these two formalisms. It examines their relation and shows that, except for a few additional constructs of CLLS, both languages are equivalent in expressive power. In terms of computational complexity, this gives us the immediate result that the complexity of the satisfiability problem of CLLS is exactly the same as that of context unification, which, unfortunately, is unknown. The thesis further investigates the complexity of the satisfiability problem of dominance constraints, an important sublanguage of CLLS, and shows that it is NP-complete. In the course of the discussion of complexity, it also briefly explains how techniques from concurrent constraint programming can be applied to implement solution algorithms for these formalisms.

Acknowledgments

First of all, I would like to thank my professors, Manfred Pinkal and Gert Smolka. Together with the other members of the CHORUS team, they have created a unique research environment and have generally given me the impression that what I was doing was significant.

I am indebted to Joachim Niehren, who has drawn me into this field of research in the first place by infecting me with his enthusiasm. He has also been an excellent discussion partner who has contributed much to the work reported here.

I am grateful to Patrick Blackburn, Manuel Bodirsky, Markus Egg, Claire Gardent, Joachim Niehren, Manfred Pinkal, Gert Smolka, and Kristina Striegnitz for helpful comments on various drafts of the thesis.

Finally, I would like to thank my family and Stefanie Schmidt, who have been an invaluable source of support and inspiration throughout all of this.

Contents

Chapter 1

Introduction

In this chapter, we give a general introduction to the topic of the thesis, explaining the basic concepts and formalisms that we will be concerned with, and stating our main results. For now, our aim is to avoid formalities and work more by intuition and example than by definition and proof; we will deliver these in later chapters.

First of all, we introduce the concept of *underspecification*. Underspecification is an approach to the treatment of all sorts of natural-language ambiguities that attempts to avoid the (expensive) enumeration of all readings of an ambiguous expression. Underspecification has been employed on many levels of linguistic analysis, but we will restrict ourselves to semantic underspecification and, in particular, the underspecified treatment of scope ambiguities.

In the next two sections, we give intuitive "definitions" of the two formalisms we will be primarily concerned with: the Constraint Language for Lambda Structures (CLLS) and context unification (CU). We show how the phenomena mentioned in the first section can be analyzed formally in these languages. Our analysis of scope ambiguities in CU will be oversimplified; we will come back to this in a later chapter.

Next, we state the contributions that this thesis makes to the research on the aforementioned constraint languages. Our first contribution is to show that the expressive powers of CLLS and CU are equal; for every constraint in one language, there is a satisfiability equivalent constraint in the other language. An immediate consequence of this result is that the complexities of CLLS and CU are the same; but at this time, neither of these is known. This makes our second contribution interesting: We show that the satisfiability problem of dominance constraints, an important sublanguage of CLLS, is NP-complete.

Finally, we briefly review two earlier approaches to scope underspecification to put our work in a broader context.

1.1 Underspecification

1.1.1 Ambiguities

It is a well-known fact that at all levels of linguistic analysis, ambiguities occur. The following is a (not at all exhaustive) list of possible sources of ambiguities.

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

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

c. Coordination:
Birds eat small worms and frogs.

d. Quantifier scope:
Every man loves a woman.

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

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

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

Traditionally, computational processing of sentences like these requires the enumeration of all their readings. What makes this a challenge for natural language processing is that the number of readings of an ambiguous sentence grows exponentially with the number of ambiguities. Consider the following well-known example.

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

Each of the two sentences in this example contains four quantifiers, which means that each sentence admits $24 = 4!$ different orderings of the quantifiers. The sentences can be disambiguated independently; so together, they have $576 = 4! \cdot 4!$ readings. Some of these readings may mean the same, but they will still be distinguished in a traditional (say, first-order logic or DRT) analysis of the sentence. On the other hand, a human listener can process a sentence like this quite easily, perhaps without even being aware of its ambiguity. This suggests that we do not have to enumerate all readings in order to understand an ambiguous sentence.

1.1.2 Underspecification

This exponential growth becomes particularly problematic when one does not only want to enumerate readings, but wants to do some work with the meaning of the sentence. This happens in virtually all applications of natural language processing in real-world systems; for example, one might want to infer more information from the natural-language input. Operations such as deduction are computationally expensive on single formulae; executing them on each of an exponential number of them in turn makes the task completely unfeasible. This is an additional motivation for looking into ways of avoiding or delaying the enumeration of readings beyond the cognitive intuition of the previous paragraph.

One such way is *underspecification*. The idea behind this approach is to represent the meaning of a sentence not as the set of its readings, but as a single, compact representation from which the readings can be extracted if necessary. Since we operate on only one semantic representation per sentence (as opposed to an exponential number if we enumerate all readings), this can be much more efficient than the traditional enumerative treatment.

In order to ensure an improvement in efficiency, two immediate requirements must be met. First, while we are trying to delay the enumeration of readings for as long as possible and hope to lose many spurious readings along the way, we may still want to know the exact set of readings when all the expensive work has been done. This means that it must be possible to derive the set of readings from the underspecified representation with reasonably efficiency. Second, it must be possible to derive an underspecified representation from some other level of representation (e.g., an underspecified semantic representation from a syntactic analysis) efficiently and systematically.

As an example that shows that underspecification can work, consider the following deduction on semantic representations.

(1.3) *Every man loves a woman.*
John is a man.
―――――――――――――――
John loves a woman.

The first premise of this argument is ambiguous, but we still find the argument valid without having to enumerate the readings of the premise. If we represent every line of the argument in an underspecified way, we can say that there is a relation of underspecified entailment between the premises and the conclusion of the argument, and we can define *direct deduction* as deduction in a calculus that respects this entailment relation. There are some subtleties to be considered in a definition of underspecified entailment (van Deemter 1996), but it can be done; there are sound and complete calculi of direct deduction.

The focus of this thesis will be on formalisms for underspecified semantics; more precisely, both formalisms we will compare can handle scope ambiguities as in Example (1.1d), and one of them (CLLS) can also describe strict/sloppy ambiguities (1.1e). But this is not the only area that underspecification has been applied to; in fact, people have given underspecified accounts of all the other items in (1.1) as well. For example, Billot and Lang (1989) make use of so-called shared parse forests for the compact representation of syntactic attachment ambiguities (b); Marcus et al. (1983) employ dominance constraints for an underspecified, monotonic treatment of various syntactic phenomena, including attachment and coordination (c). An underspecified account of lexical ambiguities arising from polysemic words such as *school* has been given by Bierwisch (1983); lexical ambiguities between unrelated lexemes, such as in (a), can be treated, for example, in Jaspars's (1997) underspecified logic. Finally, Gardent and Webber (1998) give an underspecified account of discourse ambiguities such as (f).

But of course, underspecification has its limits. While some phenomena (such as discourse or attachment ambiguities) can be processed in an incremental, left-to-right fashion by adding to the underspecified descriptions and leaving all choices open until they can be refuted, this approach can probably not be extended to, say, a parsing model that processes a sentence from left to right and does not make choices about the types of words like *that*. This is not surprising, as there are "garden-path" sentences where even human speakers commit to one reading of the prefix of a sentence and have to backtrack later and choose another.

The other extreme of making choices in the incremental processing of language is by defaults: Whenever a choice comes up, we make it immediately, and if it turns out later that the choice was wrong, we backtrack and change it. This approach is especially popular in a variant where this decision is guided by statistical data. But it is not perfect, either; it makes deduction on partially analyzed texts nonmonotonic, and it predicts garden-pathing for sentences where a human speaker doesn't garden-path. For example, Marcus et al. (1983) argue that we do not garden-path in processing a sentence such as

(1.4) *I drove my aunt from Stuttgart's car.*

A parser that is based on defaults might commit early to attaching "my aunt" as the direct object of the verb; only when "car" is encountered would such a parser realize its mistake and backtrack to attach the aunt at a lower position in the tree.

It seems reasonable to assume that the "psychological reality" of sentence processing is somewhere between these two extremes, and finding the exact balance in a cognitive model would be an interesting subject of research. However, this question is way beyond the scope of this thesis, so let us agree for now that underspecification can be used to gain a computational advantage for a wide range of phenomena and hence, is interesting to study.

To conclude our brief overview of underspecification in general, we illustrate that one seemingly plausible way to represent semantic ambiguities in a single formula, namely writing down the disjunction of all readings, is not an appropriate representation. A major problem occurs when we try to analyze negated sentences. Normally, a negated sentence is analyzed simply as the negation of the analysis of the non-negated sentence. See what happens if we do this for a "disjunctive" analysis of a lexical ambiguity.

(1.5) *Mary goes to the bank.*

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

According to the systematic analysis of negations, the meaning of the sentence (1.6) should just be the negation of the sentence (1.5). But if we represent (1.5) disjunctively, for example as

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

(where b_1 and b_2 stand for the two different meanings of the word *bank*), this would predict the meaning of (1.6) to be

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

However, the disjunctive representation of the intuitive meaning of (1.6) would be

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

and these two formulae are not equivalent.

The example of disjunction is a very nice illustration of why there should be a clean distinction between an *object language* (the language in which the different readings of the sentence will be represented) and the *meta language*, in which the underspecified representations are written. As we have just seen, using object-level disjunction for underspecification runs into problems as soon as we want to analyze negations. But disjunction on the meta-level, expressing something like "the sentence means R_1 or the sentence means R_2", does not have this problem because it doesn't interact with OL negation. The approaches we are going to look into in this thesis distinguish very clearly between object-level and meta-level.

But while meta-level disjunction allows the *correct* representation of the meaning, it is still not a *compact* representation, and hence, we would not necessarily call such a representation underspecified. To make the distinction more explicit, it is sometimes said that underspecification aims for a *non-disjunctive* representation of the meaning of a sentence.

1.1.3 Scope Underspecification

The type of ambiguity that this thesis is primarily concerned with is the class of *scope ambiguities*, as in Example (1.1d). Their name becomes clear when we look at the logical representations of their meanings. The two formulae that correspond to the two readings of (1.1d) are

(1.7) $\forall x.(man(x) \to \exists y.(woman(y) \wedge love(x, y)))$

(1.8) $\exists y.(woman(y) \wedge \forall x.(man(x) \to love(x, y)))$

If we regard the quantifiers as firmly connected to their restrictions – i.e., next to the central $love(x, y)$ subformula, we have two "fragments" $\forall x.(man(x) \to \cdot)$ and $\exists y.(woman(y) \wedge \cdot)$ –, the main difference between the two formulae is in the scopes of the quantifiers. In the first reading, it is the universal quantifier that has wide scope; in the second, it is the existential one.

In general, not only quantifiers can participate in scope ambiguities, but also other scope-bearing objects such as negations and some verbs. For instance, the sentence (1.9) has two readings that are represented by the formulae (1.10) and (1.11).

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

(1.10) $\forall x.(boy(x) \to \neg gtm(x))$

(1.11) $\neg\forall x.(boy(x) \to gtm(x))$

Early, enumerative approaches to scope ambiguities enumerated all readings of a scope ambiguity by traversing the syntax tree of the sentence and raising quantifying NPs above the sentence node; the order in which they were adjoined determined the relative scope of the quantifiers they corresponded to. For example, the *Cooper storage* (Cooper 1983) approach equipped each node of the syntax tree with a store for NPs. The tree was traversed in a bottom-up fashion, and whenever an NP was encountered, it was added to its mother's store. In this way, all NP semantics were collected. Whenever an S node was encountered, NPs could be "discharged" by applying the respective quantifiers to the preliminary lambda terms that had been constructed so far. The choice where NPs were discharged was made nondeterministically.

Over the past few years, several approaches to an underspecified treatment of this phenomenon have been proposed, the most prominent of which are Quasi Logical Form (QLF, Alshawi and Crouch 1992), Underspecified DRT (Reyle 1993), Muskens's (1995) underspecified semantics, Hole Semantics (Bos 1996), and Minimal Recursion Semantics (MRS, Copestake et al. 1997). We will give a brief overview over two of them (QLF and Hole Semantics) in Section 1.5.

The key idea of most recent underspecified accounts of scope ambiguities is to break formulae into different fragments (as we have done above) and impose constraints on the way these fragments can be assembled to obtain the readings of the sentence. This will become clear in an example. Consider Figure 1.1, which displays the fragments that occur in the description of the meaning of (1.1d).

Intuitively, this picture means that every formula which describes a reading of the sentence consists of three fragments: one containing a universal quantification over men x, one containing an existential quantification over women y, and one expressing that x loves y. The love subformula must be outscoped by both quantifiers; but there is no information on the relative scopes of the two quantifiers. One possible reading is for the universal quantifier to outscope the existential one, corresponding to reading (1.7); the other is for the existential one to outscope the universal one, yielding reading (1.8).

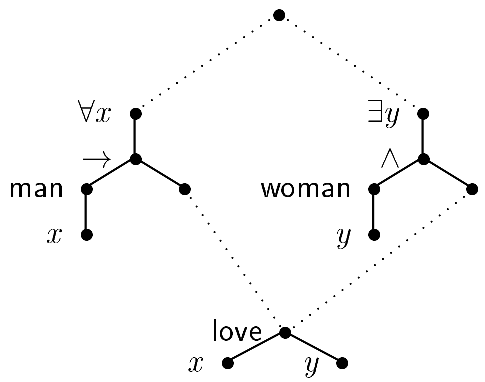


Figure 1.1: An underspecified representation of the meaning of Example 1.1d.

This intuition can be modeled formally in different ways. One is to allow "holes" in fragments and "plug" other fragments into these holes in such a way that given scoping constraints are obeyed. This is the basic idea of formalisms such as UDRT and Hole Semantics; we will get back to it in Section 1.5.

Another way of giving the graph a formal meaning relies on the close correspondence between trees and ground terms over ranked signatures (where each symbol is equipped with an arity). We will freely make use of this correspondence throughout the thesis. Once terms and formulae can be regarded as trees, sets of them can be described as the solutions of formulae in a tree logic. This is the method adopted in the approaches that we will be primarily concerned with in this thesis, namely, the Constraint Language on Lambda Structures (CLLS, Egg et al. 1998) and Context Unification (CU, Niehren et al. 1997b). These formalisms will be introduced in Sections 1.2 and 1.3 and defined formally in Chapter 2. The main topic of the thesis will be to investigate their formal relation and complexity.

CLLS and CU also capture the interaction of scope ambiguities with ellipses; in addition, CLLS correctly models the interaction of ellipses with intra-sentential anaphora. We have already seen an example of the latter (so-called *strict/sloppy ambiguities*) as Example 1.1d. The former is most obvious in so-called *Hirschbühler sentences* (Hirschbühler 1982):

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

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

1.2 CLLS

The fundamental idea underlying both CLLS and, as we will see, context unification is to regard formulae as trees and impose tree constraints that describe them. These constraints are conjunctions of atomic formulae that can be built from a small set of relation symbols with a fixed semantics; they can be satisfied by trees, and we will say that they *describe* these trees. The variables that occur in CLLS formulae denote *nodes* of trees; every node of such a tree is labeled with a symbol from a given signature Σ, and the arity of the label determines the number of children of this node. There are seven types of atomic constraints, the most primitive of which are labeling and dominance constraints. A labeling constraint $X : f(X_1, \ldots, X_n)$ is satisfied if the node denoted by X is labeled with f, and its immediate children are the nodes denoted by $X_1, \ldots, X_n$, from left to right. A dominance constraint $X \triangleleft^* Y$ is satisfied if the node denoted by X dominates (not necessarily immediately) the one denoted by Y. To avoid confusion: We take "dominance" to be the reflexive, transitive closure of "immediate dominance" throughout the thesis.

1.2.1 Scope ambiguities in CLLS

To give Figure 1.1 a formal meaning in CLLS, labeling and dominance constraints are (almost) sufficient. We can simply construct a CLLS constraint φ that is satisfied by exactly the same trees that the picture is supposed to describe. To this end, we associate a CLLS variable with every node of the graph. Whenever a node X of the graph is labeled, say, with f and its immediate children are the nodes $X_1, \ldots, X_n$, this is represented in φ as a conjunct $X{:}f(X_1, \ldots, X_n)$; whenever a dotted line goes down from a node X to a node Y, φ will contain a conjunct $X \triangleleft^* Y$.

For example, we would represent Fig. 1.1 by the following constraint:

(1.13) $X_0 \triangleleft^* X_1 \wedge X_0 \triangleleft^* X_2 \wedge$
$X_1{:}\forall x(X_3) \wedge X_3{:} \rightarrow (X_4, X_5) \wedge X_4{:}\mathsf{man}(X_6) \wedge X_6{:}x \wedge$
$X_2{:}\exists y(X_7) \wedge X_7{:} \wedge (X_8, X_9) \wedge X_8{:}\mathsf{woman}(X_{10}) \wedge X_{10}{:}y \wedge$
$X_5 \triangleleft^* X_{11} \wedge X_9 \triangleleft^* X_{11} \wedge X_{11}{:}\mathsf{love}(X_{12}, X_{13}) \wedge X_{12}{:}x \wedge X_{13}{:}y$

In this way, we have reinterpreted solid lines in the graph as immediate dominance constraints and dotted lines as dominance constraints. If we add the additional restriction that all solutions of the constraint we have just constructed only use the material we have mentioned in the constraint, its solutions will be exactly those intended by the intuitive idea of the figure. We will freely use these "constraint graphs" instead of the constraints they represent for better readability. It is essential to understand the difference between the nodes in the constraint graph and the nodes in a solution. While the former are a notational convenience and represent the variables of a CLLS constraint, the latter are part of the structures that satisfy these constraints and can be denoted by CLLS variables.

We have glossed over one problem that is worth mentioning: Our treatment of variable binding is not adequate yet. Imagine a constraint graph that is like Fig. 1.1, but in which all occurrences of y have been replaced by x – i.e., both quantifiers bind x. When the corresponding constraint is solved, it will depend on the chosen scope relation of the quantifiers which the actual binder of the variable occurrences x in the love subformula will be. This is similar to the capturing problems of lambda calculus; but the situation is even worse in that in lambda calculus, the problem can be solved by consistent renaming of variables and their binders, whereas in our example, the variables do not even have a unique binder. This means it simply does not really make sense to talk about "binders"; binders must be unique.

This is why CLLS takes resort to so-called *lambda structures*, trees that have been equipped with an additional *binding relation* between nodes. Instead of modeling variable binding by using variable names, we say that the node representing the bound variable and the node representing the binder are in the binding relation. This works not only for the first-order case we have considered so far, but also for encodings of arbitrary lambda terms. We will see in the next chapter how, exactly, this extension can be made.

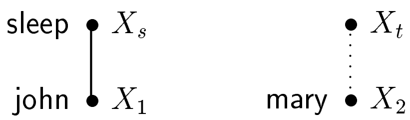
1.2.2 Ellipses

Let us now look into the CLLS treatment of ellipses. In Chapter 2, we will show in more detail how their interaction with anaphora and scope ambiguities can be handled; for now, we will restrict ourselves to a simple example to show the basic idea.

Consider the following simple ellipsis.

(1.14) *John sleeps. Mary does too.*

(1.15)



We have complete knowledge about the semantics X_s of the first sentence; it should be simply sleep(john). All we know about the semantics X_t of the second sentence without resolving the ellipsis, however, is that it should contain mary somewhere. Finally, X_t should be almost the same as X_s; but where X_s contains john, X_t should contain mary.

More formally, we can completely describe the tree corresponding to the semantics of the first sentence with labeling constraints, as in the left diagram in (1.15). In addition, we can partially describe the tree corresponding to the semantics of the second sentence with labeling and dominance constraints, as in the right diagram. But how do we express the parallelism requirement that relates the structures of the two trees?

In CLLS, this is done with a so-called *parallelism constraint*, which in this case looks like this:

$$X_s/X_1 \sim X_t/X_2.$$

This formula means that the trees below the nodes denoted by X_s and X_t, respectively, must be the same, except for the trees below X_1 and X_2, which can be different. From a different perspective, this means that the *contexts* of X_1 and X_2 in the trees below X_s and X_t must be the same, but different subtrees have been plugged into these position on both sides. We already know the tree below X_2, so we have complete information about both trees.

This works well for our simple example: The constraint we have constructed has a unique solution in which the semantics of the target sentence is determined to be sleep(mary). To cover more interesting examples, the actual definition of parallelism constraints is a bit more involved in order to take care of some subtleties, and we will defer its detailed discussion to Chapter 2.

1.3 Context Unification

Another constraint language on trees that has been used for semantic underspecification is the language of context unification (Niehren et al. 1997b), a variant of

Pinkal's (1996) "radical underspecification" approach. Although there are many parallels between the analyses provided by CLLS and CU, most of them may not be obvious at first sight, and there is a fundamental difference on the perspective on trees the two formalisms take. We will discuss these points in Section 1.4. Context unification is an interesting formalism in its own right that has been investigated in theoretical computer science; in addition, we will see that its investigation can provide results about the complexity of solving CLLS constraints.

1.3.1 Context constraints

Like CLLS, context unification is a constraint language on trees. CU formulae – conjunctions of equations between certain terms – can be satisfied by trees. Unlike CLLS, however, variables denote trees instead of nodes (or, as in the term/tree correspondence that we noted above, ground terms).

The terms t that CU constraints can equate have the following form.

$$t ::= f(t_1, \dots, t_n) \mid x \mid C(t).$$

In this definition, f is a tree constructor from the signature, x is a first-order variable that denotes a tree, and C is a so-called *context variable*. Context variables denote unary functions (called *context functions*) from trees to trees that insert their arguments into a fixed *context* – a tree with a hole, or alternatively, a tree from which a complete subtree has been cut away (Fig. 1.2). We will write contexts as terms with exactly one occurrence of the symbol $\bullet$, which represents the hole.

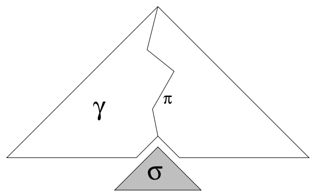


Figure 1.2: A context.

A context constraint is satisfied by a tree iff all of its variables can be mapped to trees and functions such that in every equation, the trees on the right-hand and left-hand sides are the same. For example, (1.17) is the unique solution of the context constraint (1.16). This can be seen easily by checking each equation in the constraint, from bottom to top.

(1.16) $x_0 = C(b) \wedge C(d) = f(a, C'(a)) \wedge C'(e) = g(d, e)$

(1.17) $x_0 = f(a, g(b, a))$
$C = f(a, g(\bullet, a))$
$C' = g(d, \bullet)$

Again, we will say that a context constraint *describes* a tree if the tree satisfies the constraint. *Context unification* is the problem of solving context constraints.

1.3.2 Examples

We will now take a look at how context constraints can be used as underspecified semantic representations. To this end, we will reconsider the examples of the section on CLLS and see how to treat them by context unification. In general, the linguistic coverage of context unification is the same as that of CLLS, except for phenomena related to anaphora because these cannot be represented in CU. (In fact, we will show later that CU is formally equivalent to a slightly restricted variant of CLLS.)

Recall the scope ambiguity *Every man loves a woman*, whose standard underspecified analysis was shown in Figure 1.1. We have seen in the previous section how the trees corresponding to its first-order logics semantics can be described with a CLLS constraint. The same trees can be described by the following context constraint.[1] We write the partial trees in a term notation to make the structure of the constraints more explicit. In a solution of the constraint, the variable x_0 denotes the entire tree corresponding to the semantics of the ambiguous sentence.

(1.18) $x_0 = C_1(\forall x(\rightarrow (\mathsf{man}(x), C_3(\mathsf{love}(x, y))))) \wedge$
$x_0 = C_2(\exists y(\wedge(\mathsf{woman}(y), C_4(\mathsf{love}(x, y)))))$

A closer look reveals the similarity to the graph in Figure 1.1. The fragments of the formula are still present; and instead of "dotted edges", we have used context variables as a device to leave space open. In each of the two solutions we are looking for, two of the context variables will be instantiated with the quantifier in the other constraint. The other two context variables will be instantiated with the "empty context" $\bullet$ (i.e., the context function that is the identity on trees). For example, x_0 is instantiated with the reading that assigns wide scope to the universal quantifier if we instantiate the context variables as follows:

[1] Actually, this context constraint has an additional solution that does not correspond to any linguistic reading. For sake of simplicity, however, we will go with the given constraint for now and fix things in Chapter 2.

(1.19) $C_1 = C_4 = \bullet$
$C_2 = \forall x(\to (\mathsf{man}(x), \bullet))$
$C_3 = \exists y(\wedge(\mathsf{woman}(y), \bullet))$.

To see how context constraints can be used for the description of ellipses, we will reanalyze Example 1.14. In the CLLS analysis, the crucial step towards calculating the semantics of the target sentence was that we could express that the contexts of the parallel elements within the sentences had to be the same. In context unification, we can say this even more directly by introducing a context variable C, which will denote this common context.

(1.20) *John sleeps. Mary does too.*

(1.21) $x_s = \mathsf{sleep}(\mathsf{john})$
$x_s = C(\mathsf{john}) \quad x_t = C(\mathsf{mary})$

The idea underlying this analysis is exactly the same as in CLLS: First, the semantics of the source sentence is described precisely, and then the semantics of the target sentence is determined by imposing the constraint that mary must appear in it in the same context in which john appeared in the source sentence.

Clearly, the CU analysis of ellipsis is similar to the HOU analysis of Dalrymple et al. (1991) (henceforth, DSP). But there are important differences. While the equality of higher-order terms that DSP are interested in is modulo $\alpha\beta\eta$ equivalence, the equality considered in CU is simple equality of trees. One effect of this is that CU has a much more direct handle on the actual structure of a term; this is important to express things like dominance or subtree relations. On the other hand, it means that the DSP analysis of strict/sloppy ambiguities does not carry over to CU because it crucially relies on the use of nonlinear lambda terms that can ignore arguments in beta reduction. (Context unification does not allow nonlinear functions, anyway.)

1.3.3 Computational aspects

Considering the simple structure of context constraints, it is surprisingly difficult to solve them. In fact, it is not even known if context unification is decidable.

However, it is known that the complexity of context unification is between that of string unification and that of second-order unification. Second-order unification (Goldfarb 1981) is known to be undecidable; context unification can be considered a slightly restricted form of this problem. On the other hand, string unification

(Makanin 1977), explained below, is known to be decidable, but the best known algorithm has a doubly exponential time complexity. There is a straightforward encoding of string unification problems as context unification problems.

String unification is the problem of solving a conjunction of equations between strings in which, beyond the usual characters from a given alphabet, string-valued variables can be used. A solution of such a problem is an assignment of ground strings to variables such that all equations are satisfied. To illustrate the high complexity of string unification (and with that, of context unification), we present a simple example and invite the reader to find its solutions or prove unsatisfiability. In the example, x and y are variables, and a and b are symbols from the signature.

(1.22) $axxbyx = xayyxy$

Embedding string unification into context unification is easy. For each character in the SU alphabet, we have one unary constructor in the CU signature; in addition, we have one nullary constructor ϵ in the signature to terminate the string equivalents. Trees from this signature can be read as strings from the root to the single terminal node; in this way, context variables correspond immediately to string variables. The above example, written as a context constraint, would look like this:

(1.23) $a(C(C(b(D(C(\epsilon)))))) = C(a(D(D(C(D(\epsilon))))))$

While the decidability of full context unification is an open problem that is being actively investigated, there are several known decidable fragments. Lévy (1996) restricts the full language such that every variable may only occur twice to obtain decidability. Maybe the most powerful known decidable fragment is the so-called *stratified* unification (Schmidt-Schauß 1994). But none of these contain the fragment that seems to be needed for the linguistic application.

A typical (not necessarily terminating) complete solution procedure for context constraints is the one given in (Niehren et al. 1997a, Appendix B). It attempts to infer a contradiction or specify a solution by nondeterministic application of rewrite rules. In its raw form, this procedure suffers from massive overgeneration and enormous runtimes. This can be remedied by introducing an object-language typing system and removing some of the most problematically nondeterministic rules (Koller 1998). These two changes make the procedure incomplete – some solutions of a constraint will not be found –, but the linguistically relevant examples are still found, and performance becomes acceptable. They will be explained in more detail in Chapter 4.

1.4 Contributions of this thesis

In this thesis, we make two contributions to the research on CLLS and context unification:

1. We show that for every context constraint, there is an satisfiability equivalent constraint of CLLS_0, a slightly restricted sublanguage, and vice versa.
2. We show that the satisfiability problem of the language of dominance (and labeling) constraints is NP-complete. This result stays true (with one minor restriction) if we allow the use of the other propositional connectives.

Both results were obtained in cooperation with Joachim Niehren. They have been published as (Niehren and Koller 1998) and (Koller et al. 1998).

There are several good reasons for a closer examination of the formal relation between CLLS and context unification as underspecification formalisms. First of all, while they are both tree logics, their perspectives on trees are different. In the terminology of Blackburn et al. (1995), the perspective of CLLS, whose variables denote nodes of trees, is "internal", while that of context constraints, whose variables stand for trees, is "external". Relating the two logics means relating the two perspectives.

At the same time, if we can show the equivalence of these formalisms and have any kind of information on the complexity of solving constraints on one side, we know immediately that the complexity of solving the constraints on the other side must be the same.

Finally, such a consideration is particularly relevant with respect to the project in whose context this research stands. This project, CHORUS, is a subproject of the DFG-funded Sonderforschungsbereich 378 "Ressourcenadaptive kognitive Prozesse" ("ressource-adaptive cognitive processes"). Its general goal is underspecified semantic representation, with a focus on not only achieving good linguistic coverage, but also keeping the various operations (disambiguation, direct deduction) computationally feasible. Context unification, which had been the underspecification formalism of choice in the project in 1997, was replaced by CLLS towards the end of that year for various reasons that we will explain in Chapter 2. The equivalence result between CLLS and context unification both justifies to take over old analyses and contributes to an *a posteriori* justification of some of the CU analyses based on the CLLS analyses.

The first contribution of this thesis is to show that CLLS_0, the sublanguage of CLLS that only allows labeling, dominance, and parallelism constraints, is equivalent to context constraints. It is possible to encode CLLS_0 constraints as context constraints, and vice versa, in such a way that both sides are satisfied by exactly the same trees.

This is not obvious. Fortunately, a major part of the proof can be reduced to the equivalence of context constraints and so-called *equality up-to* constraints, which was shown by Niehren et al. (1997a). In addition, there is an apparent similarity between dominance constraints and subtree constraints (Venkatamaran 1987), which can be written as context constraints in a straightforward manner. But the obvious idea of encoding dominance as a subtree relation is wrong, as the following example shows.

