Generalized Horn Constraints and Tractable Inference on Type Hierarchies with Open Specification

Stephen J. Hegner
Umeå University
Department of Computing Science
SE-901 87 Umeå, Sweden
hegner@cs.umu.se
http://www.cs.umu.se/~hegner

Revised: 7 September 2005

Abstract

Large type hierarchies are an integral part of many databases, including in particular lexicons for natural-language parsers. Due to their size and complexity, it is often impractical to provide a complete specification. Rather, an *open specification* is employed, in which properties of the hierarchy are specified by constraints. Unfortunately, the great expressive power of open specification comes at the expense of considerable computational complexity, with essential tasks rendered NP-hard. For this reason, most existing systems use a very simple set of default rules to determine the semantics, thus limiting the most desirable feature of open specification – wide expressive power.

In this work, it is shown how open specification with a relatively expressive set of constraints may be realized, while still keeping the computational complexity within reasonable limits. The key to the approach is to replace the full logical formulas of general open specification with formulas based upon a special class of disjunctions of Horn sentences. The resulting time complexity, while greater than the linear complexity of propositional Horn inference, is still better than quadratic, and close to linear in many cases.

1. Introduction

Type hierarchies are ubiquitous in knowledge representation, forming the cornerstones of topics such as description logics [3] and Formal Concept Analysis [14]. Within the context of database systems, they have a long history, beginning with early work on deductive database systems [21]; more recently, they have become one of the cornerstones of modern object-oriented database systems [2]. Such hierarchies are also central within the modern study of object-oriented programming languages, as well as within knowledge representation for artificial intelligence and natural-language processing.

For these reasons, type hierarchies have been studied extensively, in many different forms and in many different ways. The key property is unquestionably inheritance, and for this reason it has also been the most widely studied. There are however, other properties which are of substantial

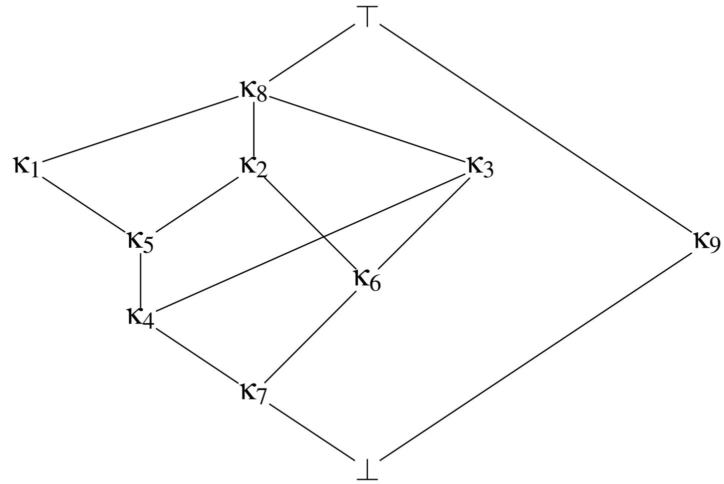

Figure 1: A graphical presentation of a type hierarchy.

importance in certain applications. One of these is *closure*. Consider the simple (parameterless) type hierarchy depicted in Figure 1. Under the convention that moving downwards in the hierarchy represents specialization, one may safely conclude that $\kappa_5 \leq \kappa_1$ and $\kappa_5 \leq \kappa_2$. To give formal meaning to these statements, assign to each type τ a set $\mathfrak{I}(\tau)$ of objects of type τ; the objects being taken from a universe $\mathfrak{U}$. The statements $\kappa_5 \leq \kappa_1$ and $\kappa_5 \leq \kappa_2$ then have the meanings $\mathfrak{I}(\kappa_5) \subseteq \mathfrak{I}(\kappa_1)$ and $\mathfrak{I}(\kappa_5) \subseteq \mathfrak{I}(\kappa_2)$, respectively. Thus, $\mathfrak{I}(\kappa_5) \subseteq \mathfrak{I}(\kappa_1) \cap \mathfrak{I}(\kappa_2)$ as well. *Meet closure* asks whether this inclusion is exact; that is, whether $\mathfrak{I}(\kappa_5) = \mathfrak{I}(\kappa_1) \cap \mathfrak{I}(\kappa_2)$ holds. In words, meet closure states that the *only* objects of type κ_5 are those which are either of type κ_1 or else of type κ_2. Meet closure is of great importance in certain applications, most notably in the context of the operation of unification on typed feature structures [6, Ch. 3], particularly in support of natural-language parsing systems based upon formalisms such as HPSG [22]. The dual notion is that of *join closure*; for example, one may ask whether $\mathfrak{I}(\kappa_1) \cup \mathfrak{I}(\kappa_2) = \mathfrak{I}(\kappa_8)$ holds, rather than just the weaker condition $\mathfrak{I}(\kappa_1) \cup \mathfrak{I}(\kappa_2) \subseteq \mathfrak{I}(\kappa_8)$. Although less widely seen, this notion also has applicability in disjunctive unification [10].

In most situations, meet closure is an all-or-nothing proposition. Either no such closure is enforced, or else it is assumed that it holds for all triples (τ_1, τ_2, τ_3), with τ_3 the greatest lower bound (glb) of τ_1 and τ_2. For such an approach to apply, the hierarchy must have a unique glb for each pair of elements. Such an assumption is implicit in many frameworks, such as ACQUILEX [9] and LKB [7], both of which are closely tied to the formal underpinnings of LFG [4], [8]. The hierarchy of Figure 1 fails to have such unique glb's, since κ_4 and κ_6 are incomparable maximal lower bounds of $\{\kappa_2, \kappa_3\}$. However, if the line from κ_3 to κ_4 is removed, the resulting hierarchy has unique glb's. A dual property relates least upper bounds (lub's) and join closure.

In the work reported in this paper, *open specification* of such constraints is studied. Rather than requiring every glb (resp. lub) to satisfy the meet closure property (resp. join closure property), such constraints are listed explicitly for those sets of types which have a glb (resp. lub). Open specification is important in many applications, because while closure may be known in some instances, in a large database, it is rarely known in all. Within the context of open specification, inference of constraints becomes central. First and foremost, it is critical to know whether a set of constraints is consistent (the property of *satisfiability*). In addition, given that some inheritance and closure constraints hold, it is important to know which others must hold as well (the property of *inference*). Of course, satisfiability may be viewed as a special case of inference.

The particular emphasis in this work is upon effective algorithms for constraint inference. As such, it builds upon the earlier work [17] and [18]. In [17], the formal groundwork for the problems considered here was developed, and it was shown that the general satisfiability problem is NP-complete. In [18], the notion of semantics was developed in detail, and a heuristic strategy for identifying models was presented. In this paper, it is shown that by limiting the class of constraints appropriately, powerful techniques developed for inference on Horn clauses [11] may be brought to bear on this problem.

The work reported here was motivated largely by the use of type hierarchies in the context of HPSG [22]. Thus, attention is focused entirely upon finite, parameterless type hierarchies. Although such hierarchies are important in their own right, it is undeniable that parameterized hierarchies of all sorts are of great importance in other contexts. The work reported herein is a complement to work such as that of, for example, [13] and others, which work with constraint inference on parameterized hierarchies. Before the issues of of complexity of open specification are understood for parameterized hierarchies, they must be mastered in the simple, parameterless setting. Both constraint inference and well-formedness of inheritance are critical concepts. Eventually, it will be appropriate to unify the two notions in a single framework, but for now, it seems prudent to study inference within a relatively simple framework.

2. Modelling of Type Hierarchies

In this section, the basic formal ideas about the syntax and semantics of type hierarchies are developed. Some of the notions parallel those developed in [18]; however, much has been re-worked to make the ideas more accessible and unified. The reader is also referred to [19] for a more formal development of the underpinnings of such hierarchies.

2.1 A taxonomy of constraints A *clean set* is any finite set P which does not contain either of the special symbols $\perp$ and $\top$. Define $\mathsf{Aug}(P) = P \cup \{\perp, \top\}$. The elements $\perp$ and $\top$ are called the *extreme types*; the elements of P are called the *base types*. As shall soon be formalized, $\top$ (resp. $\perp$) represents the greatest (resp. least) type in the hierarchy.

There is a taxonomy of constraints which will be used throughout this work; it is summarized in Table 1. This rather pedantic classification is necessary because of the need to isolate special classes of formulas for the algorithms which are developed in the following sections.

