SET CONTAINMENT INFERENCE

Paolo Atzeni
D. Stott Parker

December 1986
CSD-860012

--

Set Containment Inference

Paolo Atzeni

IASI-CNR, Viale Manzoni 30, 00185 Roma, ITALY

D. Stott Parker

UCLA Computer Science Dept., Los Angeles, CA, USA

ABSTRACT

Type hierarchies and type inclusion (*isa*) inference are now standard in many knowledge representation schemes. In this paper, we show how to determine consistency and inference for collections of statements of the form

mammal isa vertebrate.

These *containment* statements relate the contents of two sets (or types). The work here is new in permitting statements with negative information: disjointness of sets, or non-inclusion of sets. For example, we permit the following statements also:

mammal isa **non**(*reptile*)
non(*vertebrate*) *isa* **non**(*mammal*)
not(*reptile isa amphibian*)

We define general containment inference as the problem of determining the consequences of positive constraints P and negative constraints **not**(P) on sets, where positive constraints have the form

$$P:\ X_1 \cap \cdots \cap X_p \ \subseteq\ Y_1 \cup \cdots \cup Y_q .$$

Each X_i, Y_j is either a set T_k or its complement **non**(T_k). This paper considers only the binary case $p = q = 1$, with constraints like $P:\ X \subseteq Y$ or equivalently $P:\ X \cap \mathbf{non}(Y) = \varnothing$. Positive constraints therefore assert containment relations among sets, while negative constraints assert that two sets have a non-empty intersection.

We show binary containment inference is solved by rules essentially equivalent to Aristotle's *Syllogisms*. The containment inference problem can also be formulated and solved in predicate logic. When only positive constraints P are specified, binary containment inference is equivalent to *Propositional 2-CNF Unsatisfiability* (unsatisfiability of conjunctive propositional formulas limited to at most two literals per conjunct).

In either situation, necessary and sufficient conditions for consistency, as well as sound and complete sets of inference rules are presented. Polynomial-time inference algorithms are consequences, showing that permitting negative constraints does not result in intractability for this problem.

1. Introduction

In this paper we are interested in exploring inference properties of collections of statements like X *isa* Y, where X and Y are *types*. For example, we can assert

mammal isa vertebrate
reptile isa vertebrate.

Statements of this kind form an interesting class of constraints for real knowledge representation problems, since it permits declaration of containment relationships among types. 'Taxonomic' knowledge of this kind is (apparently) basic to human intelligence.

This paper departs from previous work by permitting negation in two ways:

- First, we let the types X and Y represent *complements of types*. For example, we permit statements of the form

  mammal isa **non**(*reptile*)

  where **non**(*reptile*) represents the complement of the type *reptile*. This statement asserts that *mammal*s are disjoint from *reptile*s.

- Second, we allow negative statements to be made. For example, we permit statements like

  not(*reptile isa amphibian*)

  which asserts that a *reptile* is *not* a type of *amphibian*, or equivalently, that *reptile* must intersect with **non**(*amphibian*). In other words, some non-amphibian reptiles exist.

For example, with these containment statements we can express the following statements about computer components:

	heat_sensitive_device	*isa*	*component*	
	axial_lead_device	*isa*	*component*	
	resistor	*isa*	*axial_lead_device*	
	resistor	*isa*	**non**(*heat_sensitive_device*)	
	half_watt_resistor	*isa*	*resistor*	
	diode	*isa*	*heat_sensitive_device*	
	diode	*isa*	*axial_lead_device*	
	microprocessor	*isa*	*heat_sensitive_device*	
	microprocessor	*isa*	**non**(*axial_lead_device*)	
not(	*heat_sensitive_device*	*isa*	**non**(*axial_lead_device*)	)
not(	*axial_lead_device*	*isa*	**non**(*heat_sensitive_device*)	).

The last two statements say the same thing; namely, that the *heat_sensitive_device* and *axial_lead_device* types *intersect*. For example, *diode*s are in their intersection. However, neither of these two types is contained in the other, since there are subtypes (*resistor*s and *microprocessor*s) that are contained in one but not the other.

We are interested in developing inference systems for type containment statements. The examples above are assertions that we would like to be able to store in a knowledge base and derive inferences from. For example, we would like to be able to ask

Is *half_watt_resistor* a *heat_sensitive_device* ?

and have a system correctly infer that the answer is *NO*. In general we wish to be able to make queries of the forms

$$X \ \ isa \ \ Y\,?$$
$$\textbf{not}(X \ \ isa \ \ Y\,)\,?$$

We call this problem the *Binary Containment Inference Problem,* a special case of a general containment inference problem permitting inclusion statements (and their negations) involving more than two types. It is a limited fragment of set theory of practical use.

An initial purpose of this paper was to investigate how *negation* affects inference in specific knowledge domains. In [1] we discussed a restricted version of the binary containment inference problem. Generally, of course, permitting negation causes inference to become computationally intractable; combinatorial explosions arise as soon as a negation operator is introduced. This is puzzling, since humans deal with at least certain kinds of negation without either becoming befuddled or taking long periods of time to arrive at correct conclusions.

We were pleased to discover that, for the binary containment problem, negation does not result in computational intractability. Thus binary containment assertions are at the same time expressive, yet not general enough a fragment of set theory to cause complexity problems.

Indeed, the binary containment problem can be solved somewhat elegantly. When all the statements are inclusions, we show here that the inference problem is equivalent to Propositional 2-CNF Unsatisfiability (the complement of the 2-SAT problem) [4]. More generally, the binary containment problem can be expressed and solved efficiently with predicate logic. We show that the problem is also solvable using a small set of inference rules. $O(n^3)$ time, where n is the number of types, is sufficient in all cases.

Interestingly, *syllogisms* turn out to be precisely the rules for binary containment inference. There are 24 valid syllogisms. These rules have been used for millenia, and were actually held as synonymous with the word *logic* until the mid-nineteenth century after the work of George Boole. While syllogisms were discarded eventually as being 'less general' than boolean logic, they clearly fit here naturally.

It seems likely that all the results in this paper have been discovered by other researchers at one time or another, in one form or another. After all, binary containment inference has been studied for millenia. However we are not aware of a reference covering the results here. Indeed, we were motivated by a study of existing knowledge representation systems, which uniformly lacked general containment inference processing.

It is possible to focus exclusively on logic when studying containment inference, by expressing the problem in predicate logic and then applying resolution proof techniques. The approach of this paper is broader, developing several formal systems to handle containment inference problems. This approach has certain benefits. First, it clarifies the model theory of the binary set containment problem, the problem's relationship to syllogisms, and identifies degenerate cases of constraints precisely. Second, it sets the foundation for fast algorithms not based directly on logic. Finally, it permits generalization to formal systems handling more complex types of 'syllogisms'. For example, DeMorgan studied six different kinds of syllogisms [3], including 'numerical' syllogisms such as

$$\begin{array}{c} 100\ Y\text{'s exist} \\ 70\ X\text{'s are } Y\text{'s} \\ 40\ Z\text{'s are } Y\text{'s} \\ \hline \text{at least } 10\ X\text{'s are } Z\text{'s.} \end{array}$$

The connection between sets, logic, syllogisms, and human intelligence is fascinating, and deserves further investigation.

2. Terminology

2.1 Syllogisms

Aristotle apparently defined a syllogism to be any valid inference [3], but concentrated on inferences that can be made from four kinds of propositions:

Every *S* is *P*
No *S* is *P* (i.e., Every *S* is not *P*)
Some *S* is *P*
Some *S* is not *P*

A syllogism is composed of three propositions involving three *types* *S*, *M*, and *P*, representing respectively its ‘Subject’, ‘Middle’, and ‘Predicate’. For example, the following is a syllogism:

Major premise:	every P is M
Minor premise:	some S is not M
Conclusion:	some S is not P

Since the conclusion always involves the subject and predicate, while the premises use the middle type *M* in 4 nontrivial ways (called ‘figures’ by Aristotle, although he developed only the first three figures shown below), there are a total of

$$4 \times 4 \times 4 \times 4 = 256$$

possible syllogisms, of which 24 are valid. These 24 are listed below, divided into the four figures:

S11:	every	*S*	is	*P*	**if**	every	*M*	is	*P*	**and**	every	*S*	is	*M*.	
S12:	some	*S*	is	*P*	**if**	every	*M*	is	*P*	**and**	every	*S*	is	*M*.	*
S13:	some	*S*	is	*P*	**if**	every	*M*	is	*P*	**and**	some	*S*	is	*M*.	
S14:	every	*S*	is not	*P*	**if**	every	*M*	is not	*P*	**and**	every	*S*	is	*M*.	
S15:	some	*S*	is not	*P*	**if**	every	*M*	is not	*P*	**and**	every	*S*	is	*M*.	*
S16:	some	*S*	is not	*P*	**if**	every	*M*	is not	*P*	**and**	some	*S*	is	*M*.	
S21:	every	*S*	is not	*P*	**if**	every	*P*	is not	*M*	**and**	every	*S*	is	*M*.	
S22:	some	*S*	is not	*P*	**if**	every	*P*	is not	*M*	**and**	every	*S*	is	*M*.	*
S23:	some	*S*	is not	*P*	**if**	every	*P*	is not	*M*	**and**	some	*S*	is	*M*.	
S24:	every	*S*	is not	*P*	**if**	every	*P*	is	*M*	**and**	every	*S*	is not	*M*.	
S25:	some	*S*	is not	*P*	**if**	every	*P*	is	*M*	**and**	every	*S*	is not	*M*.	*
S26:	some	*S*	is not	*P*	**if**	every	*P*	is	*M*	**and**	some	*S*	is not	*M*.	
S31:	some	*S*	is	*P*	**if**	every	*M*	is	*P*	**and**	every	*M*	is	*S*.	*
S32:	some	*S*	is	*P*	**if**	every	*M*	is	*P*	**and**	some	*M*	is	*S*.	
S33:	some	*S*	is	*P*	**if**	some	*M*	is	*P*	**and**	every	*M*	is	*S*.	
S34:	some	*S*	is not	*P*	**if**	every	*M*	is not	*P*	**and**	every	*M*	is	*S*.	*
S35:	some	*S*	is not	*P*	**if**	every	*M*	is not	*P*	**and**	some	*M*	is	*S*.	
S36:	some	*S*	is not	*P*	**if**	some	*M*	is not	*P*	**and**	every	*M*	is	*S*.	
S41:	every	*S*	is not	*P*	**if**	every	*P*	is	*M*	**and**	every	*M*	is not	*S*.	
S42:	some	*S*	is not	*P*	**if**	every	*P*	is	*M*	**and**	every	*M*	is not	*S*.	*
S43:	some	*S*	is not	*P*	**if**	every	*P*	is not	*M*	**and**	every	*M*	is	*S*.	*
S44:	some	*S*	is not	*P*	**if**	every	*P*	is not	*M*	**and**	some	*M*	is	*S*.	
S45:	some	*S*	is	*P*	**if**	every	*P*	is	*M*	**and**	every	*M*	is	*S*.	*
S46:	some	*S*	is	*P*	**if**	some	*P*	is	*M*	**and**	every	*M*	is	*S*.	

These 24 syllogisms are all valid under the assumption that the sets denoted by the types *S*, *M*, and *P* are nonempty. If this assumption does not hold, the 9 entries above marked with stars (*) are invalid. In other words, although we would normally assume that whenever

every *X* is *Y*

then also

some *X* is *Y*,

this inference is invalid when the set denoted by the type *X* is empty.

The structure of the syllogisms has fascinated philosophers and mathematicians for millenia. At this point the reader may be asking questions such as:

- Is the set of 24 rules (or 15, eliminating starred ones) complete?
- Is there a more compact presentation of these rules?
- Can the rules be used in an efficient inference system?

We address these questions later in the paper.

2.2 Syntax of Containment Propositions

Some readers will have noticed that syllogisms involve containment propositions. Specifically, if *X* and *Y* represent sets, and **non**(*Y*) represents the set complement of *Y*:

$$\begin{array}{rcl} \text{every } X \text{ is } Y & \equiv & X \subseteq Y \\ \text{some } X \text{ is } Y & \equiv & X \cap Y \neq \varnothing \\ \textbf{not} \text{ every } X \text{ is } Y & \equiv & X \cap \textbf{non}(Y) \neq \varnothing \\ \textbf{not} \text{ some } X \text{ is } Y & \equiv & X \subseteq \textbf{non}(Y) \end{array}$$

This collection of syllogistic propositions is thus equivalent to the binary propositions one can make up with the standard set predicates $\subseteq$ and $\cap{\neq}\varnothing$. In an effort to follow [1], as well as simplify notation ($\cap{\neq}\varnothing$ is tedious to write), we define the following two predicates on type descriptors X, Y:

Definition

X *isa* Y if the set denoted by X is a subset of the set denoted by Y.

Definition

X *int* Y if the set denoted by X intersects the set denoted by Y. The intersection must be *nonempty*.

Remark

$$X \textit{ int } Y \equiv \textbf{not}(\, X \textit{ isa } \textbf{non}(Y)\,).$$

The following table summarizes equivalences between notations to be observed for the remainder of the paper:

		X	$\subseteq$	Y	$\equiv$		(X	*isa*	Y)
		X	$\cap$	Y $\neq \varnothing$	$\equiv$		(X	*int*	Y)
	every	X	is	Y	$\equiv$		(X	*isa*	Y)
	some	X	is	Y	$\equiv$		(X	*int*	Y)
	every	X	is	non(Y)	$\equiv$	not	(X	*int*	Y)
	some	X	is	non(Y)	$\equiv$	not	(X	*isa*	Y)
not	every	X	is	Y	$\equiv$	not	(X	*isa*	Y)
not	some	X	is	Y	$\equiv$	not	(X	*int*	Y)
not	every	X	is	non(Y)	$\equiv$		(X	*int*	Y)
not	some	X	is	non(Y)	$\equiv$		(X	*isa*	Y)

In what follows we use *isa* and *int*.

2.3 The Set Containment Problem

Our goal in this section is to set up a framework for expressing containment problems. We differentiate between the *scheme* of a containment problem, and its *interpretations* or *models*. The scheme specifies the structure of the problem, while interpretations of the scheme give instances of objects in the types in the scheme.

Definition

A *type scheme* T/U is a collection of *type symbols* $\{U, T_1, \ldots, T_n\}$. The type symbol U is a special symbol and is called the *universe* of the type scheme.

Each type symbol T_i will denote a subset of U. The universe symbol U is needed in order to define what we mean by complements **non**(T_i) of types:

Definition

A *type descriptor* X of a type scheme T/U is either

1. a type symbol T_i or U.
2. **non**(Y), where Y is a type descriptor of T/U.

Definition

An *assignment* I of a type scheme $T/U = \{U, T_1, \ldots, T_n\}$ is a map associating to each type symbol a possibly empty subset of a finite *domain* D, subject to the following restrictions:

0. $I(U) = D$;
1. $I(\mathbf{non}(U)) = \varnothing$;
2. $I(T_i) \subseteq I(U)$, $1 \le i \le n$;
3. $I(\mathbf{non}(T_i)) \subseteq I(U)$, $1 \le i \le n$;
4. $I(T_i) \cap I(\mathbf{non}(T_i)) = \varnothing$, $1 \le i \le n$.

For each i, $I(T_i)$ is a finite subset of D. We require finite assignments here for simplicity.

An *interpretation* I is an assignment that satisfies the following additional restrictions, where '$-$' represents set difference:

5. $I(T_i) = I(U) - I(\mathbf{non}(T_i))$, $1 \le i \le n$;
6. $I(\mathbf{non}(T_i)) = I(U) - I(T_i)$, $1 \le i \le n$.

In other words, with an interpretation the type descriptor **non**(X) denotes the *complement* under U of the set denoted by X.

Remark

In any interpretation, **non**(**non**(X)) denotes the same set as X.

Definition

A type descriptor X is *trivial* in interpretation I if $I(X) = \varnothing$.
The *trivial interpretation* I assigns $I(X) = \varnothing$ for every type descriptor X.

Definition

A *positive constraint* P has the form

$$P: X_1 \cap \cdots \cap X_p \quad isa \quad Y_1 \cup \cdots \cup Y_q$$

where each X_j, $1 \le j \le p$, and each Y_k, $1 \le k \le q$, is a type descriptor.

The constraint is *satisfied* by the interpretation I if

$$I(X_1) \cap \cdots \cap I(X_p) \subseteq I(Y_1) \cup \cdots \cup I(Y_q).$$

Definition

A *negative constraint* has the form **not**(P), where P is a positive constraint. It is satisfied if P is not.

Note that positive constraints can be rewritten as

$$P: \mathbf{non}(X_1) \cup \cdots \cup \mathbf{non}(X_p) \cup Y_1 \cup \cdots \cup Y_q \ = \ U$$

or, equivalently,

$$P: X_1 \cap \cdots \cap X_p \cap \mathbf{non}(Y_1) \cap \cdots \cap \mathbf{non}(Y_q) \ = \ \varnothing$$

so the negative constraint **not**(P) is equivalent to

$$\mathbf{not}(P)\text{: } X_1 \cap \cdots \cap X_p \cap \mathbf{non}(Y_1) \cap \cdots \cap \mathbf{non}(Y_q) \ \neq \ \varnothing$$

In other words, positive constraints make assertions about *inclusions* among types, while negative constraints make assertions about *intersections* among types.

