Implementation and Optimization Techniques

Description Logic Handbook - Chapter 9

Lukasz Cwik

University of Waterloo

June 11th, 2008

Introduction

1. Motivation
2. Problem
3. Syntax and Semantics
4. Negation Normal Form

Motivation

For Description Logics to be useful in realistic applications we want:

- Expressivity
- Fast reasoners

Previous Description Logic examples:

1. European GALEN project - medical terminology
2. KL-ONE by Brachman and Schmolze circa 1985
3. CLASSIC by Patel-Schneider *et al.* circa 1991
4. GRAIL be Reactor *et al.* circa 1997

Problem

Expressive logics have high worst-case complexities so we must use highly optimized implementations of suitable reasoners.

So we will explore the following techniques:

- Lazy Unfolding
- Internalization
- Tracing
- Normalization

Syntax and Semantics

Regular Tarski style semantics $(\Delta^I, \cdot^I)$ where Δ^I is the domain and $\cdot^I$ is the interpretation function.

C, D	- arbitrary concepts
R, S	- arbitrary roles
A, P	- atomic concepts
$\sqcap$	- conjunction
$\sqcup$	- disjunction
$\neg$	- negation
$\exists R.C$	- existential role restrictions
$\forall R.C$	- universal role restrictions

Negation Normal Form

Purpose

Negations only apply to concepts.

ALC concepts can be converted to negation normal form by:

1. DeMorgan's laws
   - $\neg(C \cup D) \longleftrightarrow \neg C \cap \neg D$
   - $\neg(C \cap D) \longleftrightarrow \neg C \cup \neg D$
2. $\neg\neg C \longleftrightarrow C$
3. $\neg(\exists R.C) \longleftrightarrow (\forall R.\neg C)$
4. $\neg(\forall R.C) \longleftrightarrow (\exists R.\neg C)$

Negation Normal Form Example

Rules

1. DeMorgan's laws
   - $\neg(C \cup D) \longleftrightarrow \neg C \cap \neg D$
   - $\neg(C \cap D) \longleftrightarrow \neg C \cup \neg D$
2. $\neg\neg C \longleftrightarrow C$
3. $\neg(\exists R.C) \longleftrightarrow (\forall R.\neg C)$
4. $\neg(\forall R.C) \longleftrightarrow (\exists R.\neg C)$

Example

- $\neg(\neg C \cup D) \cap \neg\exists R.D$

Negation Normal Form Example

Rules

1. DeMorgan's laws
   - $\neg(C \cup D) \longleftrightarrow \neg C \cap \neg D$
   - $\neg(C \cap D) \longleftrightarrow \neg C \cup \neg D$
2. $\neg\neg C \longleftrightarrow C$
3. $\neg(\exists R.C) \longleftrightarrow (\forall R.\neg C)$
4. $\neg(\forall R.C) \longleftrightarrow (\exists R.\neg C)$

Example

- $\neg(\neg C \cup D) \cap \neg\exists R.D$ - apply Rule 1
- $(\neg\neg C \cap \neg D) \cap \neg\exists R.D$

Negation Normal Form Example

Rules

1. DeMorgan's laws
   - $\neg(C \cup D) \longleftrightarrow \neg C \cap \neg D$
   - $\neg(C \cap D) \longleftrightarrow \neg C \cup \neg D$
2. $\neg\neg C \longleftrightarrow C$
3. $\neg(\exists R.C) \longleftrightarrow (\forall R.\neg C)$
4. $\neg(\forall R.C) \longleftrightarrow (\exists R.\neg C)$

Example

- $\neg(\neg C \cup D) \cap \neg\exists R.D$ - apply Rule 1
- $(\neg\neg C \cap \neg D) \cap \neg\exists R.D$ - apply Rule 2
- $(C \cap \neg D) \cap \neg\exists R.D$

Negation Normal Form Example

Rules

1. DeMorgan's laws
   - $\neg(C \cup D) \longleftrightarrow \neg C \cap \neg D$
   - $\neg(C \cap D) \longleftrightarrow \neg C \cup \neg D$
2. $\neg\neg C \longleftrightarrow C$
3. $\neg(\exists R.C) \longleftrightarrow (\forall R.\neg C)$
4. $\neg(\forall R.C) \longleftrightarrow (\exists R.\neg C)$

Example

- $\neg(\neg C \cup D) \cap \neg\exists R.D$ - apply Rule 1
- $(\neg\neg C \cap \neg D) \cap \neg\exists R.D$ - apply Rule 2
- $(C \cap \neg D) \cap \neg\exists R.D$ - apply Rule 3
- $(C \cap \neg D) \cap \forall R.\neg D$