(1.24) $X : f(X_1, X_2) \wedge X_1 \lhd^* Y \wedge X_2 \lhd^* Y$

(1.25) $x{=}f(x_1, x_2) \wedge y{\ll}x_1 \wedge y{\ll}x_2$

Example (1.24), a dominance constraint in which the variables denote nodes in a tree, is unsatisfiable. X_1 and X_2 are nodes in disjoint positions of a tree, so there can't be any node Y that they both dominate. This is illustrated in Figure 1.3.

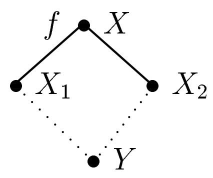


Figure 1.3: Constraint graph of (1.24).

In contrast, (1.25), a subtree constraint in which the variables denote trees, is satisfiable. A constraint $x{\ll}y$ only requires that the tree denoted by x is a subtree of that denoted by y, which is a much less strict condition than dominance of nodes. For example, one way to solve (1.25) is to assign $x = f(a, a)$ and $x_1 = x_2 = y = a$. (As it happens, this is exactly the reason why the CU analysis of the scope ambiguity in Example 1.18 is not entirely correct, as we will see later.) This shows that a naive encoding of dominance constraints as subtree constraints does not preserve satisfiability. The fundamental problem is that subtree constraints do not allow us to identify different *occurrences* of subtrees: As we have just seen, the subtree constraint above doesn't care that the subtree denoted by y appears in different places of the tree in different conjuncts. On the other hand, we can easily identify an occurrence of a subtree with a dominance constraint by specifying its root node.

We have called the idea we employ in our construction in Chapter 3 'nodes as contexts' (as opposed to the 'nodes as subtrees' approach in Example 1.24). This idea relies on the identifications of a context with the path from its root to its hole and of a node with the path leading to it from the root of the tree. If a dominance constraint is satisfied by a tree σ, we fix a tree variable $\top$ whose purpose it is to denote the entire tree σ. Then we introduce for each node variable in the dominance constraint a tree variable denoting the tree below the node (as above) and a context variable denoting the context of the node within σ. All of this can be enforced by context constraints. Finally, we recast every single dominance or labeling constraint as a context constraint. For example, dominance of nodes now simply means that the context associated with the upper node is a subcontext of that of the lower one. A detailed discussion of our encoding, along with a proof of its correctness, will follow in Chapter 3.

This equivalence in expressive power yields the immediate corollary that the complexity of the satisfiability problem of CLLS_0 constraints is the same as that of context unification; but we have seen above that the latter is unknown. A fragment about whose complexity we can say something, however – and this is the second contribution of the thesis –, is the language of dominance (and labeling) constraints (and various fragments of its full first-order language). Dominance constraints are not only a crucial fragment of full CLLS (as we have seen, they are the fundament of the CLLS treatment of scope ambiguities), but also an interesting and widely used constraint language in their own right (for other linguistic applications, see e.g. Marcus et al. 1983, Vijay-Shanker 1992, Gardent and Webber 1998).

Previous research on dominance constraints includes a first-order axiomatization by Backofen et al. (1995). In a situation where all trees can be enumerated (for instance, finite trees over a finite or countable signature), steps in a derivation from the axiomatization (which will eventually prove validity if the formula is valid) and enumeration of trees and checking if the formula is satisfied by them (which will eventually find a counterexample if one exists) can be interleaved to obtain a decision procedure for validity of first-order dominance constraints (Backofen, personal communication). In addition, Rogers and Vijay-Shanker (1994) have given a sound and complete calculus that derives so-called *quasi-trees* from (conjunctive) dominance constraints, from which tree structures solving the constraints can be extracted straightforwardly. The actual complexity of any language over the dominance constraints, however, has been unclear.

In this thesis, it is shown that the satisfiability problems of all languages of dominance constraints between the purely conjunctive constraint language (as presented above) and the existential fragment over the dominance constraints (including all propositional connectives) are NP-complete. An additional complexity result, which has been published in (Koller et al. 1998) as well, is that the first-order theory of dominance constraints has non-elementary complexity.

1.5 Previous approaches to underspecified semantics

To conclude our introduction and put the two main approaches to be considered into a broader context, we now give a brief and informal overview over earlier approaches to an underspecified treatment of scope ambiguities. From the wide variety of formalisms that we have listed above, the two we pick for a closer look are Quasi Logical Form and Hole Semantics. The former is of seminal importance for the field and has a broad coverage of linguistic phenomena. The latter is representative of a family of underspecification formalisms that is probably the most influential at this time. The most popular member of this family is UDRT (Reyle 1993; Schiehlen 1997), but Hole Semantics is much more accessible, and its basic ideas are essentially the same.

1.5.1 Quasi Logical Form

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

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

By way of example, consider the (heavily simplified) QLF representation of the scope ambiguity in Example 1.1d, repeated here as (1.26).

(1.26) *Every man loves a woman.*

(1.27) `_s:love(term(+m,` $\forall$, λX`.man(`X`)),`
`term(+w,` $\exists$, λY`.woman(`Y`)))`

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

The `love` formula is prefixed with a scoping list that is, at this point, unspecified and represented by the variable `_s`. Due to the free-variable constraint we mentioned above, any fully resolved QLF that can be derived from (1.27) must instantiate `_s` with a list that contains both `+w` and `+m`. This can be done in either order, yielding the two readings (1.28) and (1.30) below.

(1.28) `[+m,+w]:love(term(+m,` $\forall$, λX`.man(`X`)),`
`term(+w,` $\exists$, λY`.woman(`Y`)))`

(1.29) $\forall x.man(x) \rightarrow \exists y.(woman(y) \wedge love(x, y))$

(1.30) `[+w,+m]:love(term(+m,` $\forall$, λX`.man(`X`)),`
`term(+w,` $\exists$, λY`.woman(`Y`)))`

(1.31) $\exists y.woman(y) \wedge \forall x.(man(x) \rightarrow love(x, y))$

It is possible to treat various cases of ellipsis (including the interaction of VP ellipses with anaphora and scope ambiguities) in QLF (see, for example, Crouch 1995). For instance, if we continue the above scope ambiguity with (1.32) to obtain a Hirschbühler sentence, we would in a first step note the meaning of the target parallel element *several gorillas* as in (1.33). For our analysis of the target sentence, we stipulate a generalized quantifier several with the correct semantics.

(1.32) *Several gorillas do, too.*

(1.33) $\texttt{term(+g, several, }\lambda Z.\texttt{gorilla}(Z))$

To compute the meaning of the target sentence, we have to find a contextually salient antecedent for it; in this case, this will be (1.26). We can then extract a *substitution* that maps the QLF of the source parallel element to that of the target parallel element. In this case, this is

(1.34) $[\ \texttt{term(+g, several, }\lambda Z.\texttt{gorilla}(Z))\ /\ \texttt{term(+m, }\forall,\ \lambda X.\texttt{man}(X)),\ \texttt{+g/+m}\ ]$

This substitution, applied to the semantics of the source sentence, will produce the semantics of the target sentence. To ensure the correct treatment of Hirschbühler sentences, the actual application of the substitution is delayed until the scope relations in the source sentence are fully resolved. Application of the substitution will then not only replace the parallel quantifiers themselves, but will also insert the index of the target parallel element for that of the source parallel element in the scoping list. In this way, the scope parallelism of the sentences is enforced.

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

1.5.2 Hole Semantics

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

Hole Semantics is based on underspecification pictures such as Figure 1.1, which we repeat below as Fig. 1.4 in a slightly adjusted format, but gives it a different technical interpretation than CLLS. Formulae occurring in the nodes of such an underspecified representation (UR) are taken from the object language; but any subformula can be replaced by a so-called *hole* (h_0, h_1, h_2 in the picture). The function of holes is that other formulae can be *plugged* into them to obtain a larger formula. The lines in the graph are drawn from holes to formulae, and they express that the formulae must be subformulae of the formulae into whose holes they will be plugged. To take care of problems that can arise when the same formula occurs more than once in the graph, each fragment is given a unique identity, its *label* (l_1, l_2, l_3 in the picture). The graph can be represented as an upper semilattice specifying a partial order on holes and labels, and disambiguation means to make this order more specific.

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

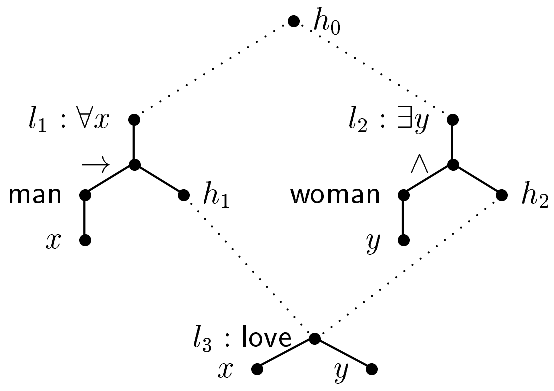


Figure 1.4: A scope ambiguity in Hole Semantics.

To see an example for such a plugging, we have equipped Fig. 1.1 with explicit holes and labels in Fig. 1.4. The UR presented in this picture has exactly two admissible pluggings. They are shown as (1.35) and (1.37), along with the predicate logic formulae they induce.

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

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

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

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

Hole Semantics also describes a simple way to obtain an underspecified model-theoretic semantics for underspecified representations from a semantics for the original object language: For every admissible plugging of a UR, it contains the (object-language) denotation of the induced formula. In itself, it does not say anything about the treatment of ellipses or about direct deduction. However, close relatives of Hole Semantics have been the subject of some research on direct deduction. For example, Reyle (1993) presents a sound and complete calculus for UDRT, and König and Reyle (1996) present another generalized underspecified logic that can be parameterized with an object language and provides a calculus of direct deduction.

The striking similarity of the graphical devices used to make CLLS and Hole Semantics representations more transparent suggests a formal connection between the technical methods of representation. Indeed, it is quite easy to encode every UR of Hole Semantics as a CLLS constraint. All we have to do is fully describe the fragments that constitute the UR with labeling constraints and then to reformulate the partial order on the holes and labels with corresponding dominance constraints. There is one subtlety, however: We must make sure that in a solution of the dominance constraint, fragments are not identified with each other. This can be done easily with inequality constraints that state that the nodes denoted by two variables must be different.

Plan of the thesis

In Chapter 2, we start off with a formal definition of the syntax and semantics of context unification and CLLS. Then we prove some basic results about the two languages; in particular, that we can allow CU formulae to contain equations between context-valued terms without changing the expressive power. We also show how to formalize some linguistic examples in both approaches.

In Chapter 3, we prove the formal equivalence between context unification and CLLS_0, the sublanguage of CLLS built up only from labeling, dominance, and parallelism constraints. More precisely, we present encodings of CU into CLLS_0 and vice versa such that both the original formula and its encoding are satisfied by exactly the same trees. The most interesting part of the proof is to show how to encode dominance constraints as context constraints.

In Chapter 4, we investigate the complexity of the satisfiability problem of dominance constraints. We present an algorithm that decides this problem and show that it is sound and complete and terminates in NP time. We also show how this algorithm can be extended to decide the satisfiability problems of larger logical languages over the dominance constraints without an increase in worst-case complexity. Next, we underpin this result by a proof of the NP-completeness of the satisfiability problem; so we cannot expect to find a faster algorithm for the full problem. To this end, we show how to encode formulae of propositional logic as satisfaction equivalent dominance constraints. Finally, we sketch an implementation of the solution algorithm and some recent results about the complexity of the first-order theory of dominance constraints, which is decidable and has non-elementary complexity.

In Chapter 5, we summarize the work reported in the thesis and present directions for further work. In particular, we will discuss requirements for a formal theory of underspecified beta reduction on CLLS constraints.

Chapter 2

CLLS and Context Unification

In this chapter, we give a formal definition of the syntax and semantics of the two constraint languages for semantic underspecification we will be looking into. We prove some basic results about trees and contexts and look into some additional examples of applying the two formalisms for linguistic analyses.

In the first section, we define CLLS. First, we define trees (in a fairly standard way) and tree structures; tree structures are trees with an additional interpretation function that interprets certain predicate symbols over the nodes of the tree. We then extend this definition to *lambda structures* – tree structures with an additional binding function. Lambda structures can be used to model lambda terms in a tree-like fashion. Based on these definitions, it is straightforward to define the syntax and semantics of CLLS.

A particularly interesting sublanguage of CLLS is the language of dominance (and labeling) constraints. Dominance constraints are a common formalism in computational linguistics, but the traditional variant of dominance constraints is different from ours. We investigate the relation between these two types of dominance constraints in Section 2.1.5, thus concluding the first section of this chapter.

In Section 2.2, we then look into the linguistic application of CLLS in some more detail. We see how the five-readings benchmark scope ambiguity *Every researcher of a company saw most samples* can be successfully analyzed in CLLS, and we demonstrate how to use CLLS for an analysis of Hirschbühler sentences.

Finally, we give a formal definition of context constraints, the language underlying context unification. This language is based on equations between terms that denote trees; these equations can make reference to so-called context variables, which denote functions that insert trees in fixed contexts. We extend the language to allow equations between context-valued terms and prove that these equations can be expressed in the original language of context constraints. As a further illustration of the linguistic application of CU, we give a correct analysis of scope ambiguities in CU.

We only introduce the core language of CLLS and don't go into various extensions that are irrelevant for our exposition and don't make a difference to complexity issues. One extension is to add *anaphoric links* (Egg et al. 1998) to lambda structures and corresponding constraints to specify them. These links can be used to model intrasentential anaphora and interact appropriately with parallelism constraints for a correct treatment of strict/sloppy ambiguities. Another extension is to add a relation of *binding equivalence* (Egg et al. 1999) to lambda structures, which can be used to soften the definition of parallelism constraints. In its simplest form (which we will adopt here), this is just the identity relation on nodes. Egg et al. (1999) show how it can be changed to solve problems that arise in the context of antecedent-contained deletion, a special case of VP ellipsis.

2.1 Syntax and Semantics of CLLS

CLLS is a constraint language that is interpreted over *lambda structures*, tree-like encodings of terms of lambda calculus. Lambda structures are based on *tree structures*, formalizations of trees that also interpret several predicate symbols. Variables in a CLLS formula denote nodes of a lambda structure. The syntax of CLLS is defined in the usual way as the language of conjunctions over applications of these predicate symbols.

Below, we will first define tree structures. We will then extend this definition to a definition of lambda structures and explain the correspondence between lambda structures and lambda terms. Finally, we define the syntax and semantics of CLLS constraints, which will be straightforward given the groundwork of the first two subsections.

2.1.1 Tree structures

Throughout the thesis, we assume that Σ is a ranked signature that contains function symbols or tree constructors $f, g, a, b, \ldots$, which are assigned arities by an arity function $\mathsf{ar} : \Sigma \to \mathbb{N}_0$. (We take $\mathbb{N}$ to be the set of positive integers and $\mathbb{N}_0$ to be the set of nonnegative integers.) We further assume that Σ contains at least two constructors, one of which is nullary. This is a minor restriction because the resulting logics become rather boring if there is only one possible finite tree, or none at all. In some cases, we will further restrict Σ to be finite, or to contain symbols of certain arities; but whenever we do, we will say this explicitly.

Following Courcelle (1983), we define a *tree domain* Δ to be a nonempty prefixed-closed subset of $\mathbb{N}^*$; i.e., the elements of Δ are words of positive integers. These words can be thought of as the paths from the root of a tree to its nodes. We write the concatenation of two words π and π' as $\pi \cdot \pi'$; whenever convenient, we leave the concatenation dot away and simply write $\pi\pi'$.

We define a *constructor tree* to be a function

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

with the additional property that for every $\pi \in \Delta$, if $\mathsf{ar}(\sigma(\pi)) = n$, then $\pi 1, \ldots, \pi n$ are in Δ, but no πk is, for any $k > n$. A *finite constructor tree* is a constructor tree whose domain is finite.

Throughout the thesis, we will always mean "finite constructor tree" whenever we say "tree".

Clearly, trees can be seen as ground terms over Σ, and we will frequently write them as such. The domain of a tree can be seen as the set of its nodes, and we will write it as $\mathrm{Dom}(\sigma)$. We write $\sigma.\pi{\downarrow}$ to mean that π is in the domain of σ.

Paths can select subtrees of a tree. Whenever σ is a tree and π is a path in $\mathrm{Dom}(\sigma)$, we define the selected subtree $\sigma.\pi$ as the function

$$\begin{array}{l} \sigma.\pi : \{\pi' \mid \sigma.(\pi\pi'){\downarrow}\} \to \Sigma \\ (\sigma.\pi)(\pi') = \sigma(\pi\pi'). \end{array}$$

$\sigma.\pi$ is undefined if π is not in the domain of σ.

The following "bottom-up" relation about selection is true:

Proposition 2.1. *If σ is a tree, π is a path such that $\sigma(\pi) = f$, and $n = \mathsf{ar}(f)$, then*

$$\sigma.\pi = f(\sigma.(\pi 1), \ldots, \sigma.(\pi n)).$$

The *tree structure* $\mathcal{M}^\sigma$ over the tree σ is a pair (σ, I), where I is an interpretation function that assigns relations on $\mathrm{Dom}(\sigma)$ to a set of fixed predicate symbols. We will use the same symbols for these relations and the predicate symbols; as the former are applied to paths and the latter are applied to CLLS variables, there is no danger of confusion. I will be fully determined by σ; so to specify a tree structure, it is sufficient to specify the underlying tree.

We now define the relations assigned to the predicate symbols by I. If $f \in \Sigma$ has arity n, the *labeling* relation $\pi{:}f(\pi_1, \ldots, \pi_n)$ is true iff $\sigma(\pi) = f$ and for all $1 \leq i \leq n$, $\pi_i = \pi i$. The *dominance* relation $\pi \triangleleft^* \pi'$ is true iff π is a prefix of π'. The *inequality* relation $\pi \neq \pi'$ is true iff π and π' are different.

Finally, there are two relations that are slightly more complex. The *similarity* relation $\pi \sim \pi'$ is true iff $\sigma.\pi = \sigma.\pi'$. The *parallelism* relation $\pi_1/\pi_1' {\sim} \pi_2/\pi_2'$ holds iff

1. there is a common "exception path" π_0, i.e. $\pi_1' = \pi_1\pi_0$ and $\pi_2' = \pi_2\pi_0$;
2. $\mathrm{Dom}(\sigma.\pi_1) - \pi_0 \cdot \mathrm{Dom}(\sigma.\pi_1') = \mathrm{Dom}(\sigma.\pi_2) - \pi_0 \cdot \mathrm{Dom}(\sigma.\pi_2')$;
3. for all π' of which π_0 is not a prefix, $\sigma(\pi_1\pi') = \sigma(\pi_2\pi')$.

Intuitively, this means that except for the subtrees below π_1' and π_2', the subtrees below π_1 and π_2 are structurally the same. The region between π_1 and π_1' (whose domain is specified in the second condition) is called a *context*; it is essentially a tree that lacks one leaf. We will say more about contexts in Section 2.3.1.

2.1.2 Lambda structures

Lambda structures are tree structures that are extended by a notion of variable binding. To model this, we assume from now on that Σ contains the nullary constructor var and the unary constructor lam.

A *lambda structure* L is a triple (σ, λ, I), where σ is a tree, I is an extended interpretation function, and $\lambda : \mathrm{Dom}(\sigma) \rightsquigarrow \mathrm{Dom}(\sigma)$ is the partial *binding function*. λ must obey the following conditions:

1. binding only holds between variables and λ-binders:
   $\forall\pi\forall\pi'.\lambda_L(\pi) = \pi' \Rightarrow (\pi{:}\mathsf{var} \wedge \exists\pi''\ \pi'{:}\mathsf{lam}(\pi''))$
2. every variable has a binder:
   $\forall\pi.\pi{:}\mathsf{var} \Rightarrow \exists\pi'\ \lambda_L(\pi) = \pi'$
3. variables are dominated by their binders:
   $\forall\pi\forall\pi'.\lambda_L(\pi) = \pi' \Rightarrow \pi' \triangleleft^* \pi$

The interpretation function I interprets all predicate symbols that the interpretation function of a tree structure does, plus the predicate symbol $\lambda(\,\cdot\,) = \,\cdot\,$. I

assigns to most predicate symbols the same relations that the interpretation function of a tree structure would; below, we only discuss the interpretation of the binding relation (which is not defined in tree structures) and of the similarity and parallelism relations (whose definitions change).

We define the *binding* relation $\lambda(\pi) = \pi'$ to be true iff the application $\lambda(\pi)$ is defined and equal to π'. We impose an additional restriction on the similarity relation:

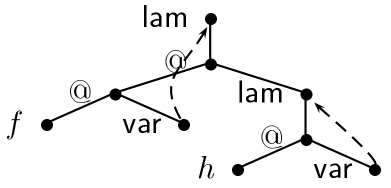
1. within the trees, binding is structurally isomorphic:
   $\forall\pi\forall\pi'\ \sigma.(\pi_1\pi)\downarrow \wedge \sigma.(\pi_1\pi')\downarrow\ \Rightarrow\ (\lambda(\pi_1\pi)=\pi_1\pi' \Leftrightarrow \lambda(\pi_2\pi)=\pi_2\pi')$
2. two variables in identical positions within the trees and bound outside the trees must be bound by the same binders:
   $\forall\pi\forall\pi'\forall\pi''\ (\pi < \pi_1 \wedge \pi' < \pi_2 \wedge \neg\pi_0{\leq}\pi'' \wedge \sigma.(\pi_1\pi'')\downarrow \wedge \lambda(\pi_1\pi'')=\pi \wedge \lambda(\pi_2\pi'')=\pi') \Rightarrow \pi = \pi'$.

The same restrictions, plus an additional condition about so-called "hanging binders" (binding nodes within a context that bind variables below the exception path), also apply to the parallelism relation $\pi_1/\pi_1'{\sim}\pi_2/\pi_2'$:

1. within the contexts, binding is structurally isomorphic:
   $\forall\pi\forall\pi'\ \neg\pi_0{\leq}\pi \wedge \sigma.(\pi_1\pi)\downarrow \wedge \neg\pi_0{\leq}\pi' \wedge \sigma.(\pi_1\pi')\downarrow \quad\Rightarrow\quad (\lambda(\pi_1\pi)=\pi_1\pi' \Leftrightarrow \lambda(\pi_2\pi)=\pi_2\pi')$
2. two variables in identical positions within their context and bound outside their context must be bound by the same binders:
   $\forall\pi\forall\pi'\forall\pi''\ (\pi < \pi_1 \wedge \pi' < \pi_2 \wedge \neg\pi_0{\leq}\pi'' \wedge \sigma.(\pi_1\pi'')\downarrow \wedge \lambda(\pi_1\pi'')=\pi \wedge \lambda(\pi_2\pi'')=\pi') \Rightarrow \pi = \pi'$.
3. there are no 'hanging' binders, i.e., var nodes below the exception positions are not bound by lam nodes inside the contexts:
   $\forall\pi\forall\pi'\ \neg(\pi_1 \leq \pi < \pi_1' \leq \pi' \wedge \lambda(\pi')=\pi)$

We can draw a lambda structure L by first drawing the tree that corresponds to the tree structure $\mathcal{M}^\sigma$, and then drawing dashed arrows from the bound nodes to the binding nodes in the binding relation. For example, the following picture displays the lambda structure that is defined by the tree $\mathsf{lam}(@(@(f,\mathsf{var}),\mathsf{lam}(@(h,\mathsf{var}))))$ and the binding function that maps the node 112 to the node ϵ, and the node 1212 to the node 12:

(2.1)



2.1.3 Syntax and semantics of CLLS

With these definitions, it is straightforward to define the syntax and semantics of CLLS constraints. Assuming a set of variables $X, Y, \ldots$, an atomic formula (or atomic constraint) φ_0 of CLLS is one of the following applications of predicate symbols to variables:

$$
\begin{array}{rcll}
\varphi_0 & ::= & X{:}f(X_1, \ldots, X_n) & f \in \Sigma,\ n = \mathsf{ar}(f) \\
& | & X {\lhd^*} Y & \\
& | & X {\neq} Y & \\
& | & X \sim Y & \\
& | & X/X' {\sim} Y/Y' & \\
& | & \lambda(X){=}Y. &
\end{array}
$$

We take the language CLLS of *constraints* over these atomic formulae to be the language of conjunctions of atomic formulae. The language of *dominance constraints* is the sublanguage of CLLS that only uses labeling and dominance constraints. CLLS_0 is the sublanguage of CLLS that does not contain binding constraints. We will mainly be interested in the constraint languages, but occasionally, we will also consider larger logical languages over them, for example the positive existential fragment (built up with all propositional connectives and positive occurrences of existential quantifiers) or the full first-order language, in which quantifiers (over nodes) can be used anywhere.

Satisfaction of an atomic constraint φ_0 is defined with respect to a pair (L, α) of a lambda structure $L = (\sigma, \lambda, I)$ and a variable assignment $\alpha : \text{Var} \to \text{Dom}(\sigma)$ that assigns nodes to the variables. The atomic constraint $R(X_1, \ldots, X_k)$ (where R is one of $:f$, $\lhd^*$, etc.) is satisfied by this pair iff $(\alpha(X_1), \ldots, \alpha(X_k)) \in I(R)$. This is extended to satisfaction of arbitrary formulae in the usual Tarskian way. If a pair (L, α) satisfies φ, we also say that φ *describes* L.

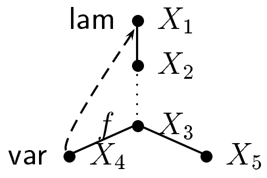
For fragments of CLLS that do not contain binding constraints (such as dominance constraints or CLLS_0), it makes no difference if we interpret its formulae over lambda structures or over tree structures. Therefore, we will only consider the (simpler) interpretations over tree structures in later chapters.

Because CLLS constraints can easily become unreadable, we will frequently use *constraint graphs* as a graphical device to represent constraints. They are essentially an alternative to the original syntax of the language. Constraint graphs are directed graphs with three kinds of edges: solid edges, used to represent immediate dominance, dotted edges, used to represent arbitrary dominance, and dashed arrows, used to represent binding constraints.

The nodes of a constraint graph represent variables of a constraint. Whenever a node of the graph is labeled with a constructor $f \in \Sigma$ of arity n, it must have exactly n children via solid edges. This configuration corresponds to a labeling constraint of the variables corresponding to the nodes. Dotted edges correspond to a dominance constraint of the corresponding variables. And finally, dashed arrows correspond to binding constraints. The other types of atomic constraints cannot easily be represented in a constraint graph.

For example, consider the constraint graph (2.2). It represents the constraint (2.3).

(2.2)



(2.3) $X_1{:}\mathsf{lam}(X_2) \wedge X_2 \triangleleft^* X_3 \wedge X_3{:}f(X_4, X_5) \wedge X_4{:}\mathsf{var} \wedge \lambda(X_4) = X_1.$

Although their pictures look very similar, it is essential not to confuse constraint graphs with lambda structures. The nodes in a constraint graph stand for variables in a constraint, and the edges stand for atomic constraints; the nodes in a lambda structure can be denoted by the variables in a constraint, and the edges are an actual part of the tree. Where constraint graphs are an alternative representation of the syntax of CLLS, lambda structures are objects of its semantics.

Note that we have only defined one direction of the relation between constraints and constraint graphs; we have shown how to read a constraint off of a graph. It is more difficult to define the converse relation. Fortunately, we can do without in this thesis, but we will take a closer look at the problem in the conclusion.

2.1.4 Lambda structures and lambda terms

The relevance of lambda structures and CLLS as the corresponding tree logic is that we can encode terms of the lambda calculus as lambda structures and talk about them in terms of trees and nodes. We have seen in the introduction that there is an inherent danger in such a tree-like treatment of lambda terms (or more generally, any kind of logical formalism that allows to bind variables) that it could become unclear which binder a variable is bound by. Lambda structures solve this problem: CLLS does not rely on variable names to determine binding, it relates the binder and the variable directly. The dashed arrows in a constraint graph can be thought of as "rubber bands"; no matter how much material is interposed between the variable and its binder, the link won't break.

The precise relation between lambda structures and lambda terms is that a lambda structure corresponds uniquely to a class of lambda terms modulo α equivalence. If we assume without loss of generality that all lambda terms are unary and ignore variable binding in a lambda term for the moment, each such term has a straightforward tree structure: For every lambda term M, we can obtain the corresponding tree as $M^\dagger$, according to the following definition of $(\cdot)^\dagger$:

$$\begin{array}{rcll} x^\dagger & = & \mathsf{var} & (x \text{ is a variable}) \\ f^\dagger & = & f & (f \in \Sigma) \\ (M(N))^\dagger & = & M^\dagger @ N^\dagger & \\ (\lambda x.M)^\dagger & = & \mathsf{lam}(M^\dagger). & \end{array}$$

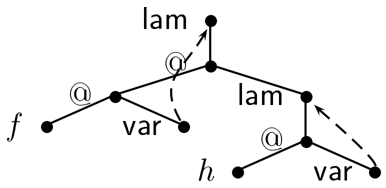
We make use of the symbol @, which is a special binary constructor that we write in left-associative infix notation.

Variable binding can be modeled by requiring that every node that is labeled with var is in the binding relation to the lam node representing its binder. The correspondence can be reversed by picking a new variable name for each lam node and naming all nodes that it binds appropriately.

As an example, the lambda term in (2.4) can be represented as the lambda structure in (2.5), and so can any other lambda term to which it is α equivalent.

(2.4) $\lambda x.f(x)(\lambda x.h(x))$

(2.5)



2.1.5 Relation to Dominance Constraints with Precedence

Dominance constraints are an important sublanguage of CLLS. They are also widely used throughout computational linguistics. But the variant of dominance constraints that is traditionally used by linguists (e.g. Marcus et al. 1983; Backofen et al. 1995) is slightly different than ours. First of all, they consider a different class of trees, where the label of a node does not determine the number of its children. This also suggests a different syntax for the constraint language, talking about labeling, immediate dominance, and precedence ("left-of" relation) separately, as in the following prototypical abstract syntax:

$$\psi \ ::= \ X \triangleleft Y \ | \ X \triangleleft^* Y \ | \ X \prec Y \ | \ X{:}f.$$

Here, $\triangleleft^*$ is the same (reflexive, transitive) dominance relation as we have defined above. $\triangleleft$ is immediate dominance, f is labeling (without specifying the immediate children), and $\prec$ is precedence: $X \prec Y$ is true if X and Y have a common ancestor Z and are dominated by different children of Z. To talk about both types more easily, we will call these constraints *precedence constraints*.

It is obvious that when interpreting over constructor trees, for every dominance constraint, there is an equivalent precedence constraint. Dominance is trivial; and labeling constraints $X{:}f(X_1, \ldots, X_n)$ can be expressed in the following way:

$$X{:}f \wedge X \triangleleft X_1 \wedge \ldots \wedge X \triangleleft X_n \wedge X_1 \prec X_2 \wedge \ldots \wedge X_{n-1} \prec X_n.$$

This works because in a constructor tree, the label f of the node denoted by X determines that there are exactly n immediate children. On an arbitrary tree, we would have to add a (quantified) formula to say that no other nodes are children of X.