Definition

A *containment scheme* is a pair $S = (T/U, C)$ where U is a universe symbol, T/U is a type scheme $\{U, T_1, \ldots, T_n\}$, and C is a set of containment constraints on type symbols in T/U.

Definition

A *model* of a containment scheme $(T/U, C)$ is an interpretation I of T/U that satisfies all constraints in C.

Containment constraints can be 'degenerate'. Consider the various constraints below:

1. X *isa* **non**(X).
2. **non**(X) *isa* X.
3. **not**(X *isa* X).

The first constraint is satisfied only when X denotes $\varnothing$. Similarly, the second is satisfied only when **non**(X) denotes $\varnothing$, so X denotes the same set as U. The first two constraints together can be satisfied only when both X and U denote $\varnothing$. In other words, the first two constraints imply that there can be only one model: the trivial one.

The third constraint is the negation of (X *isa* X), which is true of every interpretation for X. Thus, any scheme with the third constraint can satisfy no interpretation, and will have no model.

Definition

A containment scheme is *unsatisfiable* if it has no model; otherwise it is *satisfiable*.

Now consider the following general problem:

Set Containment Problem

Input: a containment scheme $(T/U,C)$, where C is a collection $\{C_j \mid j = 1, \ldots, m\}$ of set containment constraints on the type symbols of T/U.

Question: Is the containment scheme satisfiable? That is, is there an interpretation for T/U that satisfies each constraint C_j in C?

Not surprisingly, the set containment problem is NP-complete in general. This is shown in Appendix I. However, *in the special case in which all constraints are positive, the set containment problem is always satisfiable.* Specifically, the trivial interpretation (in which $I(U) = \varnothing$) is always a model satisfying C.

In this paper we are interested only in the special case $p = q = 1$, where all constraints are binary. This restricted version is called the *Binary Set Containment Problem*. In the remainder of the paper we first consider the special case where all binary constraints are positive, then study the general binary containment problem.

Example

Consider the containment scheme

$$S = (\{pacifist,\ quaker,\ republican\}/U,\ C)$$

where the constraint set C is:

republican	*isa*	**non**(*pacifist*)
republican	*int*	*quaker*
quaker	*isa*	*pacifist.*

The binary set containment problem for this scheme is to determine whether it is satisfiable or not. It turns out this scheme is *unsatisfiable*; we will see why shortly.

2.4 Containment Inference

We first recall some terminology concerning inference rules. The reader unfamiliar with this material may wish to consult texts in database theory, such as [6] and [7]. We remind the reader that classes of constraints studied in database theory do not involve negation, for the most part. That is, collections of data dependencies (functional dependencies, join dependencies, etc.) do not imply that a specific data dependency *does not* hold, but only that one *does* hold. These systems are always satisfiable. Unsatisfiability can occur with containment, as we have already seen. Therefore, the inference problem here is somewhat different than in these texts.

Implication and inference are important concepts in dealing with constraints of the general kind proposed here. If we are given a set of constraints, we are frequently interested in deducing whether other constraints must also hold.

A constraint c is *implied* by a set of constraints C on a scheme S if it holds in all models of S. Given C and c, the *inference problem* is to tell whether C implies c. Algorithms for the solution of the inference problem (called *inference algorithms*) have correctness proofs that are usually based on sound and complete sets of inference rules.

Definition

$C \models c$ if C implies c (that is, c must hold in every model of C).

If C is unsatisfiable, then C has no models, and the definition of $\models$ becomes vacuous. Thus if C is unsatisfiable, then $C \models c$ for every constraint c.

Set Containment Inference Problem

Input: a containment scheme $(T/U,C)$, and a constraint c.

Question: Is it true that $C \models c$?

This problem can be reduced to the Set Containment Problem mentioned earlier simply by determining satisfiability of the scheme with constraints $C \cup \{\textbf{not}\,(c)\}$. Alternatively, we can approach the inference problem with inference rules.

Definition

An *inference rule* $C \vdash c$ is a rule asserting that the constraint c holds whenever the set of constraints C holds. For example, the rule

$$X\ isa\ Y,\ \ Y\ isa\ Z\ \vdash\ X\ isa\ Z$$

asserts that the inclusion predicate *isa* is transitive. X, Y, and Z represent arbitrary type descriptors.

Definition

Relative to a specific set of inference rules, $C \vdash c$ if c can be derived from C using applications of the rules.

The basic requirement for each inference rule is to be *sound*, i.e., that it derive from C only constraints c such that $C \models c$. Moreover, it is important to have sets of inference rules that are *complete*, i.e., that allow the derivation of *all* the constraints c such that

$C \models c$. Thus, a set of rules is sound and complete when $\vdash$ is equivalent to $\models$.

Definition

A scheme $(T/U, C)$ is *inconsistent* if there is a constraint c such that

$$C \vdash c \quad \text{and} \quad C \vdash \mathbf{not}(c).$$

Otherwise the scheme is *consistent*.

Example

The constraint set C considered earlier

republican	*isa*	**non**(*pacifist*)
republican	*int*	*quaker*
quaker	*isa*	*pacifist*

can be shown to be inconsistent with the appropriate inference rules. The (sound) inference rule

$$X \ int\ Y,\ Y \ isa\ Z \ \vdash\ X \ int\ Z$$

implies that

$$republican\ \ int\ \ quaker,\ \ quaker\ \ isa\ \ pacifist\ \vdash\ republican\ int\ pacifist.$$

However

$$republican\ int\ pacifist$$

is equivalent to

$$\mathbf{not}(\ republican\ isa\ \mathbf{non}(pacifist)\ ),$$

contradicting the first constraint in C.

Clearly, an inconsistent scheme is also unsatisfiable. We will see later that the converse also holds.

3. Degenerate Properties of Containment Schemes

This section characterizes when a containment scheme has certain degenerate properties. We show three things.

- First, a type X is trivial in every model of a scheme if, and only if, we can infer X *isa* **non**(X).
- Second, a satisfiable scheme has only the trivial model if, and only if, for some type descriptor X we can infer both X *isa* **non**(X) and **non**(X) *isa* X.
- Finally, a scheme is unsatisfiable if, and only if, there is some type descriptor X for which we can infer X *int* **non**(X).

With a reasonable inference mechanism, then, we will be able to detect unsatisfiability or triviality of types. In later sections we develop such inference mechanism.

3.1 Model Extension Algorithm

We give first an algorithm useful in various proofs later. It takes as input a containment scheme (T/U,C), an assignment I, and a constant t (which I may or may not include in its image), and produces an assignment I' which uses t unless C is degenerate. It is called a 'model extension' algorithm since, as we show below, it always produces a model when given one.

At various points the algorithm tests whether $C \models c$, given some C and c. The Model Extension Algorithm therefore requires an algorithm for deciding implication. Such an algorithm is suggested in Appendix I, where it is shown the Set Containment Problem is in NP. Essentially, to decide whether $C \models c$, we need only look for models with domain D of size at most the number of intersection constraints in C. This is not efficient, but efficiency is not the issue here, since the Model Extension Algorithm is used only in proofs.

Model Extension Algorithm

Input:
• a containment scheme (T/U,C),
• an assignment I from symbols in T/U to subsets of D,
• a symbol t which may or may not be in D.

Output:
• an assignment I from symbols in T/U to subsets of $D \cup \{t\}$.

The algorithm constructs I' from I as follows:

1. *Test for trivial universe.*
   If $C \models$ (U *isa* **non**(U)), set $I' = I$ and halt.
2. *Preserve any previous use of t.*
   For all X with t already in $I(X)$, set

$$I'(X) = I(X) \cup \{t\}$$
$$I'(\mathbf{non}(X)) = I(\mathbf{non}(X))$$

Note t cannot be in both $I(X)$ and $I(\mathbf{non}(X))$, for then I would not be an assignment.

3. *Assign assignments for trivial types and their complements.*
For all X (including $\mathbf{non}(U)$) with $I'(X)$ currently undefined and $C \models (X \textit{ isa } \mathbf{non}(U))$, set
$$I'(X) = I(X)$$
$$I'(\mathbf{non}(X)) = I(\mathbf{non}(X)) \cup \{t\}$$
We cannot find X at this point such that both X and $\mathbf{non}(X)$ are trivial, since these would imply $C \models (U \textit{ isa } \mathbf{non}(U))$.