Negation Normal Form Example

Rules

1. DeMorgan's laws
   - $\neg(C \cup D) \longleftrightarrow \neg C \cap \neg D$
   - $\neg(C \cap D) \longleftrightarrow \neg C \cup \neg D$
2. $\neg\neg C \longleftrightarrow C$
3. $\neg(\exists R.C) \longleftrightarrow (\forall R.\neg C)$
4. $\neg(\forall R.C) \longleftrightarrow (\exists R.\neg C)$

Example

- $\neg(\neg C \cup D) \cap \neg\exists R.D$ - apply Rule 1
- $(\neg\neg C \cap \neg D) \cap \neg\exists R.D$ - apply Rule 2
- $(C \cap \neg D) \cap \neg\exists R.D$ - apply Rule 3
- $(C \cap \neg D) \cap \forall R.\neg D$ - Done!

Subsumption Testing Algorithms

1. Structural Subsumption Algorithms
2. Logical Algorithms
3. Logical Algorithms using existing Reasoners

Structural Subsumption Algorithms

To determine if one concept subsumes another, structural algorithms simply compare the (normalised) syntactic structure of the two concepts.

Advantages

- Generally easy to demonstrate the soundness of the structural inference rules
- Very efficient

Disadvantages

- Usually incomplete (may fail to infer all valid subsumption relationships)
- Difficult to extend structural algorithms to deal with more expressive logics

Logical Algorithms

These kinds of algorithm use a refutation style proof, we try to show that $C \sqsubseteq D$ by showing that for some individual x:

$$x \in C^I \text{ and } x \notin D^I \text{ is logically inconsistent.}$$

So in the case of description logics, this corresponds to testing the logical (un)satisfiability of the concept $C \sqsubseteq D$ iff $C \sqcap \neg D$ is not satisfiable.

Not satisfiability approach

- Sound theoretical basis in first order logic
- Allows for a range of logical languages by changing the set of tableaux expansion rules
- Very expressive logics
- Optimal for a number of description logic languages

Logical Algorithms using existing Reasoners

Most description logic systems using logical algorithms use existing reasoners such as LOGICS WORKBENCH, KSAT/*SAT, or SPASS

Advantages

- Easier to build a system using an existing reasoner.
- Able to use state of the art implementations of existing reasoners
- Allows to deal with more expressive description logics

Disadvantages

- Difficult to extend the reasoner to add optimizations that take advantage of specific features of a description logic
- May have to rewrite the reasoner to handle more expressive description logics

Tableaux Algorithms

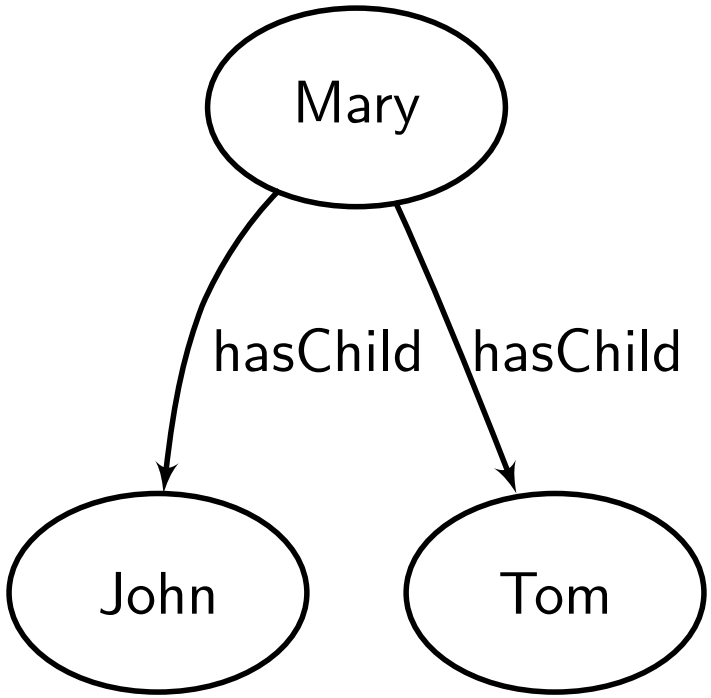
1. What is a Tableau?
2. Purpose of a Tableau
3. How a Tableau Algorithm works
4. Graph Notation/Definitions
5. Tableaux expansion rules for ALC
6. "$\sqcup$" Non-Determinism

What is a Tableau?

Definition

A *tableau* is a graph which represents a model, with nodes corresponding to individuals (elements of Δ^I) and edges corresponding to relationships between individuals (elements of $\Delta^I X \Delta^I$).