An *elementary positive meet constraint* has the form $(\bigsqcap\{\tau_1, \tau_2, .., \tau_n\} \leq \tau)$, with the τ_i's and τ members of $\mathsf{Aug}(P)$. As shown in Table 1, the set of all such constraints over P is denoted $\mathsf{ElemConstr}^+_\sqcap(P)$. In this context, $\bigsqcap$ represents a lattice-like meet operation; not logical conjunction. The normal variants associated with such lattice notation are allowed. For example, both $(\tau \geq \bigsqcap\{\tau_1, \tau_2, .., \tau_n\})$ and $(\tau_1 \sqcap \tau_2 \sqcap .. \sqcap \tau_n \leq \tau)$. are synonyms for the above formula. In particular, $(\tau_1 \leq \tau)$ is a synonym for $(\bigsqcap\{\tau_1\} \leq \tau)$. The *elementary positive join constraints*, denoted $\mathsf{ElemConstr}^+_\sqcup(P)$, are defined dually, as elaborated in the second entry of Table 1. Similar synonym conventions apply. Together, the elementary positive meet and join constraints comprise the elementary positive constraints, denoted $\mathsf{ElemConstr}^+(P)$. It does not suffice to consider meets and joins of only two elements; for example, it is quite possible to express the constraint $\bigsqcap\{\tau_1, \tau_2, \tau_3\} \leq \tau$ without

Symbol	Formula	Bindings
$\mathsf{ElemConstr}^+_\sqcap(P)$	$(\bigsqcap\{\tau_1,\tau_2,..,\tau_n\} \leq \tau)$	$\tau_1,\tau_2,..,\tau_n,\tau \in \mathsf{Aug}(P)$
$\mathsf{ElemConstr}^+_\sqcup(P)$	$(\tau \leq \bigsqcup\{\tau_1,\tau_2,..,\tau_n\})$	$\tau_1,\tau_2,..,\tau_n,\tau \in \mathsf{Aug}(P)$
$\mathsf{ElemConstr}^+(P)$	φ	$\varphi \in \mathsf{ElemConstr}^+_\sqcap(P) \cup \mathsf{ElemConstr}^+_\sqcup(P)$
$\mathsf{Constraints}^+_\sqcap(P)$	$\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$	$\varphi_1,\varphi_2,\ldots\varphi_n \in \mathsf{ElemConstr}^+_\sqcap(P)$
$\mathsf{Constraints}^+_\sqcup(P)$	$\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$	$\varphi_1,\varphi_2,\ldots\varphi_n \in \mathsf{ElemConstr}^+_\sqcup(P)$
$\mathsf{Constraints}^+(P)$	$\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$	$\varphi_1,\varphi_2,\ldots\varphi_n \in \mathsf{ElemConstr}^+(P)$
$\mathsf{ElemConstr}^-_\sqcap(P)$	$(\neg\varphi)$	$\varphi \in \mathsf{ElemConstr}^+_\sqcap(P)$
$\mathsf{ElemConstr}^-_\sqcup(P)$	$(\neg\varphi)$	$\varphi \in \mathsf{ElemConstr}^+_\sqcup(P)$
$\mathsf{ElemConstr}^-(P)$	$(\neg\varphi)$	$\varphi \in \mathsf{ElemConstr}^+(P)$
$\mathsf{Constraints}^-_\sqcap(P)$	$\varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_n$	$\varphi_1,\varphi_2,\ldots\varphi_n \in \mathsf{ElemConstr}^-_\sqcap(P)$
$\mathsf{Constraints}^-_\sqcup(P)$	$\varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_n$	$\varphi_1,\varphi_2,\ldots\varphi_n \in \mathsf{ElemConstr}^-_\sqcup(P)$
$\mathsf{Constraints}^-(P)$	$\varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_n$	$\varphi_1,\varphi_2,\ldots\varphi_n \in \mathsf{ElemConstr}^-(P)$
$\mathsf{Constraints}(P)$	φ	$\varphi \in \mathsf{Constraints}^+(P) \cup \mathsf{Constraints}^-(P)$

Table 1: Definitions of constraint sets over the set P.

any explicit specification involving $\bigsqcap\{\tau_1,\tau_2\}$, $\bigsqcap\{\tau_1,\tau_3\}$, or $\bigsqcap\{\tau_2,\tau_3\}$.

The *positive meet constraints* (resp. *positive join constraints), resp. positive constraints* on P, denoted $\mathsf{Constraints}^+_\sqcap(P)$, (resp. $\mathsf{Constraints}^+_\sqcup(P)$, resp.] $\mathsf{Constraints}^+(P)$), are built up from the elementary positive meet constraints (resp. positive join constraints, resp. positive constraints) using the usual logical connective of conjunction.

To every positive constraint φ there is a corresponding negative constraint, denoted $\overline{\varphi}$. If φ is an elementary positive constraint, then $\overline{\varphi}$ is just $(\neg\varphi)$. If $\varphi = \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$ is a general positive constraint, then $\overline{\varphi} = (\neg\varphi_1) \vee (\neg\varphi_2) \vee \ldots \vee (\neg\varphi_n)$. Logically, $\overline{\varphi}$ is always equivalent to $(\neg\varphi)$; however, it is furthermore required to have the specific syntactic form shown. This association of positive constraints to negative is clearly injective; the inverse operator is also denoted by overbar. Thus, $\overline{\overline{\varphi}} = \varphi$, in a true syntactic sense (as well as a natural semantic sense to be formalized shortly).

Let C be the name denoting any of the sets of positive constraints; *i.e.*, any of the first six elements in the first column of Table 1. Then the name for $\{\overline{\varphi} \mid \varphi \in C\}$ is obtained by replacing the superscript "$+$" with "$-$" in the name C. The name itself is obtained in the obvious way, by replacing "positive" with "negative." Thus, $\mathsf{ElemConstr}^-_\sqcup$ is the set of all *elementary negative join constraints*.

The set of *constraints* on P is just $\mathsf{Constraints}(P) = \mathsf{Constraints}^+(P) \cup \mathsf{Constraints}^-(P)$. If Φ is a set of constraints, Φ^+ denotes the positive constraints in Φ, while Φ^- denotes the negative. Clearly, $\Phi = \Phi^+ \cup \Phi^-$, with the union disjoint.

Table 2 shows abbreviations for some common constraints. The first entry is a positive meet constraint (in $\mathsf{Constraints}^+_\sqcap(P)$) , the second is a positive join constraint (in $\mathsf{Constraints}^+_\sqcup(P)$), the third is both a positive meet constraint and a positive join constraint, while the fourth is a negative constraint which is in both $\mathsf{Constraints}^{-}_{\sqcap}(P)$ and $\mathsf{Constraints}^{-}_{\sqcup}(P)$. None of these constraints is elementary. The last entry, $(\tau_1 < \tau_2)$, must be regarded as two distinct constraints, the first positive and the second negative.

Finally, there are a few notational items. For any constraint φ, $\mathsf{Length}(\varphi)$ denotes the number of symbols in the string representing φ (with each type assumed to have a length of one). $\mathsf{Count}_{\vee}(\varphi)$ denotes the number of disjuncts in φ, and is only nontrivial in the case of non-elementary negative constraints. For a set Φ of formulas, $\mathsf{Count}_{\vee}(\Phi) = \sum_{\varphi\in\Phi} \mathsf{Count}_{\vee}(\varphi)$. Similarly, $\mathsf{Disjuncts}(\varphi)$ denotes the set of disjuncts of a constraint φ.

Abbreviation	**Full Formula(s)**
$(\bigsqcap\{\tau_1,\tau_2,\ldots,\tau_n\} = \tau)$	$(\bigsqcap\{\tau_1,\tau_2,\ldots,\tau_n\} \le \tau)\wedge(\tau \le \tau_1)\wedge(\tau \le \tau_2)\wedge..\wedge(\tau \le \tau_n)$
$(\tau = \bigsqcup\{\tau_1,\tau_2,\ldots,\tau_n\})$	$(\tau \le \bigsqcup\{\tau_1,\tau_2,\ldots,\tau_n\})\wedge(\tau_1 \le \tau)\wedge(\tau_2 \le \tau)\wedge\ldots\wedge(\tau_n \le \tau)$
$(\tau_1 = \tau_2)$	$(\tau_1 \le \tau_2)\wedge(\tau_2 \le \tau_1)$
$(\tau_1 \ne \tau_2)$	$(\neg(\tau_1 \le \tau_2))\vee(\neg(\tau_2 \le \tau_1))$
$(\tau_1 < \tau_2)$	$(\tau_1 \le \tau_2), (\neg(\tau_2 \le \tau_1))$

Table 2: Common abbreviations for constraints.