4. *Propagate t through isa constraints.*
For all X with t in $I'(X)$ at this point, for all Z with $I'(Z)$ undefined such that $C \models X \textit{ isa } Z$, set
$$I'(Z) = I(Z) \cup \{t\}$$
$$I'(\mathbf{non}(Z)) = I(\mathbf{non}(Z))$$

5. *Add t to some type permitting the addition.*
If there is no X such that $I'(X)$ is undefined, then halt: I' is the completed assignment.

   Otherwise select a 'minimal' X such that $I'(X)$ is undefined. That is, find an X for which there is no nontrivial Y such that $C \models Y \textit{ isa } X$ and also $C \not\models X \textit{ isa } Y$. Such an X must exist, since otherwise Step 4 would have already yielded a definition for $I'(X)$. Then set
$$I'(X) = I(X) \cup \{t\}$$
$$I'(\mathbf{non}(X)) = I(\mathbf{non}(X)),$$
   and go to step 4. Step 4 will then propagate definition of I' for supertypes of X.

The construction is well defined, because the process always terminates. Moreover, when it terminates I' is a valid assignment, defined for all type descriptors X of T/U. In particular the algorithm cannot assign t to both $I'(X)$ and $I'(\mathbf{non}(X))$ for any X, since each step assigns t to either one or the other, and $I'(X)$ is defined at most once.

Lemma
If I is an interpretation with domain D, then I' is an interpretation with domain $D \cup \{t\}$.
If I is a model with domain D, then I' is a model with domain $D \cup \{t\}$.

Proof

I' must be an interpretation if I is, since each step of the algorithm guarantees then that $I'(Z) = I'(U) - I'(\textbf{non}(Z))$ for each type descriptor Z. Also I' satisfies all constraints X *int* Y that I does. If I is a model, the only way I' can fail to be a model is that there be some constraint V *isa* W implied by C, where t is in $I'(V)$ but not in $I'(W)$. However, steps 4 and 5 of the construction explicitly prohibit this situation. □

3.2 Triviality in Containment Schemes

Theorem 1.
A type descriptor X is trivial in every model of C **iff** $C \models X$ *isa* **non**(X).

Proof

Clearly if $C \models X$ *isa* **non**(X) then $C \models X$ *isa* **non**(U) and X must be trivial in all models. Conversely, let I be any model of C with $I(X) = \varnothing$. Let t be a symbol not in $I(U)$. If it is *not* true that $C \models X$ *isa* **non**(X), then $C \not\models X$ *isa* **non**(U) and the Model Extension Algorithm above can find a model I' with $I'(X) = \{t\}$, showing X is not trivial. □

A corollary of Theorem 1 is that a pair of constraints

$$X \textit{ isa } \textbf{non}(X)$$
$$\textbf{non}(X) \textit{ isa } X$$

must be implied by any scheme with only the trivial model. (U, and consequently both X and **non**(X), must denote $\varnothing$.)

3.3 Unsatisfiable Containment Schemes

We showed earlier that containment constraints can actually be unsatisfiable, giving as an example **not**(X *isa* X). In fact, this constraint is not only sufficient for unsatisfiability, it is also necessary:

Theorem 2.
A containment scheme $(T/U,C)$ is unsatisfiable **iff** for some X, $C \models X$ *int* **non**(X).

Proof Clearly if the scheme is satisfiable, we cannot have $C \models X$ *int* **non**(X) for any X. Conversely, if for every X it is true that $C \not\models X$ *int* **non**(X), then we can exhibit a model satisfying C. The model is built in two steps:

1. Define the assignment I_0 as follows: For each type descriptor X in $\{U, T_1, \ldots, T_n, \textbf{non}(U), \textbf{non}(T_1), \ldots, \textbf{non}(T_n)\}$ set

$$I_0(X) = \begin{cases} \varnothing & \text{if } C \models X \textit{ isa } \textbf{non}(U) \\ \{\ \{X,Y\} \mid C \models X \textit{ int } Y\ \} & \text{otherwise.} \end{cases}$$

   I_0 is not an assignment iff there is some X such that $I_0(X) \cap I_0(\textbf{non}(X)) \neq \varnothing$. But this cannot arise, since $I_0(X) \cap I_0(\textbf{non}(X)) \subseteq \{\ \{X, \textbf{non}(X)\}\ \}$, and the assumption $C \not\models X$ *int* **non**(X) implies this intersection must be empty.

2. If $I_0(X) = \varnothing$ for every X, then halt: at this point there can be no intersection constraints in C and we have a (trivial) model $I = I_0$. (Any intersection constraint X *int* Y would imply X *int* X, which when combined with X *isa* **non**(X) implies X *int* **non**(X), a contradiction.)

   If $I_0(X) \neq \varnothing$ for some X, extend I_0 to a model I by accumulatively applying the Model Extension Algorithm for each element 't' in the set

$$\{ \ \{V,W\} \mid C \models V \ int \ W \ \}.$$

We argue that, when this process completes, I is a model of C. The first step guarantees I_0 meets all 'intersection' constraints implied by C, ignoring trivial types. Since I_0 is a valid assignment, the second step extends it to an assignment I that also satisfies the 'inclusion' constraints implied by C. □

Example

Consider the type scheme $S = (\{a,b,c,d\}/U,C)$, where C is the set of constraints

$$\begin{array}{lll} a & isa & c \\ a & isa & \mathbf{non}(d) \\ c & int & d \\ b & int & \mathbf{non}(c) \end{array}$$

Here

$$\{ \ \{X,Y\} \mid C \models X \ int \ Y \ \} \ = \ \{ \ \{c,d\}, \ \{b, \mathbf{non}(c)\} \ \}.$$

The model constructed for S evolves as follows:

a	**non**(a)	b	**non**(b)	c	**non**(c)	d	**non**(d)	U	**non**(U)
$\varnothing$	$\varnothing$	$\{\{b,\mathbf{non}(c)\}\}$	$\varnothing$	$\{\{c,d\}\}$	$\{\{b,\mathbf{non}(c)\}\}$	$\{\{c,d\}\}$	$\varnothing$	$\varnothing$	$\varnothing$
$\varnothing$	$\{\{c,d\},\{b,\mathbf{non}(c)\}\}$	$\{\{b,\mathbf{non}(c)\}\}$	$\varnothing$	$\{\{c,d\}\}$	$\{\{b,\mathbf{non}(c)\}\}$	$\{\{c,d\}\}$	$\varnothing$	$\{\{c,d\},\{b,\mathbf{non}(c)\}\}$	$\varnothing$
$\varnothing$	$\{\{c,d\},\{b,\mathbf{non}(c)\}\}$	$\{\{b,\mathbf{non}(c)\}\}$	$\{\{c,d\}\}$	$\{\{c,d\}\}$	$\{\{b,\mathbf{non}(c)\}\}$	$\{\{c,d\}\}$	$\varnothing$	$\{\{c,d\},\{b,\mathbf{non}(c)\}\}$	$\varnothing$
$\varnothing$	$\{\{c,d\},\{b,\mathbf{non}(c)\}\}$	$\{\{b,\mathbf{non}(c)\}\}$	$\{\{c,d\}\}$	$\{\{c,d\}\}$	$\{\{b,\mathbf{non}(c)\}\}$	$\{\{c,d\},\{b,\mathbf{non}(c)\}\}$	$\varnothing$	$\{\{c,d\},\{b,\mathbf{non}(c)\}\}$	$\varnothing$

The first sequence shows I_0, the assignment satisfying all intersection constraints implied by C.

The second sequence shows the assignment created at step 4 of the Model Extension Algorithm with $t = \{c,d\}$. We have used the facts that $C \models \mathbf{non}(c)$ *isa* $\mathbf{non}(a)$, $C \models d$ *isa* $\mathbf{non}(a)$, $C \models b$ *isa* U, and $C \models c$ *isa* U.

The third sequence shows the final extension of I_0 created by the Model Extension Algorithm for $t = \{c,d\}$. (Note this is only one possible extension; the type descriptor **non**(b) was selected in step 5 of the algorithm.)

The fourth sequence shows the subsequent assignment obtained from the Model Extension Algorithm for $t = \{b,\mathbf{non}(c)\}$. (The type descriptor d was selected in step 5 of the algorithm.) This sequence can be verified to be a model of S.

We might have expected from the constraints c *int* d, b *int* **non**(c) in C that both $C \models$ **non**(c) *int* **non**(d) and $C \models$ **non**(b) *int* c would follow. The model just constructed contradicts the first logical implication, and the other implication would also be violated if in the third sequence b had been selected instead of **non**(b) by step 5 of the Model Extension Algorithm.