It is unclear if there is a satisfiability preserving encoding of precedence constraints into dominance constraints. However, if the signature is finite and if we additionally allow the use of *disjunctions* in our dominance constraints, we can get such an encoding of precedence constraints ψ in the following way:

- Encode an atomic constraint $X \triangleleft Y$ of ψ as

$$\bigvee_{\substack{f \in \Sigma \\ \mathsf{ar}(f)=n}} \bigvee_{i=1}^{n} X : f(Z_1, \ldots, Y, \ldots, Z_n) \qquad (Y \text{ is } i\text{-th argument, } Z_k \text{ fresh}).$$

- Encode an atomic constraint $X{:}f$ of ψ as

$$X{:}f(Z_1, \ldots, Z_n) \qquad (Z_1, \ldots, Z_n \text{ fresh}).$$

- Encode an atomic constraint $X \prec Y$ of ψ as

$$\bigvee_{\substack{f \in \Sigma \\ \mathsf{ar}(f)=n}} (Z{:}f(Z_1, \ldots, Z_n) \wedge \bigvee_{1 \leq i < k \leq n} (Z_i \triangleleft^* X \wedge Z_k \triangleleft^* y)) \qquad (Z, Z_1, \ldots, Z_n \text{ fresh}).$$

Clearly, this encoding does not work for infinite signatures; but it is easy to see that it preserves satisfiability.

2.2 Applying CLLS

In the introduction, we have seen some first examples of how CLLS can be used for the underspecified description of semantics. In this section, we consider some more interesting examples. First, we see scope underspecification in its full beauty by presenting a treatment of the five-readings sentence *Every researcher of a company saw most samples.* In addition, we show how Hirschbühler sentences such as *Every man loves a woman. Several gorillas do, too.* can be correctly addressed in CLLS.

But before we go into the details of the linguistic analyses, a word about our object language is in order. In the introduction, our semantic object language (the formulae of which we described as trees) was first-order predicate logic for simplicity. Unfortunately, first-order logic does not work very well any more once we consider more interesting cases of parallelism, such as the Hirschbühler sentence in Section 2.2.2. For cases like these, it is necessary that the representation of an NP is a contiguous subtree of the entire semantics. This is not necessarily the case in a first-order analysis, but it is in a higher-order analysis before β-reduction. Note that we do not have builtin β-reduction in our representation of lambda terms as lambda structures; we are representing raw lambda terms, not β-equivalence classes.

So from here on, we choose our object language to be the language of simply typed lambda terms. As our syntax/semantic interface, we assume a construction similar to that from Montague grammar (Montague 1974), analyzing every NP as a generalized quantifier whose type is independent of whether it is a proper name or a determiner application such as *every man*. When we draw constraint graphs or lambda structures, we will usually compress subtrees that correspond to determiners into a single node that is labeled with the name of the determiner for easier readability. For example, we might compress $\lambda P \lambda Q \forall x P(x) \rightarrow Q(x)$ into a single node labeled with every. This is just an abbreviation.

Finally, we will speak about "minimal solutions" of a constraint. We connect with this term an intuitive idea of a solution which does not contain any unnecessary material; for example, it might only contain the material that is mentioned in labeling constraints, or only solutions of minimal size. There is no formal definition of a minimal solution so far that is commonly agreed upon; we will use this notion in an informal way in our discussions of linguistic examples below, but it will not appear in the more formally oriented later chapters of the thesis. We will come back to the problem in the concluding chapter.

2.2.1 Advanced scope ambiguities

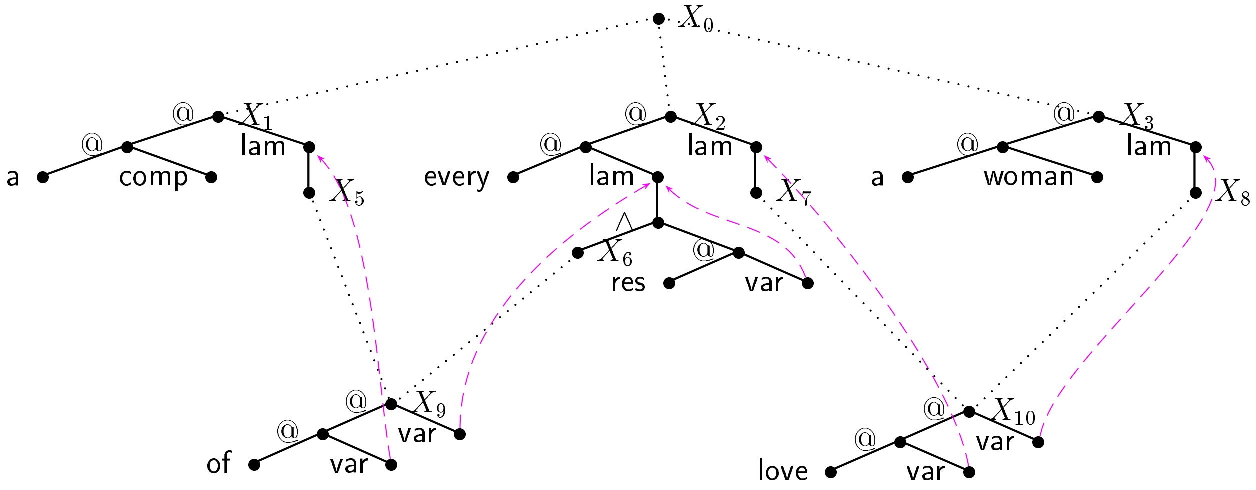
Consider the following sentence.

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

NP modifications as in this example were a problem in early approaches to scope ambiguity. These approaches would simply enumerate all permutations of the three quantifiers at the sentence node, so they would predict $6 = 3!$ readings for the sentence in the example. But this sentence has only five readings. Most recent theories treat this example correctly, but it is still an important benchmark for a theory of scope ambiguity.

CLLS does handle this sentence correctly: The constraint for the semantics of (2.6) is shown as a constraint graph below.

(2.7)



The constraint in (2.7) has a specific structure that makes it easy to see its solutions. Due to the left "dominance diamond" (whose points are the root node X_0, the left and the middle quantifier X_1 and X_2, respectively, and X_9, the tree below which models the genitive relation), there are essentially two possible places for the left quantifier in a lambda structure that solves this constraint: above the universal quantifier, or below the universal quantifier's left daughter X_6. The requirement that both quantifiers dominate X_9 leaves no other options. Likewise, the right "dominance diamond" (whose bottom point is X_{10}, the tree below which models the love relation) allows two different classes of solutions, one where the right quantifier X_3 dominates the universal quantifier X_2, and one where it occurs below the universal quantifier's right daughter X_7. It is crucial that there are two independent "dominance diamonds", each with its own "nucleus" at the bottom.

This yields five structurally different solutions to the constraint. If the existential quantifier X_1 dominates the universal quantifier X_2, the existential quantifier X_3 may be above, between, or below the other quantifiers, which gives three options. If, on the other hand, X_1 is dominated by X_2, there are only two different positions for the existential quantifier X_3, viz., either above or below the universal quantifier: The universal quantifier essentially acts as a binary constructor, ensuring that if the existential quantifiers are both dominated by (different leaves of) the universal quantifier, they don't dominate each other. This is exactly where the naive approach of enumerating the permutations of the quantifiers fails: If the universal quantifier has widest scope, the two existential quantifiers must be in *disjoint* positions in the tree, whereas the two permutations that give the universal quantifier widest scope correspond to the two possibilities for the two existential quantifiers to *dominate* each other.

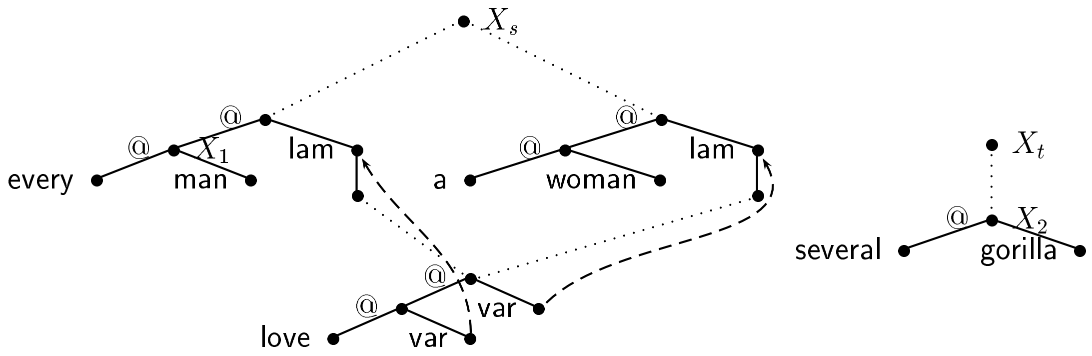
2.2.2 Hirschbühler sentences

As we have sketched in the introduction, a *Hirschbühler sentence* is an ellipsis whose source sentence contains a scope ambiguity. The interesting property of Hirschbühler sentences is that when resolving the ellipsis, the scope ambiguity is copied over to the target sentence, which might lead one to believe that the pair of sentences has four different readings (two scope ambiguities, each with two readings). But there are only two readings, as the ellipsis enforces a "scope parallelism" between the two sentences; both scope ambiguities must be resolved in the same way. For reference, we repeat Example 1.12.

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

In Section 1.2.2, we have seen how to analyze very simple ellipses with CLLS parallelism constraints. A pleasant property of CLLS is that Hirschbühler sentences can be analyzed in precisely the same way without any further modifications to the formalism. When we analyzed our simple example in the introduction, we first wrote down all we could say without resolving the ellipsis and then added the parallelism constraint. We will proceed here in the same way. The first part of the constraint is represented as a constraint graph in (2.9).

(2.9)



(a) (b)

Part (a) of this constraint is just the representation of the scope ambiguity in the source sentence that we have been using since the introduction, written in our higher-order object language. Part (b) expresses that somewhere in the semantics of the target sentence, several gorillas occur.

Now we add the parallelism constraint; what we want to say is that the semantics of the target sentence is just like that of the source sentence, but where the source sentence contained "every man", the target sentence should have "several gorillas".

(2.10) $X_s/X_1{\sim}X_t/X_2$

This constraint has exactly the two correct minimal solutions, but we should observe that the construction was only possible because the parallel elements were represented as subtrees. Parallelism constraints can only replace entire subtrees; had we attempted to do the same thing with the first-order representation of the scope ambiguity (Fig. 1.1), we would have failed because the source parallel element would have been distributed over several nodes in the tree that do not form a subtree. At this point, the difference between the CLLS/CU and the DSP analysis of ellipses becomes apparent: DSP need not require that the representation of a parallel element must be contiguous. They consider full higher-order logic with builtin β-reduction, so they do not have to worry about the exact representation of the NP semantics, and in fact, could not distinguish between the unreduced representation (where the NP semantics is in a subtree by itself) and the β-reduced representation (which is essentially first-order). In CLLS, on the other hand, we are talking about individual lambda structures instead of β-equivalence classes; here, it does make a difference if we try to resolve the ellipsis before or after β-reduction.

2.3 Context Unification

We now define context unification and its underlying logic, the language of context constraints. To this end, we first define contexts and context functions, which are used in the definition of the semantics of context constraints. Finally, we extend the language with equations between context-valued terms and prove that this extension makes no difference to the expressive power of the language.

2.3.1 Contexts and Context Functions

In analogy to tree domains, we define a *context domain* Γ to be a finite prefixed-closed subset of $\mathbb{N}^*$ for which there is a path $\pi_e \in \mathbb{N}^* - \Gamma$ such that $\Gamma \cup \{\pi_e\}$ is a tree domain. The intuition behind a context domain is that it is the set of nodes of a tree that misses one leaf. Note that a context domain must be finite and can be empty. π_e is called the *exception path* of Γ.

A *context* s is a function

$$s : \Gamma \to \Sigma,$$

where Γ is a context domain with exception path π_e, with the additional property that for every $\pi \in \Gamma$, if $\mathsf{ar}(s(\pi)) = n$, then $\pi 1, \ldots, \pi n$ are in $\Gamma \cup \{\pi_e\}$, but no πk is, for any $k > n$. π_e is called the *exception path* of s.

Intuitively, we can think of a context as a tree from which an entire subtree (including its root) has been cut away, leaving behind a hole. Consequently, we can and frequently will write a context as a ground term that contains exactly one occurrence of the special symbol $\bullet$, which we call 'hole'. Fig. 2.1 schematically shows a tree from which the subtree σ has been cut away.

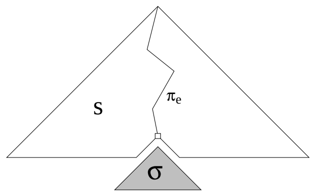


Figure 2.1: A context.

If s is a context with exception path π_e, we can define the *context function* γ_s to be the unary function from trees to trees that inserts its argument into the context s in the following way:

$$\gamma_s(\sigma)(\pi) = \begin{cases} s(\pi) & \text{if defined} \\ \sigma(\pi') & \text{if } \pi = \pi_e\pi' \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Clearly, context functions and contexts correspond uniquely to each other, and all context functions are injective. The "identity context" $\bullet$, whose domain is empty, corresponds to the identity function on trees.

Throughout, we will switch freely between contexts and context functions and use whichever is more convenient. To this end, we list some more definitions that will make this easier. We define the exception path of a context function γ_s to be the exception path of s. If s is a context and σ is a tree, we define the *insertion* $s[\sigma]$ of σ into s to be the tree $\gamma_s(\sigma)$. In term representation, we can think of this as substituting σ for the hole of s. If s and s' are contexts, we define the *extension* $s \circ s'$ of s by s' to be the context corresponding to the context function $\gamma_s \circ \gamma_{s'}$. In the term representation, we can think of this as substituting s' for the hole of s.

As for trees, we can define selection $s.\pi$ of subcontexts. Let $s : \Gamma \to \Sigma$ be a context, let π_e be its exception path, and let π be a path in $\Gamma \cup \{\pi_e\}$. Then we can define the selected subcontext $s.\pi$ as the function

$$\begin{aligned} &s.\pi : \{\pi' \mid \pi\pi' \in \Gamma \cup \{\pi_e\}\} \to \Sigma \\ &(s.\pi)(\pi') = s(\pi\pi'). \end{aligned}$$

$s.\pi$ is undefined if π is not in $\Gamma \cup \{\pi_e\}$. $s.\pi_e$ is defined; its domain is empty, so it is the identity context $\bullet$.

The following are a few simple but useful lemmas that will facilitate a lot of later work.

Lemma 2.2. *If s is a context, π is its exception path, and σ is a tree, then the domain of $s[\sigma]$ is the disjoint union*

$$\mathrm{Dom}(s[\sigma]) = \mathrm{Dom}(s) \uplus \pi \cdot \mathrm{Dom}(\sigma).$$

Proof. Obvious from the definitions of $s[\sigma]$ and of context functions. □

Lemma 2.3. *Two contexts s, s' are equal iff their exception paths are the same and there is a tree σ such that $s[\sigma] = s'[\sigma]$.*

Proof. The "$\Rightarrow$" direction is trivial. For the other direction, all we have to prove is that the domains of the functions s and s' are equal; it follows immediately that the entire functions are equal.

We know that $s[\sigma] = s'[\sigma]$; so in particular, their domains are the same. Call the common exception path of the two contexts π. According to Lemma 2.2, the following equalities hold:

$$\begin{array}{rcl}\mathrm{Dom}(s[\sigma]) & = & \mathrm{Dom}(s) \uplus \pi \cdot \mathrm{Dom}(\sigma) \\ \mathrm{Dom}(s'[\sigma]) & = & \mathrm{Dom}(s') \uplus \pi \cdot \mathrm{Dom}(\sigma)\end{array}$$

Because the unions are disjoint, it follows that $\mathrm{Dom}(s) = \mathrm{Dom}(s')$. □

Lemma 2.4. *If s_1, s_2 are contexts, then the exception path of $s_1 \circ s_2$ is the concatenation of the exception paths of s_1 and s_2.*

Proof. By definition of $s_1 \circ s_2$. □

2.3.2 Syntax and Semantics of CU

The language of *context constraints* is built up from a ranked signature Σ as in 2.1.1, a set of first-order variables $x, y, \ldots$, and a set of *context variables* $C, D, \ldots$. A context constraint ψ is a conjunction of equations $t = t'$ of terms of the following syntax:

$$t ::= x \mid f(t_1, \ldots, t_n) \mid C(t).$$

Context constraints are interpreted over trees. First-order variables (and, in general, all terms t) denote trees, and context variables denote context functions.

Variable assignments β that assign trees to first-order variables and context functions to context variables can be lifted to functions from terms t to trees homomorphically:

$$\begin{array}{rcl} \beta(f(t_1,\ldots,t_n)) & = & f(\beta(t_1),\ldots,\beta(t_n)) \\ \beta(C(t)) & = & \beta(C)(\beta(t)). \end{array}$$

A variable assignment β satisfies an equation $t = t'$ iff $\beta(t) = \beta(t')$. It satisfies a context constraint ψ iff it satisfies all equations in ψ. *Context unification* is the satisfiability problem of context constraints. As for CLLS, we can consider larger languages; in the full first-order language, we allow quantification over both first-order variables x and context variables C.

2.3.3 Context Equations

It turns out that in practical work with context unification, it is often convenient to express the equality of context-valued terms (such as context variables, extensions of context variables, contexts that are written down explicitly as terms with one occurrence of $\bullet$, etc.) instead of only of tree-valued terms. In this section, we show that these *context equations* can be expressed as standard context constraints. For easier reference, we call the terms t from the previous section *tree terms* in this section.

A *context equation* is an equation $u{=}u'$ between *context terms* of the following form:

$$u \ ::= \ C \mid s \mid u \circ u',$$

where s is a ground term over Σ with exactly one occurrence of $\bullet$ (i.e., the term representation of a context).

The application of a context term u to a tree term t can be reduced to an ordinary tree term by the following rules:

$$\begin{array}{rcl} f(t_1,\ldots,u_i,\ldots,t_n)(t) & = & f(t_1,\ldots,u_i(t),\ldots,t_n) \\ \bullet(t) & = & t \\ (u_1 \circ u_2)(t) & = & u_1(u_2(t)). \end{array}$$

As trees can be seen as a special case of tree terms, we can define the semantics of a context equation by lifting variable assignments β to context terms u such that $\beta(u)$ is the context function

$$\beta(u) : \sigma \mapsto \beta(u(\sigma)).$$

We define the language of *extended context constraints* as the language of conjunctions of equations between either tree or context terms.

To reduce extended context constraints to standard context constraints, we need to find a way to express every context equation as a finite conjunction of tree equations. This is done by the following proposition. Hence, we can safely use context equations as "abbreviations" for context constraints whenever it is convenient.

Proposition 2.5 (Expressing Context Equations). *If u, u' are context terms, σ_1, σ_2 are different ground terms, and β is a variable assignment, then β satisfies the context equation*

$$u = u'$$

iff it satisfies the context constraint

$$u(\sigma_1) = u'(\sigma_1) \ \wedge \ u(\sigma_2) = u'(\sigma_2).$$

Proof. By definition of the semantics, it is sufficient to show that any two contexts s, s' are equal iff there are two different trees σ_1, σ_2 such that $s[\sigma_1] = s'[\sigma_1]$ and $s[\sigma_2] = s'[\sigma_2]$. The direction from left to right is trivial; we show the other direction below.

Assume that the exception paths π, π' of the contexts are different. We will derive a contradiction; it follows that $\pi = \pi'$ and, by application of Lemma 2.3, that $s = s'$.

π cannot be a proper prefix of π'; otherwise, $s[\sigma_1] = s'[\sigma_1]$ would not be satisfied. Symetrically, π' cannot be a proper prefix of π. This means that π and π' must be disjoint paths.

π' is defined in s: Since it is defined in $s'[\sigma_1]$, we can conclude by Lemma 2.2 that

$$\pi' \in \mathrm{Dom}(s) \cup \pi \cdot \mathrm{Dom}(\sigma_1).$$

But we have assumed that π is no prefix (proper or not) of π', so π' must already be defined in s.

As π' and π, the exception path of s, are disjoint, the following equalities hold:

$$\begin{array}{rcccccc} s.\pi' & = & s[\sigma_1].\pi' & = & s'[\sigma_1].\pi' & = & \sigma_1 \\ s.\pi' & = & s[\sigma_2].\pi' & = & s'[\sigma_2].\pi' & = & \sigma_2. \end{array}$$

So in contradiction to our assumptions, we have derived that $\sigma_1 = \sigma_2$. It follows that π and π' cannot be disjoint and hence, must be equal. $\square$

2.4 Applying Context Unification

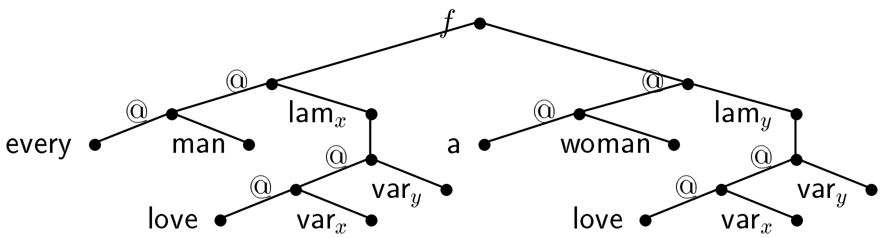
When we sketched the CU treatment of scope ambiguities in the introduction, we pointed out that our preliminary analysis allowed a third, linguistically unwanted solution. In this section, we will analyze this problem more closely and provide a solution. We will not go into accounts of more interesting linguistic examples, as the CU analysis is largely equivalent to that of CLLS, and we have already seen all the important ideas in Section 2.2.

Recall Example 1.18, our preliminary encoding of the scope ambiguity *Every man loves a woman*, repeated here as Example 2.11. We have converted the example to its higher-order variant; variable binding is modeled with special constructors lam_x and var_x. As we know from the section on CLLS, modeling binding via variable names does not work in underspecified descriptions, but it will do for now. Note, by the way, the infix use of the apply constructor.

(2.11) $X_0 = C_1(\mathsf{every}@\mathsf{man}@\mathsf{lam}_x(C_3(\mathsf{love}@\mathsf{var}_x@\mathsf{var}_y))) \wedge$
$X_0 = C_2(\mathsf{a}@\mathsf{woman}@\mathsf{lam}_y(C_4(\mathsf{love}@\mathsf{var}_x@\mathsf{var}_y)))$

Even if we ignore non-minimal solutions, this context constraint has three structurally different solutions:

1. A reading that assigns the universal quantifier wide scope (as in Example 1.19).
2. A reading that assigns the existential quantifier wide scope.
3. For any binary constructor f, a tree in which the quantifiers are in disjoint positions, each with its own copy of the $\mathsf{love}(x, y)$:



Clearly, the third reading (whose correctness we invite the reader to verify) does not correspond to any linguistically warranted reading of the sentence, and we need to change the constraint to exclude it.

What has gone wrong? Essentially, we have expressed a *subtree* relation with our context constraint when we wanted to express a *dominance* relation: We have only expressed that the trees below the quantifiers both contain the love subtree, but we did not require that both quantifiers dominate the same occurrence of this subtree. This leaves the possibility open to have two copies of this subtree, each dominated by one but not both of the quantifiers, and let the quantifiers "escape" into disjoint positions of the tree.

To repair this deficiency, we use context equations. If we equate two contexts, we force their exception paths to be the same. If, in addition, the quantifiers appear on the exception paths of these contexts, we have made sure that they must dominate a common node (namely, the node at the end of the exception path). We can then insert the love subtree at this location. This is expressed by the following extended context constraint.

(2.12)
$$\begin{aligned} C_0 &= C_1(\mathsf{every@man@lam}_x(C_3(\bullet))) \land \\ C_0 &= C_2(\mathsf{a@woman@lam}_y(C_4(\bullet))) \land \\ X_0 &= C_0(\mathsf{love@var}_x\mathsf{@var}_y) \end{aligned}$$

This constraint has only the two correct minimal solutions; the use of context equations (which can nevertheless be expressed as ordinary context constraints, as we have seen above) has given us a direct handle on nodes of the tree. It follows the analysis of Niehren et al. (1997b), and it can serve as a first example of the encoding of dominance constraints as context constraints that we will present in the next chapter and that will give a systematic account of this problem.

2.5 Conclusion

In this chapter, we have defined the syntax and semantics of CLLS and context constraints. Starting with a definition of trees and tree structures, we have introduced the notion of lambda structures, which can be used to model lambda terms in a tree-like fashion while avoiding problems with variable binding. Then we have defined the syntax and semantics of CLLS, the conjunctive language over a selection of atomic constraints, including labeling, dominance, parallelism, and binding constraints. The most interesting sublanguages of CLLS are the fragment of (labeling and) dominance constraints and CLLS_0, the fragment that does not use binding constraints. We have compared our variant of dominance constraints to a more common one, and we have found out that if the signature is finite, both are equivalent. Finally, we have applied CLLS to some more interesting linguistic examples.

In the second half of the chapter, we have discussed context unification, the satisfiability problem of context constraints. After defining contexts and context functions in analogy to the trees of the first section, we have defined the syntax and semantics of context constraints; they are equations between tree-valued terms which can contain first-order variables denoting trees and context variables denoting context functions. Afterwards, we have shown that context equations (equations between context-function-valued terms) can be added to context constraints to obtain extended context constraints without changing the expressive power. We have revisited scope ambiguities in CU and corrected a deficiency that resulted from a confusion of subtree and dominance constraints.

In the course of our definitions, we have introduced various classes of formulae and objects. To avoid confusion, we comprehensively list them once again, along with the letters that we use to range over them:

- φ are CLLS, CLLS_0, or dominance constraints; ψ are context constraints. We have generally distinguished constraints (purely conjunctive formulae) from formulae in general, which we write with the respective capital Greek letters.
- X are variables denoting nodes; x are variables denoting trees; C are context variables, denoting context functions.
- α are variable assignments for CLLS, mapping variables to nodes in a lambda structure; β are variable assignments for context constraints, mapping tree variables to trees and context variables to context functions.
- σ are trees; s are contexts; γ are context functions. Trees can be written as ground terms, and contexts can be written as ground terms with exactly one occurrence of the hole symbol $\bullet$.

Chapter 3

Relating Context Unification and CLLS

Now that we have our definitions straight, we turn to a proof of the equivalence of CLLS_0 and Context Unification in this section. More precisely, we show that for every constraint of CLLS_0, there is a satisfiability equivalent context constraint, and vice versa. The encoding of CLLS_0 into context unification can be extended in a straightforward way to an encoding of the first-order theories.

The main obstacle that we must overcome in our encoding of CLLS_0 into CU is to provide the power to talk about *occurrences* of subtrees. In CLLS_0 (and even in dominance constraints), we can do this easily because we can talk about the nodes of a tree (i.e., the roots of occurrences); it is not clear at all that this is possible in the external perspective that CU takes, relating only trees and not their nodes. We have seen in the introduction (Section 1.4) that the naive encoding of dominance constraints as subtree constraints, which can be considered a purely external sublanguage of context constraints, does not preserve satisfiability. More precisely, the unsatisfiability of dominance constraints as in Example 1.24, repeated here as Example 3.1, is not preserved by the encoding because subtree constraints don't talk about occurrences of trees; a tree satisfying the subtree constraint can have more than one occurrence of the subtree denoted by y, and nothing says that all occurrences of y must refer to the same occurrence of this tree.

(3.1) $X{:}f(X_1, X_2) \wedge X_1 \lhd^* Y \wedge X_2 \lhd^* Y$

(3.2) $x{=}f(x_1, x_2) \wedge y \ll x_1 \wedge y \ll x_2$

But as we have seen in Section 2.4, the expressive power to talk about occurrences of subtrees is, in fact, available in CU, if only in a slightly awkward manner. The first section of this chapter will be concerned with extending the intuitive idea of the previous chapter to a systematic encoding of dominance constraints as (extended) context constraints. Our encoding will not only preserve satisfiability, but even the solutions of the constraints: In a sense to be defined, a dominance constraint and its encoding are satisfied by exactly the same trees. We first define the encoding for the constraint language; afterwards, we extend it to the first-order theories and prove its correctness for the more general case, from which the correctness of the encoding for constraints follows. The basic idea of our encoding will be to identify the domain of a tree with the set of contexts in this tree that start at its root; under this condition, a context is uniquely identified by its exception path. We can axiomatize context variables to denote these contexts and then recast dominance and labeling constraints as context equations.

In the second section, we will complete the proof of the equivalence of the two languages by covering parallelism. In one direction, we extend the encoding of the first section by an encoding of parallelism and similarity constraints; this is not difficult once we have the intuitions and formal groundwork of the encoding of dominance constraints available. The other direction is much less obvious. Fortunately, we can make use of so-called *equality up-to constraints*. On one hand, Niehren et al. (1997a) have shown that they are equivalent to context constraints; on the other hand, they are sufficiently similar to CLLS_0's parallelism constraints to allow a simple encoding of equality up-to as parallelism constraints. The bulk of this direction of the proof is to show the equivalence between context and equality up-to constraints (which Niehren et al. already did); once that is out of the way, the rest of the proof is simple.

Throughout the chapter, we will freely switch between trees σ and tree structures $\mathcal{M}^\sigma$, and contexts s and context functions γ_s, respectively, whenever it is convenient. Furthermore, we will ignore the binding function of lambda structures in this and the next chapter; as we have seen above, it is sufficient to interpret CLLS_0 constraints over tree structures.

3.1 Encoding dominance constraints

In this section, we show how to encode dominance constraints as extended context constraints such that they have the same solutions. The correctness of our construction follows from a more general encoding of the full first-order theories. Nevertheless, we first present the encoding for the constraints and extend it to the first-order theories later.

We call the encoding function of dominance into extended context constraints $[\![\cdot]\!]$. For the proof, we will construct a mapping from pairs of tree structures and variable assignments for dominance constraints to variable assignments for context constraints that we call $[\![\cdot,\cdot]\!]$. With this terminology, the key result (Prop. 3.5) of our correctness proof (that makes the term "have the same solutions" precise) is

$$(\mathcal{M}^\sigma, \alpha) \models \Phi \Leftrightarrow [\![\mathcal{M}^\sigma, \alpha]\!] \models [\![\Phi]\!],$$

where Φ is an arbitrary closed first-order formula over the dominance constraints.

The central idea of the encoding is to identify nodes of a tree with their contexts. We associate with every variable X appearing in a dominance constraint φ a context variable C_X (whose purpose it is to denote the context starting at the root of the tree and whose hole is the node denoted by X) and a tree variable x (whose purpose it is to denote the tree below X). In addition, we introduce a new tree variable $\top$ that we want to denote the entire tree.