2.2 Specifications and interpretations An *open specification* is a pair (P,Φ), in which P is a finite clean set and $\Phi \subseteq \mathsf{Constraints}(P)$. The adjectives *positive*, *negative*, and *elementary* are used precisely when every element of Φ has the associated property.

An *interpretation* for P is a pair $M = (\mathfrak{U},\mathfrak{I})$, in which $\mathfrak{U}$ is a finite nonempty set, called the *universe of objects*, and $\mathfrak{I} : \mathsf{Aug}(P) \to \mathbf{2}^{\mathfrak{U}}$ is a function which associates a subset of $\mathfrak{U}$ to each type in $\mathsf{Aug}(P)$, subject to the conditions that $\mathfrak{I}(\top) = \mathfrak{U}$ and $\mathfrak{I}(\bot) = \emptyset$. The constraint $(\bigsqcap\{\tau_1,\tau_2,..,\tau_n\} \le \tau)$ is *satisfied* by M if $\mathfrak{I}(\tau_i)\cap\mathfrak{I}(\tau_2)\cap..\cap\mathfrak{I}(\tau_n) \subseteq \mathfrak{I}(\tau)$. Similarly, $(\tau \le \bigsqcup\{\tau_1,\tau_2,..,\tau_n\})$ is *satisfied* by M if $\mathfrak{I}(\tau) \subseteq \mathfrak{I}(\tau_i)\cup\mathfrak{I}(\tau_2)\cup..\cup\mathfrak{I}(\tau_n)$. Satisfaction for other constraints is obtained by applying these definitions to their representations in the obvious way. More precisely, $\varphi_1\wedge\varphi_2\wedge\ldots\wedge\varphi_n$ is satisfied by M if each φ_i, $1 \le i \le n$, is satisfied by M; $\varphi_1\vee\varphi_2\vee\ldots\vee\varphi_n$ is satisfied by M if at least one φ_i, $1 \le i \le n$, is satisfied by M; and $(\neg\varphi)$ is satisfied by M if φ is not satisfied by M. In general, if φ is a constraint and M is an interpretation, then $M \models \varphi$ denotes that φ is satisfied by M. If Φ is a set of constraints, $M \models \Phi$ holds iff $M \models \varphi$ for each $\varphi \in \Phi$. A *model* of (P,Φ) is an interpretation M for P for which $M \models \varphi$ holds for each $\varphi \in \Phi$.

Let $M = (\mathfrak{U},\mathfrak{I})$ be an interpretation for P. For each nonempty subset $\mathfrak{V} \subseteq \mathfrak{U}$, define the $\mathfrak{V}$-*projection* of M to be $M_{|\mathfrak{V}} = (\mathfrak{V},\mathfrak{I}_{|\mathfrak{V}})$, with $\mathfrak{I}_{|\mathfrak{V}} : \mathsf{Aug}(P) \to \mathbf{2}^{\mathfrak{V}}$ given by $S \mapsto \mathfrak{I}(S)\cap\mathfrak{V}$. Conversely, let $M_i = (\mathfrak{U}_i,\mathfrak{I}_i)$ be interpretations for P for $i = 1,2$. Assume further that $\mathfrak{U}_1\cap\mathfrak{U}_2 = \emptyset$. The *product interpretation* $M_1 \times M_2$ is given by $(\mathfrak{U}_1\cup\mathfrak{U}_2,\mathfrak{I}_1\times\mathfrak{I}_2)$, with $\mathfrak{I}_1\times\mathfrak{I}_2 : \mathsf{Aug}(P) \to \mathbf{2}^{\mathfrak{U}_1\cup\mathfrak{U}_2}$ given by $S \mapsto \mathfrak{I}_1(S)\cup\mathfrak{I}_2(S)$. This definition extends easily to any finite number of interpretations. Note that, for any interpretation $M = (\mathfrak{U},\mathfrak{I})$, $\prod_{a\in\mathfrak{U}} M_{|\{a\}}$ is just M, up to a renaming of $\mathfrak{I}$.

The *size* of an interpretation $M = (\mathfrak{U}, \mathfrak{I})$ is the cardinality of $\mathfrak{U}$; an interpretation of size m is often called an *m-element interpretation*.

2.3 Theorem — Characterization of models *Let (P,Φ) be an open specification, and let $(\mathfrak{U}, \mathfrak{I})$ be an interpretation for P. Then M is a model for (P,Φ) iff the following two conditions are satisfied.*

(a) *For each $\varphi \in \Phi^+$ and every $a \in \mathfrak{U}$, $M_{|\{a\}} \models \varphi$.*

(b) *For each $\varphi \in \Phi^-$, there is an $a \in \mathfrak{U}$ with the property that $M_{|\{a\}} \models \varphi$.*

PROOF: For $\Phi \subseteq \mathsf{ElemConstr}^+(P)$, the result is immediate, since the basic set operations involved ($\cup$, $\cap$, and $\subseteq$) are all defined pointwise. For $\varphi = \varphi_1 \wedge \varphi_2 \wedge \ldots \varphi_n \in \mathsf{Constraints}^+(P)$, with each $\varphi_i \in \mathsf{ElemConstr}^+(P)$, it suffices to note that $M \models \varphi$ iff $M \models \varphi_i$ for each i, and then to replace Φ with $(\Phi \cup \{\varphi_1, \varphi_2, \ldots, \varphi_n\}) \setminus \{\varphi\}$. Thus, the results holds whenever $\Phi \subseteq \mathsf{Constraints}^+(P)$.

For $(\neg\varphi) \in \mathsf{ElemConstr}^-(P)$, $(\neg\varphi) \models M$ iff $M \not\models \varphi$. However, in view of the argument of the previous paragraph, $M \not\models \varphi$ holds iff $M_{|\{a\}} \not\models \varphi$ holds for some $a \in \mathfrak{U}$. Thus, the result holds whenever $\Phi \subseteq \mathsf{ElemConstr}^-(P)$. For $\varphi = (\neg\varphi_1) \vee (\neg\varphi_2) \vee \ldots \vee (\neg\varphi_n) \in \mathsf{Constraints}^-(P)$, $M \models \varphi$ iff $M \models (\neg\varphi_i)$ holds for some i, which is true iff $M \not\models \varphi_i$ holds for some i, which in turn holds iff $M_{|\{a\}} \not\models \varphi_i$ for some $a \in \mathfrak{U}$. Thus, the result holds for $\Phi \subseteq \mathsf{Constraints}^-(P)$ as well. Now, since $\Phi^+ = \Phi \cap \mathsf{Constraints}^+(P)$, and $\Phi^- = \Phi \cap \mathsf{Constraints}^-(\mathrm{P})$, to obtain the result for an arbitrary Φ it suffices to combine the results for $\mathsf{Constraints}^+(P)$ and $\mathsf{Constraints}^-(P)$. $\square$

2.4 Corollary — Adequacy of one-element models *Let (P,Φ) be an open specification for which Φ^- contains at most one element. In other words, assume that all of the constraints in Φ, except possibly one, are positive. Then (P,Φ) has a model iff it has a one-element model.*

PROOF: Let M be any model of (P,Φ). In view of 2.3(a), for every $a \in \mathfrak{U}$, $M_{|a}$ is a model of Φ^+. Thus, if $\Phi^- = \emptyset$, $M_{|a}$ is a model of Φ for every $a \in \mathfrak{U}$. On the other hand, if $\Phi^- \neq \emptyset$, then by assumption it is a singleton; *i.e.*, $\Phi^- = \{\varphi\}$ for some $\varphi \in \mathsf{Constraints}^-(P)$. In view of 2.3(b), there is an $a \in \mathfrak{U}$ for which $M_{|a}$ is a model of φ. By 2.3(a), $M_{|a}$ is also a model of every member of Φ^+, whence it is a model of Φ, as required. Thus, if Φ has a model, then it has a one-element model.