4. Positive Binary Containment

Let us now devote our attention to the containment problem. Consider first the important special case of the set containment inference problem where all constraints are of the form

$$X \textit{ isa } Y$$

with X and Y type descriptors denoting subsets of U.

In [1], a similar problem was studied and solved. A complete set of inference rules is presented for two containment predicates, *isa* and *dis*. The proposition (X *isa* Y) states that the type X denotes a subset of the type Y, while the proposition (X *dis* Y) states that X and Y denote disjoint sets.

In [1] it is shown that, for arbitrary types X, Y, Z, the following rules for *isa* and *dis* are sound and complete:

I1. $\vdash X \textit{ isa } X.$
I2. $X \textit{ isa } Y, Y \textit{ isa } Z \vdash X \textit{ isa } Z.$
M1. $X \textit{ dis } Y, Z \textit{ isa } X \vdash Z \textit{ dis } Y.$
M2. $X \textit{ dis } X \vdash X \textit{ isa } Y.$
D1. $X \textit{ dis } X \vdash X \textit{ dis } Y.$

We can use set complementation to simplify these rules. Since, for example,

$$X \textit{ dis } Y \;\equiv\; X \textit{ isa } \mathbf{non}(Y)$$

we can informally replace the five rules above with

I1. $\vdash X \textit{ isa } X.$
I2. $X \textit{ isa } Y, Y \textit{ isa } Z \vdash X \textit{ isa } Z.$
D1. $X \textit{ isa } \mathbf{non}(X) \vdash X \textit{ isa } Y.$

provided we let X, Y, Z be arbitrary type descriptors, and treat $\mathbf{non}(\mathbf{non}(X))$ as being equivalent to X.

In the statements X *isa* Y, X *dis* Y of [1], X is required to be a type symbol (not a type descriptor). Without this requirement the 5 rules above are incomplete. For example, the rule

$$X \textit{ isa } Y \;\vdash\; \mathbf{non}(Y) \textit{ isa } \mathbf{non}(X)$$

is not inferrable from the 5 rules, but is sound.

Consider, then, the following set of rules:

ISA0.	$\vdash X\ isa\ U$.
ISA1.	$\vdash X\ isa\ X$.
ISA2.	$X\ isa\ Y, Y\ isa\ Z \vdash X\ isa\ Z$.
ISA3.	$X\ isa\ Y \vdash \mathbf{non}(Y)\ isa\ \mathbf{non}(X)$.
TRIV0.	$U\ isa\ \mathbf{non}(U) \vdash X\ isa\ \mathbf{non}(U)$.
TRIV1.	$X\ isa\ \mathbf{non}(X) \vdash X\ isa\ \mathbf{non}(U)$.
TRIV2.	$X\ isa\ \mathbf{non}(U) \vdash X\ isa\ Y$.
EQ0.	$\alpha(\mathbf{non}(\mathbf{non}(X))) \vdash \alpha(X)$ [α any expression].
EQ1.	$\alpha(X) \vdash \alpha(\mathbf{non}(\mathbf{non}(X)))$ [α any expression].

Theorem 3.
The set of rules ISA0-3,TRIV0-2,EQ0-1 is sound and complete for positive binary containment inference. That is, if C contains only positive constraints and $\vdash$ represents derivability using these rules, then

$$C \vdash (X\ isa\ Y)\ \textbf{iff}\ C \models (X\ isa\ Y).$$

Proof

We omit proofs of the soundness of the rules, as they follow from basic axioms of set inclusions. Also, we use rules EQ0 and EQ1 implicitly throughout the proof here.

For each containment scheme $S = (T/U,C)$, we show that *isa* constraints not derivable from C by the rules cannot be implied by C. Suppose that $c = (X\ isa\ Y)$ is a constraint implied by C, but not derivable from the rules. We construct a counterexample model I satisfying C but violating c.

First note that c cannot be of the forms $(X\ isa\ X)$ or $(X\ isa\ U)$, since ISA0 and ISA1 preclude these. Also since $(X\ isa\ Y)$ is not derivable from the rules, then $(X\ isa\ \mathbf{non}(U))$ and $(X\ isa\ \mathbf{non}(X))$ are not either, for otherwise $(X\ isa\ Y)$ would be derivable with rules TRIV1 and TRIV2. Finally $C \nvdash (U\ isa\ \mathbf{non}(U))$, since the TRIV rules would yield $(X\ isa\ Y)$.

We construct an assignment I_0 as follows: For each type descriptor Z, let $I_0(Z) = \{t\}$ if $(X\ isa\ Z)$ is derivable from C, and $I_0(Z) = \varnothing$ if $(X\ isa\ \mathbf{non}(Z))$ is derivable. In addition, put $I_0(Y) = \varnothing$ and $I_0(\mathbf{non}(Y)) = \{t\}$.

I_0 will be a valid assignment if we can guarantee the following properties:

1. $I(U) = \{t\}$.
2. $I(\mathbf{non}(U)) = \varnothing$.
3. $I(\mathbf{non}(X)) = \varnothing$.
4. There is no Z such that both $(X\ isa\ Z)$ and $(X\ isa\ \mathbf{non}(Z))$.

These properties must hold. $I_0(U) = \{t\}$ since $\vdash (X$ *isa* $U)$. If $I_0(\textbf{non}(U)) \neq \emptyset$, then we can derive $(X$ *isa* $\textbf{non}(U))$, contradicting our earlier observation. Similarly if $I_0(\textbf{non}(X)) \neq \emptyset$, then we can derive $(X$ *isa* $\textbf{non}(X))$, again contradicting our observation. Finally, if there is a Z such that both $(X$ *isa* $Z)$ and $(X$ *isa* $\textbf{non}(Z))$, we can use rule ISA3 on the second constraint to infer $(Z$ *isa* $\textbf{non}(X))$. Then ISA2 and the first constraint derive $(X$ *isa* $\textbf{non}(X))$, contradicting our earlier observation that this was not derivable. So I_0 is a valid assignment.

Now, extend this assignment to an interpretation I by applying the Model Extension Algorithm, with $\models$ replaced by $\vdash$, to I_0 and constant t. Since $C \vdash X$ *isa* X, $I(X)=\{t\}$. Also, I does not satisfy X *isa* Y because $I(Y) = I_0(Y) = \emptyset$. We claim I is a model for S. Suppose V *isa* W is a constraint in C. If $I(V) = \emptyset$, I satisfies the constraint. If $I(V) = \{t\}$, then $C \vdash X$ *isa* V. But then, by rule ISA2, $C \vdash X$ *isa* W also, so $I(W) = \{t\}$, and I satisfies the constraint again. Since this is true for all such constraints, I is a model. □

5. General Binary Containment

We show in this section that the 24 syllogisms listed earlier are essentially the rules we need for general set containment inference. However, although these 24 rules are elegant, they are also somewhat verbose. We show first that we can reduce the syllogism rules to a set of 5 simple rules (involving really only two basic syllogisms, S11 and S13). We then show that an extension of these 5 rules dealing with degenerate conditions is a sound and complete set for containment inference.

5.1 Compressed Syllogisms

Let X, Y, Z be type descriptors. Consider the following rules:

R1: every X is Z **if** every Y is Z **and** every X is Y.
R2: some X is Z **if** every Y is Z **and** some X is Y.
R3: some X is Y **if** every X is Y.
R4: some X is Y **if** some Y is X.
R5: every X is **non**(Y) **if** every Y is **non**(X).

R1 and R2 are syllogisms mentioned earlier. Rule R3 requires the assumption that types are nonempty; this is actually necessary only where the existential quantifier "some" implies actual existence of some object, as it often does in natural language. R4 and R5 state that both intersection and disjointness of types are symmetric relations.

Theorem 4. All valid syllogisms follow from the rules R1-R5.

Proof A simple case analysis shows this, and is instructive about the structure of the 24 syllogisms. Let S*ij*(M/**non**(M)) denote the *ij*-th syllogism with type descriptor M replaced by **non**(M), etc. We simply list the rules and the 'variable substitutions' needed to derive each syllogism.

