Theory and Application of Ontologies (CS6852)

C ONTOLOGY REPORT

Rahul Kejriwal (CS14B023) Ajmeera Balaji Naik (CS14B034) Bikash Gogoi (CS14B039)

Table of Contents:

Summary

1. Classes:

a. For the purpose of analysis, we split the classes used in the ontology into 2 parts - atomic entities (models atomic language constructs) and complex entities (models non-atomic language constructs).

b. The C ontology has classes that models atomic entities like variables, literals, library, type qualifiers, storage classes.

   i. Variable Class contains all identifiers that are used for storage locations.
   ii. Literal is a class containing any value (integer, string etc.) hard coded in a program.
   iii. Type Qualifiers consists of const and volatile subclass to signify constants and volatile (content can change without code action).
   iv. Storage Class consisting of auto, extern, static and register which are the 4 basic keywords used in C for specifying storage classes(by default - auto).

c. It also models more complex entities like conversion, declaration, expressions, extern definitions, scope, statement, type (includes derived types and hence not atomic), unsorted via appropriate subclass relationships.

   i. Conversion - models the different kinds of data type conversions that are possible.
   ii. Declaration models different kinds of identifier declarations that can take place - class, array, enum, function, pointer, template, variable etc.
   iii. Expression models any string of tokens whose computation results in a resultant/return value.
   iv. External definitions declare identifiers as being defined outside the current program file.
   v. Scope models blocks and functions, basically constructs that introduce their own new scope.
   vi. Statement models individual statements of the program delimited by semi-colons.
   vii. Type models the various different data types possible - both primitive and derived. While primitive types are atomic entities, derived types are complex entities.
   viii. Unsorted models entities belonging to an unordered collection like case statements in a switch block.

d. Any C program is composed of a string of tokens each of which are objects of atomic entities. A simple pass over a C program will yield triples defining the type of each atomic entity.

e. Next, these tokens or atomic entities are strung together to form more complex constructs like loops etc., which are modelled via complex entities. This utilizes context information.

f. The classes convey some information about the token and the ABox thus, becomes an encoding of the program itself (translations can be made such that no information of the C program are lost, specially if the IRIs convey token location information in the program).

2. Object Properties:

- refersTo - pointer refering to function or variable
- referredBy - inverse of refersTo
- hasType - type of variable or return type of function
- hasStorageClass - storage class of variable
- hasSize - size of variable
- hasScope - scope of a variable or function
- hasParameterList - list of formal parameters of a function
- hasParameter - parameters in the parameter list
- hasName - name of a variable
- hasInitialValue - initial value of a variable which is assigned during its declaration
- hasForTest - for condition
- hasForInit - for initialization
- hasForIncr - for increment
- hasElseBody - else body of if-else statement
- hasDefinition - function definition
- hasDeclaration - declaration of a function or variable
- hasCondition - condition of a conditional statement like if, while
- hasLeftOperand - left operand of a binary operator
- hasRightOperand - right operand of a binary operator
- hasOperand - operand of a unary operator
- hasBody - body of a compound statement or iteration statements.
- hasArrayRank - dimension of an array
- hasArgumentList - list of arguments to a function call
- hasArgumentExpression - expression in a argument list
- definedBy - a statement which defines a variable
- declaredBy - a statement which declares a variable
- consistsOf - statements in a compound statement
- hasAlignment - alignment of memory location where the object to be stored

3. Data Type Properties:

a. hasName to provide names for different tokens like variables (identifiers).
b. hasValue to provide current value of the object.

4. Individuals:

a. Individuals for varous data types are made like: float, double, long long int, signed_long etc.
b. Individuals for storage class keywords: auto, extern, register and static.
c. Individuals for derived Types like function_type, array_type, enum_type etc.

5. Things we didn’t understand:

a. ‘UnknownClass’ Class
b. ‘Library‘ Class
c. ‘hasName’ & ‘hasValue’ appear as both Data Property & Object Property.

Critique

1. Limitations

a. The C code is assumed to be preprocessed. No classes for preprocessor directives exist in their ontology.

b. Literals are considered as sibling class of Expressions. But in languages like C, literals actually model a value that can be assigned, used in computations etc. In that regard, it might be useful to actually make literals, a subclass of Expression.

c. Similarly, variables itself can also be an expression. It may also be placed under Expressions class rather than as a sibling class. This would be more inline with C grammar.

d. Conversion class is a child class of owl:Thing whereas CastOp is child class of Expression. We should model Conversion class as subclass of CastOp to model casting between different types.

e. The ontology does not support inline functions.

f. Due to the assumption that the code is preprocessed, the ABox must already have assertions for library functions code. In our sample program, we did not consider the initial code for library functions that would have been added by the preprocessor. This would modify the line numbers.

g. Addition of a single line modifies the line numbers for many individuals and their IRIs need to be updated.

h. Object properties have not defined range and domains. We could add this to do some form of type checking.

i. There is a class for IOStatement appearing as a subclass of unsorted. Perhaps it is better modelled as a child class of FunctionCall since all IO in C occurs via functions.

2. Extensions

a. Another extension would be to encode preprocessor directive information into classes.

b. Could possibly bring Literals and Variables under Expressions.

c. Could possibly bring Conversion class as child class of CastOp.

d. Could add class for inline and add support for it.

e. Could add domain and range for the object properties to do type checking.

f. Could transfer IOStatement as a child class of FunctionCall.

g. An alternate naming scheme for individuals could be used that is more meaningful to a human reader. For example, if statements could be modelled as if1 for the first if statement appearing in the program and so on. This delivers more meaning than simple w:x,y:z where w and y are line numbers for start and end of entity and x and z are column numbers for start and end of entity.

   It also has the added advantage that changes in the code, like insertion of a printf statement would not affect the naming of the if statement which would in the current scheme mess up all the line numbers for entities that appear after the new printf statement.

Sample C Program

```c
#include <stdio.h>

int pow(int a, int n){
	if(n == 0){
		return 1;
	}
	else{
		int temp_res = pow(a, n/2);
		if(n%2 == 0)
			return temp_res * temp_res;
		else
			return temp_res * temp_res * a;
	}
}

int main(){
	int a = 5;

	printf("%d\n", pow(7,3));

	return 0;
}
```

1. We translated the above program into ABox assertions and added them to the ontology in Protege.
2. On running the reasoner, we got a consistent ontology.
3. PFA the ABox assertions in the ‘c.owl’ file. It can be viewed by loading it up in Protege.