The converse is trivial. $\square$

2.5 Examples Let $P_0 = \{\kappa_1, \kappa_2, \kappa_3, \kappa_4, \kappa_5, \kappa_6, \kappa_7, \kappa_8, \kappa_9\}$. This set will be used as the base types for three different examples, presented below. Each of these examples is consistent with the diagram of Figure 1, although none is completely represented by it. In all cases, the constraints are represented formally; the diagram is just an incomplete visual approximation of them.

First, let $\Phi_0 = \{(\bigsqcap\{\kappa_1, \kappa_2\} = \kappa_5), (\bigsqcap\{\kappa_2, \kappa_3\} = \kappa_6), (\bigsqcap\{\kappa_1, \kappa_3\} = \kappa_4), (\bigsqcap\{\kappa_3, \kappa_9\} = \bot), (\kappa_1 \leq \kappa_8), (\kappa_2 \leq \kappa_8), (\kappa_3 \leq \kappa_8), (\kappa_4 \leq \kappa_5), (\kappa_7 \leq \kappa_4), (\kappa_7 \leq \kappa_6)\}$. All of the constraints in Φ_0 are positive. Thus, in view of 2.3, if it has a model, (P_0, Φ_0) has a one-element model. Indeed, one may define $M_{00} = (\{a_0\}, \mathfrak{I}_{00})$ with $\mathfrak{I}_{00}(\tau) = \emptyset$ for all $\tau \in \mathsf{Aug}(P) \setminus \{\top\}$, and $\mathfrak{I}_{00}(\top) = \{a_0\}$. Given a set of positive constraints which does not include any (equivalents of) joins of the form $(\bigsqcup S = \top)$, it is always possible to assign the empty set to every type in P. Of course, other models are also possible. For example, one may define $M_{01} = (\{a_0\}, \mathfrak{I}_{01})$ with $\mathfrak{I}_{00}(\tau) = \emptyset$ for $\tau \in \{\bot, \kappa_7\}$, and $\mathfrak{I}_{00}(\tau) = \{a_0\}$ otherwise. Many other, similar models are possible, since there are no constraints which force elements to be distinct (other than $\bot$ and $\top$).

Next, some distinctness constraints are added. Let $\Phi_1 = \Phi_0 \cup \{(\kappa_1 \neq \kappa_2),\ (\kappa_1 \neq \kappa_3),\ (\kappa_2 \neq \kappa_3)\}$. It is no longer possible to construct a one-element model. Indeed, since κ_1, κ_2, and κ_3 must have distinct values, a domain with at least two elements is required. First of all, define $M_{11} = (\{a_1\}, \mathfrak{I}_{11})$, with $\mathfrak{I}_{11}(\kappa_1) = \mathfrak{I}_{11}(\kappa_8) = \{a_1\}$, and $\mathfrak{I}_{11}(\tau) = \emptyset$ otherwise. Likewise, define $M_{12} = (\{a_2\}, \mathfrak{I}_{12})$, with $\mathfrak{I}_{12}(\kappa_2) = \mathfrak{I}(\kappa_8) = \{a_2\}$, and $\mathfrak{I}_{11}(\tau) = \emptyset$ otherwise. It is easy to see that both M_{11} and M_{12} are models of Φ_0. Furthermore, M_{11} is a model of $(\kappa_1 \neq \kappa_2)$ and $(\kappa_1 \neq \kappa_3)$, while M_{12} is a model of $(\kappa_1 \neq \kappa_2)$ and $(\kappa_2 \neq \kappa_3)$. Thus, in view of 2.3, $M_{11} \times M_{12}$ is a model of Φ_1. Of course, for symmetry, it is possible to add a third model $M_{13} = (\{a_3\}, \mathfrak{I}_{13})$, with $\mathfrak{I}_{12}(\kappa_3) = \mathfrak{I}(\kappa_8) = \{a_3\}$, and $\mathfrak{I}_{11}(\tau) = \emptyset$ otherwise. However, the product of just two models suffices. In general, if k elements are required to be distinct, a model with at least $\lceil \log_2(k) \rceil$ elements is required.

Now, suppose that $(\neg(\kappa_4 \leq \kappa_6))$ is specified as well; let $\Phi_2 = \Phi_0 \cup \{(\neg(\kappa_4 \leq \kappa_6))\}$. Since $\kappa_4 = \kappa_4 \sqcap \kappa_5 = (\kappa_1 \sqcap \kappa_3) \sqcap (\kappa_1 \sqcap \kappa_2) = \kappa_1 \sqcap \kappa_2 \sqcap \kappa_3 \leq \kappa_2 \sqcap \kappa_3 = \kappa_6$, it follows that $\Phi_1 \models (\kappa_4 \leq \kappa_6)$. Thus, Φ_2 is unsatisfiable.

To close this section, the basic complexity results for questions of satisfiability are recalled. The proof of 2.7 may be found in [17].

2.6 Important problems on open specifications Let $\mathbf{S} = (P, \Phi)$ be an open specification.

Satisfiability: Does $\mathbf{S}$ have a model? In other words, does there exist an interpretation M such that $M \models \Phi$?

Query solution: Given $\varphi \in \mathsf{Constraints}(P)$, does $\Phi \models \varphi$ hold?

2.7 Theorem — complexity results Let $\mathbf{S} = (P, \Phi)$ be an open specification.

(a) The satisfiability problem for $\mathbf{S}$ is NP-complete.

(b) The query solution problem for $\mathbf{S}$ is co-NP-complete.

These results remain valid even when the context is limited to positive constraints. The size of an instance is the length of the associated formula; that is, $\mathsf{Length}(\Phi)$ for the satisfiability problem for $\mathbf{S}$ and $\mathsf{Length}(\Phi \cup \{\varphi\})$ for the query solution problem. □

Thus, the problems of satisfiability and query solution do not admit tractable solutions unless $P = NP$ [15]. In the following two sections, a more restrictive but nonetheless useful framework is identified for which these problems admit solutions which are no worse than deterministically quadratic.

3. Translation to Propositional Logic

In this section, it is shown how to translate constraints into formulas in propositional logic, so that well-known inference algorithms within that context may be brought to bear on the problem of inference. The basic ideas appear in [18], although the direction which they are taken is quite different in that paper.

3.1 One-element models and propositional logic Let P be a set of clean types. Define the propositional logic $\mathcal{L}_P$ to have as proposition symbols the set $\{\mathfrak{r}_\tau \mid \tau \in \mathsf{Aug}(P)\}$. Also, within this context, **false** and **true** will be used to denote special propositions which always have the values false and true, respectively. For $\varphi \in \mathsf{ElemConstr}^+(P)$, associate a formula $\mathfrak{F}(\varphi)$ according to Table 3.

Constraint φ	Associated Logical Formula $\mathfrak{F}(\varphi)$
$(\bigsqcap\{\tau_1, \tau_2, .., \tau_n\} \leq \tau)$	$(\mathfrak{r}_{\tau_1} \wedge \mathfrak{r}_{\tau_2} \wedge \ldots \wedge \mathfrak{r}_{\tau_n}) \Rightarrow \mathfrak{r}_\tau$
$(\tau \leq \bigsqcup\{\tau_1, \tau_2, .., \tau_n\})$	$\mathfrak{r}_\tau \Rightarrow (\mathfrak{r}_{\tau_1} \vee \mathfrak{r}_{\tau_2} \vee \ldots \vee \mathfrak{r}_{\tau_n})$

Table 3: Translation from constraints to propositional logic.

The definition of $\mathfrak{F}(\varphi)$ for a formula $\varphi \in \mathsf{Constraints}(P)$ is the obvious extension of that in Table 3. More precisely, $\mathfrak{F}(\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n) = \mathfrak{F}(\varphi_1) \wedge \mathfrak{F}(\varphi_2) \wedge \ldots \wedge \mathfrak{F}(\varphi_n)$, $\mathfrak{F}(\varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_n) = \mathfrak{F}(\varphi_1) \vee \mathfrak{F}(\varphi_2) \vee \ldots \vee \mathfrak{F}(\varphi_n)$, and $\mathfrak{F}((\neg\varphi)) = (\neg\mathfrak{F}(\varphi))$.

