

1 Extracting Ontologies from Prolog Source Code

The method used to hypothesise ontological constraints from source code is based on Clark's completion algorithm [1]. This normally is used to strengthen the definition of a predicate given as a set of single-implication Horn clauses into a definition with double implication. Consider, for example, the predicate $member(E, L)$ which is true if E is an element of the list, L :

$$\begin{aligned} member(X, [X|T]) \\ member(X, [H|T]) &\leftarrow member(X, T) \end{aligned}$$

The Clark completion of this predicate is:

$$member(X, L) \leftrightarrow L = [X|T] \vee (L = [H|T] \wedge member(X, T)) \quad (1)$$

Use of this form of predicate completion allows us to hypothesise ontological constraints. For example, if we were to assert that $member(d, [a, b, c])$ is a true statement in some problem description then we can deduce that this is inconsistent with our use of $member$ as constrained by its completion in expression 1 above because the implication below (which is an instance of the double implication in expression 1) is not satisfiable.

$$member(c, [a, b]) \rightarrow [a, b] = [c|T] \vee ([a, b] = [H|T] \wedge member(c, T)) \quad (2)$$

Normally Clark's completion is used for transformation of logic programs where we are concerned to preserve the equivalence between original and transformed code. It therefore is applied only when we are sure that we have a complete definition for a predicate (as we had in the case of $member$). However, we can still apply it in "softer" cases where definitions are incomplete. Consider, for example, the following incomplete definition of the predicate $animal(X)$:

$$\begin{aligned} animal(X) &\leftarrow mammal(X) \\ animal(X) &\leftarrow fish(X) \end{aligned}$$

Via completion as above, we could derive the constraint:

$$animal(X) \rightarrow mammal(X) \vee fish(X)$$

This constraint is over-restrictive since it assumes that animals can only be mammals or fish (and not, for instance, insects). Nevertheless, it is useful for two purposes:

- As a basis for editing a more general constraint on the use of the predicate '*animal*'. We describe the beginnings of an editor for these sorts of constraints in Section 2.
- As a record of the constraints imposed by this *particular* use of the predicate '*animal*'. We describe an automated use of constraints under this assumption in Section 3.

2 A Constraint Extraction Tool

We have produced a basic system for extracting from Prolog source code ontological constraints of the sort described above. Our tool can be applied to any standard Prolog program but only is likely to yield useful constraints for predicates which contain no control-effecting subgoals (although non-control-effecting goals such as *write* statements are accommodated). We demonstrate the current tool by example. Although we have applied the tool to significantly sized logic programs constructed by others, it is easier to explain what the tool does if we show its application to a simpler example.

Figure 1 shows the tool being applied to a simple example of animal classification, following the introduction of the previous section. The Prolog code is:

```

animal(X) :- mammal(X).
animal(X) :- fish(X).
mammal(X) :- vertebrate(X), warm_blooded(X), milk_bearing(X).
fish(X) :- vertebrate(X), cold_blooded(X), aquatic(X), gill_breathing(X).

```

and this corresponds to the Horn Clauses:

$$\begin{aligned}
animal(X) &\leftarrow mammal(X) \\
animal(X) &\leftarrow fish(X) \\
mammal(X) &\leftarrow vertebrate(X) \wedge warm_blooded(X) \wedge milk_bearing(X) \\
fish(X) &\leftarrow vertebrate(X) \wedge cold_blooded(X) \wedge aquatic(X) \wedge gill_breathing(X)
\end{aligned}
\tag{3}$$

The constraints extracted for this program (seen in the lower window of Figure 1) are:

$$\begin{aligned}
animal(X) &\rightarrow mammal(X) \vee fish(X) \\
fish(X) &\rightarrow vertebrate(X) \wedge cold_blooded(X) \wedge aquatic(X) \wedge gill_breathing(X) \\
mammal(X) &\rightarrow vertebrate(X) \wedge warm_blooded(X) \wedge milk_bearing(X)
\end{aligned}
\tag{4}$$