To ensure that these new variables interact correctly, we impose the following constraint, which we call $Root(\varphi)$:

$$Root(\varphi) = \bigwedge_{X \in \mathcal{FV}(\varphi)} \top{=}C_X(x),$$

where $\mathcal{FV}(\varphi)$ stands for the set of free variables of φ. (As φ does not contain quantifiers, this is the entire set of variables.)

Intuitively, the Root constraint expresses the following facts:

1. $\top$ denotes the entire tree. More precisely, the tree it denotes contains all subtrees that we can refer to with a (tree or context) variable, which is sufficient for our purposes.
2. The context C_X starts at the root node of $\top$; the tree below its hole is denoted by x.

Now we can define the encoding $[\![\cdot]\!]$ proper as in Figure 3.1. To encode a dominance constraint φ as an extended context constraint, we simply conjoin $Root(\varphi)$ and $[\![\varphi]\!]$.

Proposition 3.1 (Encoding Constraints). *Let φ be a dominance constraint. φ describes a tree σ iff there is a variable assignment β that satisfies $Root(\varphi) \wedge [\![\varphi]\!]$ and assigns $\beta(\top) = \sigma$.*

This proposition is a consequence of the more general Theorem 3.2. For the moment, let us review an example. In Example 1.24, we presented a dominance

$[\![X\triangleleft^*Y]\!]$	$=$	$C_X \circ C = C_Y$ C is a fresh context variable	
$[\![X{:}f(X_1,\dots,X_n)]\!]$	$=$	$\bigwedge_{1\leq i\leq n} C_{X_i} = C_X \circ f(x_1,\dots,\bullet,\dots,x_n)$	if $n \geq 1$
$[\![X{:}a]\!]$	$=$	$x{=}a$	
$[\![\varphi_1 \wedge \varphi_2]\!]$	$=$	$[\![\varphi_1]\!] \wedge [\![\varphi_2]\!]$	

Figure 3.1: Encoding dominance constraints φ as context equations.

constraint (which we repeat here as Example 3.3) and showed how the naive subtree encoding failed: The dominance constraint was unsatisfiable, but the subtree constraint wasn't. Here, we demonstrate that our new encoding preserves the unsatisfiability.

(3.3) $X : f(X_1, X_2) \wedge X_1\triangleleft^*Y \wedge X_2\triangleleft^*Y$

(3.4) $\top = C_X(x) \wedge \top = C_{X_1}(x_1) \wedge \top = C_{X_2}(x_2) \wedge \top = C_Y(y)\wedge$
$C_{X_1} = C_X \circ f(\bullet, x_2) \wedge C_{X_2} = C_X \circ f(x_1, \bullet)\wedge$
$C_{X_1} \circ C = C_Y \wedge C_{X_2} \circ C' = C_Y$

The extended context constraint in Example 3.4 is the encoding of the dominance constraint according to Proposition 3.1: The first line is the Root formula, the second line is the encoding of the labeling constraint, and the third line, the encoding of the dominance constraints.

The context constraint is unsatisfiable. Suppose it had a solution, i.e., there existed a satisfying variable assignment β, and consider the exception path π of $\beta(C_Y)$. We can conclude from the constraint in the third line and Lemma 2.4, we can conclude that the exception paths both of $\beta(C_{X_1})$ and of $\beta(C_{X_2})$ must be prefixes of π. But the second line requires that these paths are different in at least one place. This is a contradiction, so the constraint must be unsatisfiable.

The major conceptual difference to the encoding as subtree constraints is that we are no longer just talking about the subtrees; we can explicitly talk about occurrences of subtrees by identifying them with their contexts, denoted by a context variable.

To continue with the proof of the proposition, we now lift the encoding of dominance constraints to arbitrary closed first-order formulae over this language by defining it for a complete set of first-order connectives (Fig. 3.2). Note that now that we have

$[\![X \lhd^* Y]\!]$	$=$	$\exists C. C_X \circ C = C_Y$	C is a fresh context variable
$[\![X{:}f(X_1, \ldots, X_n)]\!]$	$=$	$\bigwedge_{1 \le i \le n} C_{X_i} = C_X \circ f(x_1, \ldots, \bullet, \ldots, x_n)$	if $n \ge 1$
$[\![X{:}a]\!]$	$=$	$x{=}a$	
$[\![\Phi_1 \wedge \Phi_2]\!]$	$=$	$[\![\Phi_1]\!] \wedge [\![\Phi_2]\!]$	
$[\![\neg\Phi]\!]$	$=$	$\neg[\![\Phi]\!]$	
$[\![\exists X.\Phi]\!]$	$=$	$\exists C_X \exists x.(\top = C_X(x) \wedge [\![\Phi]\!])$	

Figure 3.2: Encoding first-order dominance formulae Φ as first-order context equations.

negation in our language, it is not sufficient to introduce fresh context variables for the encoding of $\lhd^*$; we must close them off with existential quantifiers. For this encoding, the following theorem holds.

Theorem 3.2 (Encoding First-Order Formulae). *A closed first-order formula Φ over the language of dominance constraints describes a tree σ iff there is a variable assignment β that satisfies $[\![\Phi]\!]$ and assigns $\beta(\top) = \sigma$.*

As an immediate consequence, it follows that our encoding preserves both satisfiability and validity of formulae. Note that the theorem does not mention $Root(\Phi)$. For closed formulae, $Root(\Phi)$ is the empty conjunction. This is sufficient because the conjuncts of the Root formula are distributed over the encodings of the quantifiers.

It is easy to see that Proposition 3.1 is a corollary of Theorem 3.2. If we close off a dominance constraint by adding existential quantifiers for all of its variables, we do not change the set of trees that it describes. We can apply the theorem to this formula and encode it as an extended context formula; we can then remove all existential quantifiers and gather the occurrences of constraints of the form $\top = C_X(x)$ in the Root formula. In this way, we have obtained just the encoding of dominance constraints that we constructed above.

As mentioned above, we exploit the correspondence between nodes in a tree and their contexts for the proof of this theorem. This correspondence is expressed formally by the following lemma.

Lemma 3.3. *Let σ be a tree, $\pi_1, \pi_2 \in \mathrm{Dom}(\sigma)$, and π_1 a prefix of π_2; say, $\pi_2 = \pi_1\pi$. Then there is a unique context function $\Gamma^\sigma_{\pi_1,\pi_2}$ such that*

$$\sigma.\pi_1 = \Gamma^\sigma_{\pi_1,\pi_2}(\sigma.\pi_2).$$

The exception path of $\Gamma^\sigma_{\pi_1,\pi_2}$ is π.

Proof. First, we show the existence of a context function γ such that $\sigma.\pi_1 = \gamma(\sigma.\pi_2)$. We do this by defining a context s for whose context function γ_s the condition is satisfied.

Define

$$\begin{array}{rcl} D & = & \{\pi' \mid \sigma.\pi_1\pi'\downarrow \text{ and } \pi_2 \not\leq \pi_1\pi'\} \\ s & = & \sigma|D, \end{array}$$

where $\sigma|D$ is the restriction of σ to the domain D. s is a context because D is a finite tree domain, and the arities are respected in all nodes except for the parent of π_2, which lacks one child. The exception path of s (the path to the "missing child" in D) is π.

Now we show that $\gamma_s(\sigma.\pi_2) = \sigma.\pi_1$. On a path π' of which π is a prefix (say, $\pi' = \pi\pi''$),

$$\begin{array}{rcll} \gamma_s(\sigma.\pi_2)(\pi') & = & (\sigma.\pi_2)(\pi'') & \text{(by definition)} \\ & = & (\sigma.\pi_1)(\pi') & (\pi_2 = \pi_1\pi). \end{array}$$

On the other hand, if π is not a prefix of π',

$$\begin{array}{rcll} \gamma_s(\sigma.\pi_2)(\pi') & = & s(\pi') & \text{(by definition)} \\ & = & \sigma(\pi_1\pi') & \text{(by definition)} \\ & = & (\sigma.\pi_1)(\pi'), & \end{array}$$

and s is defined on π' iff $\pi_2 \not\leq \pi_1\pi'$ or, equivalently, whenever $\pi \not\leq \pi'$ and it is defined in $\sigma.\pi_1$.

This concludes the proof of existence. For the uniqueness proof, we assume that there are two context functions γ_1, γ_2 that satisfy the condition. These two functions must assume the same value on $\sigma.\pi_2$, and their exception paths must be the same; so $\gamma_1 = \gamma_2$ by Lemma 2.3. □

A simple but important property of the Γ functions is expressed in the following lemma.

Lemma 3.4. *Let $\pi_1 \leq \pi_2 \leq \pi_3$ be paths in σ. Then the following equation holds:*

$$\Gamma^\sigma_{\pi_1,\pi_2} \circ \Gamma^\sigma_{\pi_2,\pi_3} = \Gamma^\sigma_{\pi_1,\pi_3}.$$

Proof. By Lemmata 3.3 and 2.4, the exception paths are the same. Moreover, both sides take the same value on the argument $\sigma.\pi_3$. □

In addition, we map pairs of tree structures and variable assignments that satisfy a dominance formula to a variable assignment that satisfies an extended context formula. The following function, $[\![\cdot,\cdot]\!]$, is such a mapping, as Proposition 3.5 says.

$$\begin{array}{lcl}
[\![\mathcal{M}^\sigma,\alpha]\!](\top) & = & \sigma \\
[\![\mathcal{M}^\sigma,\alpha]\!](x) & = & \sigma.\alpha(X) \;\text{ for all variables } x \text{ such that } \alpha(X) \text{ is defined} \\
[\![\mathcal{M}^\sigma,\alpha]\!](C_X) & = & \Gamma^\sigma_{\epsilon,\alpha(X)} \text{ for all variables } C_X \text{ such that } \alpha(X) \text{ is defined.}
\end{array}$$

Proposition 3.5. *Let $\mathcal{M}^\sigma$ be a tree structure, α a variable assignment, and Φ a first-order formula over the dominance constraints. Φ is satisfied by $(\mathcal{M}^\sigma,\alpha)$ iff $[\![\Phi]\!]$ is satisfied by $[\![\mathcal{M}^\sigma,\alpha]\!]$.*

Proof. We prove the proposition by structural induction. First, we show that it is true for the atomic constraints; towards the end of the proof, we conduct the induction steps.

Throughout the proof, we write

$$\beta = [\![\mathcal{M}^\sigma,\alpha]\!]$$

for brevity.

- $X \lhd^* Y$

  "⇒" Assume that $(\mathcal{M}^\sigma,\alpha)$ satisfies $X \lhd^* Y$; we conclude that β satisfies the context equation on the right-hand side.

  Our assumption means that $\alpha(X) \leq \alpha(Y)$. Hence, we can construct a variable assignment β' that is like β, but assigns $\Gamma^\sigma_{\alpha(X),\alpha(Y)}$ to C. With this definition, $\beta'(C_Y) = \beta'(C_X \circ C)$, by Lemma 3.4.

  As a consequence, the existentially quantified context equation is satisfied by β.

  "⇐" Assume that the context equation is satisfied by β. Then there must be a context function γ such that $\beta(C_X) \circ \gamma = \beta(C_Y)$; hence, $\alpha(X)$ must be a prefix of $\alpha(Y)$, and $(\mathcal{M}^\sigma,\alpha)$ satisfies the dominance formula.

- $X{:}f(X_1, \ldots, X_n)$, where $n \geq 1$

"$\Rightarrow$" Assume that $(\mathcal{M}^\sigma, \alpha)$ satisfies $X{:}f(X_1, \ldots, X_n)$; we conclude that β satisfies each of the context equations on the right side.

For any $i = 1, \ldots, n$, let $\gamma_1 = \beta(C_X \circ f(x_1, \ldots, \bullet, \ldots, x_n))$ and $\gamma_2 = \beta(C_{X_i})$. We first show that the values of γ_1 and γ_2 on $\beta(x_i)$ are equal; then we show that their exception paths are the same. From Lemma 2.3, we can then conclude that γ_1 and γ_2 are equal and hence, the context equation is satisfied.

By definition, $\gamma_2(\beta(x_i)) = \sigma$. By the definition of the concatenation of context terms,

$$\begin{array}{rcl} \gamma_1(\beta(x_i)) & = & \beta(C_X)(f(\beta(x_1), \ldots, \beta(x_i), \ldots, \beta(x_n))) \\ & = & \Gamma^\sigma_{\epsilon,\alpha(X)}(f(\sigma.\alpha(X_1), \ldots, \sigma.\alpha(X_n))). \end{array}$$

As $(\mathcal{M}^\sigma, \alpha)$ satisfies the labeling constraint, we know that $\sigma(\alpha(X)) = f$ and that for all i, $\alpha(X_i) = \alpha(X) \cdot i$. So by Lemma 2.1 and the definition of $\Gamma^\sigma_{\epsilon,\alpha(X)}$, we conclude that $\gamma_1(\beta(x_i)) = \sigma$.

By definition, the exception path of γ_2 is $\alpha(X_i)$. On the other hand, by Lemma 2.4, the exception path of γ_1 is $\alpha(X) \cdot i$, which we already know to be equal to $\alpha(X_i)$. Hence, we can apply Lemma 2.3 and have shown this direction.

"$\Leftarrow$" Assume that β satisfies the conjunction of context equations on the right-hand side; we conclude that for all i, $\alpha(X_i) = \alpha(X) \cdot i$ and $\sigma(\alpha(X)) = f$ and hence, that the labeling constraint is satisfied by $(\mathcal{M}^\sigma, \alpha)$.

First, we show the condition on the paths. By assumption, we know that for all i, $\Gamma^\sigma_{\epsilon,\alpha(X_i)} = \Gamma^\sigma_{\epsilon,\alpha(X)} \circ \gamma_{f(x_1,\ldots,\bullet,\ldots,x_n)}$. If we reduce this equation to the exception paths, we have shown the first part.

Second, we show the condition on the label. By inserting from the previous argument, we know that

$$\begin{array}{rcl} \sigma & = & \Gamma^\sigma_{\epsilon,\alpha(X_i)}(\sigma.\alpha(X_i)) \\ & = & (\Gamma^\sigma_{\epsilon,\alpha(X)} \circ \gamma_{f(x_1,\ldots,\bullet,\ldots,x_n)})(\sigma.\alpha(X_i)) \\ & = & \Gamma^\sigma_{\epsilon,\alpha(X)}(f(\sigma.\alpha(X_1), \ldots, \sigma.\alpha(X_n))). \end{array}$$

Hence, the node at path $\alpha(X)$ must be labeled with f.

- $X{:}a$

"$\Rightarrow$" Assume that $(\mathcal{M}^\sigma, \alpha)$ satisfies $X{:}a$. This means that $\sigma(\alpha(X)) = a$ and that there is no $\pi \neq \epsilon$ such that $\alpha(X)\pi \in \mathrm{Dom}(\sigma)$. So by Lemma 2.1, $\beta(X) = \sigma.\alpha(X) = a$.

"$\Leftarrow$" Assume that β satisfies $X = a$; then $\beta(X) = \sigma.\alpha(X) = a$. In other words, the label of $\alpha(X)$ is a, which shows that the left-hand side is satisfied.

Of the complex cases, negation and conjunction are trivial. Existential quantification is more interesting:

- $\exists X.\Phi$

  "$\Rightarrow$" We assume that $(\mathcal{M}^\sigma, \alpha)$ satisfies $\exists X.\Phi$; so there is a path π such that $(\mathcal{M}^\sigma, \alpha[\pi/X])$ satisfies Φ. We call this new variable assignment α'. By induction hypothesis, $[\![\mathcal{M}^\sigma, \alpha']\!]$ satisfies $[\![\Phi]\!]$; it also satisfies $\top = C_X(x)$, by construction. $[\![\mathcal{M}^\sigma, \alpha']\!]$ and β agree on all arguments except possibly for x and C_X, so we can conclude that β satisfies the formula on the right-hand side.

  "$\Leftarrow$" We assume that β satisfies the formula on the right-hand side; this means that there are a tree σ' and a context function γ such that $\beta' = \beta[\sigma'/x, \gamma/C_X]$ satisfies both $\top = C_X(x)$ and $[\![\Phi]\!]$. We show that there is a variable assignment α' that only differs from α in X such that $\beta' = [\![\mathcal{M}^\sigma, \alpha']\!]$; from this and the induction hypothesis, satisfiedness of the left-hand side follows immediately.

  Let π be the exception path of γ. Then we know that as β' satisfies $\top = C_X(x)$, $\gamma(\sigma')$ must be equal to σ, and σ' must be equal to $\sigma.\pi$. We choose $\alpha' = \alpha[\pi/X]$ and verify that $\beta' = [\![\mathcal{M}^\sigma, \alpha']\!]$.

  $[\![\mathcal{M}^\sigma, \alpha']\!]$ agrees with β on all arguments except for x and C_X, to which it assigns $\sigma.\pi = \sigma'$ and $\Gamma^\sigma_{\epsilon,\pi}$, respectively. As the exception paths of γ and $\Gamma^\sigma_{\epsilon,\pi}$ are equal and they assume the same value on the argument σ', these context functions are the same, and we have shown that $\beta' = [\![\mathcal{M}^\sigma, \alpha']\!]$.

□

From the proposition, the theorem follows easily.

Proof (Theorem 3.2). Let σ be any tree. Clearly, if Φ is satisfied by $\mathcal{M}^\sigma$ and an arbitrary variable assignment α (remember that Φ must be closed), then $[\![\mathcal{M}^\sigma, \alpha]\!]$ satisfies $[\![\Phi]\!]$, according to the proposition. $[\![\mathcal{M}^\sigma, \alpha]\!]$ also assigns σ to the variable $\top$, by construction.

For the other direction of the proof, observe that the only free (context or tree) variable of $\Psi = [\![\Phi]\!]$ is the newly introduced $\top$. If β is a variable assignment that satisfies Ψ and assigns the tree σ to $\top$, we can restrict it to an assignment β' that *only* assigns σ to $\top$ and does not assign anything to any other variable; β' will satisfy Ψ as well. By definition, $\beta' = [\![\mathcal{M}^\sigma, \alpha_0]\!]$, where α_0 is the "empty variable assignment" whose domain is empty. Now, by Proposition 3.5, $(\mathcal{M}^\sigma, \alpha_0)$ satisfies Φ, so Φ describes σ. □

3.2 Parallelism

Now that we know how to encode dominance constraints as context constraints, it is surprisingly easy to finish the construction and encode parallelism and similarity constraints as context constraints and vice versa. Once we have done this, we have completed our proof that the expressive powers of context unification and of CLLS_0 are equal in the sense that for each constraint in one language, there is a satisfiability equivalent constraint in the other language. Unfortunately, there is no straightforward way to lift the strong result from the previous section, where the encoding even preserved solutions. This is because if we allow arbitrary context constraints (and not just encodings, as in the previous section), there might not be any tree variable that is mapped to the entire tree; in addition, the nodes denoted by the node variables must all be in the same tree. So we would have to construct a large tree around the trees mentioned in a satisfying variable assignment β, which would lead to a rather complicated correspondence between the solutions of the CLLS_0 and CU constraints and doesn't really seem to be worth the trouble.

At first sight, it is not even obvious that such an extended encoding can easily be obtained at all. This becomes much clearer, however, if we briefly review the theory of *equality up-to constraints*, as developed by Niehren et al. (1997a).

An equality up-to constraint is a conjunction of atomic constraints of the following form:

$$x/x'{=}y/y' \mid x{=}f(x_1, \ldots, x_n).$$

Variables are interpreted as trees; constraints can be satisfied by variable assignments β just like context constraints. The second type of constraint is satisfied if $\beta(x)$ is a tree whose top node is labeled with f and whose children are $\beta(x_1), \ldots, \beta(x_n)$. The first type of constraint is satisfied iff there is a context function γ such that $\beta(x) = \gamma(\beta(x'))$ and $\beta(y) = \gamma(\beta(y'))$. Intuitively, this is the case iff the trees denoted by x and y are *equal*, *up to* trees below a common subnode – hence the name.

The most important result about equality up-to constraints is that they are equivalent to context constraints. More precisely, there are satisfiability preserving en-

$$\begin{array}{|lcl|}\hline [\![X \sim Y]\!] & = & x = y \\ [\![X/X' {\sim} Y/Y']\!] & = & C_{X'} = C_X \circ C \wedge C_{Y'} = C_Y \circ C \quad (C \text{ a fresh context variable}) \\ \hline \end{array}$$

Figure 3.3: Encoding parallelism and similarity constraints as context equations. This completes the encoding from Fig. 3.1. To close off the resulting context constraint, add $\exists C$ to the second clause.

codings of equality up-to constraints into context constraints (Niehren et al. 1997a, Prop.15) and vice versa (dto., Prop. 17). Note that their result is only about the constraint languages, not about the first-order theories; so we can't state our theorem for the first-order theories, either. It would probably be possible to repair their proofs to cover first-order theories, but as first-order languages are not our main concern here, we will not do so and instead, restrict ourselves to the constraint languages.

With this result, the proof of the following theorem, stating the equivalence of CLLS_0 and context unification, is reasonably easy.

Theorem 3.6 (Encoding Parallelism). *For every $CLLS_0$ constraint φ, there is a satisfiability equivalent context constraint ψ, and vice versa.*

Proof. First, we present the satisfiability preserving encoding of CLLS_0 into CU (Fig. 3.3); then we look into an encoding of context constraints into CLLS_0 via equality up-to constraints (Fig. 3.4).

For the first direction, we extend our encoding $[\![\cdot]\!]$ by the clauses in Fig. 3.3, covering parallelism and similarity constraints. For both types of atomic constraints, we show the base cases for the structural induction of Proposition 3.5; it follows that Proposition 3.1, extended with the definitions in Fig. 3.3, still holds, which implies our claim.

- $X \sim Y$

  "$\Rightarrow$" Assume that $(\mathcal{M}^\sigma, \alpha)$ satisfies $X \sim Y$; call $\pi_1 = \alpha(X)$ and $\pi_2 = \alpha(Y)$, then $\sigma.\pi_1 = \sigma.\pi_2$. As $[\![\mathcal{M}^\sigma, \alpha]\!](x) = \sigma.\pi_1$ and $[\![\mathcal{M}^\sigma, \alpha]\!](y) = \sigma.\pi_2$. the context constraint is satisfied as well.

  "$\Leftarrow$" We assume that $[\![\mathcal{M}^\sigma, \alpha]\!]$ satisfies $x = y$. Then as above, $\sigma.\pi_1 = \sigma.\pi_2$, and the similarity constraint is satisfied.

$$\llbracket x/x'{=}y/y' \rrbracket^{-1} = X/X''{\sim}Y/Y'' \wedge X' \sim X'' \wedge Y' \sim Y'' \ (X'', Y'' \text{ fresh})$$

$$\llbracket x{=}f(x_1, \ldots, x_n) \rrbracket^{-1} = X{:}f(X_1', \ldots, X_n') \wedge X_1 \sim X_1' \ldots X_n \sim X_n' \quad (X_1', \ldots, X_n' \text{ fresh})$$

Figure 3.4: Encoding equality up-to constraints in CLLS_0.

- $X/X'{\sim}Y/Y'$

  "$\Rightarrow$" Assume that $(\mathcal{M}^\sigma, \alpha)$ satisfies $X/X'{\sim}Y/Y'$; call the paths that α assigns to these variables π_1, π_1', π_2, and π_2', respectively. We need to show that there is a context function γ such that $\Gamma^\sigma_{\epsilon,\pi_1'} = \Gamma^\sigma_{\epsilon,\pi_1} \circ \gamma$ and $\Gamma^\sigma_{\epsilon,\pi_2'} = \Gamma^\sigma_{\epsilon,\pi_2} \circ \gamma$; from this, the satisfiedness of the right-hand side follows by definition.

  Rephrasing the semantics of the parallelism constraint, we know that $\Gamma^\sigma_{\pi_1,\pi_1'}$ and $\Gamma^\sigma_{\pi_2,\pi_2'}$ have a common exception path, π_0, and that their corresponding contexts are equal. Clearly, $\Gamma^\sigma_{\pi_1,\pi_1'} = \Gamma^\sigma_{\pi_2,\pi_2'}$.

  Now, from Lemma 3.4, we know that $\Gamma^\sigma_{\epsilon,\pi_1'} = \Gamma^\sigma_{\epsilon,\pi_1} \circ \Gamma^\sigma_{\pi_1,\pi_1'}$ and $\Gamma^\sigma_{\epsilon,\pi_2'} = \Gamma^\sigma_{\epsilon,\pi_2} \circ \Gamma^\sigma_{\pi_2,\pi_2'}$, so we have found the context function γ we needed to find.

  "$\Leftarrow$" Assume that $\llbracket \mathcal{M}^\sigma, \alpha \rrbracket$ satisfies $\exists C.(C_{X'} = C_X \circ C \wedge C_{Y'} = C_Y \circ C)$; this means that there is a context function γ such that $\Gamma^\sigma_{\epsilon,\alpha(X')} = \Gamma^\sigma_{\epsilon,\alpha(X)} \circ \gamma$ and $\Gamma^\sigma_{\epsilon,\alpha(Y')} = \Gamma^\sigma_{\epsilon,\alpha(Y)} \circ \gamma$. We need to show that there is a path π_0 such that $\alpha(X') = \alpha(X)\pi_0$ and $\alpha(Y') = \alpha(Y)\pi_0$, and that the contexts between these nodes are equal.

  The first part is trivial, as by Lemma 2.4, the exception path of the context function γ is such a common extension. For the other part, observe that by Lemma 3.4, $\Gamma^\sigma_{\epsilon,\alpha(X')} = \Gamma^\sigma_{\epsilon,\alpha(X)} \circ \Gamma^\sigma_{\alpha(X),\alpha(X')}$ and $\Gamma^\sigma_{\epsilon,\alpha(Y')} = \Gamma^\sigma_{\epsilon,\alpha(Y)} \circ \Gamma^\sigma_{\alpha(Y),\alpha(Y')}$. As context functions are injective, the first of these equations implies that $\gamma = \Gamma^\sigma_{\alpha(X),\alpha(X')}$ and the second one, that $\gamma = \Gamma^\sigma_{\alpha(Y),\alpha(Y')}$; hence, the two context functions and, consequently, their contexts are equal.

This concludes the first half of the proof. For the other half of the proof, we have to encode an arbitrary context constraint ψ as a CLLS_0 constraint. This encoding proceeds in three steps. First, we encode ψ as an equality up-to constraint ψ' according to (Niehren et al. 1997a). Next, we choose a node variable X for every tree variable x that appears in ψ'; the purpose of these nodes variables is to denote nodes the trees under which look like the trees denoted by x. Finally, we apply the encoding $[\![\cdot]\!]^{-1}$ in Fig. 3.4.

It is obvious that this encoding preserves satisfiability. The encoding of the $\sim$ constraint expresses that somewhere below the nodes X and Y, there are nodes X'' and Y'' the trees below which look just like the trees below the nodes X' and Y', and the contexts between X and X'' and Y and Y'' are equal. (Note that this is a weaker condition than parallelism itself; it does not say anything about the locations of the nodes denoted by X' and Y'.) The encoding of the "labeling" equality up-to constraint works similarly: It expresses that X is labeled with f and that its subtrees look just like the subtrees below the $X_1, \ldots, X_n$. $\square$

3.3 Conclusion

The major result of this chapter is that the two constraint languages under investigation in this thesis, CLLS_0 and context unification, are equivalent: Every CLLS_0 constraint can be encoded as a satisfiability equivalent context constraint, and vice versa.

The most involved part of our proof was to embed dominance constraints as context constraints; once that was out of the way, the same intuitions carried over to an embedding of similarity and parallelism constraints (which, incidentally, generalize dominance constraints). We described a node by specifying both its context and the tree below it; Lemma 3.3 stated that this is always possible uniquely in a formal way. We even showed the more general result that the encoding in this direction does not only preserve satisfiability, it also preserves the satisfying trees in a certain sense; in addition, the result not only holds for constraints, but for all first-order sentences.

The other direction, the encoding of context constraints in CLLS_0, proceeded in two steps: First, we encoded them as equality up-to constraints, which have a much more restricted syntax that makes proofs easier; then we encoded the resulting equality up-to constraint in CLLS_0 in a straightforward manner that obviously preserved satisfiability.

This correspondence between the two languages has two important consequences. For one, it allows us to transfer complexity results between them; if we can determine the complexity of one language, we immediately know that the complexity of the other language must be the same. As the complexity of context unification is actively being investigated by several scientists throughout Europe, we can hope that this will eventually give us a complexity result for CLLS_0. For the time being, however, the complexity of either language is unknown, and it is reasonable to investigate the complexity of sublanguages. We will do this for dominance constraints, the most interesting sublanguage of CLLS_0, in the next two chapters.

The other consequence is a clarification of earlier papers on the linguistics application of context unification. We have seen in the previous chapter that for a correct treatment of scope ambiguities in context unification, one has to write down slightly unintuitive constraints. The exact type of constraints we used was introduced by Niehren et al. (1997b); but it was not entirely clear why, exactly, the encoding as context equations worked, and if the more complex encoding really solved all problems. In the light of the results of this chapter, we easily recognize the context equations in Example 2.12 as the encoding of a dominance constraint.

Chapter 4

Complexity of Dominance Constraints

As we have seen in the previous chapter, the complexities of solving $CLLS_0$ and context constraints are the same, but we know neither of these complexities. Hence, it is reasonable to investigate the complexity of sublanguages, both from a systematic point of view (if the complexity of the entire language is too difficult to determine, maybe we can say something about fragments?) and from a practical, implementation-oriented point of view.

In this chapter, we explore the complexity of the language of dominance constraints, consisting only of atomic labeling and dominance constraints, and show that the satisfiability problems of dominance constraints and, in case of a finite signature, also of the propositional language and the positive existential first-order fragment over the language are NP-complete. As we have seen in the introduction, dominance constraints are not only the most interesting sublanguage of CLLS, providing the expressive power to deal with scope ambiguities, but also a popular formalism for linguistic analysis in their own right.

In the first section, we present an algorithm that decides the satisfiability of a dominance constraint. This algorithm saturates the constraint by adding entailed conjuncts and checking for clashes; it runs in nondeterministically polynomial time. We prove that it is sound and complete for the problem and show how to raise the result to the positive existential fragment.

In the second section, we show that the satisfiability problem even of the purely conjunctive language of dominance constraints is NP-hard. To this end, we encode formulae of propositional logic in conjunctive normal form with three literals per clause (3-CNF) as satisfaction equivalent dominance constraints; the satisfiability problem of these formulae, 3SAT, is a classical NP-complete problem. The key to this encoding is the "dominance triangle", a special constraint that allows to express disjunctions. Together with the result from the first section, we have thus shown the complexity result we claimed above.