For a set $\Phi \subseteq \mathsf{Constraints}(P)$, define $\mathfrak{F}(\Phi) = \{\mathfrak{F}(\varphi) \mid \varphi \in \Phi\}$. Further, for $\varphi \in \mathsf{Constraints}(P)$, define $\overline{\mathfrak{F}(\varphi)} = \mathfrak{F}(\overline{\varphi})$ and $\mathsf{Disjuncts}(\mathfrak{F}(\varphi)) = \{\mathfrak{F}(\psi) \mid \psi \in \mathsf{Disjuncts}(\varphi)\}$. $\mathsf{Length}(\mathfrak{F}(\varphi))$ is defined to be the actual length of the string. It is important to note that $O(\mathsf{Length}(\varphi)) = O(\mathsf{Length}(\mathfrak{F}(\varphi)))$, although the actual lengths may differ.

There is a natural association of semantics corresponding to this syntactic pairing. Let $M = (\{a\}, \mathfrak{I})$ be any one-element semantics for P. For $\mathcal{L}_P$, define the interpretation $\mathfrak{F}(M)$ to be true on the propositions in the set $\{\mathfrak{r}_\tau \in \mathsf{Aug}(P) \mid \mathfrak{I}(\tau) = \{a\}\}$, and false otherwise. The following result is immediate; however, note carefully that it applies only to one-element models. More complex models must be realized as products, as described further on in this section.

3.2 Characterization of one-element models Let (P, Φ) be an open specification, and let $M = (\{a\}, \mathfrak{I})$ be a one-element semantics for P. Then M is a model of (P, Φ) iff $\mathfrak{F}(M)$ is a model of $\mathfrak{F}(\Phi)$. $\square$

3.3 Examples The translations to propositional logic of the three sets of 2.5 are now presented.

First, $\mathfrak{F}(\Phi_0) = \{((\mathfrak{r}_{\kappa_1} \wedge \mathfrak{r}_{\kappa_2}) \Rightarrow \mathfrak{r}_{\kappa_5}),\ (\mathfrak{r}_{\kappa_5} \Rightarrow \mathfrak{r}_{\kappa_1}),\ (\mathfrak{r}_{\kappa_5} \Rightarrow \mathfrak{r}_{\kappa_2}),\ ((\mathfrak{r}_{\kappa_2} \wedge \mathfrak{r}_{\kappa_3}) \Rightarrow \mathfrak{r}_{\kappa_6}),\ (\mathfrak{r}_{\kappa_6} \Rightarrow \mathfrak{r}_{\kappa_2}),\ (\mathfrak{r}_{\kappa_6} \Rightarrow \mathfrak{r}_{\kappa_3}),\ ((\mathfrak{r}_{\kappa_1} \wedge \mathfrak{r}_{\kappa_3}) \Rightarrow \mathfrak{r}_{\kappa_4}),\ (\mathfrak{r}_{\kappa_4} \Rightarrow \mathfrak{r}_{\kappa_1}),\ (\mathfrak{r}_{\kappa_4} \Rightarrow \mathfrak{r}_{\kappa_3}),\ ((\mathfrak{r}_{\kappa_3} \wedge \mathfrak{r}_{\kappa_9}) \Rightarrow \textbf{false}),\ (\mathfrak{r}_{\kappa_1} \Rightarrow \mathfrak{r}_{\kappa_8}),\ (\mathfrak{r}_{\kappa_2} \Rightarrow \mathfrak{r}_{\kappa_8}),\ (\mathfrak{r}_{\kappa_3} \Rightarrow \mathfrak{r}_{\kappa_8}),\ (\mathfrak{r}_{\kappa_4} \Rightarrow \mathfrak{r}_{\kappa_5}),\ (\mathfrak{r}_{\kappa_7} \Rightarrow \mathfrak{r}_{\kappa_4}),\ (\mathfrak{r}_{\kappa_7} \Rightarrow \mathfrak{r}_{\kappa_6})\}$. A few notes are in order. First of all, constraints are written in atomic form. Thus, the translation of $(\bigsqcap\{\kappa_1, \kappa_2\} = \kappa_5)$ is expressed as the set $\{((\mathfrak{r}_{\kappa_1} \wedge \mathfrak{r}_{\kappa_2}) \Rightarrow \mathfrak{r}_{\kappa_5}),\ (\mathfrak{r}_{\kappa_5} \Rightarrow \mathfrak{r}_{\kappa_1}),\ (\mathfrak{r}_{\kappa_5} \Rightarrow \mathfrak{r}_{\kappa_2})\}$. It could just as well have been expressed with the single formula $((\mathfrak{r}_{\kappa_1} \wedge \mathfrak{r}_{\kappa_2}) \Rightarrow \mathfrak{r}_{\kappa_5}) \wedge (\mathfrak{r}_{\kappa_5} \Rightarrow \mathfrak{r}_{\kappa_1}) \wedge (\mathfrak{r}_{\kappa_5} \Rightarrow \mathfrak{r}_{\kappa_2})$. Second, the constraint $(\bigsqcap\{\kappa_3, \kappa_9\} = \bot)$ is represented by $((\mathfrak{r}_{\kappa_3} \wedge \mathfrak{r}_{\kappa_9}) \Rightarrow \textbf{false})$. Strictly following the translation rules, the formulas $(\textbf{false} \Rightarrow \mathfrak{r}_{\kappa_3})$ and $(\textbf{false} \Rightarrow \mathfrak{r}_{\kappa_7})$ should be included as well, but each is trivially true, and so omitted.

$\mathfrak{F}(\Phi_1) = \mathfrak{F}(\Phi_0) \cup \{(\neg(\mathfrak{r}_{\kappa_1} \Rightarrow \mathfrak{r}_{\kappa_2})) \vee (\neg(\mathfrak{r}_{\kappa_2} \Rightarrow \mathfrak{r}_{\kappa_1})),\ (\neg(\mathfrak{r}_{\kappa_1} \Rightarrow \mathfrak{r}_{\kappa_3})) \vee (\neg(\mathfrak{r}_{\kappa_3} \Rightarrow \mathfrak{r}_{\kappa_1})),\ (\neg(\mathfrak{r}_{\kappa_2} \Rightarrow \mathfrak{r}_{\kappa_3})) \vee (\neg(\mathfrak{r}_{\kappa_3} \Rightarrow \mathfrak{r}_{\kappa_2}))\}$, which may be written as $\{(\mathfrak{r}_{\kappa_1} \wedge (\neg\mathfrak{r}_{\kappa_2})) \vee (\mathfrak{r}_{\kappa_2} \wedge (\neg\mathfrak{r}_{\kappa_1})),\ (\mathfrak{r}_{\kappa_1} \wedge (\neg\mathfrak{r}_{\kappa_3})) \vee (\mathfrak{r}_{\kappa_3} \wedge (\neg\mathfrak{r}_{\kappa_1})),\ (\mathfrak{r}_{\kappa_2} \wedge (\neg\mathfrak{r}_{\kappa_3})) \vee (\mathfrak{r}_{\kappa_3} \wedge (\neg\mathfrak{r}_{\kappa_2}))\}$.

$\mathfrak{F}(\Phi_2) = \mathfrak{F}(\Phi_0) \cup \{(\neg(\mathfrak{r}_{\kappa_4} \Rightarrow \mathfrak{r}_{\kappa_6}))\}$, which is equivalent to $\mathfrak{F}(\Phi_0) \cup \{\mathfrak{r}_{\kappa_4} \wedge (\neg\mathfrak{r}_{\kappa_6})\}$. While this is also logically equivalent to $\mathfrak{F}(\Phi_0) \cup \{\mathfrak{r}_{\kappa_4}, (\neg\mathfrak{r}_{\kappa_6})\}$, it is not appropriate to break up $\mathfrak{r}_{\kappa_4} \wedge (\neg\mathfrak{r}_{\kappa_6})$ into two separate formulas. The reason is that the algorithms to follow which build models for the corresponding constraints are based upon the ideas of 2.3. For a negative constraint to hold, the entire constraint must hold on a single one-element projection of the model. It is not adequate to have each conjunct hold on a different projection.

