Functional Logic Programming: $\mathcal{FL}^-$ subsumption check in Prolog

June 17, 2010

Introduction

$\mathcal{FL}^-$ is one of the most simple DL available. $\mathcal{FL}^-$ (being a DL) is decidable, moreover its limited expressive power allows us to define an algorithm to check subsumption that is PTime. Due to its simplicity the algorithm for deciding subsumption works only on the structure of the formula, i.e. only at the syntactical level. Other Description Logics (e.g. $\mathcal{ALC}$) do not enjoy this property because they are too expressive. In such cases algorithms based on Tableaux are usually implemented.

We developed a prolog program that, given two $\mathcal{FL}^-$ complex concepts C and D, decides whether $C \sqsubseteq D$. In particular after loading the program, at the prompt it is possible to pose the query:

```
?-  subs( c , d ).
```

and the interpreter returns True if $c \sqsubseteq d$, and False otherwise.[1]. Moreover it is possible to define complex concept to use inside other concepts by typing:

```
?- define(c, all(child,adult)).
?- subs( and(all(child,adult),some(child)), c)
```

The list of defined concepts can be retrieved with

```
?- listdefine.
```

[1] Concepts are expressed using the syntax defined in Section 2.

1 $\mathcal{FL}^-$

$\mathcal{FL}^-$is a structural DL with a limited expressive power where we can only talk about concepts. Roles can be used to express complex concepts, but we cannot say anything directly about them.

1.1 Syntax

Considering A as an atomic concept, R as a role and C,D as complex concepts, we can define $\mathcal{FL}^-$syntax in one line:

$$C, D := A|C \sqcap D|\exists R|\forall R.C$$

We also introduce a different syntax to express $\mathcal{FL}^-$that is called *functional syntax*:

$$\begin{aligned} \textit{concept} := \quad & < \textit{atomic-concept} > | \\ & : \textit{and} < \textit{concept} > .... < \textit{concept} > | \\ & : \textit{some} < \textit{atomic-role} > | \\ & : \textit{all} < \textit{atomic-role} >< \textit{concept} > \end{aligned}$$

Once concepts are defined (in either syntax) we can define the $\mathcal{T}Box$ by defining the inclusion assertions, that in $\mathcal{FL}^-$are limited to the ones of the form:

$$C \sqsubseteq D$$

with C,D complex concepts. To better understand what this means we are going to briefly introduce the semantic of $\mathcal{FL}^-$, althought this is not needed since the subsumption problem is solved at a purely syntactical level.

1.2 Semantics

As for all DLs, an interpretation $\mathcal{I}$ is a tuple $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ where

- $\Delta^{\mathcal{I}}$ is a non empty set
- $\cdot^{\mathcal{I}}$ is a mapping function that associates to every concept C a subset $C^{\mathcal{I}}$of $\Delta^{\mathcal{I}}$ and to every role R a subset $R^{\mathcal{I}}$ of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$

We can define the extension of $\cdot^{\mathcal{I}}$ for the $\mathcal{FL}^-$operators as:

$$\begin{aligned} (C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\ (\forall R.C)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} | \forall y.(x,y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\} \\ (\exists .R)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} | \exists y.(x,y) \in R^{\mathcal{I}}\} \end{aligned}$$

Now we can define more formally the subsumption problem as:

$$C \sqsubseteq D \text{ iff } C^{\mathcal{I}} \subseteq D^{\mathcal{I}} \ \forall \mathcal{I}$$

1.3 Subsumption checking algorithm

Given two complex concepts C,D we want to tell whether $C \sqsubseteq D$. The idea of this algorithm is that in order for $C^{\mathcal{I}}$ to be a subset of $D^{\mathcal{I}}$, all the concepts appearing in D (or a more *specific* version) must also appear in C.

To do so we need to proceed as follows

Convert C and D into a normal form. This conversion is split in two steps:

1. Flatten the formula,
2. Factorize the universal quantifiers.

In the first step we take all the conjunctions that are nested and we flatten them: eg. $A \sqcap (B \sqcap C)$ become $A \sqcap B \sqcap C$. Note that this step is necessary since we define a syntax in which $\sqcap$ is binary.

The second step takes all the universal quantifiers that have the same role, and returns the conjunction of the concepts: eg $\forall R.C \sqcap \forall R.D$ becomes $\forall R.(C \sqcap D)$.

Recursively check subsumption of the concepts in C and D . For all the elements D_i of D, we need to check:

If D_i is an atomic-concept or an existential: there must exists a C_i in C s.t. $C_i = D_i$

If D_i is a universal: then, given $D_i = \forall R.D'$ there must be $C_i = \forall R.C'$ in C s.t. $C' \sqsubseteq D'$.

This short description is, in fact, the whole algorithm to check subsumption.