Finally, we investigate implementations. We have just said that the general problem of solving dominance constraints is intractable; but the implementation that we sketch in the third section runs efficiently at least in those cases that seem to be of linguistic relevance. This implementation, taken from (Duchier and Gardent 1999), employs set constraints in the programming language Oz. In addition, we sketch a streamlined implementation of a solver for context constraints. This procedure does not necessarily terminate, but on linguistic examples, it runs with reasonable efficiency. We present it as a first stab at an implementation of parallelism; at this time, there is no known algorithm for solving parallelism constraints within CLLS yet, and we must take resort to the more general formalism of context unification.

Throughout the chapter, we will use the symbol $=$ to denote that two variables must be mapped to the same node; $X{=}Y$ is simply an abbreviation for the conjunction $X \lhd^* Y \wedge Y \lhd^* X$. Furthermore, we assume a signature that not only contains a nullary constructor a, but also a constructor g of arity $r \geq 2$. We will use these two constructors to simulate constructors of all other arities. As before, we will consider satisfaction of dominance constraints over tree (and not lambda) structures.

4.1 Solving Dominance Constraints

We now present an algorithm that decides the satisfiability of a conjunction over labeling, dominance, and negated dominance constraints $\neg X \lhd^* Y$; for the most part of this section, we deviate from the original definition by considering negated dominance constraints atomic as well. We prove that the algorithm is sound and complete and that it runs in nondeterministically polynomial time.

4.1.1 The algorithm

The algorithm proceeds in three steps. First, we guess for each pair X, Y of variables in φ if X dominates Y or not, and add the corresponding atomic constraint to φ. This can be expressed as in the following rule, where $\vee$ expresses nondeterministic choice.

$$\mathbf{true} \quad\to\quad X \lhd^* Y \vee \neg X \lhd^* Y$$

In the second step, we saturate φ according to the following deterministic *propagation* rules.

(Refl)	$\mathbf{true} \quad\to\quad X \lhd^* X$	(X occurs in φ)
(Trans)	$X \lhd^* Y \wedge Y \lhd^* Z \quad\to\quad X \lhd^* Z$	
(Decomp)	$X{:}f(X_1, \ldots, X_n) \wedge Y{:}f(Y_1, \ldots, Y_n) \wedge X{=}Y \quad\to\quad \bigwedge_{i=1}^n X_i{=}Y_i$	
(Disj)	$X{=}X' \wedge X{:}f(\ldots, Y, \ldots) \wedge X'{:}f(\ldots, Z, \ldots) \quad\to\quad \neg Y \lhd^* Z \wedge \neg Z \lhd^* Y$	
(Dom)	$X{:}f(\ldots, Y, \ldots) \quad\to\quad X \lhd^* Y$	
(Parent)	$X{=}Y \wedge X'{:}f(\ldots, X, \ldots) \wedge Y'{:}g(\ldots, Y, \ldots) \quad\to\quad X'{=}Y'$	
(Child)	$X \lhd^* Y \wedge X{:}f(X_1, \ldots, X_n) \wedge \bigwedge_{i=1}^n (\neg X_i \lhd^* Y) \quad\to\quad Y \lhd^* X$	

(In the Disj rule, Y is the i-th and Z is the k-th argument of the labeling constraints, where $i \neq k$; in the Parent rule, f and g need not be different.)

In the third step, we detect unsatisfiable constraints by applying the following *clash* rules.

(Clash1)	$X{:}f(\ldots) \wedge Y{:}g(\ldots) \wedge X{=}Y \quad\to\quad \mathbf{false}$,	if $f \neq g$
(Clash2)	$X \lhd^* Z \wedge Y \lhd^* Z \wedge \neg X \lhd^* Y \wedge \neg Y \lhd^* X \quad\to\quad \mathbf{false}$	
(Clash3)	$X \lhd^* Y \wedge \neg X \lhd^* Y \quad\to\quad \mathbf{false}$	
(Clash4)	$X{:}f(X_1, \ldots, X_i, \ldots, X_n) \wedge X_i \lhd^* X \quad\to\quad \mathbf{false}$	

As we have guessed the dominance relations between all variables in the first step, the second step can never consistently add a new constraint; either the constraint is already known, or it clashes, by the Clash3 rule. In other words, the propagation rules don't really propagate anything. We could have rewritten them all as Clash rules, but writing them as propagation rules simplifies some proofs. As a naive application strategy, we could simply try all instances of all propagation and clash rules; if one of these applications leads to a clash, we reject, and otherwise, we accept. Even this simple strategy only takes a polynomial number of steps because every rule only has a polynomial number of instances, so the runtime of the algorithm is in NP. In the next two subsections, we show that it also produces correct results by proving its soundness and completeness.

4.1.2 Soundness

We call a constraint φ *consistent* if it has a saturation with respect to the above rule system that does not contain **false**. It is inconsistent if every saturation contains **false**.

Proposition 4.1 (Soundness). *If a dominance constraint φ is satisfiable, then it is consistent.*

Proof. Assume that the constraint φ is satisfiable. We first prove that there is a choice of atomic (possibly negated) dominance constraints whose conjunction with φ is satisfiable as well; we call this conjunction φ'. Then we prove that the left-hand side of every Clash rule is unsatisfiable. Finally, we show for each propagation rule that if its left-hand side is satisfiable, so is its conjunction with the right-hand side. It follows that we can saturate φ' with propagation rules such that no clash rule is applicable on the result; hence, this particular saturation cannot contain **false**, and φ is consistent.

For the first part of the proof, let $(\mathcal{M}, \alpha)$ be such that it satisfies φ. For every pair X, Y of variables in φ, we consider the paths $\alpha(X)$ and $\alpha(Y)$; if the former is a prefix of the latter, we add $X \triangleleft^* Y$ to φ, otherwise we add $\neg X \triangleleft^* Y$ to φ. Clearly, each of these conjuncts is satisfied by $(\mathcal{M}, \alpha)$, so the entire conjunction φ' is too.

Now consider a clash rule; we show that its left-hand side must be unsatisfiable. This is very simple to prove; we only show, as an example, that it is true for the Clash2 rule. Suppose there is a pair $(\mathcal{M}, \alpha)$ that satisfies the conjunction $X \triangleleft^* Z \wedge Y \triangleleft^* Z \wedge \neg X \triangleleft^* Y \wedge \neg Y \triangleleft^* X$; then $\alpha(X) \leq \alpha(Z)$ and $\alpha(Y) \leq \alpha(Z)$. However, by the other two conjuncts, neither of $\alpha(X)$ and $\alpha(Y)$ can be a prefix of the other, so they must differ in one position. But this contradicts the assumption that they have a common extension $\alpha(Z)$, so the left-hand side of the Clash2 rule is unsatisfiable.

Finally, we show that satisfiability of the left-hand side L of a propagation rule implies satisfiability of the conjunction $L \wedge R$, where R is the right-hand side of the rule. Again, this is generally simple, and we only show it for the Parent rule, by way of example. Assume that $(\mathcal{M}, \alpha)$ satisfies the left-hand side of this rule; we show that it also satisfies the right-hand side, and hence, the conjunction. As $X{=}Y$ is satisfied, $\alpha(X)$ must be the same as $\alpha(Y)$ because they are prefixes of each other. Furthermore, by definition of the semantics of labeling constraints, $\alpha(X) = \alpha(X')\cdot i$ if X is the i-th argument of the constraint. By the same argument, $\alpha(Y) = \alpha(Y')\cdot k$ if Y is the k-th argument of the constraint. Now, as the constraint on the LHS is satisfied, we can conclude that not only must i and k be the same integer, but also $\alpha(Y') = \alpha(X')$, which means that $(\mathcal{M}, \alpha)$ also satisfies the constraint $X'{=}Y'$. $\square$

4.1.3 Completeness

Theorem 4.2 (Completeness). *If a dominance constraint φ is consistent, then it is satisfiable.*

According to the premise of the theorem, there is one saturation of φ that does not contain **false**. We call the set of atomic conjuncts in this saturation S.

For the proof, we proceed in six steps.

1. Define an auxiliary graph G' from S. G' mirrors all conjuncts in the saturation, including redundant dominances. Its nodes are equivalence classes of the variables of φ, and its edges correspond to dominance and labeling constraints.
2. Show that G' is well-defined.
3. Show that G' is 'well-behaved' in a certain sense; for example, that it is almost acyclic, that there are no unresolved ambiguities, etc. (We make this more precise below.)
4. Build another graph G that is like G', but does not contain redundant dominances. We could call G a *solved form* of φ. Show that G is even more 'well-behaved' than G'; it is essentially a forest.
5. Build a satisfying tree structure and a variable assignment from G.
6. Show that this pair satisfies φ.

It is trivial that whenever both $X \triangleleft^* Y$ and $\neg X \triangleleft^* Y$ are in S, S is inconsistent by the Clash3 rule. We will not mention this rule application below, due to its frequency, and simply infer that $X \triangleleft^* Y$ is in S whenever $\neg X \triangleleft^* Y$ isn't, and vice versa.

1. Define G'. So first of all, let us define G'. $G' = (V', E', L_{V'}, L_{E'})$ is a directed graph with node and edge labels; the alphabet for the node labels is Σ, and the alphabet for the edge labels is $\mathbb{N} \cup \{\triangleleft^*\}$. $L_{E'}$ is a total function; $L_{V'}$ does not have to be.

The relation $=_S$, defined by

$$X =_S Y \quad \Leftrightarrow \quad (X = Y) \in S,$$

is an equivalence relation. This follows from saturation of S under the Refl and Trans rules. We define the node set V' of G' to be the set of equivalence classes of the variables of φ under this equivalence relation.

As the set E' of edges of G', we take the set of pairs $(\bar{X}, \bar{Y})$ of equivalence classes such that there are representatives X, Y for which either $X \triangleleft^* Y$ or $X{:}f(\dots, Y, \dots)$ are in S.

Finally, we define

$$L_{V'}(\bar{X}) = \begin{cases} f & \text{if there is a representative } X \text{ s.t. } X{:}f(\dots) \in S; \\ \mathit{undef} & \text{otherwise} \end{cases}$$

and

$$L_{E'}(\bar{X}, \bar{Y}) = \begin{cases} i & \text{if there are representatives } X, Y \text{ s.t. } X{:}f(\dots, Y, \dots) \in S, \\ & \text{where } Y \text{ is the } i\text{-th argument;} \\ \triangleleft^* & \text{otherwise.} \end{cases}$$

2. Well-definedness of G'. In order to show that G' is well-defined, we need to show that $L_{V'}$ and $L_{E'}$ are; i.e., that every node of G' is assigned only one label and that every $\mathbb{N}$-edge is assigned only one number. (Trivially, no edge is labeled both with $\triangleleft^*$ and a number.)

For the first claim, assume that a node $\bar{X}$ has representatives X, X' such that $X{:}f(\dots), X'{:}g(\dots) \in S$ and $f \neq g$ (which is the situation that would make L_V ill-defined). As $X{=}X' \in S$, we can derive failure with the Clash1 rule, in contradiction to our assumption that φ is consistent.

For the second claim, assume that $L_{E'}$ is ill-defined; this means that there are nodes $\bar{X}, \bar{Y}$ with representatives X, X' and Y, Y', respectively, such that $Y{:}f(\dots, X, \dots), Y'{:}g(\dots, X', \dots) \in S$, where X is the i-th and X' is the k-th argument, and $i \neq k$. (f and g need not be different.) By construction, $X{=}X' \in S$; so by the Parent rule, we can derive $Y{=}Y' \in S$. As S is consistent, it must be the case that $f = g$; otherwise, the Clash1 rule would make S inconsistent. But now, by the Disj rule, $\neg X \triangleleft^* X' \in S$, so S is inconsistent.

Some properties of G'. G' is a graph that contains edges that are labeled with $\triangleleft^*$ and edges that are labeled with an integer. We collectively call the second type of edge $\mathbb{N}$-*edge*. Furthermore, we call a cycle in G' that contains an $\mathbb{N}$-edge an $\mathbb{N}$-*cycle* and a cycle that contains only $\triangleleft^*$-edges a $\triangleleft^*$-*cycle*. To express the fact that there is an edge from $\bar{X}$ to $\bar{Y}$ that is labeled with a, we write $\bar{X}[a]\bar{Y}$. Note that the source node of an $\mathbb{N}$-edge is always labeled; and if a node is labeled with a non-nullary constructor, it has an outgoing $\mathbb{N}$-edge.

If there is a path in G' from $\bar{X}$ to $\bar{Y}$ (over edges of any types), we say that $\bar{X}$ is *reachable* from $\bar{Y}$ and write $\bar{X} \rightsquigarrow_{G'} \bar{Y}$. Whenever the context makes it clear, we leave the subscript away; we will see later (Lemma 4.8) that the $\rightsquigarrow$ relation is the same on both graphs that we consider, anyway. It has the important property that a node is reachable from another iff a corresponding dominance constraint is in S, as we show now.

Lemma 4.3. *For any two variables X, Y, $X \triangleleft^* Y \in S$ iff $\bar{X}[\triangleleft^*]\bar{Y}$ or $\bar{X}[i]\bar{Y}$ for an $i \in \mathbb{N}$.*

Proof. The left-to-right direction is trivial. The right-to-left direction is not much more difficult. If $\bar{X}[\triangleleft^*]\bar{Y}$, there are representatives X', Y' such that $X' \triangleleft^* Y', X{=}X', Y{=}Y' \in S$. Hence, by the Trans rule, $X \triangleleft^* Y$ is in S as well. If $\bar{X}[i]\bar{Y}$ for any i, there are representatives X', Y' such that $X'{:}f(\ldots, Y', \ldots) \in S$. By the Dom rule, $X' \triangleleft^* Y'$ is in S as well, and we can continue as above. □

Lemma 4.4. *For any two variables X, Y, $\bar{X} \rightsquigarrow_{G'} \bar{Y}$ iff $\bar{X}[\triangleleft^*]\bar{Y}$ or $\bar{X}[i]\bar{Y}$ for an $i \in \mathbb{N}$.*

Proof. The right-to-left direction is trivial. For the left-to-right direction, let $e_1, \ldots, e_n$ be an arbitrary path of length $n \geq 1$ from $\bar{X}$ to $\bar{Y}$; we show the lemma by induction over n. (For a path of length 0, the lemma also holds because of Lemma 4.3 and the Refl rule.)

<u>$n = 1.$</u> trivial.

<u>$n - 1 \to n.$</u> Let $\bar{Z}$ be the goal node of e_1. $e_2, \ldots, e_n$ is a path of length $n-1$; so by the induction hypothesis, we know that either $\bar{Z}[\triangleleft^*]\bar{Y}$ or $\bar{Z}[i]\bar{Y}$, for an $i \in \mathbb{N}$. Whichever is the case, we know by Lemma 4.3 that there are representatives Y, Z such that $Z \triangleleft^* Y \in S$.

Now e_1 can either be an $\mathbb{N}$-edge or a $\triangleleft^*$-edge; whichever is the case, we know by Lemma 4.3 that there is a representative X of $\bar{X}$ such that $X \triangleleft^* Z \in S$. By application of the Trans rule, we know that $X \triangleleft^* Y \in S$, and by the other direction of Lemma 4.3, we conclude that there is an edge from $\bar{X}$ to $\bar{Y}$.

□

Corollary 4.5. *For any two variables X, Y, $X \triangleleft^* Y \in S$ iff $\bar{X} \rightsquigarrow_{G'} \bar{Y}$.*

Proof. By Lemmas 4.3 and 4.4. □

Corollary 4.6. $\rightsquigarrow_{G'}$ *is a partial order on* V'.

Proof. Using Corollary 4.5, transitivity of $\rightsquigarrow$ follows from closure of S under the Trans rule, reflexivity follows from closure under the Refl rule, and antisymmetry follows from the construction of V'. □

3. Well-behavedness of G'. As a prerequisite for the later parts of the proof, we need to know that G' is 'well-behaved' in a certain sense. The following lemma makes this notion precise.

Lemma 4.7. *G' has the following properties:*

1. *The only cycles that G' contains are the dominance edges $(\bar{X}, \bar{X})$.*
2. *If a node in G' is labeled with f ($\mathsf{ar}(f) = n$), it has exactly one i-child for each $1 \leq i \leq n$ and no other $\mathbb{N}$-children. All of its $\mathbb{N}$-children are different.*
3. *No node in G' has two incoming $\mathbb{N}$-edges.*
4. *If a node in G' has two incoming edges, one of its parents is reachable from the other parent.*

Proof.

1. Suppose G' contains an $\mathbb{N}$-cycle; w.l.o.g., let $e_1, \ldots, e_n$ be its edges such that $e_1 = (\bar{X}, \bar{Y})$ is an $\mathbb{N}$-edge. Then there must be representatives X of $\bar{X}$ and Y of $\bar{Y}$ such that $X{:}f(\ldots, Y, \ldots) \in S$.

   $\bar{X}$ is reachable from $\bar{Y}$; so by Corollary 4.5, $X \triangleleft^* Y \in S$. By the Clash4 rule, S is inconsistent, so G' cannot contain an $\mathbb{N}$-cycle.

   If, on the other hand, G' contains a $\triangleleft^*$-cycle, we know for any two nodes $\bar{X}$ and $\bar{Y}$ on the cycle that $\bar{X} \rightsquigarrow \bar{Y}$ and $\bar{Y} \rightsquigarrow \bar{X}$; so for any representatives X, Y, we conclude by Corollary 4.5 that $X{=}Y \in S$. Hence, $\bar{X}$ and $\bar{Y}$ are the same nodes, and the cycle is just a loop.

2. Let $\bar{X}$ be a node in G' such that $L_{V'}(\bar{X}) = f$ and $\mathsf{ar}(f) = n$. By definition, we know that $\bar{X}$ has a representative X such that there is a labeling constraint $X{:}f(X_1, \ldots, X_n)$ in S. Furthermore, for every $1 \leq i \leq n$, there is an i-edge from $\bar{X}$ to $\bar{X}_i$.

   For the uniqueness result, assume that there are nodes $\bar{Y}, \bar{Z}$ and an i such that $\bar{X}[i]\bar{Y}$ and $\bar{X}[i]\bar{Z}$. By definition, there are representatives X, X', Y, Z such that there are labeling constraints $X{:}f(\ldots, Y, \ldots), X'{:}f(\ldots, Z, \ldots)$ in

S, where both Y and Z are i-th children. (We can show that the labels must be the same as in the proof of well-definedness of G'.) By the Decomp rule, this implies that $Y{=}Z \in S$, so $\bar{Y} = \bar{Z}$.

To show that all $\mathbb{N}$-children are different, assume otherwise. Suppose there are different integers $1 \leq i, k \leq n$ and a node $\bar{Y}$ such that both $\bar{X}[i]\bar{Y}$ and $\bar{X}[k]\bar{Y}$. This means that there are representatives X, X' of $\bar{X}$ and Y, Y' of $\bar{Y}$ such that all of $X{:}f(\ldots, Y, \ldots), X'{:}f(\ldots, Y', \ldots), X{=}X', Y{=}Y'$ are in S. By the Disj rule, we can conclude that $\neg Y \lhd^* Y' \in S$, which means that S is inconsistent by the Clash3 rule.

Finally, to show that $\bar{X}$ has no other $\mathbb{N}$-children, assume it does have a k-child $\bar{Y}$ with $k > n$. This means that there are representatives for which a labeling constraint with a constructor of arity higher than n is in S. But as we have shown above, all labeling constraints for a representative of $\bar{X}$ must be labelings with the n-ary label f.

3. Suppose that the node $\bar{Z}$ has two incoming $\mathbb{N}$-edges, i.e. there are nodes $\bar{Y}, \bar{Z}$ and numbers i, k such that $\bar{X}[i]\bar{Z}$ and $\bar{Y}[k]\bar{Z}$. This means that there are representatives X, Y, Z, Z' such that $Z{=}Z', X{:}f(\ldots, Z, \ldots), Y{:}g(\ldots, Z', \ldots) \in S$ (f, g not necessarily different). By the Parent rule, $X{=}Y \in S$ and hence, $\bar{X} = \bar{Y}$. Furthermore, all $\mathbb{N}$-children of $\bar{X}$ are different, so i and k must be equal.

4. Assume that $\bar{Z}$ has two incoming edges, say, from the nodes $\bar{X}$ and $\bar{Y}$. As we have just seen, it is not possible that both edges are $\mathbb{N}$-edges; so one of the following cases must be true:

   (a) Both edges are $\lhd^*$-edges. Then there must be representatives X, Y, Z, Z' such that $X \lhd^* Z, Y \lhd^* Z' \in S$. By the Trans and Dom rules, we conclude that $Y \lhd^* Z$ is in S as well.
   Suppose that neither $X \lhd^* Y$ nor $Y \lhd^* X$ were in S; then both $\neg X \lhd^* Y$ and $\neg Y \lhd^* X$ must be in S. By the Clash2 rule, this is inconsistent, in contradiction to our assumption. So one of the two dominances must be in S, and hence, there must be a $\lhd^*$-edge between $\bar{X}$ and $\bar{Y}$ (in one direction).

   (b) One edge is an $\mathbb{N}$-edge, and the other is a $\lhd^*$-edge; without loss of generality, assume that $\bar{X}[i]\bar{Z}$ and $\bar{Y}[\lhd^*]\bar{Z}$. Then there must be representatives X, Y, Z, Z' such that $X{:}f(\ldots, Z, \ldots), Y \lhd^* Z', Z{=}Z' \in S$. By the Dom rule, we conclude that $X \lhd^* Z \in S$; by the Trans rule, we conclude that $Y \lhd^* Z \in S$. From this point, we can proceed as in the previous case.

$\square$

4. Construct G. As we have just shown, the auxiliary graph G' has a rather pleasant structure. However, it still contains a lot of redundant dominance edges; Lemma 4.4 says that if we can reach one node from another in G' via any path, we can do so via a single edge because all paths of length > 1 can be abbreviated by going over a dominance edge. This redundancy gets in the way of the construction of a satisfying tree structure, so we define a new graph G that has essentially the same structure as G', but does not contain the redundant edges.

G has the same nodes and node labeling function as G'. We define its edge set E to be

$$\begin{array}{rcll} E & = & \{e \in E' & \mid \quad L_{E'}(e) \in \mathbb{N}\} \\ & \cup & \{e \in E' & \mid \quad e \neq (\bar{X}, \bar{X}) \text{ and there is no cycle-free path } e_1, \ldots, e_k \ (k > 1) \\ & & & \quad \text{in } E' \text{ such that } start(e_1) = start(e) \text{ and } goal(e_k) = goal(e)\} \end{array}$$

and obtain L_E as the restriction of $L_{E'}$ to E.

The fact that G has essentially the same structure as G' is expressed by the following lemma.

Lemma 4.8. *For any nodes $\bar{X}, \bar{Y} \in V$, there is a path from $\bar{X}$ to $\bar{Y}$ in G iff there is a path from $\bar{X}$ to $\bar{Y}$ in G'.*

Proof. One direction of this is trivial: $E \subseteq E'$, so if there is such a path in G, it also exists in G'.

For the other direction, we prove that every cycle-free path $e_1, \ldots, e_k$ from $\bar{X}$ to $\bar{Y}$ in G' of maximal length also exists in G, by induction over k. From this, the lemma follows, as existence of a path from $\bar{X}$ to $\bar{Y}$ implies existence of a cycle-free path of maximal length.

$\underline{k = 0.}$ Such a path does not use any edges, so it also exists in G.

$\underline{k = 1.}$ As the longest cycle-free path from $\bar{X}$ to $\bar{Y}$ has length 1, $e_1 \in E$ by definition.

$\underline{k - 1 \to k.}$ Let $\bar{Z}$ be the goal of the edge e_1. $e_2, \ldots, e_n$ is a maximal cycle-free path from $\bar{Z}$ to $\bar{Y}$: If there was a longer cycle-free path from $\bar{Z}$ to $\bar{Y}$, we could build a longer cycle-free path from $\bar{X}$ to $\bar{Y}$ from it by prefixing it with e_1 because by Lemma 4.7, we know that the only cycles that G' contains are the reflexive loops. Furthermore, e_1 is a longest cycle-free path from $\bar{X}$ to $\bar{Z}$, by the same argument. Hence, by the induction hypothesis, the path $e_1, \ldots, e_k$ also exists in G.

□

Corollary 4.9. *For any two variables X, Y, $X \lhd^* Y \in S$ iff $\bar{X} \rightsquigarrow_G \bar{Y}$.*

Proof. Follows from Lemma 4.8 and Corollary 4.5. □

Lemma 4.10. *G has the following properties:*

1. *G has no cycles.*
2. *If a node in G is labeled with f ($\mathsf{ar}(f) = n$), it has exactly one i-child for each $1 \leq i \leq n$ and no other $\mathbb{N}$-children. All of its $\mathbb{N}$-children are different.*
3. *If a node in G is labeled, it has no $\lhd^*$-children.*
4. *If $\bar{X}$ and $\bar{Y}$ are two nodes in G, there is at most one path from $\bar{X}$ to $\bar{Y}$.*
5. *No node in G has two incoming edges.*

Proof.

1. According to Lemma 4.7, the only cycles in G' are loops. These loops are not in E, and no additional edges that could be part of cycles were added.
2. Follows from Lemma 4.7 because E and E' contain the same $\mathbb{N}$-edges.
3. Assume that $\bar{X}$ is labeled with f, and that $\bar{X}[\lhd^*]\bar{Y}$. Then there must be representatives X, X' of $\bar{X}$, a representative Y of $\bar{Y}$, and variables $X_1, \ldots, X_n$ ($n \geq 0$) such that $X{:}f(X_1, \ldots, X_n), X' \lhd^* Y \in S$. By the Trans rule, $X \lhd^* Y$ is also in S. Now we distinguish the following cases (if $n = 0$, the first two cases can never occur) and derive a contradiction from each case.
   (a) For one X_i, $X_i{=}Y \in S$. Then $\bar{Y} = \bar{X}_i$, and both a $\lhd^*$-edge and an $\mathbb{N}$-edge lead from $\bar{X}$ to this node, which is not even possible in G'.
   (b) For one X_i, $\bar{X}_i \rightsquigarrow \bar{Y}$ but $\bar{X}_i \neq \bar{Y}$. Then the $\lhd^*$-edge from $\bar{X}$ to $\bar{Y}$ abbreviates the path of length > 1 from $\bar{X}$ to $\bar{X}_1$, which is not possible by definition of E.
   (c) For no X_i, $\bar{X}_i \rightsquigarrow \bar{Y}$. Then by Corollary 4.9, no $X_i \lhd^* Y$ is in S, so for all i, $\neg X_i \lhd^* Y \in S$. Now suppose $\bar{X}$ was not reachable from $\bar{Y}$. Then, again by Corollary 4.9, $\neg Y \lhd^* X \in S$. But by the Child rule, $Y \lhd^* X$ must be in S as well, which means that S would be inconsistent. So $\bar{X}$ must be reachable from $\bar{Y}$. But this means that the $\lhd^*$-edge from $\bar{X}$ to $\bar{Y}$ is a loop, which is impossible by the first point of this lemma.

4. Assume that there are nodes $\bar{X}$ and $\bar{Y}$ in G such that there are two different paths $e_1, \ldots, e_k$ and $e'_1, \ldots, e'_l$ from $\bar{X}$ to $\bar{Y}$. Assume further that $\bar{X}$ and $\bar{Y}$ are such that $e_k \neq e'_l$, and that k is minimal among all such paths. Let $\bar{Z}$ be the source of e'_1.

   $\bar{Y}$ has two incoming edges, e_k and e'_l. By Lemma 4.7, one of its parents is reachable from the other (by Lemma 4.8, reachability in G is the same as reachability in G'). Without loss of generality, let the source of e'_l be reachable from the source of e_k. So we can go to $\bar{Y}$ from the source of e_k on two different paths; either by going to the source of e'_l and then over e'_l, or by going over e_k. As k was chosen to be minimal, we can conclude that $k = 1$.

   If $l > 1$, there must be an i such that e_1 is an i-edge because E does not contain $\triangleleft^*$-edges that abbreviate paths of length > 1. In addition, e'_1 must be a $\triangleleft^*$-edge: it can neither be an i-edge, or else it would be the same as e_1, nor can it be a k-edge for a $k \neq i$; otherwise, there would be representatives X, X', Y, Z and a constructor f such that $X{=}X', X{:}f(\ldots, Y, \ldots){:}X'{:}f(\ldots, Z, \ldots) \in S$, where Y is the i-th argument and Z is the k-th argument, so by the Disj rule, $\neg Z \triangleleft^* Y \in S$, and hence, $\bar{Y}$ would not be reachable from $\bar{Z}$. But by the previous point of this proof, this cannot be true: G does not contain any nodes that have both an $\mathbb{N}$-child and a $\triangleleft^*$-child.

   So either there is only one path between $\bar{X}$ and $\bar{Y}$, or the second path has only length 1 as well; but even in G', there is only one edge between two nodes, so $e_k = e'_l$, in contradiction to our assumption.

5. Suppose $\bar{Z}$ had two incoming edges; then one of its parents would be reachable by the other, by Lemma 4.7. This means that there are two paths from one parent to $\bar{Z}$: either directly or via the other parent. This is in contradiction to (4).

□

5. Construct a satisfying tree structure and variable assignment for φ from G. The graph G need not be connected. We call G's connected components $G_1, \ldots, G_m$. The connected components are very tree-like, as the following lemma expresses.

Lemma 4.11. *Every connected component G_i of G has exactly one minimal element r_i with respect to the partial order $\rightsquigarrow_G$ on nodes, and there is exactly one path from r_i to every other node in G_i. We call r_i the* root *of G_i.*

Proof. As the number of nodes in G_i is finite and $\rightsquigarrow_G$ is a partial order, it is clear that minimal elements exist. To show that the minimal element is unique, we simply show the second result in the lemma, namely, that every other node in G_i is reachable from a minimal element. Given this, it follows that two minimal elements are reachable from each other; as G is cycle-free, they must be the same.