S11:	R1
S12:	R1 & R3
S13:	R2
S14:	S11(P/**non**(P))
S15:	S12(P/**non**(P))
S16:	S13(P/**non**(P))
S21:	S11(P/**non**(P)) & R5
S22:	S21 & R3
S23:	S13(P/**non**(P)) & R5
S24:	S21(M/**non**(M))
S25:	S22(M/**non**(M))
S26:	S23(M/**non**(M))

S31:	S13 & R3 & R4
S32:	S13 & R4
S33:	S32(*P*/*S*,*S*/*P*)
S34:	S31(*P*/**non**(*P*))
S35:	S32(*P*/**non**(*P*))
S36:	S33(*P*/**non**(*P*))
S41:	S11(**non**(*S*)/*P*,*P*/*S*) & R5
S42:	S41 & R3
S43:	R5 & S34
S44:	R5 & S35
S45:	S12(*S*/*P*,*P*/*S*) & R3
S46:	S13(*S*/*P*,*P*/*S*) & R3

□

This theorem saves us considerable effort in relating the results here to syllogisms.

5.2 Formal Containment Rules

Consider the following rules:

INT0.	X *int* $Y \vdash X$ *int* U.
INT1.	X *int* $Y \vdash X$ *int* X.
INT2.	X *int* $Y \vdash Y$ *int* X.
INT3.	X *int* Y, Y *isa* $Z \vdash X$ *int* Z.
INT4.	X *int* U, X *isa* $Y \vdash X$ *int* Y.
INC0.	X *int* **non**(X) $\vdash Y$ *isa* Z.
INC1.	X *int* **non**(X) $\vdash Y$ *int* Z.
ISA0.	$\vdash X$ *isa* U.
ISA1.	$\vdash X$ *isa* X.
ISA2.	X *isa* Y, Y *isa* $Z \vdash X$ *isa* Z.
ISA3.	X *isa* $Y \vdash$ **non**(Y) *isa* **non**(X).
TRIV0.	U *isa* **non**(U) $\vdash X$ *isa* **non**(U).
TRIV1.	X *isa* **non**(X) $\vdash X$ *isa* **non**(U).
TRIV2.	X *isa* **non**(U) $\vdash X$ *isa* Y.
EQ0.	α(**non**(**non**(X))) $\vdash \alpha(X)$ [α any expression].
EQ1.	$\alpha(X) \vdash \alpha$(**non**(**non**(X))) [α any expression].

These rules extend the compressed syllogism rules R1-R5 to deal with trivial types and unsatisfiable constraint sets, but otherwise express the same information.

Theorem 5. The rules INT0-4,INC0-1,ISA0-3,TRIV0-2,EQ0-1 are sound and complete for general binary containment inference. That is, if C is set of binary constraints, c is a binary constraint, and $\vdash$ represents derivability using these rules, then

$$C \vdash c \textbf{ iff } C \models c.$$

Proof For the containment scheme ($T/U,C$), we show that constraints not derivable from C by the rules cannot be implied by C.

Suppose that c is a constraint such that $C \models c$ but $C \nvdash c$. We know then that, for every X, $C \nvdash (X\ int\ \mathbf{non}(X))$ since otherwise INC1 or INC2 would give $C \vdash c$. Analogously, $C \nvdash (X\ int\ \mathbf{non}(U))$, since otherwise ISA0, ISA3, INT1 would derive $C \vdash (X\ int\ \mathbf{non}(X))$.

Let us consider the case where $c = (X\ int\ Y)$. Then $C \models c$ implies C must contain some intersection constraints. In three steps we construct a counterexample model I satisfying C but violating c, much like the model in Theorem 2:

1. Let C' be C with c explicitly negated. That is,

$$C' \;=\; C \cup \{\mathbf{not}(c)\} \;=\; C \cup \{X\ isa\ \mathbf{non}(Y)\}.$$

2. Define the assignment I_0 as follows: For each type descriptor V in $\{U, T_1, \ldots, T_n, \mathbf{non}(U), \mathbf{non}(T_1), \ldots, \mathbf{non}(T_n)\}$ by

$$I_0(V) \;=\; \begin{cases} \varnothing & \text{if } C' \vdash V\ isa\ \mathbf{non}(V) \\ \{\ \{V,W\} \mid C' \vdash V\ int\ W\ \} & \text{otherwise.} \end{cases}$$

   Notice that we define I_0 in terms of what can be derived from C' by the rules.

3. Extend I_0 to I by applying a variant of Model Extension Algorithm accumulatively with the scheme ($T/U,C'$), each time picking a new element 't' from the set

$$\{\ \{V,W\} \mid C' \vdash V\ int\ W\ \}.$$

   The variant of the Model Extension Algorithm used here is exactly like the algorithm, but replaces all uses of $\models$ with $\vdash$. Thus again the rules are used in constructing I.

The Model Extension Algorithm with $\models$ replaced by $\vdash$ will generate an assignment satisfying at least as many constraints as its predecessor, except when step 1 determines that $C' \vdash (U\ isa\ \mathbf{non}(U))$. If this were to happen, however, since we know C contains some intersection constraint $(V\ int\ W)$ we could show with INT0, ISA0, ISA3, INT3 that $C' \vdash (V\ int\ \mathbf{non}(V))$. However, since $C \nvdash (V\ int\ \mathbf{non}(V))$ we can obtain a contradiction. The proof involves analysis of several cases, which we sketch here.

Recall $C' = C \cup \{X\ isa\ Y\}$. Since we can derive $C' \vdash (V\ int\ \mathbf{non}(V))$, either rules INT3 or INT4 must be used in the derivation. (INC1 may be ignored here without loss of generality.) Use of INT3 implies that a Z exists such that

$$C' \vdash (V\ int\ Z)$$
$$C' \vdash (Z\ isa\ \mathbf{non}(V))$$

while use of INT4 implies

$$C' \vdash (V \; int \; U)$$
$$C' \vdash (V \; isa \; \textbf{non}(V))$$

In either of these situations **not**(c) = (X *isa* **non**(Y)) must be used somehow in one or both of the derivations, since $C \nvdash$ (V *int* **non**(V)).

1. In the INT3 case, depending upon whether (X *isa* **non**(Y)) is used in the first, the second, or both derivations, and upon how it is used, we get several possibilities:

   $$C \vdash \{(V \; int \; Z),\; (Z \; isa \; X),\; (\textbf{non}(Y) \; isa \; \textbf{non}(V))\}$$
   $$C \vdash \{(V \; int \; Z),\; (Z \; isa \; Y),\; (\textbf{non}(X) \; isa \; \textbf{non}(V))\}$$
   $$C \vdash \{(V \; int \; X),\; (\textbf{non}(Y) \; isa \; Z),\; (Z \; isa \; \textbf{non}(V))\}$$
   $$C \vdash \{(V \; int \; Y),\; (\textbf{non}(X) \; isa \; Z),\; (Z \; isa \; \textbf{non}(V))\}$$
   $$C \vdash \{(V \; int \; X),\; (\textbf{non}(Y) \; isa \; Z),\; (Z \; isa \; X),\; (\textbf{non}(Y) \; isa \; \textbf{non}(V))\}$$
   $$C \vdash \{(V \; int \; X),\; (\textbf{non}(Y) \; isa \; Z),\; (Z \; isa \; Y),\; (\textbf{non}(X) \; isa \; \textbf{non}(V))\}$$
   $$C \vdash \{(V \; int \; Y),\; (\textbf{non}(X) \; isa \; Z),\; (Z \; isa \; X),\; (\textbf{non}(Y) \; isa \; \textbf{non}(V))\}$$
   $$C \vdash \{(V \; int \; Y),\; (\textbf{non}(X) \; isa \; Z),\; (Z \; isa \; Y),\; (\textbf{non}(X) \; isa \; \textbf{non}(V))\}.$$

   For each possibility ISA2, ISA3, INT2 and INT3 yield $C \vdash (X \; int \; Y)$. But then $C \vdash c$, a contradiction.

2. The INT4 case is simpler. We discover we must have

   $$C \vdash \{(V \; int \; U), (V \; isa \; X), (V \; isa \; Y)\},$$

   from which INT4, INT2, and INT3 again obtain $C \vdash c$, a contradiction.

Hence $C' \nvdash$ (V *int* **non**(V)) for any such V, and the algorithm does not halt on its first step.

Thus I has the right properties if it is a model. If it is not a model, there is some constraint in C' that I violates.

1. This constraint cannot be (V *int* W). I is constructed to satisfy this constraint, unless W is **non**(V); but we have just determined that $C' \nvdash$ (V *int* **non**(V)), precluding this possibility.

2. The constraint cannot be (V *isa* W). If it were, clearly W could be neither V nor U, since then ISA0 or ISA1 would give a contradiction. Similarly W cannot be **non**(U) or **non**(V), since the TRIV rules would force $I(V) = \varnothing$ in the construction of I, and I would satisfy the constraint. Thus W must be an ordinary type descriptor, and there must be some element $\{\{Z_1, Z_2\}\}$ in $I(V)$ that is not in $I(W)$. However step 4 of the Model Extension Algorithm precludes this possibility also.