3.4 A meta-algorithm for constructing models of open specifications In Figure 2, a meta-algorithm for constructing models of open specifications is presented. It is a "meta-algorithm" because it does not elaborate on data structures or upon how certain subroutines are implemented.

```
/* Construct a model for the open specification (P,Φ),
   or report failure if no such model is possible. /*
Success ← true;
Pos_constraints ← {𝔉(φ) | φ ∈ Constraints⁺(Φ)};
Pool ← {𝔉(φ) | φ ∈ Constraints⁻(Φ)};
Model ← ∅;
if (Pool = ∅)
  then
    Model ← Build_model(Pos_constraints,true);
endif;
while (Success and (Pool ≠ ∅)) do
  Neg_constraint ← Select_and_delete(Pool);
  Local_model ← Build_model(Pos_constraints,Neg_constraint});
  if (Local_model ≠ ∅)
  then
     Model ← Combine_models(Model,Local_model);
  else
     Success ← false;
  endif
endwhile;
```

Figure 2: The meta-algorithm for constructing models.

The main idea is to combine the characterization of models presented in 2.3 with the translation of 3.1, in realization of a strategy which operates within propositional logic. The control structure is completely straightforward. Given an open specification (P,Φ), the variable `Pos_constraints` is set to the propositional representation of the associated set Φ^+ of positive constraints, while `Pool` is set to the propositional representation of the set Φ^- of negative constraints. If `Pool` is empty initially, a model of `Pos_constraints` is the only goal. Otherwise, elements are removed nondeterministically, one-by-one, from `Pool` into the variable `Neg_constraint`, and a model of each such set of the form `Pos_constraints` $\cup$ {`Neg_constraint`} is sought. The product of these models is the final result.

The description of a few details is in order. The function `Build_model` does what its names conveys. It takes a formula in the propositional logic $\mathcal{L}_P$, and produces a model. For convenience, the constraints are given as two arguments, one a set of positive constraints, and the other a single negative one. In the next section, a detailed expansion of this procedure for certain special classes of constraints will be provided. For now, the details are left open.

The call `Combine_models(Model,Local_model)` computes the product of `Model` and `Local_model`, as defined in 2.2, provided `Model` is not $\emptyset$. If `Model` $= \emptyset$, then the call returns `Local_model`.

The meta-algorithm is a bit naïve in that it computes a model for each negative constraint. Thus, for example, on the set Φ_1 of 2.5 and 3.3, it will compute and combine three one-element models, even though a solution consisting of only two such models is possible, as illustrated in 2.5. To avoid computing such extra models, a strategy of testing and re-use is necessary.

The dominant component of this meta-algorithm, in terms of time complexity, is line 14. In the general case, to build a model of the sentence presented to the procedure will require time which is exponential in the size of the formula. However, by judiciously choosing the type of formulas which are considered in this step, the complexity may be reduced dramatically. It is to this point that the discussion now turns.

4. A Computationally Tractable Class of Open Specifications

The results of 2.7 indicate that the management of general, open type hierarchies is likely to be inherently intractable computationally. This does not mean, of course, that no such management is possible. Rather, it indicates that compromises need to be made in certain directions. One such direction was developed in [18], in which full management was eschewed in favor of the ability to handle certain classes of queries efficiently. In this work, a different direction is taken, in which the class of open specifications itself is pared back, with the result that all queries may be supported in a tractable fashion.

4.1 Horn clauses and Horn sentences In a propositional logic, a *Horn clause* is one in which at most one of the literals is positive. Thus, the most general form of such a clause is $\neg p_1 \vee \neg p_2 \vee .. \vee \neg p_n \vee q$, with the p_i's and q propositions from the extant logic. Horn clauses are of central importance in computer science because, on the one hand, they are useful in the formal representation of and inference upon a wide variety of important concepts [20], and, on the other hand, satisfiability of a set of such sentences may be determined in time linear in the size of the formulas [11].

There are four classes of Horn clauses. If there are no negative literals (*i.e.*, $n = 0$), the Horn clause is called a *fact*, and written as simply p. If there are both negative and positive literals, the above clause is often in *implication form*, as $p_1 \wedge p_2 \wedge .. \wedge p_n \Rightarrow q$, and is called a *rule*. If there are no positive literals, the clause $\neg p_1 \vee \neg p_2 \vee .. \vee \neg p_n$ is often re-written as $p_1 \wedge p_2 \wedge .. \wedge p_n \Rightarrow$ **false**, and is called a *constraint*. Finally, the empty clause (which is always false) is also a Horn clause, and is written **false**.

A conjunction of Horn clauses is called a *Horn sentence*, and certain abbreviations are used in this regard. Specifically, a formula of the form $(p_1 \wedge p_2 \wedge \ldots \wedge p_n) \Rightarrow (q_1 \wedge q_2 \wedge \ldots \wedge q_m)$ is to be regarded as an abbreviation for the conjunction $((p_1 \wedge p_2 \wedge \ldots \wedge p_n) \Rightarrow q_1) \wedge ((p_1 \wedge p_2 \wedge \ldots \wedge p_n) \Rightarrow q_2) \wedge \ldots \wedge ((p_1 \wedge p_2 \wedge \ldots \wedge p_n) \Rightarrow q_m)$. In this case, the p_i's are called the *antecedents*, and the q_i's the *consequents*, of the sentence.

For any set Ψ of Horn sentences, define $\mathsf{Facts}(\Psi)$ to be the set of all propositions which are true in every model of Ψ. Define $\mathsf{MinMod}(\Psi)$ to be the interpretation which is true on those propositions in $\mathsf{Facts}(\Psi)$, and false otherwise. It is a remarkable property of Horn sentences that if Ψ is satisfiable, then $\mathsf{MinMod}(\Psi)$ is a model of Ψ.

The idea behind the model-building algorithm is extremely simple, and in fact resembles classical rule algorithms for expert systems [5]. Let `F` denote a variable which is to be set at completion to $\mathsf{Facts}(\Psi)$ if Ψ is satisfiable; otherwise, failure will be flagged. One begins by setting `F` to the set of clauses in Ψ which are facts, and and "fires" those rules, all of whose antecedents lie in `F`. The consequents of the fired rules are added to `F`, and the process continues. If any constraint fires, there is no model. Otherwise, $\mathtt{F} = \mathsf{Facts}(\Psi)$ when the process terminates. The nontrivial aspect is to employ data and control structures which render this process computable in linear time. For details, the reader is referred to the original paper [11], or to the report [16, Sec. 3], which contains an exposition of this algorithm for propositional logic as background to an extended version for feature structures.

A *Horn disjunction* is a sentence of the form $\psi_1 \vee \psi_2 \vee \ldots \vee \psi_n$, with each ψ_i a Horn sentence. Since any formula which is in disjunctive normal form (DNF) [12, pp. 48-49] is a Horn disjunction, any sentence is equivalent to a Horn disjunction. Nonetheless, there exists an algorithm which determines whether the formula is satisfiable, and builds a model if one exists, which has the same time complexity as the algorithm for Horn sentences.

4.2 Proposition — Complexity of satisfiability for disjunctions *Let $\psi = \psi_1 \vee \psi_2 \vee \ldots \vee \psi_n$ be any disjunction of Horn sentences in propositional logic. The time required to test ψ for satisfiability is bounded by $O(\mathsf{Length}(\psi))$. In the case that ψ is satisfiable, a model may be constructed within these same time bounds.*

PROOF: One simply applies the Horn satisfiability test and minimal model construction to the ψ_i's, in turn, until one is found which is true, or until all have been tested. □

Note that while the algorithm for Horn disjunctions has the same complexity as that for Horn sentences, there is no corresponding notion of a least model ($\mathsf{MinMod}(\{\psi\})$), since one may need to chose among several alternatives.

4.3 Horn constraints The members of $\mathsf{Constraints}(P)$ which correspond to Horn sentences and Horn disjunctions are of central importance. Formally, let P be a set of clean types. $\varphi \in \mathsf{Constraints}(P)$ is called a *simple Horn constraint* if $\mathfrak{F}(\varphi)$ is a Horn sentence in the logic $\mathcal{L}_P$, and a *disjunctive Horn constraint* if $\mathfrak{F}(\varphi)$ is a Horn disjunction in the logic $\mathcal{L}_P$. Strictly speaking, every simple Horn constraint φ is a disjunctive Horn constraint, since φ may be viewed as the disjunction of exactly one Horn sentence. However, to avoid any confusion, the term *generalized Horn constraint* will be used to identify any constraint which is either simple Horn or disjunctive Horn.