So now, we show that for any node $\bar{X}$ in G_i, there is a path from r_i to $\bar{X}$. G_i is connected; so between any two nodes in it, there is a cycle-free path in $(E^*(E^{-1})^*)^*$, where E^{-1} is the set of edges in E with their directions reversed. As we know from Lemma 4.10, there are no nodes in G that have two incoming edges; so in particular, all these paths must be in $(E^{-1})^*E^*$. But r_i has no incoming edges, otherwise it would not be minimal, so all paths from r_i to another node in G_i are in E^*, which is what we wanted to show. □

We will now extract a part of a satisfying tree structure from each of the G_k. Intuitively, we will build this tree structure by traversing G_k, starting at the root, and translating paths in G_k to paths in a tree. Whenever we meet a labeled node, we put the label in our tree structure and add the children. If we meet an unlabeled node that has dominance children, things are a bit more difficult. If there is only one child and there is a negated dominance constraint between a node that we can reach and the current node, we choose a unary constructor and add the dominance child as a child of this unary constructor. (In this way, we ensure the proper dominance.) If there is only one child, but no negated dominance requirement, we simply identify the two nodes. If there are multiple children, we choose a constructor of appropriate arity and add our children as the children of this constructor.

For this construction, we need to simulate that our signature contains at least one constructor of each signature. As we have assumed that we have at least one constructor g of arity $r \geq 2$ and at least one nullary constructor a in our signature, this is not difficult to do. First, we simulate a binary constructor by closing off the $r-2$ rightmost children of an occurrence of g with a; then we simulate a constructor of arbitrary arity of at least 2 by sequences of the binary constructor. Similarly, we can simulate a unary constructor by closing off the $r-1$ rightmost children of g instead of the $r-2$ rightmost ones. We have illustrated the construction in Fig. 4.1. For any trees $\sigma_1, \dots, \sigma_n$, we define the tree $f_n(\sigma_1, \dots, \sigma_n)$ as the tree obtained by plugging the σ_i into this construction in a left-to-right fashion. For our purposes, the "simulated constructors" f_n behave just like ordinary n-ary constructors, but we describe them a bit more formally anyway.

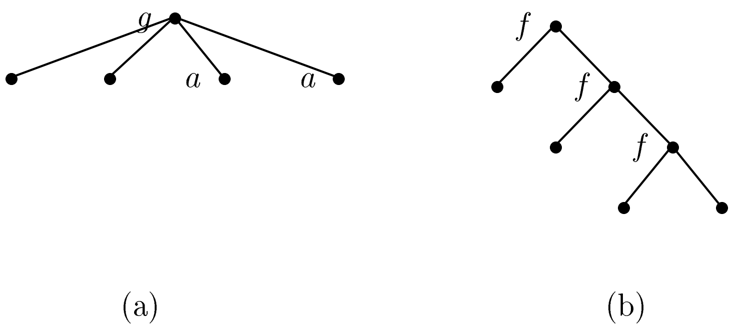


Figure 4.1: (a) Simulation of a binary constructor with a constructor g of arity $r \geq 2$. (b) Simulation of a 4-ary constructor with a binary constructor f.

If we define the path $p_n(i)$ as the path from the root of f_n to its i-th hole (into which children can be plugged), it can be described as follows:

$$p_n(i) = \begin{cases} 2^{n-1} & \text{if } i = n; \\ 2^{i-1} \cdot 1 & \text{otherwise.} \end{cases}$$

All the paths in a tree $f_n(\sigma_1, \ldots, \sigma_n)$ that do not start with one of these $p_n(i)$ are artifacts of the construction. We call the set of these paths $\mathcal{D}(f_n)$. For $n > 1$, this set has the form

$$\mathcal{D}(f_n) = \bigcup_{k=0}^{n-2} \{2^k, 2^k \cdot 3, \ldots 2^k \cdot r\};$$

for $n = 1$, it is even simpler:

$$\mathcal{D}(f_1) = \{2, \ldots, r\}.$$

Now we can formally define, for each $1 \leq k \leq m$, a function $\pi_k : V_k \to \mathbb{N}^*$, where V_k is the node set of G_k:

$$
\begin{array}{lll}
\pi_k(r_k) = \epsilon; & & \\
\pi_k(\bar{X}) = \pi, \bar{X}[i]\bar{Y} & \Rightarrow \quad \pi_k(\bar{Y}) = \pi \cdot i; & \\
\pi_k(\bar{X}) = \pi, \bar{X}[\triangleleft^*]\bar{Y} & \Rightarrow \quad \pi_k(\bar{Y}) = \pi \cdot p_1(1) & \text{if } \bar{Y} \text{ is the only child of } \bar{X} \text{ and there} \\
& & \text{is a node } \bar{Z} \text{ that is reachable from } \bar{X} \\
& & \text{and representatives } X, Z \text{ such that} \\
& & \neg Z \triangleleft^* X \in S; \\
\pi_k(\bar{X}) = \pi, \bar{X}[\triangleleft^*]\bar{Y} & \Rightarrow \quad \pi_k(\bar{Y}) = \pi & \text{if this is not the case and } \bar{Y} \text{ is the} \\
& & \text{only child of } \bar{X}; \\
\pi_k(\bar{X}) = \pi & \Rightarrow \quad \pi_k(\bar{X}_i) = \pi \cdot p_n(i) & \text{if } \bar{X} \text{ has } n > 1 \triangleleft^*\text{-children,} \\
& & \text{and } \bar{X}_i \text{ is the } i\text{-th } \triangleleft^*\text{-child.}
\end{array}
$$

Now we define trees σ_k that "satisfy" the connected components G_k in the following way.

$$
\begin{array}{rcl}
\mathcal{D}_k & = & \pi_k(V_k) \\
& \cup & \{\pi_k(\bar{X}) \cdot \mathcal{D}(f_n) \mid \bar{X} \text{ has } n > 1 \triangleleft^*\text{-children}\} \\
& \cup & \{\pi_k(\bar{X}) \cdot \mathcal{D}(f_1) \mid \bar{X} \text{ has exactly one } \triangleleft^*\text{-child, and there are a node } \bar{Z} \text{ that} \\
& & \text{is reachable from } \bar{X} \text{ and representatives } X, Z \text{ such that } \neg Z \triangleleft^* X \in S\} \\
\sigma_k(\pi) & = & \begin{cases} f & \text{if there is a } \bar{X} \in V_k \text{ such that } L_V(\bar{X}) = f \text{ and } \pi_k(\bar{X}) = \pi; \\ g & \text{if this is not the case and } \pi \text{ has exactly } r \text{ successors in } \mathcal{D}_k; \\ a & \text{if this is not the case and } \pi \text{ has no successors in } \mathcal{D}_k. \end{cases}
\end{array}
$$

Lemma 4.12. *For any k, σ_k is a tree.*

Proof. For one, every $\mathcal{D}_k$ is a finite tree domain, by construction. Furthermore, for every k and every path $\pi \in \mathcal{D}_k$, the arity of $\sigma_k(\pi)$ agrees with the number of children of π; this is trivial if the label is defined by the second or third branch of the definition of σ_k, and it follows from the structure of G_k if it is defined by the first branch. $\square$

Finally, we assemble the pieces by inserting them into a simulated m-ary constructor: Let $\sigma_1, \ldots, \sigma_m$ be the tree structures that we just extracted from the connected components $G_1, \ldots, G_m$, then our final tree is just $f_m(\sigma_1, \ldots, \sigma_m)$. The subtree corresponding to the i-th connected component is located below the path $p_m(i)$; so we can define a mapping $\pi : V \to \mathrm{Dom}(\sigma)$ as $\pi(\bar{X}) = p_m(i) \cdot \pi_i(\bar{X})$ if $\bar{X}$ is in G_i. We claim that $\mathcal{M}^\sigma$, together with the variable assignment $\alpha(X) = \pi(\bar{X})$, satisfies the original constraint.

6. Prove satisfiedness. For the proof that this construction really satisfies φ, we collect some observations about its structure in a lemma.

Lemma 4.13. *The tree σ and the mapping π that we have just constructed have the following properties:*

1. *If $\bar{X} \rightsquigarrow \bar{Y}$, then $\pi(\bar{X}) \leq \pi(\bar{Y})$.*
2. *If $\pi(\bar{X}) < \pi(\bar{Y})$, then $\bar{X} \rightsquigarrow \bar{Y}$.*
3. *If $\pi(\bar{X}) = \pi(\bar{Y})$, then there is a chain $\bar{X}_1[\lhd^*]\dots[\lhd^*]\bar{X}_r$ such that all of the nodes $\bar{X}_1, \dots, \overline{X_{r-1}}$ have only one outgoing edge and either $\bar{X}_1 = \bar{X}$ and $\bar{X}_r = \bar{Y}$ or vice versa.*
4. *If $\bar{X} \rightsquigarrow \bar{Y}$ and there are representatives X, Y such that $\neg Y \lhd^* X \in S$, then $\pi(\bar{X}) < \pi(\bar{Y})$.*
5. *If the node $\bar{X}$ is labeled with f and has the $\mathbb{N}$-children $\bar{X}_1, \dots, \bar{X}_n$, then $\sigma(\pi(\bar{X})) = f$ and for all $1 \leq i \leq n$, $\pi(\bar{X}_i) = \pi(\bar{X}) \cdot i \in \mathcal{D}^\sigma$, but for all $i > n$, $\pi(\bar{X}) \cdot i \notin \mathrm{Dom}(\sigma)$.*

Lemma 4.14. *$(\mathcal{M}^\sigma, \alpha)$ satisfies φ.*

Proof. We show the stronger result that for every atomic constraint $C \in S$, $(\mathcal{M}, \alpha)$ satisfies C by considering all cases for C.

<u>$C = X \lhd^* Y$.</u> By Corollary 4.9, we know that $\bar{X} \rightsquigarrow_G \bar{Y}$; hence, they are in the same connected component, and we know that $\pi(\bar{X})$ is a prefix of $\pi(\bar{Y})$, which is what we wanted to show.

<u>$C = X{:}f(X_1, \dots, X_n)$.</u> By construction of G, $L_V(\bar{X}) = f$, so $\sigma(\pi(\bar{X})) = f$. Furthermore, for any $1 \leq i \leq n$, we know that $\bar{X}[i]\bar{X}_i$ in G, and hence, that $\pi(\bar{X}_i) = \pi(\bar{X}) \cdot i$.

<u>$C = \neg X \lhd^* Y$.</u> Assume that $\pi(\bar{X}) \leq \pi(\bar{Y})$; we derive a contradiction. We know that $\bar{X}$ and $\bar{Y}$ are in the same connected component of G, say in G_k. Let us distinguish the two possible cases for the relation of $\pi(\bar{X})$ and $\pi(\bar{Y})$.

1. $\pi(\bar{X}) < \pi(\bar{Y})$; then we know that $\bar{Y}$ is reachable from $\bar{X}$. Hence, by Corollary 4.9, there is also a constraint $X \lhd^* Y$ in S. This is not possible, as in this case, we could have derived an inconsistency with Clash3.

2. $\pi(\bar{X}) = \pi(\bar{Y})$. There must be a chain of nodes $\bar{X}_1, \ldots, \bar{X}_s$ such that $\bar{X}_1[\triangleleft^*] \ldots [\triangleleft^*]\bar{X}_s$, where every node is the single child of its predecessor and either $\bar{X}_1 = \bar{X}$ and $\bar{X}_s = \bar{Y}$, or vice versa. In the first case, we can again conclude that $\bar{Y}$ is reachable from $\bar{X}$ and can proceed as above. In the second case, we know by the above lemma that $\pi(\bar{Y}) < \pi(\bar{X})$, a contradiction.

□

4.1.4 Larger logical languages

The algorithm we have defined solves dominance constraints that are pure conjunctions of labeling, dominance, and negated dominance constraints. We now extend it to allow additionally disjunctions and, later, negations in formulae over these atomic constraints.

1. Disjunctions. First, let us consider the language that contains conjunctions and disjunctions, but no negations (other than the negated dominance constraints, which we still consider atomic for the moment). To decide the satisfiability of such a formula φ, we first apply a function A that nondeterministically builds a conjunction from φ; then we apply the algorithm from Section 4.1.1 to this conjunction.

A is a recursive function that maps a formula φ to a conjunction of atomic constraints. We define it as follows:

$$\begin{array}{rcl} A(\varphi_1 \wedge \varphi_2) & = & A(\varphi_1) \wedge A(\varphi_2) \\ A(\varphi_1 \vee \varphi_2) & = & A(\varphi_i) \\ A(\varphi_0) & = & \varphi_0, \end{array}$$

where i is chosen nondeterministically to be either 1 or 2, and φ_0 is atomic. Clearly, A runs in nondeterministic linear time. The key result about the function is the following lemma.

Lemma 4.15. *A pair $(\mathcal{M}, \alpha)$ satisfies φ iff it satisfies one of the possible results of $A(\varphi)$.*

Proof. By structural induction on φ.

<u>Atoms.</u> trivial.

$\underline{\varphi = \varphi_1 \wedge \varphi_2.}$ Consider a pair $(\mathcal{M}, \alpha)$. By induction hypothesis, this pair satisfies φ_1 iff it satisfies one result $A(\varphi_1)$, and it satisfies φ_2 iff it satisfies one result $A(\varphi_2)$. Hence, it satisfies φ iff it satisfies one possible value of $A(\varphi)$.

$\underline{\varphi = \varphi_1 \vee \varphi_2.}$ By induction hypothesis, a pair $(\mathcal{M}, \alpha)$ satisfies φ_1 iff it satisfies a possible result of $A(\varphi_1)$; call this particular result φ_1'. Likewise, $(\mathcal{M}, \alpha)$ satisfies φ_2 iff it satisfies the possible result φ_2' of $A(\varphi_2)$. Now φ is satisfied by $(\mathcal{M}, \alpha)$ iff either φ_1 or φ_2 is, which is equivalent to the satisfaction of either φ_1' or φ_2'. Hence, there is a choice such that $A(\varphi)$ is satisfied; and conversely, satisfiedness of φ_i' implies satisfiedness of φ_i, which implies satisfiedness of φ.

□

We decide satisfiability of a formula φ by first running A on it and then feeding the result into the algorithm from the previous sections. Again, we call φ *consistent* if there is a result of $A(\varphi)$ that has a saturation which does not contain **false**. For this, the following result holds.

Proposition 4.16. *A formula φ of disjunctions and conjunctions over the dominance constraints is satisfiable iff it is consistent.*

Proof. First, let φ be satisfiable. We know by Lemma 4.15 that there is a result φ' of running A on φ that is satisfiable as well. By Proposition 4.1, this implies that φ' can be saturated in such a way that the saturation does not contain **false**. So φ is consistent.

Conversely, let φ be consistent; then there is a result φ' of running A on φ that has a saturation that does not contain **false**. By Theorem 4.2, φ' is satisfiable; but by Lemma 4.15, this means that φ is satisfiable, as well. □

2. Negations. Now we add negation to the list of connectives that we are allowed to use. So far, the only instance of negation we could use was as negated dominance constraints, but we considered those as atomic constraints. Now, we lift this restriction and return to the original definition of atomic constraints. As we are testing for satisfiability of formulae, it does not matter if we additionally allow positive occurrences of existential quantifiers; by renaming variables, we can always get an equivalent quantifier-free formula. The encoding that we present does not work for the general case; we must require a finite signature. But for practical purposes, this restriction does not hurt at all.

Clearly, we can reduce a formula that contains negations to an equivalent formula where the only negations are single negations of atomic formulae in linear time. In addition, our original algorithm can handle negated dominance constraints. So the only difficulty is to extend the algorithm to handle negated labeling constraints.

We do this by simply replacing all negated labeling constraints in φ by equivalent formulae that do not contain negated labeling constraints. To be precise, we replace a constraint $\neg X{:}f(X_1, \dots, X_n)$ by

$$\left(\bigvee_{\substack{g \neq f \\ m = \mathsf{ar}(g)}} X{:}g(X'_1, \dots, X'_m) \right) \vee \; (X{:}f(X''_1, \dots, X''_n) \wedge (\neg X''_1{=}X_1 \vee \dots \vee \neg X''_n{=}X_n)),$$

where the X'_i and X''_i are fresh variables. Now all we need to show is that a negated labeling constraint φ is satisfied by a pair $(\mathcal{M}, \alpha)$ iff its encoding φ' is satisfied by $(\mathcal{M}, \alpha')$, where α' agrees with α on α's domain.

As this is quite simple, we only sketch the proof. By definition of the semantics, φ is satisfied by $(\mathcal{M}^\sigma, \alpha)$ iff either $\sigma(\alpha(X)) \neq f$ or there is an i such that $\alpha(X_i) \neq \alpha(X) \cdot i$. The first case is equivalent with saying that $\sigma(\alpha(X))$ is a constructor different than f; this is exactly the situation where the first disjunct of φ' is satisfied. The second case is exactly the situation where the first conjunct of the second disjunct of φ' and one of the disjuncts of the other conjunct is satisfied. Hence, φ and φ' are equivalent.

4.2 NP-Hardness

As we have seen in the previous section, a wide selection of propositional languages over the dominance constraints has satisfiability problems that can be solved in NP time. In this section, we supplement this result by showing that even for purely conjunctive dominance constraints, this problem is NP-hard. To this end, we encode propositional formulae in 3-CNF as satisfaction equivalent dominance constraints.

We proceed in two steps. First, we go through the encoding of a specific example to make the general ideas and intuitions clear (Section 4.2.1). In the second step, we present a systematic encoding and prove its correctness (Section 4.2.2). For the casual reader, it will be much more worthwhile to read the first part and the first page or so of the second than the rest of the second part, as the correctness proof is rather tedious and does not provide any new insights.

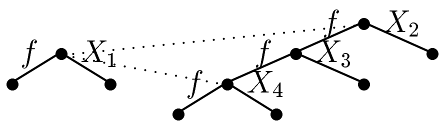
4.2.1 An Example

To prove the NP-hardness of the satisfiability problem of dominance constraints, we reduce the problem 3SAT, which is well known to be NP-complete, to it. 3SAT is the satisfiability problem of propositional formulae in conjunctive normal form, where every conjunct is a disjunction of exactly three literals. This special type of conjunctive normal form is called 3-CNF. To fix names and avoid confusion, we define the syntax of 3-CNF formulae as follows:

$$\begin{array}{lrcl} \text{formulae} & \psi & = & C_1 \wedge \ldots \wedge C_m \\ \text{clauses} & C_i & = & L_{i1} \vee L_{i2} \vee L_{i3} \\ \text{literals} & L_{ij} & = & X_k \text{ or } \bar{X}_k. \end{array}$$

We write $\bar{X}_i$ to mean the negation of the variable X_i; we assume that the variables that occur in ψ are $X_1, \ldots, X_n$. As an alternative representation of a literal, we will sometimes write X^t, where X is a variable and $t \in \{\mathbf{true}, \mathbf{false}\}$ is a truth value. We take $X^{\mathbf{true}}$ to mean X and $X^{\mathbf{false}}$ to mean $\bar{X}$; so for any t, $t^t = \mathbf{true}$.

The central construction that we use to model clauses even in the conjunctive language of dominance constraints is the *dominance triangle*. This is a subconstraint whose graph looks like this:



If $(\mathcal{M}^\sigma, \alpha)$ is a solution of this constraint, α must map exactly one of the variables X_2, X_3, X_4 to the same node as X_1 because $\alpha(X_2)$ must be a prefix of $\alpha(X_1)$, which in turn must be a prefix of $\alpha(X_4)$. We can exploit this effect to model three-way disjunction – just what we need to encode a clause.

As an example of a 3-CNF formula, let us consider the formula ψ in (4.1).

(4.1) $(X_1 \vee \bar{X}_2 \vee X_3) \wedge (\bar{X}_1 \vee X_2 \vee X_3)$

The constraint graph in Fig. 4.2 represents the dominance constraint φ which is the encoding of ψ. We are drawing the constraint graph in a somewhat simplified manner by leaving away all labels of inner nodes and most variable names; all inner nodes should be read as being labeled with a fixed binary constructor f. The signature we use is $\{f{:}2, \mathbf{true}{:}0, \mathbf{false}{:}0\}$.

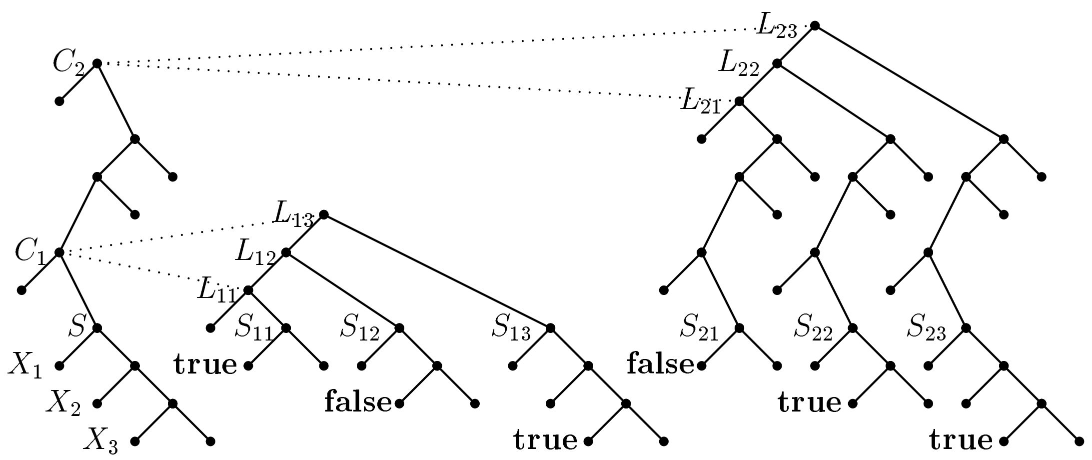


Figure 4.2: An encoding of $(X_1 \vee \bar{X}_2 \vee X_3) \wedge (\bar{X}_1 \vee X_2 \vee X_3)$ as a dominance constraint.

We claim that φ is satisfiable iff ψ is satisfiable. To understand this, let us take a closer look at the various parts of the diagram.

The lower left part of the graph (below the node S) holds a variable assignment: For each of the variables X_k that occur in ψ, there is one node. In a solution, each of these nodes must be labeled with either **true** or **false**, but not both.

We can view ψ as a constraint on admissibility of variable assignments by calling a variable assignment admissible if it satisfies ψ. Each clause imposes such a restriction on the variable assignments; within a clause, we have a choice between three different options for satisfying the constraint.

The dominance constraint expresses the very same thing.

Because it is part of a dominance triangle, C_1 must be identified with one of the L_{1j} in any solution. But once we have identified C_1 with one of the three L_{1j}, we have decided which of the clause C_1's literals we want to satisfy: The right daughter of the chosen L_{1j} node is identified with S, some entries in the variable assignment subtree may be skipped, and then a value restriction is imposed on one of the variables X_k. In the example, L_{11} forces the label of X_1 to be **true**; L_{12} forces the label of X_2 to be **false**; etc. We have imposed a constraint on the variable assignment that is obviously equivalent to that imposed by the first clause in ψ.

The second clause is represented similarly: The dominance triangle between L_{21}, C_2, and L_{23} allows a choice which literal of this clause we want to satisfy. Whichever literal we pick, its right daughter first skips the entry for C_1 (identifying S with one of the S_{2j}), and then it selects a variable entry and imposes a value constraint.

An important detail of this encoding is the presence of more nodes than just the C_i in the main branch of the graph (for example, there are two additional nodes between C_1 and C_2 in the constraint graph). These nodes are "rubbish dumps" which can be used to store unneeded material in such a way that it won't interfere with anything else. Suppose we identified C_1 and L_{12} in a solution of φ. Then L_{11} will be identified with the left daughter of C_1, and L_{13} will be identified with the mother of C_1. Clearly, we do not want any other part of the constraint to say anything about the right child of C_1's mother because otherwise, we might run into unnecessarily unsatisfiable dominance constraints. This means that above each C_i node, we need two additional nodes to drop material from the identification process. We do not need any additional nodes below the C_i because the unnecessary material is then a left child of the selected literal node and can safely be stored below C_i's left daughter.

To summarize, the encoding of 3-CNF formulae as dominance constraints consists of the following parts: a) a dedicated subtree to hold variable assignments; b) for each clause, a dominance triangle to allow the selection of literals; c) for each literal, a subtree that skips lower clauses, selects a variable in the variable assignment section, and imposes a value restriction on this variable. It is intuitively clear that ψ and φ are satisfaction equivalent; unsatisfiability of ψ means that one variable would necessarily have to take two values at once, and in such a situation, the labeling requirements on the representing node in φ would clash as well.

4.2.2 NP-Completeness of Dominance Constraints

Now that we have made the intuition clear, we define a systematic encoding and prove its correctness.

In a simple way, we build the constraint graph that corresponds to φ from the "building blocks" in Fig. 4.3. Larger building blocks can include several copies of smaller building blocks. For most of the building blocks, we have specified with arrows an upper and a lower attachment site where it can be composed with other blocks by identifying the two attachment sites; we write such compositions as trees whose labels are the two building blocks. Furthermore, we take a block with a superscript s (such as Skip with superscript $i-1$ in X_i) to mean s-fold composition of building blocks. So we want the X_i block to consist of $i-1$ occurrences of

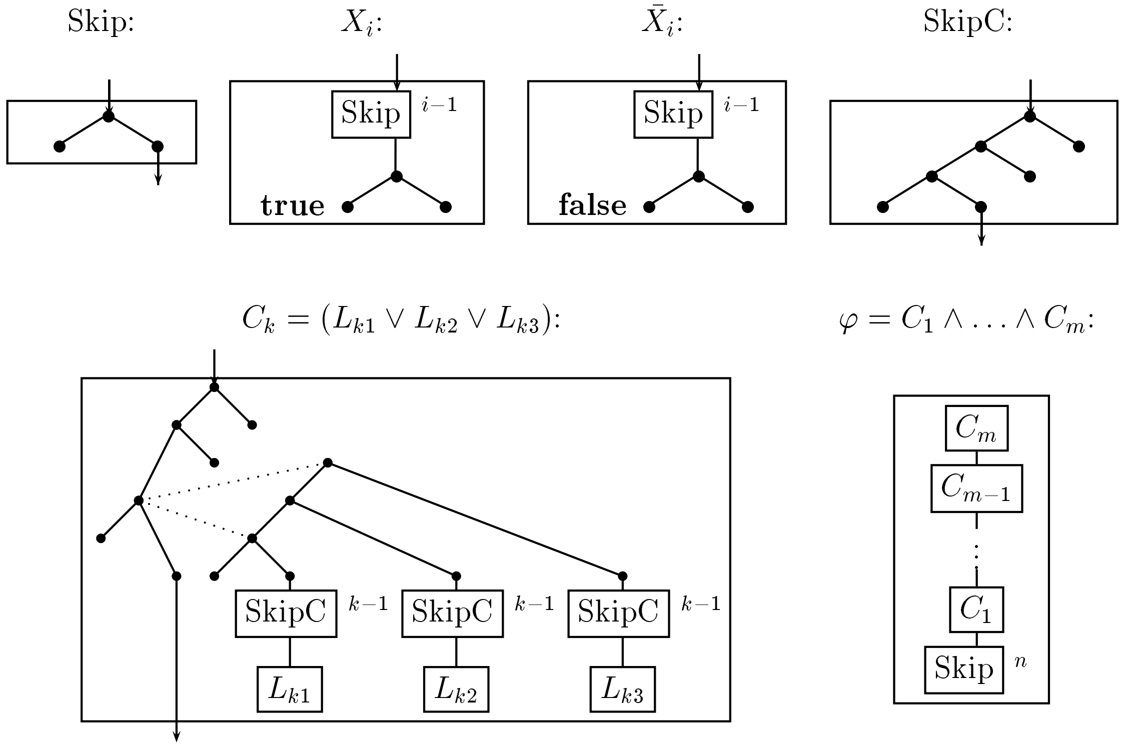


Figure 4.3: Building blocks for the encoding of 3SAT as a dominance constraint.

Skip blocks and two additional nodes that are immediate children of the lowest attachment site in the sequence of Skips, the left of which is labeled with **true**.

It is easy to see that the constraint graph from the previous section was built according to this scheme. The overall structure consists of m entries for the clauses, below which n Skip blocks hold a variable assignment. Within each C_i block, there is a dominance triangle that allows the selection of a literal, together with a sufficient number of SkipC blocks to skip lower clauses. Finally, the encoding of a literal selects a propositional variable and imposes a value restriction.

After the intuitive explanation in the previous section, it should be rather straightforward to see how the encoding works. Below, we prove more formally that it is correct, but the proof provides no further insights and can be safely skipped. Even so, we may not always spell out every single detail.

Soundness. First, we show a soundness result: If ψ is satisfiable, φ is satisfiable as well. (We will show the corresponding completeness result below.) To this end, we assume that ψ is satisfied by a valuation V and explicitly construct a pair $(\mathcal{M}, \alpha)$ of a tree structure and a variable assignment that satisfies φ. As V satisfies ψ, every clause C_k of ψ contains one literal L_{kj_k} that is satisfied by V. Even if the result looks a bit intimidating, it is helpful to write down the entire constraint explicitly instead of relying on the constraint graph, which we do now. Throughout the proof, we will say that the variable involved in the literal L_{kl} is $X_{h_{kl}}$ and its polarity is t_{kl}; in other words, $L_{kl} = X_{h_{kl}}^{t_{kl}}$.

First of all, let us fix a set of variables for use in the constraint representation of φ. We will use five different families of variables, VA, C, SC, LI, and SK; a variable of a family A has the name X^A and typically several additional indices. Each family of variables corresponds roughly to one of the building blocks; indices in superscript indicate different instances of the same building block, whereas indices in subscript indicate different positions in an instance of a building block. More precisely, the variables we use are the following:

variables	ranges	encodings of ...
$X^{C,k}_{1,\ldots,14}$	$1 \leq k \leq m$	C_k
$X^{VA,l}_{1,\ldots,3}$	$1 \leq l \leq n$	variable assignment
$X^{SC,k,l,i}_{1,\ldots,7}$	$1 \leq k \leq m, l = 1,2,3, 1 \leq i \leq k-1$	i-th SkipC block for literal L_{kl}
$X^{LI,k,l}_{1,\ldots,3}$	$1 \leq k \leq m, l = 1,2,3$	lower part of block for L_{kl}
$X^{SK,k,l,i}_{1,\ldots,3}$	$1 \leq k \leq m, l = 1,2,3, 1 \leq i \leq h_{kl}-1$	i-th Skip in L_{kl} block.