Thus there can be no constraints in C that are violated, and I is a model of C.

The case where $c = (X \; isa \; Y)$ is similar. □

Corollary A containment scheme ($T/U,C$) is unsatisfiable if and only if it is inconsistent.

Proof

If the constraint set C is inconsistent, there is some constraint c such that both $C \vdash c$ and $C \vdash \mathbf{non}(c)$. Clearly then C is not satisfiable.

Conversely, if C is unsatisfiable we know from Theorem 2 that $C \models (X \textit{ int } \mathbf{non}(X))$ for some X. By Theorem 5, then, $C \vdash (X \textit{ int } \mathbf{non}(X))$. But $C \vdash (X \textit{ isa } X)$, and $(X \textit{ isa } X) = \mathbf{not}\ (X \textit{ int } \mathbf{non}(X))$. So C is inconsistent. □

The rules listed above involve an amount of 'overhead' in dealing with trivial types and unsatisfiable constraint sets. This overhead complicates ordinary use in making inferences. In most real situations we would assume that:

1. *the constraints are satisfiable;*
2. *no type is trivial except* **non**(U).

To close this section, we show that the rules simplify considerably if these assumptions are known to be true.

Since the type descriptor **non**(U) is always trivial, we first introduce notation to distinguish it from others.

Definition

$X \not\equiv T$ iff X is none of the type descriptors $\{\ T, \mathbf{non}(\mathbf{non}(T)), \cdots\ \}$.

Now, if we verify that our containment scheme is satisfiable, and formally make the non-triviality assumptions,

INT0. $X \not\equiv \mathbf{non}(U) \vdash X \textit{ int } U$.
INT1. $X \not\equiv \mathbf{non}(U) \vdash X \textit{ int } X$.
ISA0. $\vdash X \textit{ isa } U$.
ISA1. $\vdash X \textit{ isa } X$.

then the rules analyzed earlier can be replaced with the following set:

INT2. $X \textit{ int } Y \vdash Y \textit{ int } X$.
INT3. $X \textit{ int } Y, Y \textit{ isa } Z \vdash X \textit{ int } Z$.
INT4. $X \not\equiv \mathbf{non}(U), X \textit{ isa } Y \vdash X \textit{ int } Y$.
ISA2. $X \textit{ isa } Y, Y \textit{ isa } Z \vdash X \textit{ isa } Z$.
ISA3. $X \textit{ isa } Y \vdash \mathbf{non}(Y) \textit{ isa } \mathbf{non}(X)$.
EQ0. $\alpha(\mathbf{non}(\mathbf{non}(X))) \vdash \alpha(X)$ [α any expression].
EQ1. $\alpha(X) \vdash \alpha(\mathbf{non}(\mathbf{non}(X)))$ [α any expression].

These rules are intuitive, and match the compressed syllogism rules R1-R5 closely.

6. Containment and Logic

It is possible to map all containment constraints into first-order predicate logic. The table below shows how this can be done, giving not only a logical equivalent, but also its 'skolemized' clausal form.

Constraint	*Predicate Logic Equivalent*	*Clause Equivalent*
X *isa* Y	$\forall x \; U(x) \supset (X(x) \supset Y(x))$	**not** $U(x) \vee$ **not** $X(x) \vee Y(x)$
X *int* Y	$\exists x \; U(x) \wedge X(x) \wedge Y(x)$	$U(\gamma) \wedge X(\gamma) \wedge Y(\gamma)$

For example, we have the following translations from containment constraints to clauses:

mammal isa vertebrate	**not** $U(x) \vee$ **not** *mammal* $(x) \vee$ *vertebrate* (x)
non(*vertebrate*) *isa* **non**(*mammal*)	**not** $U(x) \vee$ **not** *mammal* $(x) \vee$ *vertebrate* (x)
mammal isa **non**(*reptile*)	**not** $U(x) \vee$ **not** *mammal* $(x) \vee$ **not** *reptile* (x)
not(*vertebrate isa amphibian*)	$U(\gamma) \wedge$ *reptile* $(\gamma) \wedge$ **not** *amphibian* (γ)

Here γ is a (unique) skolem constant representing an object that is simultaneously a member of the types X, Y, and U. *Resolution* can be used directly on the clauses to derive inferences. For example, the resolvent of the clauses (**not** $X(x) \vee Y(x)$) and (**not** $Y(x) \vee Z(x)$) is (**not** $X(x) \vee Z(x)$), mirroring the inference rule ISA2. A thorough introduction to resolution proof techniques may be found in [5].

The use of U in the clauses is necessary to handle degeneracy properly. The key problem is that the predicate logic sentence

$$\forall x \, P(x)$$

is not well-defined when the quantification is over an empty domain. Including U in each clause makes the domain of quantification explicit.

Specifically, with a set of positive constraints C implying both (X *isa* **non**(X)) and (**non**(X) *isa* X), we now know from Theorem 1 that U must be trivial, and only the trivial interpretation is a model for C. However, if we were to omit the use of $U(x)$ in the clauses corresponding to C, resolution would be able to derive from C the clauses (**not** $X(x) \vee$ **not** $X(x)$) and ($X(x) \vee X(x)$), i.e.,

$$(\textbf{not}\, X(x)) \text{ and } (X(x)).$$

The resolvent of these two clauses is **false**, indicating incorrectly that C is unsatisfiable. By including U we obtain the clauses

$$(\textbf{not}\, U(x) \vee \textbf{not}\, X(x)) \text{ and } (\textbf{not}\, U(x) \vee X(x)),$$

with resolvent (**not** $U(x)$). This resolvent asserts there is no x for which $U(x)$ is true. In other words, U denotes $\varnothing$, as desired.

There are other connections between containment constraints and logic. Boole [2] points out that containment constraints correspond to propositional logic equations. He gives essentially the following table:

Containment constraint	*Propositional equation*
T_1 *isa* T_2	$T_2 = V \wedge T_1$
T_1 *isa* **non**(T_2)	**not** $T_2 = V \wedge T_1$
T_1 *int* T_2	$V \wedge T_2 = V \wedge T_1$
T_1 *int* **non**(T_2)	$V \wedge$ **not** $T_2 = V \wedge T_1$
non(T_1) *isa* T_2	$T_2 = V \wedge$ **not** T_1
non(T_1) *isa* **non**(T_2)	**not** $T_2 = V \wedge$ **not** T_1
non(T_1) *int* T_2	$V \wedge T_2 = V \wedge$ **not** T_1
non(T_1) *int* **non**(T_2)	$V \wedge$ **not** $T_2 = V \wedge$ **not** T_1

A new propositional variable V is generated for each containment constraint. We can infer then that (T_1 *isa* T_2) if we can show that $T_1 = W \wedge T_2$, and infer (T_1 *int* T_2) if we can show that $W_1 \wedge T_1 = W_2 \wedge T_2$, where W, W_1, W_2 are conjunctions of zero or more (possibly negated) variables.

In fact, when only positive constraints are considered, there is a direct connection between containment inference and logic:

Theorem 6.
Positive Binary Set Containment Inference and Propositional 2-CNF Unsatisfiability are equivalent. That is, for every scheme $(T/U,C)$ there is a propositional formula $f(C)$ in conjunctive normal form with at most two literals per clause, such that

1. C implies (X *isa* Y) **iff** $f(C) \wedge (X) \wedge$ (**not** Y) is unsatisfiable.
2. C has only the trivial model **iff** $f(C)$ is unsatisfiable.

Proof

Propositional 2-CNF instances are formulas f on variables $V_1, \ldots, V_n$ in conjunctive normal form,

$$f = D_1 \wedge \cdots \wedge D_N$$

where each disjunct D_i has one of the two forms

$$D_i = (L_{i1})$$
$$D_i = (L_{i1} \vee L_{i2}),$$

and each 'literal' L_{ij} is either a variable V or its complement **not** V.

Now in a containment scheme $(T/U,C)$, C is a set of constraints

$$C = \{(X_1 \ \textit{isa} \ Y_1), (X_2 \ \textit{isa} \ Y_2), \cdots, (X_m \ \textit{isa} \ Y_m)\}.$$

Define a corresponding 2-CNF formula

$$f(C) = (\textbf{not}\, X_1 \vee Y_1) \wedge (\textbf{not}\, X_2 \vee Y_2) \wedge \cdots \wedge (\textbf{not}\, X_m \vee Y_m) \wedge U \wedge \bigwedge_{i=1}^{n} (\textbf{not}\, T_i \vee U).$$

We claim that $f(C)$ is satisfiable iff C is nontrivially satisfiable. In fact, if C is nontrivially satisfiable then it has a model I such that $I(U) = \{t\}$, a singleton set.