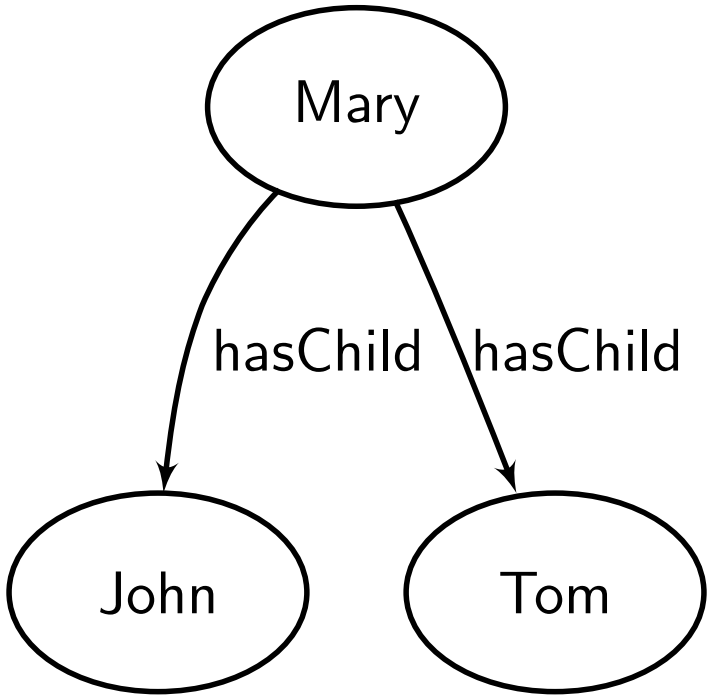
Example



Purpose of a Tableau

The purpose of a tableau is to try to prove the satisfiability of a concept C by constructing a *model*, and interpretation I in which C^I is not empty

Example



How a Tableau Algorithm works

A typical algorithm will start with a single individual satisfying D and try to construct a tableau, by inferring the existence of additional individuals or of additional constraints on individuals.

The inference mechanism consists of applying a set of expansion rules which correspond to the logical constructs of the language, and the algorithm terminates either when the structure is complete (no further inferences are possible) or obvious contradictions have been revealed.

A contradiction or *clash* is detected when $C, \neg C$ occur.

Graph Notation

1. each node x is labelled with a set of concepts $(L(x) = C_1, C_2, ..., C_n)$
2. each edge $< x, y >$ is labelled with a role $(L(< x, y >) = R)$.
3. If $C \in L(x)$, it represents a model in which the individual corresponding with x is in the interpretation of C
4. If $L(< x, y >) = R$, it represents a model in which the tuple corresponding with $< x, y >$ is in the interpretation of R
5. A node y is called an R-successor of a node x if there is an edge $< x, y >$ labelled R, x is called the predecessor of y, also if y is an R-successor of z, then x is called an ancestor of z