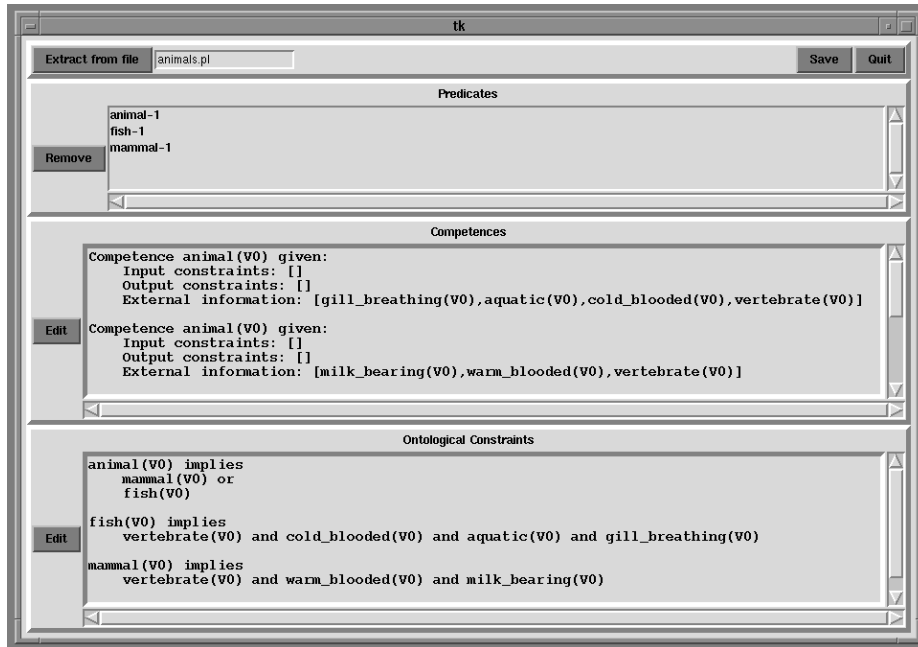


Figure 1: Ontology extraction tool

We show in Section 3 how these constraints, which we extracted completely automatically from the Prolog source code, can be used automatically to check another Prolog program purporting to use the same ontology.

3 Ontological “Safe Envelopes”

The idea of running programs within ontological “safe envelopes” was introduced in [2]. Programs are run according to the normal execution control regime of the language concerned but a record is kept of the cases where the execution uses terminology which does not satisfy a given set of ontological constraints. When this happens we say the execution has strayed outside its safe envelope from an ontological point of view. This sort of checking is not intended to alter the execution of the program in any significant way, only to pass back retrospective information about the use of terminology during an execution. This style of checking can be implemented elegantly for languages such as Prolog which permit meta-interpretation, allowing us to define the control structure for execution explicitly and then augment this

with appropriate envelope checking. The Horn clauses shown in expression 5 provide a basic example (extended versions of this appear in [2]).

$$\begin{aligned}
& \text{solve}(\text{true}, \{\}) \\
& \text{solve}((A \wedge B), E_a \cup E_b) \leftarrow \text{solve}(A, E_a) \wedge \text{solve}(B, E_b) \\
& \text{solve}((A \vee B), E) \leftarrow \text{solve}(A, E) \vee \text{solve}(B, E) \\
& \text{solve}(X, E \cup \{C \mid (X \rightarrow C \wedge \text{not}(C))\}) \leftarrow \text{clause}(X, B) \wedge \text{solve}(B, E)
\end{aligned}
\tag{5}$$

In the expressions above, $\text{clause}(X, B)$ means that there is a clause in the program satisfying goal X contingent on conditions, B (where there are no conditions, B has the value true). The implication $X \rightarrow C$ is an ontological constraint of the sort we are able to derive in the extraction tool of Section 2. The operators \leftarrow , \wedge , \vee , and \cup are the normal logical operators for (left)implication, conjunction, disjunction and union, while $\text{not}(C)$ is the closed-world negation of condition C .

The effect of the meta-interpreter above is to test against the available ontological constraints each successful goal in the proof tree for a query. For example, suppose that we have the following Prolog database of information about animals, a1 and a2 , using the animal ontology of Section 2.

```

animal(a1).
vertebrate(a1).
warm_blooded(a1).
milk_bearing(a1).
animal(a2).
vertebrate(a2).
cold_blooded(a2).
terrestrial(a2).

```

We could query this database in the normal way, for example by giving the goal $\text{animal}(X)$ for which yields solutions with $X = \text{a1}$ and $X = \text{a2}$. If we want to perform the same query while checking for violations of the ontological constraints we extracted in Section 2 then we pose the query via the meta-interpreter we defined above - the appropriate goal being $\text{solve}(\text{animal}(X), C)$. This will yield two solutions, as before, but each one will be accompanied by corresponding ontological constraint violations (as corresponding instances of the variable C). The two solutions are:

$$\begin{array}{ll}
X = \text{a1} & C = \{\} \\
X = \text{a2} & C = \{\text{mammal}(\text{a2}) \vee \text{fish}(\text{a2})\}
\end{array}$$

These solutions tell us that the solution for `animal(a1)` violates no ontological constraint but the solution for `animal(a2)` is obtained despite the fact that (according to the ontological constraints) `a2` is neither a mammal nor a fish.