To see this, note that if C is nontrivially satisfiable, then all of the constraints in C are satisfied. Since the i-th constraint

$$X_i \ isa \ Y_i$$

is equivalent to

$$(\mathbf{non}(X_i) \cup Y_i) = U.$$

It follows that the set expression

$$(\mathbf{non}(X_1) \cup Y_1) \cap (\mathbf{non}(X_2) \cup Y_2) \cap \cdots \cap (\mathbf{non}(X_m) \cup Y_m)$$

evaluates to U if, and only if, all constraints in C are satisfied.

We now assert that there is a nontrivial model I of C such that $I(U) = \{t\}$, a singleton set. To see this, note that we can use the Model Extension Algorithm given earlier on the trivial interpretation, which is a model since we have only positive constraints. The resulting interpretation I is a model with $I(U) = \{t\}$, and $I(T_i) \subseteq \{t\}$ for each type symbol T_i in T.

With this model we can construct a truth assignment

$$truth(T_i) = \begin{cases} \mathbf{true} & \text{if } I(T_i) = \{t\} \\ \mathbf{false} & \text{if } I(T_i) = \varnothing \end{cases}$$

Analogously, then, $f(C)$ evaluates to **true** under this truth assignment if, and only if, the interpretation I is a model satisfying all constraints in C. Hence C is nontrivially satisfiable if and only if $f(C)$ is satisfiable. □

This theorem shows there is an inference procedure using resolution on the propositional equivalent $f(C)$ of a set of positive clauses. We check only for a proof of **false** from the propositional logic equivalent of the containment constraints conjoined with the negation of the constraint whose inferrability we wish to test. The algorithm is guaranteed to complete in time at most $O(n^3)$, where n is the number of types in the scheme.

Example

Take the positive binary constraint set

$$C = \{ \ a \ isa \ \mathbf{non}(b), \ \ c \ isa \ b, \ \ \mathbf{non}(d) \ isa \ c \ \}.$$

The corresponding propositional formula $f(C)$ is

$$\begin{aligned} f(C) \ = \ & (\mathbf{not}\, a \vee \mathbf{not}\, b) \wedge (\mathbf{not}\, c \vee b) \wedge (d \vee c) \\ & \wedge U \wedge (\mathbf{not}\, a \vee U) \wedge (\mathbf{not}\, b \vee U) \wedge (\mathbf{not}\, c \vee U) \wedge (\mathbf{not}\, d \vee U). \end{aligned}$$

Just as ISA2 and ISA3 imply

$$C \vdash (a \ isa \ d),$$

we determine that $(\mathbf{not}\, a \vee d)$ is a resolvent of $f(C)$, whence

$$f(C) \text{ logically implies } (\textbf{not } a \vee d)$$

or equivalently

$$f(C) \wedge (a) \wedge (\textbf{not } d) \quad \text{is unsatisfiable.}$$

7. Concluding Remarks

This paper generalizes the results in [1] to consider negation in various ways. The results are encouraging, in that only a few rules are needed for complete binary containment inference. As long as we are interested only in binary properties of containment among sets, this gives us a complete inference system for set theory.

Perhaps the first work to be done is in developing algorithms using the inference systems presented here. Algorithms are beyond the scope of what we wished to present here, but the inference systems developed in this paper all run in polynomial time. A simple upper bound is $O(n^3)$ where n is the number of types. Improved bounds will follow where more is known about the type structure. For instance, few real type hierarchies seem to be very deep. Even the standard biological taxonomy of living creatures is only about 10 levels deep.

When we begin to consider more complex forms of knowledge about types, such as sentences like

$$(\mathit{amphibian} \cap \mathbf{non}(\mathit{tailed})) \subseteq (\mathit{frog} \cup \mathit{toad})$$

the containment problem becomes NP-complete, and the inference problem co-NP-complete. Still, it would be interesting to extend the binary rules in this paper for the more general inference problem.

Other directions for further research lie in exploring graphical representations of the constraints, as in [1], and in providing efficient algorithms for containment problems. Also, there are a variety of ways to generalize the problems discussed here that the reader has no doubt already considered. These include investigating alternative types of 'syllogisms', restricting values of U in interpretations (for example, specifying $I(U)$ in advance), and so forth. The general area of containment inference is a new area in which many problems wait to be studied.

Acknowledgement

Richard Huntsinger, Karen Lever, and Tom Verma gave the paper a careful reading and suggested important improvements in the presentation.

Appendix I: NP-Completeness of the Set Containment Problem

NP-completeness of the Set Containment Problem can be shown by reducing the well-known SAT problem [4] directly to it.

To show the problem is in NP, notice that a set containment scheme $(T/U,C)$ has a model if and only if it has a model over a domain D with cardinality at most N_{int}, the number of intersection constraints in C. (Given a model I having domain D with cardinality greater than N_{int}, a model I' restricting I to a subdomain of cardinality N_{int} can be constructed by repeatedly discarding members of D that are not solely used to satisfy some intersection constraint.) Thus we can solve the satisfiability problem for set containment schemes by nondeterministically guessing an interpretation I having domain of size N_{int}, then checking that I satisfies the scheme.

To make the reduction from SAT to set containment, suppose we are given a propositional formula in conjunctive form,

$$f = D_1 \wedge \cdots \wedge D_m$$

on variables $V_1, \ldots, V_n$. Each disjunct D_i is given by

$$D_i = (L_{i1} \vee \cdots \vee L_{ir_i}),$$

where each L_{ij} is either some variable V or its complement **not** V.

We construct a corresponding set containment scheme $S = (T/U,C)$ by putting $T/U = \{U, V_1, \ldots, V_n\}$, and defining

$$C = \{U \ int \ U\} \cup \{ C_i \mid 1 \leq i \leq m \}$$

as follows.

For each literal L_{ij} $1 \leq i \leq m$, $1 \leq j \leq r_i$, in the disjuncts $D_i = (L_{i1} \vee \cdots \vee L_{ir_i})$, define

$$K_{ij} = \begin{cases} V_k & \text{if } L_{ij} = V_k \\ \textbf{non}(V_k) & \text{if } L_{ij} = \textbf{not } V_k \end{cases}$$

and let C_i be the constraint

$$C_i\text{: } (K_{i1} \cup \cdots \cup K_{ir_i}) = U.$$

for $1 \leq i \leq m$. Since any constraint

$$\textbf{non}(X_1) \cup \cdots \cup \textbf{non}(X_p) \cup Y_1 \cup \cdots \cup Y_q = U$$

is equivalent to

$$X_1 \cap \cdots \cap X_p \ \ isa \ \ Y_1 \cup \cdots \cup Y_q,$$

C consists of the positive constraints C_i, $1 \leq i \leq m$, and the negative constraint $U \ int \ U$. This defines S.

We claim that S is satisfiable if and only if the original formula f is satisfiable. If f is satisfiable, then there is a truth assignment *truth* satisfying f. Letting $I(U) = \{t\}$ (a singleton set),

$$I(V_k) = \begin{cases} \{t\} & \text{if } truth(V_k) = \textbf{true} \\ \varnothing & \text{if } truth(V_k) = \textbf{false} \end{cases}$$

for $1 \le k \le n$. This interpretation can be seen to satisfy all the constraints in C since the corresponding truth assignment satisfies f, and U *int* U is satisfied iff $I(U)$ is nonempty.

Conversely, if S has a model, then as shown above since it has one intersection constraint it must have a model I that assigns $I(U) = \{t\}$, a domain of cardinality one. Reversing the argument above, we find I induces a truth assignment satisfying f.

Acknowledgement

Richard Huntsinger, Karen Lever, and Tom Verma gave the paper a careful reading and suggested important improvements in the presentation.

References

1. Atzeni, P. and D. Stott Parker, Formal Properties of Net-based Knowledge Representation Schemes, *Proc. 2nd Conference on Data Engineering*, Los Angeles, CA, February 1986.
2. Boole, G., *An Investigation of The Laws of Thought,* Dover, 1958.
3. Gardner, M., *Logic Machines and Diagrams,* University of Chicago Press, 1982.
4. Garey, M.R. and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W.H. Freeman and Co., San Francisco, 1979.
5. Loveland, D., *Automated Theorem Proving,* North-Holland, 1976.
6. Maier, D., *The Theory of Relational Data Bases,* Computer Science Press, Rockville, MD, 1983.
7. Ullman, J.D., *Principles of Data Base Systems, 2nd Ed.,* Computer Science Press, Rockville, MD, 1984.