Tableaux expansion rules for ALC

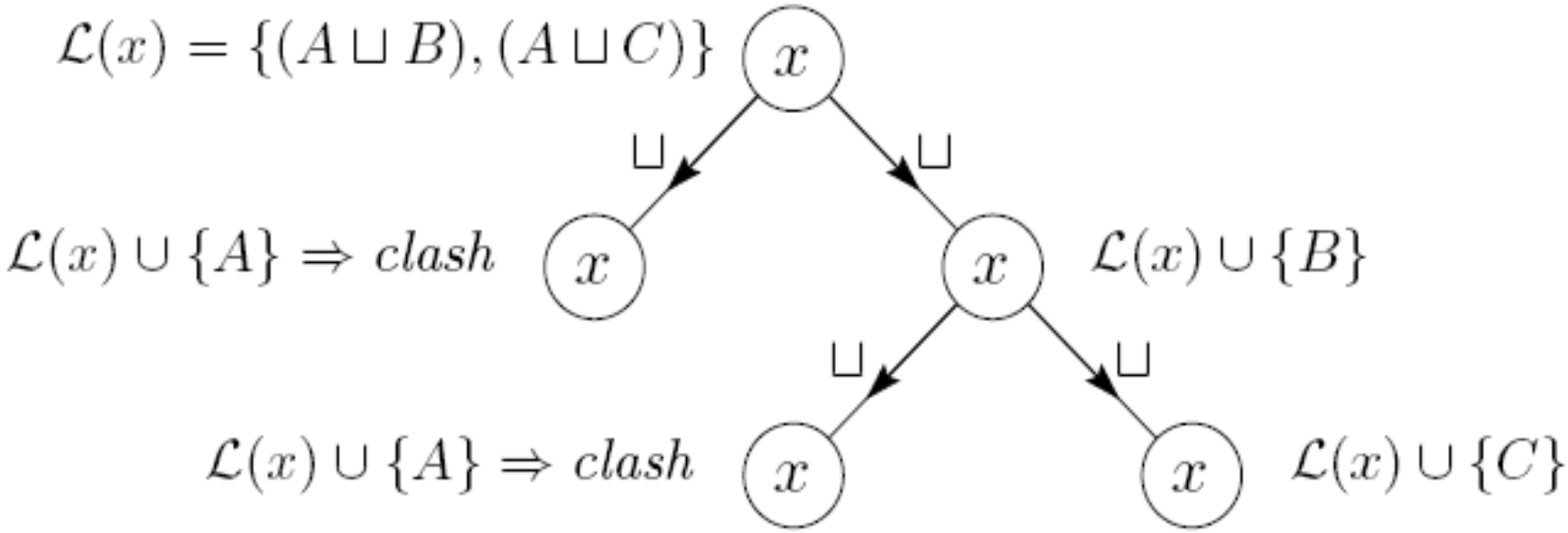
$\sqcap$-rule	if 1.	$(C \sqcap D) \in L(x)$
	2.	$\{C, D\} \not\subseteq L(x)$
	then	$L(x) \longrightarrow L(x) \cup \{C, D\}$
$\sqcup$-rule	if 1.	$(C \sqcup D) \in L(x)$
	2.	$\{C, D\} \cap L(x) = \emptyset$
	then	*either* $L(x) \longrightarrow L(x) \cup \{C\}$
		or $L(x) \longrightarrow L(x) \cup \{D\}$
$\exists$-rule	if 1.	$\exists R.C \in L(x)$
	2.	there is no y s.t. $L(< x, y >) = R$ and $C \in L(y)$
	then	create a new node y and edge $< x, y >$
		with $L(y) = \{C\}$ and $L(< x, y >) = R$
$\forall$-rule	if 1.	$\forall R.C \in L(x)$
	2.	there is some y s.t. $L(< x, y >) = R$ and $C \notin L(y)$
	then	then $L(y) \longrightarrow L(y) \cup \{C\}$

"⊔" Non-Determinism

Non-determinism is dealt with by searching different possible expansions.

The concept is unsatisfiable if every expansion leads to a contradiction and is satisfiable if any possible expansion leads to the discovery of a complete non-contradictory structure.

Example (If A is unsatisfiable, B and C satisfiable)



Reducing a DL to Satisfiability and Subsumption Testing

1. Unfolding
2. Internalization

Unfolding

Purpose

Subsumption testing can be made to be independent of T:

$$T \models C \sqsubseteq D \longleftrightarrow \emptyset \models \mathit{Unfold}(C, T) \sqsubseteq \mathit{Unfold}(D, T)$$

1. $A \equiv D$, substitute A with D everywhere it occurs in C
2. $A \sqsubseteq D$, substitute A with the concept $A' \sqcap D$ where A' is a new concept name

Unfolding

Purpose

Subsumption testing can be made to be independent of T:

$$T \models C \sqsubseteq D \longleftrightarrow \emptyset \models \mathit{Unfold}(C, T) \sqsubseteq \mathit{Unfold}(D, T)$$

1. $A \equiv D$, substitute A with D everywhere it occurs in C
2. $A \sqsubseteq D$, substitute A with the concept $A' \sqcap D$ where A' is a new concept name

Questions

1. Why should the definitions be acyclic? $(\{A \sqsubseteq \exists R.A\} \notin T)$

Unfolding

Purpose

Subsumption testing can be made to be independent of T:

$$T \models C \sqsubseteq D \longleftrightarrow \emptyset \models \mathit{Unfold}(C, T) \sqsubseteq \mathit{Unfold}(D, T)$$

1. $A \equiv D$, substitute A with D everywhere it occurs in C
2. $A \sqsubseteq D$, substitute A with the concept $A' \sqcap D$ where A' is a new concept name

Questions

1. Why should the definitions be acyclic? $(\{A \sqsubseteq \exists R.A\} \notin T)$
2. Why should the definitions be unique? $(\{A \equiv C, A \equiv D\} \notin T)$

Internalization

Purpose

The concept axioms in T can be reduced to the form $\top \sqsubseteq C$, which is equivalent to testing satisfiability of a concept

1. $A \equiv B \longleftrightarrow \top \sqsubseteq (A \sqcup \neg B) \sqcap (\neg A \sqcup B)$
2. $A \sqsubseteq B \longleftrightarrow \top \sqsubseteq \neg A \sqcup B$