In 2.5 and 3.3, Φ_0 is a set of simple Horn constraints, since there is no negation involved. Φ_1 is a set of generalized Horn constraints; the translations of the inequality constraints are disjunctive Horn sentences, but not ordinary Horn sentences. As for Φ_2, since the translation of the constraint $(\neg(\tau_4 \leq \tau_6))$ is a Horn sentence, the constraint itself is simple Horn, and so Φ_2 is also a set of simple Horn constraints.

Indeed, the only members of $\mathsf{Constraints}(P)$ which are not disjunctive Horn are those which involve positive disjunction which cannot be otherwise expressed. For example, the constraint $(\kappa_1 \vee \kappa_2 = \kappa_8)$ is not a disjunctive Horn constraint. More precisely, the constraints which fail to be disjunctive Horn are those which lie in $\mathsf{Constraints}^{+}_{\sqcup}(P)$, but not in $\mathsf{Constraints}^{+}_{\sqcap}(P)$. This is formalized in the following. The proof is a simple verification, which is omitted.

4.4 Proposition — Characterization of Horn constraints *Let P be a set of clean types.*

(a) *Every element of* $\mathsf{Constraints}^{+}_{\sqcap}(P) \cup \mathsf{ElemConstr}^{-}(P)$ *is a simple Horn constraint.*

(b) *Every element of* $\mathsf{Constraints}^{+}_{\sqcap}(P) \cup \mathsf{Constraints}^{-}(P)$ *is a disjunctive Horn constraint.* $\square$

4.5 The `Build_model` procedure for Horn constraints In Figure 3, a meta-algorithm for the `Build_model` procedure of Figure 2 which is specialized to Horn constraints is presented. This procedure operates directly upon propositional formulas; the translation from constraints to such formulas is performed in the main procedure of Figure 2. The input consists of two arguments; the first is a set of Horn sentences Ψ, and the second is a single Horn disjunction $\psi_1 \vee \psi_2 \vee \ldots \vee \psi_m$. The crux of the algorithm is to decompose the Horn disjunction into its disjuncts $\{\psi_1, \psi_2, \ldots, \psi_m\}$ (Line 4), and then test iteratively $\Psi \cup \{\psi_1\}$, $\Psi \cup \{\psi_2\}$, $\ldots$, $\Psi \cup \{\psi_n\}$ for satisfiability (Lines 9-13), stopping when a satisfiable set is found. The selection of a test element is nondeterministic; the ψ_i's may be processed in any order.

```
       /* Construct a model of a Horn disjunction,
          or report failure if no such model is possible. /*
1.     Procedure Build_model (Pos_constraints: set_of Horn_sentences,
2.                            Neg_constraint:  Horn_disjunction): Model;
3.     Success ← false;
4.     Pool ← Disjuncts(Neg_constraint);
5.     if (Pool = ∅)
6.       then
7.         Model ← Build__min_model(Pos_constraints);
8.     else
9.         while ((not Success) and (Pool ≠ ∅)) do
10.          Neg_constraint ← Select_and_delete(Pool);
11.          Local_model ← Build_min_model(Pos_constraints∪{Neg_constraint});
12.          if (Local_model ≠ ∅) then Success ← true; endif;
13.        endwhile;
14.     endif;
15.   if Success then Return Local_model;
16.                    else Return false; endif;
17.   endProcedure;
```

Figure 3: Constructing minimal models of disjunctive Horn constraints.

Each $\Psi \cup \{\psi_i\}$ is a set of Horn sentences, and so the satisfaction-testing and model-building step (Line 11) may be realized in time $\mathsf{Length}(\Psi \cup \{\psi_i\})$. This, in turn, yields the following complexity result.

4.6 Proposition — Complexity *Let* (P, Φ) *be an open specification with* Φ *consisting of generalized Horn constraints. The meta-algorithm of Figure 2, using the procedure of Figure 3, may be realized with a worst-case time complexity bounded by* $O(\mathsf{Count}_{\vee}(\Phi^{-}) \cdot \mathsf{Length}(\Phi^{+}) + \mathsf{Length}(\Phi^{-}))$. *This complexity is no worse than* $O(\mathsf{Length}(\Phi)^2)$.

PROOF: Throughout this proof, it is assumed that the procedure `Build_min_model` in the procedure of Figure 3 may be executed in time $O(n)$, with n the length of the formula which is its argument. This assumption is based upon the work of [11], mentioned in 4.1 above.

If Φ is empty, there is nothing to be done. Otherwise, there are two cases to consider. If Φ^- is empty, then the procedure `Build_model` of Figure 3 is called exactly once, with $\mathfrak{F}(\Phi^+)$ its first argument, and $\emptyset$ its second. In this case, the procedure `Build_min_model` is called only once, on the single set of Horn sentences $\mathfrak{F}(\Phi^+)$. This takes time $O(\mathsf{Length}(\Phi))$. Otherwise, the main procedure (Figure 2) calls `Build_model` once for each element $\psi \in \mathfrak{F}(\Phi^-)$ as the second argument, each time with $\mathfrak{F}(\Phi^+)$ as the first argument. In view of 4.2, the time for such a call is in $O(\sum_{\psi_i \in \mathsf{Disjuncts}(\psi)}(\mathsf{Length}(\Phi) + \mathsf{Length}(\psi_i)) = O(\mathsf{Count}_\vee(\{\psi\}) \cdot \mathsf{Length}(\Phi^+) + \mathsf{Length}(\psi))$, since each such call to `Build_min_model` operates on a set of the form $\mathfrak{F}(\Phi^+ \cup \{\psi_i\})$, with $\psi_i \in \mathsf{Disjuncts}(\psi_i)$. The total time over all such calls is thus $O(\sum_{\psi\in\Phi^-}(\mathsf{Count}_\vee(\{\psi\}) \cdot \mathsf{Length}(\Phi^+) + \mathsf{Length}(\psi)) = O(\mathsf{Count}_\vee(\Phi^-) \cdot \mathsf{Length}(\Phi^+) + \mathsf{Length}(\Phi^-))$, as required. The (extremely conservative) bound $O(\mathsf{Length}(\Phi)^2)$ follows from the fact that $\mathsf{Count}_\vee(\Phi^-) \leq \mathsf{Length}(\Phi)$. □

4.7 Examples Consider once again the example constraints of 2.5 and 3.3. Φ_0 consists entirely of positive constraints, and thus `Build_model` is called only once, which in turn calls `Build_min_model` once.

Next, note that $\Phi_1^+ = \Phi_0$, while $\Phi_1^- = \{(\kappa_1 \neq \kappa_2),\ (\kappa_1 \neq \kappa_3),\ (\kappa_2 \neq \kappa_3)\}$. Thus, `Build_model` is called three times, once for each element of $\{\mathfrak{F}(\Phi_0) \cup \{\mathfrak{F}((\kappa_1 \neq \kappa_2))\},\ \mathfrak{F}(\Phi_0) \cup \{\mathfrak{F}((\kappa_1 \neq \kappa_3))\},\ \mathfrak{F}(\Phi_0) \cup \{\mathfrak{F}((\kappa_2 \neq \kappa_3))\}\}$. In turn, an element of the form $\mathfrak{F}(\kappa_i \neq \kappa_j)$ is represented as the disjunction of two Horn sentences, so each call to `Build_model` results in two calls to `Build_min_model`. According to 4.6, there can be a maximum of six calls to `Build_min_model`, since there are three constraints in Φ_1^-, and each such constraint consists of exactly two disjuncts. Thus, the algorithm of Figure 2 calls `Build_model` at most three times, and each such call involves at most two calls to `Build_min_model`. However, there will actually be only three such calls, since for each element of Φ_1^-, both disjuncts are satisfiable, and a model will be found in Line 11 of Figure 3 for whichever is chosen first.

Observe that $\Phi_2^+ = \Phi_0$ as well, while $\Phi_2^- = \{(\neg(\kappa_4 \leq \kappa_6))\}$. As noted in 3.3, $\mathfrak{F}(\Phi_2^-)$ must be written using the single formula $\{\mathfrak{r}_{\kappa_4} \wedge (\neg \mathfrak{r}_{\kappa_6})\}$. Thus, there will be only one (unsuccessful) call to `Build_model` and to `Build_min_model` in this case.

Finally, the result of 4.6 may be used to obtain much better bounds on the complexity of the satisfiability and query solution problems identified in 2.6 than are provided in the general case of 2.7.