Using these variables, the encoding φ can be written as follows:

$$
\begin{array}{ll}
& \bigwedge_{l=1}^{n} X_1^{VA,l}{:}f(X_2^{VA,l}, X_3^{VA,l}) \wedge \bigwedge_{l=2}^{n} X_1^{VA,l}{=}X_3^{VA,l-1} \\
\wedge & \bigwedge_{k=1}^{m} (X_1^{C,k}{:}f(X_2^{C,k}, X_3^{C,k}) \wedge X_2^{C,k}{:}f(X_4^{C,k}, X_5^{C,k}) \wedge X_4^{C,k}{:}f(X_6^{C,k}, X_7^{C,k})) \\
\wedge & \bigwedge_{k=2}^{m} X_7^{C,k}{=}X_1^{C,k-1} \wedge X_7^{C,1}{=}X_1^{VA,1} \\
\wedge & \bigwedge_{k=1}^{m} (X_8^{C,k}{:}f(X_9^{C,k}, X_{10}^{C,k}) \wedge X_9^{C,k}{:}f(X_{11}^{C,k}, X_{12}^{C,k}) \wedge X_{11}^{C,k}{:}f(X_{13}^{C,j}, X_{14}^{C,k}) \\
& \wedge X_{14}^{C,k}{=}X_1^{SC,k,1,1} \wedge X_{12}^{C,k}{=}X_1^{SC,k,2,1} \wedge X_{10}^{C,k}{=}X_1^{SC,k,3,1} \\
& \wedge X_8^{C,k} \lhd^* X_4^{C,k} \wedge X_4^{C,k} \lhd^* X_{11}^{C,k}) \\
\wedge & \bigwedge_{k=1}^{m} \bigwedge_{l=1}^{3} \bigwedge_{i=1}^{k-1} (X_1^{SC,k,l,i}{:}f(X_2^{SC,k,l,i}, X_3^{SC,k,l,i}) \wedge X_2^{SC,k,l,i}{:}f(X_4^{SC,k,l,i}, X_5^{SC,k,l,i}) \\
& \wedge X_4^{SC,k,l,i}{:}f(X_6^{SC,k,l,i}, X_7^{SC,k,l,i})) \\
\wedge & \bigwedge_{k=1}^{m} \bigwedge_{l=1}^{3} \bigwedge_{i=2}^{k-1} X_1^{SC,k,l,i}{=}X_7^{SC,k,l,i-1} \\
\wedge & \bigwedge_{k,l} \big(\bigwedge_{i=1}^{h_{kl}-1} X_1^{SK,k,l,i}{:}f(X_2^{SK,k,l,i}, X_3^{SK,k,l,i}) \\
& \wedge \bigwedge_{i=2}^{h_{kl}-1} X_1^{SK,k,l,i}{=}X_3^{SK,k,l,i-1} \wedge X_1^{SK,k,l,1}{=}X_7^{SC,k,l,k-1} \\
& \wedge X_1^{LI,k,l}{=}X_3^{SK,k,l,h_{kl}-1} \wedge X_1^{LI,k,l}{:}f(X_2^{LI,k,l}, X_3^{LI,k,l}) \wedge X_2^{LI,k,l}{:}t_{kl}\big)
\end{array}
$$

Now we define a satisfying tree structure and variable assignment. First, we define auxiliary trees $S(k,j,t)$ inductively in the following way:

$$\begin{array}{rcl} S(1,1,t) & = & f(t,\mathbf{false}) \\ S(1,j,t) & = & f(\mathbf{false},S(1,j-1,t)) \\ S(k,j,t) & = & f(f(f(\mathbf{false},S(k-1,j,t)),\mathbf{false}),\mathbf{false}). \end{array}$$

(Intuitively, $S(k,j,t)$ consists of first $k-1$ SkipC blocks, then $j-1$ Skip blocks, then $f(t,\mathbf{false})$, where all open leaves have been labeled with **false**. This is just the structure around the L_{kl} blocks in the diagrams.)

Now we inductively define trees $\sigma_0,\sigma_1,\ldots,\sigma_m$ in the following way:

$$\begin{array}{rcl} \sigma_0 & = & f(V(X_1),f(V(X_2),\ldots,f(V(X_n,\mathbf{false}))\ldots) \\ \sigma_k & = & f(f(f(D_{k1},\sigma_{k-1}),D_{k2}),D_{k3}), \end{array}$$

where the definition of the subtrees D_{ki} depends on the index j_k of the satisfied literal:

$$D_{k1} = \begin{cases} \mathbf{false} & \text{if } j_k = 1 \\ f(\mathbf{false},S(k,h_{k1},t_{k1})) & \text{if } j_k = 2 \\ f(f(\mathbf{false},S(k,h_{k1},t_{k1})),S(k,h_{k2},t_{k2})) & \text{if } j_k = 3; \end{cases}$$

$$D_{k2} = \begin{cases} S(k,h_{k2},t_{k2}) & \text{if } j_k = 1 \\ S(k,h_{k3},t_{k3}) & \text{if } j_k = 2 \\ \mathbf{false} & \text{if } j_k = 3; \end{cases}$$

$$D_{k3} = \begin{cases} S(k,h_{k3},t_{k3}) & \text{if } j_k = 1 \\ \mathbf{false} & \text{otherwise.} \end{cases}$$

Let $\sigma=\sigma_m$, and let $\mathcal{M}=\mathcal{M}^\sigma$. The following lemma holds about these trees.

Lemma 4.17. *1. For any k,j,t, the domain of $S(k,j,t)$ is*

$$\begin{array}{rl} & \bigcup_{i'=0}^{k-2}(112)^{i'}\cdot\{\epsilon,1,2,11,12,111\} \\ \cup & (112)^{k-1}\cdot\bigcup_{j'=0}^{j-1}\{2^{j'},2^{j'}1\} \\ \cup & \{(112)^{k-1}\cdot 2^j\}. \end{array}$$

2. The domain of σ_0 is

$$\bigcup_{j=0}^{n}\{2^j,2^j1\}\cup\{2^{n+1}\}.$$

3. *For any* k, j, t,
$$S(k,j,t)((112)^{k-1} \cdot 2^{j-1} \cdot 1) = t.$$

4. *For any* k, l,
$$\sigma_0(2^{h_{kl}-1} \cdot 1) = t_{kl}.$$

Proof. 1. We proceed along the various stages of the definition. Clearly, the domain of $S(1,1,t)$ is $\{\epsilon, 1, 2\}$.

For $j > 1$, the following recursive relation about the domain of $S(1,j,t)$ holds:
$$\mathrm{Dom}(S(1,j,t)) = \{\epsilon, 1\} \cup 2 \cdot \mathrm{Dom}(S(1,j-1,t)).$$
By induction over j, it follows easily that
$$\mathrm{Dom}(S(1,j,t)) = \bigcup_{i=0}^{j-1} \{2^i, 2^i 1\} \cup \{2^j\}.$$
Similarly, the following recursive relation about the domains of the $S(k,j,t)$ (for $k > 1$) is obvious:
$$\mathrm{Dom}(S(k,j,t)) = \{\epsilon, 1, 2, 11, 12, 111\} \cup 112 \cdot \mathrm{Dom}(S(k-1,j,t)).$$
Another induction (this time over k) proves the equation in the lemma. As the base case of this induction, we use the general equation for the domain of $S(1,j,t)$.

3. Follows by induction. The claim is obvious for $j = k = 1$; we can prove it for arbitrary j by induction (as in the first claim), and then by another induction, for arbitrary k.

2.,4. Obvious.

□

Now we define a variable assignment α by defining the path that is assigned to each variable.

var	$\alpha(var)$	var	$\alpha(var)$
$X_1^{C,k}$	$(112)^{m-k}$	$X_8^{C,k}$	$(112)^{m-k} \cdot 1^{j_k-1}$
$X_2^{C,k}$	$(112)^{m-k} \cdot 1$	$X_9^{C,k}$	$(112)^{m-k} \cdot 1^{j_k-1} \cdot 1$
$X_3^{C,k}$	$(112)^{m-k} \cdot 2$	$X_{10}^{C,k}$	$(112)^{m-k} \cdot 1^{j_k-1} \cdot 2$
$X_4^{C,k}$	$(112)^{m-k} \cdot 11$	$X_{11}^{C,k}$	$(112)^{m-k} \cdot 1^{j_k-1} \cdot 11$
$X_5^{C,k}$	$(112)^{m-k} \cdot 12$	$X_{12}^{C,k}$	$(112)^{m-k} \cdot 1^{j_k-1} \cdot 12$
$X_6^{C,k}$	$(112)^{m-k} \cdot 111$	$X_{13}^{C,k}$	$(112)^{m-k} \cdot 1^{j_k-1} \cdot 111$
$X_7^{C,k}$	$(112)^{m-k} \cdot 112$	$X_{14}^{C,k}$	$(112)^{m-k} \cdot 1^{j_k-1} \cdot 112$
$X_1^{SC,k,l,i}$	$\alpha(X_8^{C,k}) \cdot 1^{3-l} \cdot 2 \cdot (112)^{i-1}$	$X_1^{VA,l}$	$(112)^m \cdot 2^{l-1}$
$X_2^{SC,k,l,i}$	$\alpha(X_8^{C,k}) \cdot 1^{3-l} \cdot 2 \cdot (112)^{i-1} \cdot 1$	$X_2^{VA,l}$	$(112)^m \cdot 2^{l-1} \cdot 1$
$X_3^{SC,k,l,i}$	$\alpha(X_8^{C,k}) \cdot 1^{3-l} \cdot 2 \cdot (112)^{i-1} \cdot 2$	$X_3^{VA,l}$	$(112)^m \cdot 2^{l-1} \cdot 2$
$X_4^{SC,k,l,i}$	$\alpha(X_8^{C,k}) \cdot 1^{3-l} \cdot 2 \cdot (112)^{i-1} \cdot 11$	$X_1^{SK,k,l,i}$	$\alpha(X_7^{SC,k,l,k-1}) \cdot 2^{i-1}$
$X_5^{SC,k,l,i}$	$\alpha(X_8^{C,k}) \cdot 1^{3-l} \cdot 2 \cdot (112)^{i-1} \cdot 12$	$X_2^{SK,k,l,i}$	$\alpha(X_7^{SC,k,l,k-1}) \cdot 2^{i-1} \cdot 1$
$X_6^{SC,k,l,i}$	$\alpha(X_8^{C,k}) \cdot 1^{3-l} \cdot 2 \cdot (112)^{i-1} \cdot 111$	$X_3^{SK,k,l,i}$	$\alpha(X_7^{SC,k,l,k-1}) \cdot 2^{i-1} \cdot 2$
$X_7^{SC,k,l,i}$	$\alpha(X_8^{C,k}) \cdot 1^{3-l} \cdot 2 \cdot (112)^{i-1} \cdot 112$		
$X_1^{LI,k,l}$	$\alpha(X_3^{SK,k,l,h_{kl}-1})$		
$X_2^{LI,k,l}$	$\alpha(X_3^{SK,k,l,h_{kl}-1}) \cdot 1$		
$X_3^{LI,k,l}$	$\alpha(X_3^{SK,k,l,h_{kl}-1}) \cdot 2$		

We claim that $(\mathcal{M}, \alpha)$ satisfies φ, and we show this by verifying that every single conjunct is satisfied. But first, we must verify that every path that α assigns to a variable really exists in $\mathcal{M}$.

Lemma 4.18. *1. For every $0 \leq r \leq m$, $\sigma.(112)^r = \sigma_{m-r}$.*

2. For every variable X that appears in φ, σ is defined on $\alpha(X)$.

Proof. 1. If k is an arbitrary number between 1 and m, we know by definition that $\sigma_k.112 = \sigma_{k-1}$. The claim follows by induction over r.

2. We prove the claim separately for each family of variables. We write $X^{F,\cdots}$ for an arbitrary variable $X_i^{F,\cdots}$ of the family F.

<u>$X^{VA,l}$.</u> By the first part of this lemma, all the $\sigma.\alpha(X^{VA,l})$ are defined iff all the $\sigma_0.(2^{l-1} \cdot \{\epsilon, 1, 2\})$ are. But as $l \leq n$, all of these paths are defined by Lemma 4.17.

<u>$X^{C,k}$.</u> The definedness of the paths assigned to $X_1^{C,k}, \ldots, X_7^{C,k}$ follows from the first part of the lemma and the definition of σ_k. For the other variables $X^{C,k}$, we distinguish cases by the value of j_k and verify the definedness of the paths $1^{j_k-1} \cdot \{\epsilon, 1, 2, 11, 12, 111, 112\}$ in σ_k.

$\underline{j_k = 1.}$ Obvious; we don't even have to look inside the D_{ki}.

$\underline{j_k = 2.}$ Definedness of most paths is obvious without looking inside the D_{ki}. The only interesting cases are the paths 1111 and 1112, which are defined iff the paths 1 and 2 are defined in D_{k1}, which is the case for $j_k = 2$.

$\underline{j_k = 3.}$ The interesting cases are the four paths $1111, 1112, 11111, 11112$. They are defined in σ_k iff the paths $1, 2, 11, 12$ are defined in D_{k1}, which they are for $j_k = 3$.

$\underline{X^{SC,k,l,i}.}$ The paths that are assigned to these and the next two families of variables all start with $p(m,k,l) = (112)^{m-k} \cdot 1^{j_k-l+1} \cdot 2$. By inspection of the structure of the σ_k and D_{ki}, we notice that

$$\sigma.p(m,k,l) = \begin{cases} \sigma_{k-1} & \text{if } j_k = l; \\ S(k, h_{kl}, t_{kl}) & \text{otherwise.} \end{cases}$$

Now, suppose that $j_k = l$. Then the paths assigned to the variable $X^{SC,k,l,i}$ are defined in σ iff the paths $\epsilon, 1, 2, 11, 12, 111, 112$ are defined in σ_{k-i}, by the above equation for $\sigma.p(m,k,l)$ and the first part of this lemma; clearly, this is true. On the other hand, if $j_k \neq l$, definedness of these paths follows from Lemma 4.17.

$\underline{X^{SK,k,l,i}.}$ First, consider the case of $j_k = l$. Then by the above result about $p(m,k,l)$ and the first part of this lemma, the $\sigma.\alpha(X^{SK,k,l,i})$ are defined iff all paths $2^{i-1} \cdot \{\epsilon, 1, 2\}$ are defined in σ_0. But by assumption, $i \leq n$, so by Lemma 4.17, this is true.

On the other hand, suppose $j_k \neq l$. Then the path is defined iff all paths $2^{i-1} \cdot \{\epsilon, 1, 2\}$ are defined in $S(k, h_{kl}, t_{kl})$; as $i \leq h_{kl} - 1$, this is true by Lemma 4.17.

$\underline{X^{LI,k,l}.}$ Analogous to the previous case.

□

An immediate consequence of the definition of α and Lemma 4.18 is that all binary labeling constraints in φ are satisfied by $(\mathcal{M}, \alpha)$: It is easily verified that all variables that appear as parents in a binary labeling constraint denote internal nodes, and hence, must be labeled with f. To prove that $(\mathcal{M}, \alpha)$ satisfies φ, it remains to verify the equivalence (=), dominance, and unary labeling constraints in φ.

By definition, an equivalence constraint $X{=}Y$ is satisfied by $(\mathcal{M}, \alpha)$ iff $\alpha(X) = \alpha(Y)$. α satisfies this condition, as is easily verified for every equivalence constraint in φ. For dominance constraints $X \lhd^* Y$, we require that $\alpha(X) \leq \alpha(Y)$; for the dominance constraints in φ, this is true, as

$$\begin{array}{rcl} \alpha(X_8^{C,k}) & = & (112)^{m-k} \cdot 1^{j_k - 1} \\ \alpha(X_4^{C,k}) & = & (112)^{m-k} \cdot 11 \\ \alpha(X_{11}^{C,k}) & = & (112)^{m-k} \cdot 1^{j_k-1} \cdot 11, \end{array}$$

and $j_k - 1 \leq 2$. Finally, consider the unary labeling constraints; we need to show that for all k, l,

$$\sigma(\pi) = t_{kl},$$

where $\pi = \alpha(X_2^{LI,k,l}) = p(m,k,l) \cdot (112)^{k-1} \cdot 2^{h_{kl}-1} \cdot 1$. This is equivalent to proving that $(\sigma.p(m,k,l))((112)^{k-1} 2^{h_{kl}-1} 1) = t_{kl}$, which we do for both possible cases of the relation of j_k and l:

$j_k = l$. As we have seen in the proof of Lemma 4.18, $\sigma.p(m,k,l) = \sigma_{k-1}$, so $\sigma.(p(m,k,l) \cdot (112)^{k-1}) = \sigma_0$. But as Lemma 4.17 expresses, the label of the node selected in σ_0 by the path $2^{h_{kl}-1}1$ is t_{kl}.

$j_k \neq l$. We know that $\sigma.p(m,k,l) = S(k, h_{kl}, t_{kl})$. By Lemma 4.17, the rest of the path selects a subtree with root label t_{kl} in this tree.

Hence, all atomic conjuncts of φ are satisfied, which completes the proof of soundness.

Completeness. We now show that if φ is satisfiable, ψ is as well. For this proof, we proceed in three steps:

1. There is a solution of a slightly restricted form that essentially looks like the σ_k's we defined in the soundness proof. We call this solution $(\mathcal{M}^\sigma, \alpha)$.

2. For every $1 \leq k \leq m$, there is a $j_k \in \{1,2,3\}$ such that the j_k-th literal node (i.e., $X_8^{C,k}$, $X_9^{C,k}$, or $X_{11}^{C,k}$) is identified with $X_4^{C,k}$. Let $L_{kj_k} = X_j^t$.

3. For this j_k, $\alpha(X_2^{LI,k,j_k}) = \alpha(X_2^{VA,j})$.

Once we have completed these three steps, the rest of the proof is simple. If we define a propositional valuation V to map a variable X_j to $\sigma(\alpha(X_2^{VA,j}))$, we know by the third part of the proof that $V \models L_{kj_k}$ for all k. As V satisfies a literal of each clause of the formula, it also satisfies the entire formula.

For the first part, let $(\mathcal{M}^{\sigma'}, \alpha')$ be a solution of φ. It is easy to see that for all $2 \le k \le m$, there are trees $\sigma_{k1}, \sigma_{k2}, \sigma_{k3}$ such that

$$\sigma'.\alpha'(X_1^{C,k}) = f(f(f(\sigma_{k1}, \sigma'.\alpha'(X_1^{C,k-1})), \sigma_{k2}), \sigma_{k3}).$$

For let $2 \le k \le m$ be arbitrary, then if we call $\pi = \alpha'(X_1^{C,k})$, the satisfiedness of the labeling and equivalence constraints in φ requires the following assignments of paths and labels:

var	$\alpha'(var)$	label
$X_1^{C,k}$	π	f
$X_2^{C,k}$	$\pi 1$	f
$X_3^{C,k}$	$\pi 2$	
$X_4^{C,k}$	$\pi 11$	f
$X_5^{C,k}$	$\pi 12$	
$X_6^{C,k}$	$\pi 111$	
$X_7^{C,k}$	$\pi 112$	
$X_1^{C,k-1}$	$\pi 112$	

This implies not only the above structure of the trees, but by a simple induction over k, also the relation

$$\alpha'(X_1^{C,k}) = (112)^{m-k}.$$

By the same argument, there are trees $\sigma_{11}, \sigma_{12}, \sigma_{13}$ such that

$$\sigma'.\alpha'(X_1^{C,1}) = f(f(f(\sigma_{11}, \sigma'.\alpha'(X_1^{VA,1})), \sigma_{12}), \sigma_{13}).$$

Furthermore, $\alpha'(X_1^{C,m})$ must dominate all other variables. So if we call $\sigma = \sigma'.\alpha'(X_1^{C,m})$ and let α be such that $\alpha'(X) = \alpha'(X_1^{C,m}) \cdot \alpha(X)$, we know that $(\mathcal{M}^{\sigma}, \alpha)$ satisfies φ as well.

For the second point, the following relations must hold to satisfy φ:

$$\alpha(X_8^{C,k}) \le \alpha(X_4^{C,k}) \le \alpha(X_{11}^{C,k}) = \alpha(X_8^{C,k}) \cdot 11.$$

Furthermore, $\alpha(X_9^{C,k}) = \alpha(X_8^{C,k}) \cdot 1$. So $\alpha(X_4^{C,k})$ must be identical to the denotation of one of the other three variables. Let a be the function that maps literal indices to their respective variable indices (for example, it maps 3 to 8), then pick j_k such that $\alpha(X_4^{C,k}) = \alpha(X_{a(j_k)}^{C,k})$.

Finally, we prove the third part by a series of arguments similar to the main argument of the first part: From the satisfiedness of φ, we derive relations between the paths denoted by variables $X_1^{\dots,r,\dots}$ and $X_1^{\dots,r-1,\dots}$ and use inductions to extend this to a result for the path denoted by a variable $X_1^{\dots,r-s,\dots}$. We will not spell out these arguments in detail any more.

We can conclude that $\alpha(X_4^{C,k}) = (112)^{m-k} \cdot 11$. In particular, $\alpha(X_4^{C,1}) = (112)^{m-1} \cdot 11$, so $\alpha(X_1^{VA,1}) = (112)^m$. This implies that $\alpha(X_2^{VA,j}) = (112)^m \cdot 2^{j-1} \cdot 1$. Furthermore, we know that $\alpha(X_{a(j_k)}^{C,k}) = \alpha(X_4^{C,k}) = (112)^{m-k} \cdot 11$; we conclude that $\alpha(X_1^{SC,k,j_k,1}) = (112)^{m-k+1}$. By two more inductions, we can conclude that $\alpha(X_1^{SK,k,j_k,1}) = (112)^m$ (by going through $k-1$ SkipC blocks) and that $\alpha(X_1^{LI,k,j_k}) = (112)^m \cdot 2^{j-1}$ (by going through $j-1$ Skip blocks). So by the respective binary labeling constraint, $\alpha(X_2^{LI,k,j_k}) = (112)^m \cdot 2^{j-1}1$, as we wanted to show.

This concludes the proof of completeness. Taking all the results from this and the previous section together, we have just shown:

Theorem 4.19. *The satisfiability problems of the language of dominance constraints and of the language of conjunctions and disjunctions over dominance constraints are NP-complete. If the signature is finite, the satisfiability problems of the propositional language over the dominance constraints and of the positive existential fragment are NP-complete, as well.*

4.3 Implementations

To close our discussion of the problem of solving dominance constraints, we briefly go into how they can be implemented by constraint programming, as laid out by Duchier and Gardent (1999). Furthermore, we briefly show how to implement context unification, as presented by Koller (1998). We will not go into either implementation very deeply, but refer the reader to the respective articles.

4.3.1 Implementing Dominance Constraints

The algorithm for solving dominance constraints that we have presented in Section 4.1 guesses for each pair of variables if one dominates the other in its first step. This makes the proof of termination in NP time very convenient, but renders the algorithm useless for implementation.

Koller et al. (1998) present another algorithm for solving dominance constraints which replaces this guessing step by a set of *distribution* rules. The strategy of this algorithm is to apply propagation rules, starting with the original constraint, until no further rules can be applied; if the result does not contain **false**, one distribution rule is applied, and the process starts over. This strategy of "propagate & distribute" is the basic paradigm of Constraint Programming.

A language that provides very powerful constraint programming techniques is the concurrent programming language Oz (Smolka 1995; Oz Development Team 1999). Its basic programming model is that a set of concurrent threads operate on a *constraint store*. If one of these "agents" detects a certain situation in the constraint store, it can add more information to the store. The propagation and clash rules of the solution algorithm can be programmed very elegantly as agents that wait for the rule preconditions to be present in the constraint and then add the right-hand sides. Once no agent can contribute any more information, a distribution rule can be applied to distinguish cases.

One particularly nice feature of Oz is that it provides *set constraints* (Müller and Müller 1997). These constraints allow to express equations and inequations between terms (such as unions and intersections) over variables that denote finite sets of integers, as well as (non-)membership of integers in these sets. The implementation of set constraints used in Oz is very efficient.

Set constraints have an immediate application to a solver for dominance constraints, as noticed by Denys Duchier (Duchier and Gardent 1999). We can associate every variable X of a dominance constraint with set variables that denote the sets of variables denoting nodes equal to, properly above, properly below, and in disjoint positions of the node denoted by X, respectively. We can then translate all dominance constraints to set constraints of their associated set variables.

For example, we can say that for each node variable X, its associated set variables must form a partition of the set of all variables in the dominance constraint. Furthermore, a dominance constraint $X \triangleleft^* Y$ translates to a conjunction that expresses that all variables that dominate X must also dominate Y, all variables dominated by Y must also be dominated by X, and all variables that are disjoint to X must also be disjoint to Y.

Finally, a distribution rule can be added by taking into account that any two variables X, Y must either denote the same node, X must properly dominate Y, Y must properly dominate X, or X and Y must be disjoint. Whenever no propagation rule can contribute information, these cases are tested in turn, producing new information in each case.

Despite the general intractibility of the problem that we have proved in the previous section, the implementation runs very efficiently on real problems from the linguistic domain (see Fig. 4.5).

(Subst) $X{=}t \longrightarrow \textbf{true}$ if X does not appear free in t $\mid X \mapsto t$

(Decomp) $a(t_1,...,t_n){=}a(t'_1,...,t'_n) \longrightarrow \bigwedge_{i=1..n} t_i{=}t'_i \mid Id$

(Proj) $t{=}C(t') \longrightarrow t{=}t' \mid C \mapsto \lambda X.X$

(Imit) $a(t_1,...,t_n){=}C(t') \longrightarrow t_i{=}C'(t') \mid C \mapsto \lambda X.a(t_1,..,t_{i-1},C'(X),t_{i+1},..,t_n)$

(Flex-Flex1) $C(t){=}C'(t') \longrightarrow t{=}C''(t') \mid C' \mapsto \lambda X.C(C''(X))$

(Flex-Flex2) $C(t){=}C'(t') \longrightarrow \textbf{true} \mid C \mapsto \lambda Y.C_1(a(\,\pi(\overline{X},C_2(Y),C_3(t'))\,)),$
$C' \mapsto \lambda Z.C_1(a(\,\pi(\overline{X},C_2(t),C_3(Z))\,))$
where π is a permutation

Figure 4.4: The algorithms for context unification. The rules below the line are removed to make the complete algorithm more tractable.

4.3.2 Implementing Context Unification

As we have seen in Chapter 2, the CLLS analysis of ellipsis relies on parallelism constraints, which we have shown to be equivalent to context constraints in the previous chapter. So even if the problem turns out to be decidable, we are confronted with a much higher level of intractibility when attempting to implement them; but it is nonetheless necessary, and with some effort, it can be done with reasonable efficiency. As there is currently no known solution procedure for solving parallelism constraints, we present an implementation for the equivalent problem of context unification.

Fortunately, even though the problem is not known to be decidable, there are known complete solutions algorithms; however, they do not necessarily terminate. One such procedure, taken from (Niehren et al. 1997a), is shown in Figure 4.4.

Implemented in its raw form, however, this procedure not only does not necessarily terminate, but even for the cases when it does terminate, it suffers from two major problems:

1. Massive overgeneration: The implementation does not output minimal or most general solutions of any kind, but a wide variety of (linguistically uninteresting) partial instantiations.

(1) *Every man loves a woman.*
(2) *Every researcher of a company saw most samples.*
(3) *Peter likes Mary. John does too.*

Implementation	Ex. (1) fail	Ex. (1) sol	Ex. (1) time	Ex. (2) fail	Ex. (2) sol	Ex. (2) time	Ex. (3) fail	Ex. (3) sol	Ex. (3) time
complete CU	3000	600	40 sec	n/a			13000+	2200+	2+ hrs
modified CU	71	2	1 sec	6500	5	40 sec	111	2	1 sec
dom. constraints	0	2	<1 sec	0	5	1 sec	n/a		

Figure 4.5: Comparison of the performance of the implementations for dominance constraints and context unification.

2. Enormous runtimes: It takes about 40 seconds to solve even the simplest scope ambiguity, and Hirschbühler sentences take hours.

In (Koller 1998), both of these problems were alleviated. To solve the first problem, a step was made from considering (encodings of) untyped lambda terms as the space of possible solutions to restricting the possible solutions to well-typed lambda terms. Because the terms are present in an encoded form, a distinction was made between τ-types of subtrees and κ-types of constructors f, describing how to compute the τ-type of the tree $f(t_1, \ldots, t_n)$ from those of the trees $t_1, \ldots, t_n$; the syntax of these types was as follows:

$$\begin{array}{lcl} \tau & ::= & e \mid t \mid \langle \tau_1, \tau_2 \rangle \\ \kappa & ::= & \langle \tau_1, \ldots, \tau_n \rangle \rightsquigarrow \tau \qquad (n \geq 0). \end{array}$$

By requiring all solutions to be well-typed, the number of solutions was cut down immensely, the only type of wrong solutions that were still found being those that involved additional quantifiers while still being well-typed.

The second problem was addressed by a more drastic measure: The last two rules of the algorithm, which contributed most to the nondeterminism that made the search space that the implementation had to traverse so huge, were removed. This rendered the implementation incomplete, but on all examples that were tested, the linguistically relevant solutions were still found. A reason for this is that the first removed rule can be simulated in a way by sequences of Imitation and Projection rules, and the second removed rule invents new material and introduces it into a solution; in the linguistic application, this is only warranted in very special cases.

These two changes to the algorithm made its runtimes much more pleasant (and in most cases, feasible at all). Fig. 4.5 shows a comparison of the performance of the complete and the modified algorithm; for each implementation and each example, the number of failed search paths, the number of found solutions, and the runtime are shown. But the table also shows that on scope ambiguities, where the solver for dominance constraints can be applied as well, the latter by far outperforms even the streamlined implementation of context unification. This gives rise to the hope that once it is known how to solve parallelism constraints, a combined implementation that solves the entire language of CLLS will perform much better than the implementation of CU.

4.4 Conclusion

In this chapter, we have analyzed the complexity of various logical languages over the dominance constraints. We have shown for all languages between the purely conjunctive constraint language and the positive existential fragment that this problem can be decided in NP by a saturation algorithm (in the latter cases, only if the signature is finite) and that it is NP-hard, as we can use it to express the satisfiability problem of formulae of propositional logic in 3-CNF. Finally, we have sketched implementations that solve dominance and context constraints.

In the light of this complexity result, it is natural to look into analogous results about the full first-order language, for which not even decidability is obvious. In fact, it has been known for a while that the problem is decidable. Backofen et al. (1995) have given an axiomatization of the first-order theory of dominance constraints over finite trees, and while this axiomatization is not complete, we can interleave steps of first-order deduction over this set of axioms (which will eventually show validity if a formula is valid) and steps of enumerating all finite trees and checking satisfiedness (which will eventually show that the formula is not valid if it isn't) to obtain a decision procedure (Backofen, personal communication). This proof, however, is hardly straightforward and does not say anything about the complexity of the problem.

Koller et al. (1998) remedy this situation by giving a more straightforward proof of the decidability and pinpointing the complexity of the problem to be non-elementary (i.e. there is no tower of exponentials of any fixed height that isn't exceeded by the running time for large instances). The decidability proof generalizes a similar result by Rogers (1994), who showed decidability for finite signatures by encoding dominance constraints in the monadic second-order logic SnS, which was shown to be decidable by Rabin (1969), to signatures of bounded arity. The key idea of the complexity proof is to encode the equivalence problem of regular languages with concatenation, union, and complement, as first-order dominance formulae.