2 Prolog implementation

By choosing appropriate data structures we can exploit many features of prolog to write this program. In particular we decided to use a syntax that is similar to the functional one presented in 1.1. In particular we map the $\mathcal{FL}^-$elements to the following prolog elements:

- Atomic Concepts $\rightarrow$ Constants (c)
- Atomic Roles $\rightarrow$ Constants (r)
- Conjunction $\rightarrow$ $and(C1, C2)$
- Existential $\rightarrow$ $some(R)$
- Universal $\rightarrow$ $all(R, C)$

The use of functors allows us to exploit unification to do pattern matching. Here we define $and/2$ as a binary atom, but during the flattening step we will need to have more conjuncts. Thus we decided to use lists (internally) to represent conjunction. ie. $and(a, and(b, c))$ becomes `[a,b,c]`.

The implementation strictly resambles the description above:

1. `normalize/2` takes a $\mathcal{FL}^-$ formulas and returns the normal form,

2. `r_subs/2` takes two $\mathcal{FL}^-$ normalized concepts and performs the syntactic check.

A short description of the different rules is provided here for reference. Unfortunately prolog programs require much more effort to be understood from the code, so we thought it would be helpful to provide a general description of what those rules are for.

2.1 Normal Form

We build the normal form by performing the steps described previously. The only addition is step 1 that is used to implement the `define/2` functionality.

1. `remove_aliases/2` recursively navigates C and D and subsitutes the complex concepts that have been defined using the `define/2` command. This command modifies the program facts database, using the `assert/1` clause: eg. `assert(alias(c,and(a,b))).`

2. All nested conjunctions are flattened. ie. $A \sqcap (B \sqcap C) \rightarrow A \sqcap B \sqcap C$. This is easy to do with lists: `[A,[B,C]]` $\rightarrow$ `[A,B,C]`. In particular this operation is performed by `flatten/2`, that performs a smart list concatenation (`concat/3`) in which nested list are flattened (`flatlist/2`).

3. Conjunctions of universal quantifications are factorized. ie $\forall R.C \sqcap \forall R.D \rightarrow \forall R.(C \sqcap D)$: `[all(R,C),all(R,D)]` $\rightarrow$ `[all(R,[C,D])]`. This is performed by `factorize/2` that first sorts the list of concepts (`sort_concept/2`) and then performs the real recursive factorization `factorize_/2`. The sorting is performed in order to have a simpler factorization factor: this way if two universal refer to the same role they are adjacent in the list, and there's no need to scan the whole list. The sort itself is an insertion sort (`insert_sort/3`) that makes use of a partial ordering function (`bigger/2`)[2].

2.2 Structural check

There are 4 rules with similar names:

- `subs/2` is the main rule, that calls `lsubs/2` with a cut to avoid backtracking (`once`). This was added because subsumption is a decision problem, so we are interested in a Yes/No answer, and we don't want to search for alternatives answers.

- `lsubs/2` calls `r_subs/2` after normalizing C and D,

- `r_subs/2` performs the real recursive subsumption check by calling `r_subs_i/2` on the head element of D, and making the recursion on the tail;

[2] Note the use of negation as failure to avoid the need of defining a converse rule *smaller*/2

- `r_subs_i/2` uses `find/2` to look in C for the element D_i in case D_i it's either an atomic concept or an existential. Otherwise, if $D_i = \forall R.D'$ it calls `get_role_concept/3` that returns $C_j = \forall R.C'$ (if present in C) and then runs the subsumption check on C' and D'.

3 Problems encountered

Most of the problems were related to the use of not purely logical predicates like `atom` and `cut`. In fact it is not possible to run this query:

```
?- subs(X,c).
```

What is still possible is to ask for `?- r_subs(X,[c]).` by skipping the normalization. This query will return (infinitely many) correct results. We consider this not to be a big loss, since given a concept D there are infinitely many concepts C s.t. $C \sqsubseteq D$.

We didn't have to debug too much, since we tried to keep rules simple and as atomic as possible. Nevertheless when the program was almost complete, a nasty typo forced us to spend a lot of time on debugging. In this case we found the graphical trace interface (`guitracer/0`) to be particularly convinient to use.

After spending some time debugging we decided to introduce some change managment in our project by adding test units. We found out that SWI-Prolog defines some interesting methods to write real test units, so we decided to write test for relevant rules and include also some subsumption examples:

	C	D	$C \sqsubseteq D$
1.	$\forall CHILD.Adult \sqcap \exists CHILD$	$\forall CHILD.Adult$	True
2.	$Adult \sqcap Male$	$Adult$	True
3.	$Adult \sqcap Male \sqcap Rich$	$Adult \sqcap Male$	True
4.	$\forall CHILD.(Adult \sqcap Male)$	$\forall CHILD.Adult$	True
5.	$\forall CHILD.Adult$	$\exists CHILD$	False
6.	$\exists CHILD$	$\forall CHILD.Adult$	False
7.	$\forall CHILD.Adult \sqcap \exists CHILD$	$\forall CHILD.Man$	False

4 Conclusions

We believe that this project was a success and our goals were achieved because the code operates correctly, and we were able to use many of the features inherent in the prolog programming language. Also, with the short descriptions of the program that we have included, we think that the code is quite understandable and even moreso, the examples make it clear how it is designed to be used and allow for other users to test the subsumption of their own complex concepts.

References

[1] http://www.inf.unibz.it/~franconi/dl/course/slides/struct-DL/flminus.pdf