Optimization Techniques

1. Theory vs. Practice
2. Optimization Types
3. Lazy Unfolding
4. Internalization Technique
5. Trace Technique
6. Normalization

Theory vs. Practice

Many times there are gaps between the theory and the implementation so in practice we also need to consider:

- The efficiency of the algorithm, in the theoretical (worst case) sense
- The efficiency of the algorithm, in a practical (average case) sense
- How to use the the algorithm with unfoldable, general and cyclic knowledge bases
- How to optimize the implementation of the algorithm for the average case

Optimization Types

1. Preprocessing, makes classification and subsumption testing easier
2. Parital ordering, minimize the number of subsumption tests required
3. Subsumption optimizations that replace expensive satisfiability tests with cheaper ones
4. Satisfiability optimizations that try to improve the typical case performance of the underlying satisfiability tester

Lazy Unfolding

U_1-rule if 1. $A \in L(x)$ and $(A \equiv C) \in T$
2. $C \notin L(x)$
then $L(x) \longrightarrow L(x) \cup \{C\}$

U_2-rule if 1. $\neg A \in L(x)$ and $(A \equiv C) \in T$
2. $\neg C \notin L(x)$
then $L(x) \longrightarrow L(x) \cup \{\neg C\}$

U_3-rule if 1. $A \in L(x)$ and $(A \sqsubseteq C) \in T$
2. $C \notin L(x)$
then $L(x) \longrightarrow L(x) \cup \{C\}$

Internalization Technique

I_1-rule if 1. $(C \equiv D) \in T$
2. $(D \sqcup \neg C) \notin L(x)$
then $L(x) \longrightarrow L(x) \cup \{(D \sqcup \neg C)\}$

I_2-rule if 1. $(C \equiv D) \in T$
2. $(\neg D \sqcup C) \notin L(x)$
then $L(x) \longrightarrow L(x) \cup \{(\neg D \sqcup C)\}$

I_3-rule if 1. $C \sqsubseteq D \in T$
2. $(D \sqcup \neg C) \notin L(x)$
then $L(x) \longrightarrow L(x) \cup \{(D \sqcup \neg C)\}$

Internalization Technique

I_1-rule if 1. $(C \equiv D) \in T$
2. $(D \sqcup \neg C) \notin L(x)$
then $L(x) \longrightarrow L(x) \cup \{(D \sqcup \neg C)\}$

I_2-rule if 1. $(C \equiv D) \in T$
2. $(\neg D \sqcup C) \notin L(x)$
then $L(x) \longrightarrow L(x) \cup \{(\neg D \sqcup C)\}$

I_3-rule if 1. $C \sqsubseteq D \in T$
2. $(D \sqcup \neg C) \notin L(x)$
then $L(x) \longrightarrow L(x) \cup \{(D \sqcup \neg C)\}$

Question

1. Why do these rules seem bad in a performance sense?

Trace Technique

Motivation

- Minimize space usage
- Sensible way of organizing the expansion and flow of control

How?

Choose a good ordering on what rules we apply

ALC Trace Rule

$\exists\forall$-rule if 1. $\exists R.C \in L(x)$
2. there is no y s.t. $L(< x, y >) = R$ and $C \in L(y)$
3. neither the $\sqcap$-rule nor the $\sqcup$-rule is applicable to $L(x)$

then create a new node y and edge $< x, y >$
with $L(y) = C \cup D | \forall R.D \in L(x)$ and $L(< x, y >) = R$

Normalization

Why?

Why only detect clashes when $\{C, \neg C\} \subseteq L(x)$?
Why not a direct contradiction between $(C \sqcap D)$ and $(\neg D \sqcup \neg C)$?

Normalization

Why?

Why only detect clashes when $\{C, \neg C\} \subseteq L(x)$?
Why not a direct contradiction between $(C \sqcap D)$ and $(\neg D \sqcup \neg C)$?

How?

Re-use work done in structural subsumption algorithms

Example

- Put an order on concept names
- Use DeMorgan's laws
- ...

Normalization Pros/Cons

Advantages

- Easy to implement and could be used with most logics
- Subsumption/satisfiability problems can be simplified or completely avoided
- Complements lazy unfolding
- May lead to more compact storage of the knowledge base

Disadvantages

- Adds a little overhead to everything for potentially no gain
- For unstructured knowledge bases, may even increase the size

Conclusion

We have looked at a variety of techniques and there are many more worth investigating such as:

- Absorption
- Satisfiability caching
- Semantic branching search
- Dealing with thrashing
- Pruning using Backjumps
- ...