This yields the following state of affairs in the complexities of languages over the dominance constraints:

- The satisfiability problems of dominance constraints and, if the signature is finite, also of the propositional language over dominance constraints and of the positive existential fragment over the dominance constraints are NP-complete.
- The satisfiability (and validity) problem of the first-order language over the dominance constraints is decidable and has non-elementary complexity.
- The satisfiability problems of all languages over dominance constraints with precedence (Section 2.1.5) are in the same complexity classes as their counterparts over our variant of dominance constraints. We have seen in Section 2.1.5 that it is easy to express our dominance constraints in this language; on the other hand, we can also encode precedence constraints as dominance constraints once we have access to disjunction, and the latter can be decided in NP.

One important restriction that we made throughout the chapter was that the signature must contain one constructor of arity at least 2. This restriction was necessary both to simulate constructors of arbitrary arity in the completeness proof of the algorithm and to make space for the "rubbish dumps" in the NP-hardness proof; it is also necessary for the proof of non-elementarity of the first-order language. While the problem is clearly still in NP if we leave this restriction away, it is unclear if we can maintain the NP-hardness result. From a practical perspective, however, this question is of marginal interest.

An open question of immediate practical impact is if the complexity can be reduced by sufficient amounts of inequality constraints between variables to prevent the overlaps or variable identifications that were so crucial in our encoding. In particular, it is unclear if the complexity of dominance constraints doesn't become deterministically polynomial if for any two variables that participate in a labeling constraint, we impose an inequality constraint. We will come back to this restriction, which seems to be harmless for the linguistic application and essentially reduces the expressive power of dominance constraints to that of Hole Semantics or UDRT, in a broader context in the concluding chapter.

Chapter 5

Conclusions and Outlook

5.1 Summary

In this thesis, we have explored the formal properties of two computational logics that have been used as representation formalisms in natural-language semantics.

First, we have shown that CLLS_0, the fragment of the language CLLS that talks about trees (and not the more powerful lambda structures), is equivalent to the language of context unification, thereby establishing that both satisfiability problems have the same complexity. Afterwards, we have considered dominance constraints, the most interesting sublanguage of CLLS_0. We have proved that their satisfiability problem is NP-complete; along the way, we have also shown how to decide satisfiability and how to implement this in a way that runs efficiently for linguistic problems. Both of these results are of general interest, especially the second one, as dominance constraints are widely used in computational linguistics.

Within the CHORUS project, an additional benefit of the result of equivalence between the two representation languages is that old results and analyses can essentially be taken over; the migration to a new formalism has not changed the expressive power. However, the CLLS analysis of, say, scope ambiguities is actually *simpler* than the CU one. The CU analysis of scope ambiguities in (Niehren et al. 1997b) avoids the problems addressed in Section 2.4, but is rather intransparent. As we have argued in Chapter 2, this is so because it is just the encoding of a dominance constraint. So in truth, the equivalence result is not really useful for taking over older analyses into the new formalism, but for an *a posteriori* justification of the old analyses based on the new one. This result also establishes a connection between the language CLLS and the ongoing research on the formal properties of context unification.

As CHORUS also strives to achieve computational feasibility, the investigation of the computational complexity of the formalisms used is clearly of central relevance for the project as well. Our result about the complexity of solving dominance constraints comes as a slightly inconvenient truth in this respect, but as we have seen, implementations of the dominance constraints that we really see are efficient anyway.

5.2 Further Work

There are a number of highly interesting open questions that will be considered in the next phase of the CHORUS project. Here, we briefly discuss some of them, especially inasmuch as they are related to the results in this thesis, and consider ideas for answering them. Most of the space will be devoted to perspectives for direct deduction and underspecified beta reduction; but we will also consider linguistic issues and open questions of solving CLLS constraints (in extension to the solution algorithm of Chapter 4).

5.2.1 Constraint solving

The solution algorithm for dominance constraints in Chapter 4 performs with pleasant efficiency. However, some questions are still open: Why is it that despite the NP-hardness of the problem, the NP algorithm performs so well? How can we solve parallelism constraints? And finally, is there a way to convert the algorithm to a solution algorithm for constraint graphs instead of constraints?

Parallelism constraints. CLLS was designed as a description language not only for scope ambiguities, but also for ellipses and anaphora. Clearly, one needs to be able to solve anaphoric and parallelism constraints to fully achieve this goal.

While it seems to be fairly trivial to solve anaphoric constraints, it is currently unclear how parallelism constraints can be solved. This is not surprising, as they approach the difficult problem of context unification from an unusual direction. As we have shown in Chapter 3, we can always encode a CLLS constraint involving parallelism as a context constraint and then solve the context constraint, but this is neither conceptually nor computationally acceptable. So this problem will need to be investigated in the near future.

Efficient solution of dominance constraints. Furthermore, as we have said variously throughout the thesis, it is striking that although the problem of solving dominance constraints is NP-hard, those dominance constraints that do appear in the linguistic analysis can be solved efficiently. It will be worthwhile to investigate what special property distinguishes the constraints in the linguistic application from the general case and makes the problem of solving them so easy, and to derive better complexity results for the restricted problem.

A promising candidate for this property is to require that any two variables that participate in a labeling constraint in the same role must denote different nodes – i.e., if a dominance constraint has both a conjunct $X{:}f(X_1, \ldots, X_n)$ and $Y{:}g(Y_1, \ldots, Y_m)$, X and Y must denote different nodes, and any X_i and X_k must too, even if f and g are the same constructor. This can be expressed by inequality constraints. The relevance of this becomes apparent when we consider the *fragments* of the constraint graph, i.e. the maximal subgraphs that are connected with $\mathbb{N}$-edges. Fragments correspond to parts of the tree that we describe (hence the name). If the constraint satisfies the above labeling condition, no solutions can be obtained by overlapping fragments. We can define a *fragment graph* whose nodes are the fragments of the constraint graph and in which an edge is drawn from one fragment to another if there is a dominance constraint between a node in the first fragment and one in the second, and solving the dominance constraint then degenerates to a tree-like ordering of the fragments that respects the dominance relations in the fragment graph. (This is exactly the same problem as disambiguating underspecified representations of Hole Semantics, where we have a partial order on fragments to begin with.) While this is by no means certain, it does look like this problem might be simpler than the general satisfiability problem of dominance constraints.

Constraint graphs. This construction illustrates the importance of the graph view on dominance constraints that we have been using throughout the thesis. Unfortunately, the general notion of "constraint graph" has not yet been formally defined. The reader may have noticed that while we have talked about the constraint that corresponds to a given constraint graph (and considered the graph just as a more intuitive notation), we have carefully avoided talking about the converse construction, a constraint graph corresponding to a constraint. This is because there is considerable freedom in the degree of redundancy and explicity of representation one chooses in the definition of constraint graphs. For example, what should be the condition for drawing a dominance edge in the graph? Should we do so exactly if there is a corresponding dominance constraint; should it be a very explicit representation with an edge for every entailed dominance relation (similarly to the graph G' we defined in the completeness proof of the satisfiability algorithm for dominance constraints); or should it be very irredundant and only contain a dominance edge if it can't be inferred from the rest of the graph (similarly to the graph G in the completeness proof)? Similar problems arise for inequality constraints. It is unclear which choice leads to the most convenient definition.

One particularly important aspect of this choice is that the resulting notion of constraint graphs should admit a reasonably convenient algorithm for disambiguating them; we will see in our discussion of underspecified beta reduction below that having such a solution algorithm on graphs can be very useful. A solution algorithm could be a graph rewriting system d that disambiguates the constraint by eliminating ambiguous subgraphs (e.g., nodes with two incoming edges). We can call such a system sound and complete if its normal forms are solved forms (similarly to the definition in Chapter 4), and these solved forms correspond exactly to the solutions of the underlying constraint. Typically, there will be more than one result of disambiguating the same ambiguous subgraph (this is what "ambiguity" means, after all); but sometimes, all but one of the options can be excluded, and we call such a step a *propagation* step p. In a reasonable solution algorithm, it should be possible to permute applications of p and d rules (but not necessarily applications of two different d rules).

Minimal solutions. One issue that we have only hinted at so far is that we still have no clear idea of what a "minimal solution" is. We briefly used this notion in Section 2.2, where we wanted to exclude that arbitrary amounts of semantic material could be filled into the gaps left open by dominance constraints. It is obviously unsatisfactory to leave such an important point open, but unfortunately, there are at least two promising ways how we could define "minimal", and both have their advantages.

One of them is to require that all solutions must be formed *with known material*. Put more formally, this means that every node in a solution must be denoted by a variable that participates in a labeling constraint as the head. This definition corresponds nicely to the idea of solving a constraint by simply arranging the fragments of the constraint graph, because it implies that every node in a solution must correspond to one of the nodes in a fragment.

The alternative is to define some order on trees (or as it were, lambda structures) and to only consider solutions that are minimal with respect to this order. For example, we could only consider trees with a minimal number of nodes. The advantage of this version is that it works well with the ideas on reinterpretation that we will sketch in the next section.

It should be noted that the choice among these alternatives for defining minimality makes a significant difference. For example, there are constraints that are satisfiable (and, hence, have minimal solutions by the second version of the definition), but that do not have solutions that only use known material:

(5.1) $X{:}a \wedge Y{:}b$

can be solved over the tree $f(a,b)$; but the root of this tree (which is labeled with f) is not mentioned in a labeling constraint and hence, was not built with known material.

5.2.2 Linguistic coverage

Now we will briefly sketch three of the various linguistic perspectives of the CHORUS project: the modeling of dynamic semantics, reinterpretation, and syntax/semantics interaction.

Dynamic semantics. As is well known, the potential positions of the antecedent of a given pronoun are restricted. For example, one common assumption that DRT (Kamp and Reyle 1993) and DPL (Groenendijk and Stokhof 1991) make about this *accessibility relation* is that the antecedent cannot appear within a disjunction, negation, or universal quantification that precedes the anaphor. The way that this is usually modeled is by interpreting a sentence as an operator that changes the anaphoric potential of the discourse, i.e. the list of potential antecedents for an anaphor in the next sentence; this "dynamic" interpretation is in contrast to traditional "static" semantic analyses.

In our context, the most interesting aspect of these phenomena is that they interact with scope ambiguities, as in the following discourse, where the anaphor restricts the source sentence to the reading that assigns *a woman* wide scope; in the other reading, the antecedent would be located within a universal quantification and hence, be inaccessible.

(5.2) *Every man loves a woman. Her name is Mary.*

One idea to model this interaction, which was presented in (Koller and Niehren 1999), is to impose restrictions on the relation between the start and end node of the (currently unrestricted) anaphoric links that CLLS uses to model anaphor/antecedent relations. For example, it is straightforward to rewrite the

DPL accessibility rules as conditions on paths in trees. With this definition, the reading of (5.2) that assigns *every man* wide scope would not be well-formed because one would have to pass through a universal quantifier on the way from the anaphoric node to the antecedent.

Furthermore, it is not difficult to define a simple (incomplete) inference rule on the constraint level that can make the effects of such an anaphoric link on the structure of a solution explicit as a dominance constraint. In our example, this rule would cause the disambiguation of the scope ambiguity *on the level of underspecified representations*. The kind of constraint we need to express this is of the form

(5.3) $\neg(X \triangleleft^* Y \wedge Y \triangleleft^* Z)$;

it basically says that Y must not intervene between X and Z, and it seems to be useful in a variety of other situations as well. We know from Section 4.1 that adding disjunctions of negated dominance constraints does not affect the worst-case complexity of the satisfiability problem. In addition, it seems that even the efficient actual running times of the implementation via set constraints that we sketched in Section 4.3.1 could be maintained.

What is nice about this approach from a linguistic perspective is that it is very modular: We can easily plug in whatever accessibility relation we want to. On the other hand, it is an important step towards a truly underspecified account of semantics that we can resolve this kind of interaction between anaphora and scope without enumerating readings. Note, by the way, that we are only using the word "dynamic" in a very broad sense; our semantics is fully static (i.e., does not care about context change potential).

Even so, numerous questions about this issue are still open. For example, it is not entirely clear how we can account for sentences like

(5.4) *Every pilot who shot at it hit the MIG that chased him.*

that contain kataphoric references, and if we can still process this type of reference efficiently. Further open questions include how to raise the analysis to the higher-order case, how to find antecedents if coindexation is not given, and accommodation. We are only beginning to understand our analysis of these phenomena, and it will be intriguing to investigate it further.

Reinterpretation. Consider the following example.

(5.5) *Wine is standing on the table.*

In this sentence, it is not the liquid that is standing on the table, but a container, e.g. a bottle, that contains wine. The semantics of the container, which has no explicit linguistic representation, is introduced by a so-called *reinterpretation* of the sentence; similar effects can take place for aspect.

Traditionally, this kind of phenomenon is treated by destructively introducing a reinterpretation operator into the semantics of the sentence after recognizing the sortal conflict between the mass noun *wine* and the VP *stand on the table*, which expects a physical object as its argument.

Egg (1999) proposes a CLLS-based analysis where the sentence meaning is not described directly, but in a slightly underspecified fashion. The key construction is that the wine is not connected to the rest of the sentence via a labeling constraint, but only via a dominance constraint. This kind of constraint allows a solution where the reinterpretation operator is added in the right place; in fact, to obtain a well-sorted solution, it is necessary to add the operator. If we think about this in terms of "minimal solutions", we essentially change the requirement on a solution we are interested in from being minimal among all possible solutions to being minimal among the well-sorted solutions.

This account has several advantages over the traditional analysis. First of all, semantic composition is still a monotonic process, and the introduction of reinterpretation operators, which is based on sortal information and world knowledge, is a process that can be clearly distinguished from the linguistic side of semantic construction. Next, the CLLS analysis does not only cover cases of "type coercion" as in Example 5.5, but only other reinterpretation patterns. Finally, it seems to be possible to apply reinterpretation in such a way that it can be done before enumerating the readings of a scope ambiguity.

Syntax/semantics interaction. A final idea concerns the interaction of syntax and semantics. As we have seen in the introduction, there are several constraint-based approaches to the underspecification of syntactic ambiguity; CLLS is a constraint-based approach to the underspecification of (some classes of) semantic ambiguity. So it might be worthwhile to investigate if their interaction could not be modeled by an additional layer of constraints that translates information from both sides to each other.

Both syntax and semantics would be described in an underspecified fashion in this model, and before any disambiguation takes place, propagation of constraints to the other level would be allowed. Then the newly obtained constraints could be further propagated on the other side, potentially yielding new information that helps in the disambiguation of the first side, and so on. The hope would be that once we do have to enumerate readings, most options would already be excluded by the mutual constraint propagation, thus reducing the search space.

At the moment, we do not know any clear examples where such syntax-semantics interaction (to set the idea apart from the unidirectional idea of a syntax-semantics interface) can be fruitfully applied, but the idea is certainly something that warrants further research.

The bottom line of this section is that in its current state, there are a large number of unexplored applications of CLLS. This is remarkable in that an underspecification formalism is usually obtained by stretching the expressive power of an existing language, as we have seen in the introduction. CLLS was developed from scratch with underspecification in mind and basically describes trees, an extremely flexible data structure; so far, it has not really been stretched at all, and combined with the formal foundations as laid out in this thesis, promises to be useful for a wide range of linguistic applications.

5.2.3 Towards Underspecified Beta Reduction

Finally, we present some thoughts about an important aspect of underspecification that we have briefly hinted at in the introduction: underspecified deduction. After a brief reintroduction to the basic intuition underlying this operation, we sketch some more concrete ideas on how to define a simpler, but still very interesting related operation: underspecified beta reduction. The details of this construction are unclear at the moment, but it seems likely that on a large scale, the ideas laid out here should go a long way towards a clean definition, and they are concrete enough to illustrate some of the basic problems.

Direct deduction. Recall Example 1.3 from the introduction:

(5.6) *Every man loves a woman.*
John is a man.
____
John loves a woman.

This argument is clearly valid, and we notice this without enumerating all readings, although the premise is ambiguous. To model this intuition, it should be possible to infer the conclusion from the premises on the level of underspecified representations. This does not only seem natural, it is also the very idea of underspecification to do as much work as possible *before* disambiguation.

It is crucial to understand that this inference does not have much to do with the usual inference on underspecified representations (e.g. the inference rules for dominance constraints that we defined in Section 4.1). The kind of inference we have just seen (and which we will call *direct deduction*) derives a description of the logical consequences of the described formulae from a description; it really cares about the semantic content of the represented formulae. On the other hand, simple inference on a CLLS formula can only serve to make the description more explicit.

It is an interesting question in itself what conclusions direct deduction should allow. In the example, the answer is very simple: *Every* solution of the underspecified description of the premise entails the (unique) solution of the underspecified description of the conclusion, so the second description should surely be inferrable from the first one. But for more difficult cases, for instance with ambiguous conclusions, the answer is less obvious. There is a wide range of choices for an exact definition of underspecified inference (see, e.g., van Deemter 1996), all with different logical properties and calculi, and we will not go into them in any detail here. Common to all of them, however, is that they lift an operation on the semantic representations of the single readings of a sentence to an operation on the underspecified descriptions of these representations, and that this inference must be defined in a way that is justified by some condition on the relation between the solutions of the underspecified representations on both sides. For example, the inference in (5.6) would be justified by a condition on the solutions that says that every solution of the (underspecified representation of the) premise must entail every solution of the (representation of the) conclusion. (This is not necessarily the most intuitive definition for underspecified entailment; it is just a random pick.)

Underspecified beta reduction. Instead of direct deduction, we will consider underspecified beta reduction in more detail, for several reasons. For one thing, lambda structures correspond to lambda terms up to α but not β equivalence; so clearly, beta reduction is a natural and extremely important operation on these structures. In particular, it is necessary to apply beta reductions to lambda structures until they represent a first-order formula before one can apply deduction rules at all, and we would like to do this on the underspecified representations. In addition, beta reduction is a much simpler operation than first-order deduction, and we expect that the experiences gathered with underspecified beta reduction will be helpful when considering direct deduction.

The basic idea of underspecified beta reduction is the same as that of direct deduction: lift an operation on the described structures (here, beta reduction on lambda structures) to an operation on the underspecified representations in a way that is justified by a relation on the described terms. This relation, too, is very simple

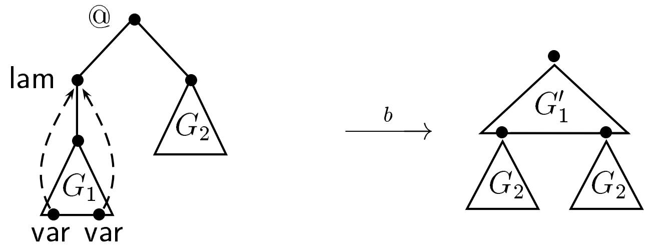


Figure 5.1: Underspecified β-reduction.

here: The solutions of the resulting constraint must be exactly the set of terms that can be obtained by applying beta reduction at a common redex to the solutions of the original constraint.

To make this a bit more precise, assume that φ_1 is a constraint that contains the description of a β-reduction redex; i.e., a subconstraint of the form

$$X_1{:}@(X_2, X_3) \wedge X_2{:}\mathsf{lam}(X_4) \wedge \lambda(X'_1){=}X_2 \wedge \ldots \wedge \lambda(X'_r){=}X_2.$$

Then every solution $(\mathcal{M}, \alpha)$ of φ_1 will have a redex at the position $\alpha(X_1)$. Hence, we can define that φ_2 is the result of underspecified beta reduction of φ_1 at the redex X_1 iff

$$S(\varphi_2) = \{(\mathcal{M}', \alpha') \;\mid\; \begin{array}{l}(\mathcal{M}, \alpha) \in S(\varphi_1),\ \mathcal{M}' \text{ is the result of } \beta\text{-reduction of } \mathcal{M} \\ \text{at redex } \alpha(X_1), \text{ and } \alpha' \text{ is "appropriate"}\},\end{array}$$

where $S(\varphi)$ is the set of solutions of the constraint φ. We will not make "appropriate" more precise here, as the exact definition of α' is simple and tedious.

This definition is nice, but what we really want is an operational characterization, for example as a graph rewriting system that transforms constraint graphs. One such system (let us call it b) that looks very natural is shown in Fig. 5.1. If the graph contains a description of a redex as in the left diagram, this subgraph can be replaced by a graph as in the right diagram; the description of the redex itself is removed, and the bound variables are replaced by the description G_2 of the argument.

At first glance, Fig. 5.1 looks just like usual beta reduction. However, it is important to remember that G_1 and G_2 are not necessarily trees, but arbitrary constraint graphs, and there can even be dominance constraints between nodes within G_1 and G_2 and nodes outside of these subgraphs, which does make a difference. Of course, if this is not the case, i.e. G_1 and G_2 are trees of $\mathbb{N}$-edges and no node of these subgraphs is involved in a dominance or parallelism constraint, b does, in fact, degenerate to usual β-reduction.

Problems with underspecified beta reduction. For the graph rewriting rule b to be faithful to the original definition of underspecified beta reduction, we have to require two properties: on the one hand, that the resulting constraint is satisfied by all β-reduced solutions of the original (a kind of completeness result), and on the other hand, that all solutions of the result can be obtained from solutions of the original by β-reduction (a soundness result). Below, we discuss some concrete problems of both directions, and sketch ideas for solving them.

Problems: Completeness. An obvious source of problems with completeness is that a solution of the original constraint φ_1 may contain more var nodes bound by the lam binder in the redex than the constraint mentions explicitly. In such a situation, the number of copies of G_2 in φ_2 will be too small for such a solution; so the result of applying β-reduction to this solution will not satisfy φ_2. It seems that this problem could be solved by only considering "minimal" solutions, by an appropriate definition of "minimal"; for example, we could require that a solution can only contain those var nodes that the constraint mentions explicitly. But as we have seen above, minimality is not a trivial concept, and some thought would have to be put into this first.

Another source of problems is parallelism. Consider a constraint that contains two copies of a description of a β-reduction redex and where a similarity constraint is imposed on the roots of these descriptions. Then it is clearly possible that while the original constraint is satisfiable, the constraint obtained by one b step on one (but not the other) of the redex descriptions is unsatisfiable – a clear violation of completeness in the above sense. Parallelism also causes some tricky problems for notions of minimality, and as we have seen above, there is no algorithm for solving parallelism constraint so far, which makes the development of a correctness proof of b very difficult. In general, parallelism is so inconvenient that it seems reasonable to set it aside for now, consider underspecified beta reduction on a restricted language, and try to extend it to the full language later.

Problems: Soundness. These two kinds of restrictions seem to solve (or work around) all problems involving completeness. However, there can still be problems with soundness: We can get spurious solutions, i.e. solutions of the resulting constraint graph that cannot be obtained from solutions of the original by β-reduction.

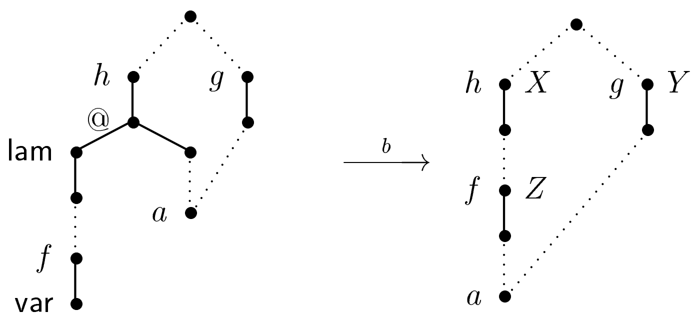


Figure 5.2: b can produce constraints with spurious solutions.

These situations usually involve a functor and/or argument of the β-reduction that is not sufficiently specified.

One problem arises when the argument contains an ambiguity and the functor binds more than one var node. In this case, b will produce a constraint that contains several copies of the ambiguous argument, and these copies can be disambiguated independently. That is, the subtrees satisfying different copies can be different; but in the β-reduction of a solution of the original constraint, they will be several copies of the *same* tree. What we would need to make sure in order to maintain soundness is that all copies of G_2 are disambiguated in the same way. An easy way to do this would be to impose similarity constraints on them; but as we have just seen, parallelism comes with problems of its own. So the most convenient way to work around this problem seems to be to restrict ourselves to linear lambda terms for now.

Finally, even in this restricted class of constraints, b can be unsound if the original constraint is overly ambiguous. Consider, by way of example, Fig. 5.2. All minimal solutions of the left constraint graph can be β-reduced to either $h(f(g(a)))$ or $g(h(f(a)))$ because the fragment with the g can be either above the application node or within the argument. But the right constraint graph, the result of applying the graph transformation b to the left graph, which should have just these two minimal solutions, has an additional minimal solution $h(g(f(a)))$. This constraint is a good illustration that, as we said above, the subgraphs G_1 and G_2 in Fig. 5.1 need not be trees and can have dominance relations to nodes outside the redex.

There are several conceivable ways of excluding this type of mistake. One way could be to simply impose a "non-intervention" constraint like we used in the context of dynamic semantics above. In the example, the problem would go away immediately if we imposed the additional constraint

$$\neg(X \lhd^* Y \wedge Y \lhd^* Z)$$

on the result.

A different idea of how to approach this problem is to restrict the applicability of the rewriting rule b. For example, we could require that for b to be applicable on the description of a redex, the argument has to be fully specified. Alternatively, we could require that there is a path from the lam node to the var node in the functor that consists of $\mathbb{N}$-edges: There is considerable freedom in balancing the restrictions among the descriptions of the functor and the argument.

At first sight, it seems that this restriction greatly reduces the value of underspecified beta reduction. But if b happens not to be applicable on a particular constraint graph, we can always disambiguate until it does become applicable; as we have said above, on fully disambiguated constraint graphs, b degenerates to usual β-reduction, so we can always arrive at a situation where b is applicable. We may need more disambiguations than absolutely necessary, but we will still be better than if we generally did β-reduction only on the satisfying lambda structures. In addition, it seems that it might be possible to find restrictions that allow (sound) application of b on most examples found in semantics.

Proving correctness. The changes and restrictions listed above seem to solve all problems; but of course, it will be necessary to prove that b really models underspecified beta reduction on the restricted cases.

One way to do this that looks very promising is based on using a sound and complete solution algorithm on constraint graphs, as sketched in Section 5.2.1. Given such an algorithm, correctness of b could be shown by proving a bisimulation argument expressing that rewriting steps in b and rewriting steps in d permute: In this case, a sequence of applications of d plus one application of b can be replaced by first one application of b and then a sequence of (possibly different) applications of d, and vice versa. In this way, applications of b can always be reduced to applications of b on solved forms, which is essentially usual beta reduction. It might be necessary to show that applications of p do not have any effect on the set of solutions of a constraint and that hence, we can apply p rules freely to normalize constraint graphs.

But clearly, such a proof relies heavily on the exact rules in the rewriting systems d and p. So it seems that the most challenging piece of work that has to be done before correctness of the restricted case of underspecified beta reduction can be proved is to find a solution algorithm for constraint graphs and prove its soundness and completeness.

Bibliography

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

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

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

Bierwisch, M. (1983). Semantische und konzeptionelle Repräsentation lexikalischer Einheiten. In R. Ruzicka and W. Motsch (eds), *Untersuchungen zur Semantik*, 61–99. Berlin: Akademie-Verlag.

Billot, S. and B. Lang (1989). The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th ACL*, Vancouver, 143–151.

Blackburn, P., W. Meyer-Viol, and M. de Rijke (1995). A proof system for finite trees. In H. K. Büning (ed.), *Computer Science Logic. Selected Papers of the 9th International Workshop CSL '95 (Paderborn, Germany)*, Number 1092 in LNCS, 86–105. Springer-Verlag, Berlin.

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

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

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

Courcelle, B. (1983). Fundamental properties of infinite trees. *Theoretical Computer Science 25*(2), 95–169.

Crouch, R. (1995). Ellipsis and quantification: A substitutional approach. In *Proceedings of the 7th EACL*, Dublin, 229–236.

Dalrymple, M., S. Shieber, and F. Pereira (1991). Ellipsis and higher-order unification. *Linguistics & Philosophy 14*, 399–452.

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

Egg, M. (1999). Reinterpretation by underspecification. Submitted.

Egg, M., A. Koller, J. Niehren, and P. Ruhrberg (1999). Constraints over lambda structures, antecedent contained deletion, and quantifier identities. Submitted. `http://www.coli.uni-sb.de/~koller/papers/acd.html`.

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

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

Goldfarb, W. D. (1981). The undecidability of the second-order unification problem. *Theoretical Computer Science 13*, 225–230.

Groenendijk, J. and M. Stokhof (1991). Dynamic predicate logic. *Linguistics & Philosophy 14*, 39–100.

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

Jaspars, J. (1997). Minimal logics for reasoning with ambiguous expressions. Technical Report 94, Universität des Saarlandes, Saarbrücken. `ftp://ftp.coli.uni-sb.de/pub/CLAUS/claus94.ps`.

Kamp, H. and U. Reyle (1993). *From Discourse to Logic*. Dordrecht: Kluwer.

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. and J. Niehren (1999, February). Towards underspecified processing of dynamics. Workshop on Dynamic Logic, Schloß Dagstuhl.

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

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.

Lévy, J. (1996). Linear second order unification. In *International Conference on Rewriting Techniques and Applications*. Springer-Verlag.

Makanin, G. (1977). The problem of solvability of equations in a free semigroup. *Soviet Akad. Nauk SSSR 223*(2).

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

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

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. `http://www.ims.uni-stuttgart.de/ftp/pub/papers/DYANA2/95copy/R2.2.C/Mus%kens.ps.gz`.

Niehren, J. and A. Koller (1998). Dominance Constraints in Context Unification. In *Proceedings of the Third Conference on Logical Aspects of Computational Linguistics*, Grenoble, France.

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.

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

Pinkal, M. (1996). Radical underspecification. In *Proceedings of the 10th Amsterdam Colloquium*, 587–606.

Rabin, M. (1969). Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society 141*, 1–35.

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

Rogers, J. (1994). *Studies in the Logic of Trees with Applications to Grammar Formalisms*. Ph. D. thesis, University of Delaware.

Rogers, J. and K. Vijay-Shanker (1994). Obtaining trees from their descriptions: An application to tree-adjoining grammars. *Computational Intelligence 10*, 401–421.

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

Schmidt-Schauß, M. (1994). Unification of stratified second-order terms. Technical Report 12/94, J. W. Goethe-Universität Frankfurt, Fachbereich Informatik. `http://www.ki.informatik.uni-frankfurt.de/papers/D-uni-SO-9-95.ps`.

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

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.

Venkatamaran, K. N. (1987). Decidability of the purely existential fragment of the theory of term algebra. *J. ACM 34*(2), 492–510.

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