4.8 Theorem — Tractable cases of satisfiability and query solution *Let* $\mathbf{S} = (P, \Phi)$ *be an open specification with* Φ *consisting of generalized Horn constraints.*

(a) *The question of whether* $\mathbf{S}$ *has a model is decidable in worst-case time* $O(\mathsf{Count}_\vee(\Phi^-) \cdot \mathsf{Length}(\Phi^+) + \mathsf{Length}(\Phi^-))$.

(b) *Let* $\varphi \in \mathsf{Constraints}^+(P) \cup \mathsf{Constraints}^-_\sqcap(P)$*, and set* $\Phi' = \Phi \cup \{\neg\varphi\}$. *The query which asks whether* $\Phi \models \varphi$ *holds is answerable in time* $O(\mathsf{Count}_\vee(\Phi'^-) \cdot \mathsf{Length}(\Phi'^+) + \mathsf{Length}(\Phi'^-))$.

PROOF: Part (a) follows immediately from 4.6. For part (b), the key is to observe that if $\varphi \in \mathsf{Constraints}^+(P) \cup \mathsf{Constraints}^-_\sqcap(P)$, then $\neg\varphi \in \mathsf{Constraints}^-(P) \cup \mathsf{Constraints}^+_\sqcap(P)$. This can be seen easily from the formulations in Table 1. Thus, in view of 4.4(b), $\Phi \cup \{\neg\varphi\}$ is a set of disjunctive Horn constraints, whence the result follows from 4.6. □

By using the characterization of 4.4(b) and the simplified complexity characterization in the last line of 4.6, a simplified view of the above theorem may be obtained. Indeed, NP-completeness are co-NP-completeness are replaced by a bound which is deterministically quadratic.

4.9 Corollary *Let* $\mathbf{S} = (P, \Phi)$ *be an open specification with the property that* $\Phi \subseteq \mathsf{Constraints}^{+}_{\sqcap}(P) \cup \mathsf{Constraints}^{-}(P)$.

(a) *The question of whether* $\mathbf{S}$ *has a model is decidable in worst-case time* $O(\mathsf{Length}(\Phi)^2)$.

(b) *Let* $\varphi \in \mathsf{Constraints}^{+}(P) \cup \mathsf{Constraints}^{-}_{\sqcap}(P)$. *The query which asks whether* $\Phi \models \varphi$ *holds is answerable in time* $O((\mathsf{Length}(\Phi) + \mathsf{Length}(\varphi))^2)$. □

5. Conclusions and Further Directions

An algorithm for efficient inference on a class of open specifications for type hierarchies has been presented. The constraints which are supported include all meet constraints and inequality constraints, as well as type subsumption. This class is more substantial than that which is typically supported in closed systems. Nonetheless, the complexity of the algorithm is very competitive, with a running time between linear and quadratic in the length of the specification, depending upon the exact flavor of the constraints.

A key topic for future study is that of open queries. In the system described in this paper, queries of the form $\Phi \models (\prod S = \tau)$ are supported. However, it is often desirable to compute a glb, and not just confirm or deny a conjecture. Thus, it is important to support queries of the form $\Phi \models (\prod S = x)$, with x a variable which is to be bound to a type. The obvious approach is to extend the framework of this paper to one with a basic form of quantification over types; that is, to a first-order logic. However, such a direct approach is likely to result in unacceptably high computational overhead. A more promising approach is to maintain compiled information about glb's. For closed, static hierarchies, such a technique has been proposed in [1]. One direction which is currently being developed is to employ such a strategy of precompiled information for fast computation of glb's, but with dynamic maintenance features which admit modification as constraints are added and deleted.

A further topic which is under investigation is the extension of the existing framework to at least a limited class of positive join constraints. As noted previously, including all positive constraints results in the associated problems becoming NP- or co-NP- complete. This is the case even when various natural restrictions are placed upon properties such as height and fanout in the hierarchy [17, Sec. 2.3]. Thus, current investigations are focused upon employing heuristic techniques related to fast algorithms for satisfiability [23].

References

[1] H. Aït-Kaci, R. Boyer, P. Lincoln, and R. Nasr, Efficient implementation of lattice operations, *ACM Trans. Programming Languages and Systems*, **11**(1989), 115–146.

[2] M. P. Atkinson, F. Bancilhon, D. J. DeWitt, K. R. Dittrich, D. Maier, and S. B. Zdonik, The object-oriented database system manifesto, in *Deductive and Object-Oriented Databases, Proceedings of the First International Conference on Deductive and Object-Oriented Databases (DOOD'89), Kyoto research Park, Kyoto, Japan, 4-6 December, 1989* (W. Kim, J.-M. Nicolas, and S. Nishio, eds.), pp. 223–240, North-Holland, 1990.

[3] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, eds., *The description logic handbook: Theory, implementation, and applications*, Cambridge University Press, 2003.

[4] J. Bresnan, *Lexical-Functional Syntax*, Oxford: Blackwell Publishers Ltd, 2001.

[5] L. Brownston, R. Farrell, E. Kant, and N. Martin, *Programming expert systems in OPS5: an introduction to rule-based programming*, Addison-Wesley, 1985.

[6] B. Carpenter, *The Logic of Typed Feature Structures*, Cambridge University Press, 1992.

[7] A. Copestake, *Implementing Typed Feature Structure Grammars*, CSLI Publications, 2002.

[8] M. Dalrymple, *Lexical Functional Grammar*, Academic Press, 2001.

[9] V. de Paiva, Types and constraints in the LKB, in *Inheritance, Defaults, and the Lexicon* (T. Briscoe, V. de Paiva, and A. Copestake, eds.), pp. 164–189, Cambridge University Press, 1993.

[10] J. Dörre and A. Eisele, Feature logic with disjunctive unification, in *Proceedings of the COLING 90, Volume 2*, pp. 100–105, 1990.

[11] W. F. Dowling and J. H. Gallier, Linear-time algorithms for testing the satisfiability of propositional Horn clauses, *J. Logic Programming*, **3**(1984), 267–284.

[12] H. B. Enderton, *A Mathematical Introduction to Logic*, Academic Press, 1972.

[13] A. Formica, H. D. Groger, and M. Missikoff, An efficient method for checking object oriented database schema correctness, *ACM Trans. Database Systems*, **23**(1998), 333–369.

[14] B. Ganter and R. Wille, *Formal Concept Analysis*, Springer-Verlag, 1999.

[15] M. R. Garey and D. S. Johnson, *Computers and Intractability*, W. H. Freeman, 1979.

[16] S. J. Hegner, Horn clauses and feature-structure logic: Principles and unification algorithms, LLI Technical Report No. 1 , University of Oslo, Language, Logic, and Information, 1993.

[17] S. J. Hegner, Distributivity in incompletely specified type hierarchies: Theory and computational complexity, in *Computational Aspects of Constraint-Based Linguistic Description II, DYANA-2, ESPRIT Basic Research Project 6852, Deliverable R1.2B* (J. Dörre, ed.), pp. 29–120, DYANA, 1994, Also available as LLI Technical Report No. 4, University of Oslo, Department of Linguistics.

[18] S. J. Hegner, Computational and structural aspects of openly specified type hierarchies, in *Logical Aspects of Computational Linguistics, Third International Conference, LACL '98 Grenoble, France, December 1998, Selected Papers* (M. Moortgat, ed.), pp. 48–69, Springer-Verlag, 2001.

[19] S. J. Hegner, Characterization of type hierarchies with open specification, in *Semantics in Databases: Second International Workshop, Dagstuhl Castle, Germany, January 7-12, 2001, Revised Papers*, pp. 100–117, Springer-Verlag, 2003.

[20] J. A. Makowsky, Why Horn formulas matter in computer science: Initial structures and generic examples, *J. Comput. System Sci.*, **34**(1987), 266–292.

[21] J. R. McSkimin and J. Minker, The use of a semantic network in a deductive question answering system, in *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pp. 50–58, 1977.

[22] C. Pollard and I. A. Sag, *Head-Driven Phrase Structure Grammar*, University of Chicago Press, 1994.

[23] B. Selman and H. Kautz, Domain-independent extensions to gsat: Solving large structured satisfiability problems, in *Proc. Thirteenth IJCAI*, pp. 290–295, 1993.