4 Extracting ontologies from other sorts of KB systems

The majority of knowledge sources are not in Prolog so for our extraction tool to be widely applicable it must be able to deal with other sorts of source code. This would be very hard indeed if it were the case that the ontological constraints we extract have to encompass the entire semantics of the code. Fortunately, we are not in that position because it is sufficient to extract some of the ontological constraints from the source code - enough to give a partial match when brokering or to give a starting point for constraint editing. The issue when moving from a logic-based language, like Prolog, to a more procedural language is how much of the ontological structure we can extract. We discuss this using CLIPS as an example.

Suppose we have the following CLIPS facts and rules:

```
(deftemplate person "the person template"
  (slot name)
  (slot gender (allowed-symbols female male) (default female))
  (slot pet) )

(deftemplate pet "the pet template"
  (slot name)
  (slot likes) )

(deffacts dating-agency-clients
  (person (name Fred) (pet Tiddles) (gender male))
  (person (name Sue) (pet Claud) )
  (person (name Tom) (pet Rover) (gender male))
  (person (name Jane) (pet Squeak) )
  (pet (name Tiddles) (likes Claud) )
  (pet (name Claud) (likes Tiddles) )
  (pet (name Rover) (likes Rover) )
  (pet (name Squeak) (likes Claud) ) )
```

```
(defrule compatible
  (person (name ?person1) (pet ?pet1))
  (person (name ?person2) (pet ?pet2))
  (pet (name ?pet1) (likes ?pet2))
=>
  (assert (compatible ?person1 ?person2)) )
```

To extract ontological constraints from these using the tool as it currently stands we must translate these CLIPS rules into Horn clauses. We can define this sort of transformation for knowledge in the form expressed above by following a simple algorithm, described intuitively below to save space:

- For each CLIPS rule, take the assertion of the rule as the head of the Horn clause and the preconditions as the body of the clause.
- Consider each head, body or CLIPS fact as an object term.
- For each object term, refer to its `deftemplate` definition and translate it into a series of binary relations as follows:
 - Invent an identifier, I , for the instance of the object.
 - The relation $object(T, I)$ gives the type of object, T , referred to by instance I .
 - The relation $A(I, V)$ gives the value, V , for an attribute A of instance I .

Applying this algorithm to our CLIPS example yields the Horn clauses shown below:

$$compatible(Person1, Person2) \leftarrow \begin{aligned} &object(person, O1) \wedge name(O1, Person1) \wedge pet(O1, Pet1) \wedge \\ &object(person, O2) \wedge name(O2, Person2) \wedge pet(O2, Pet2) \wedge \\ &object(pet, O3) \wedge name(O3, Pet1) \wedge likes(O3, Pet2) \end{aligned}$$

```
object(person, p1)  name(p1, fred)  gender(p1, male)   pet(p1, tiddles)
object(person, p2)  name(p2, sue)   gender(p2, female) pet(p2, claud)
object(person, p3)  name(p3, tom)   gender(p3, male)   pet(p3, rover)
object(person, p4)  name(p4, jane)  gender(p4, female) pet(p4, squeak)
```

```
object(pet, x1)  name(x1, tiddles)  likes(x1, claud)
object(pet, x2)  name(x2, claud)   likes(x2, tiddles)
object(pet, x3)  name(x3, rover)    likes(x3, rover)
object(pet, x4)  name(x4, squeak)  likes(x4, claud)
```

This does not capture the semantics of the original CLIPS program, since for example it does not express notions of state necessary to describe the operation of CLIPS working memory. It does, however, chart the main logical dependencies, which is enough for us then to produce ontological constraints directly from our tool. This translation-based approach is the most direct route to constraint extraction using our current tool but we anticipate more sophisticated routes which perhaps do not translate so immediately to Horn clauses.

References

- [1] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978. ISBN 0-306-40060.
- [2] Y. Kalfoglou and D. Robertson. Use of formal ontologies to support error checking in specifications. In *Proceedings of the 11th European Workshop on Knowledge Acquisition, Modelling and Management (EKAW-99), Germany*, pages 207–221. Springer Verlag (Lecture Notes in Computer Science 1621), 1999.