TRABAJO PRÁCTICO 3
INFORME ONTOLOGÍAS

INTRODUCCIÓN A LA INTELIGENCIA ARTIFICIAL

REALIZADO POR

Bini Valentina
Zimmermann Sebastián

Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniería y Agrimensura

21/05/2020

1. Introducción

Existen numerosos lenguajes de programación, cada uno tiene propiedades y utilidades diferentes. Según qué tipo de programa uno desee realizar, será más o menos conveniente usar cierto lenguaje.

Por un lado, cada lenguaje tiene un sistema de tipado, un nivel de abstracción, un conjunto de paradigmas.

Por el otro lado, cada programa tiene objetivos y utilidades distintos. Existen sistemas operativos, programas de aplicación, etc. Cada uno de estos posee ciertas características, como la posibilidad de correr en paralelo o cierto tamaño límite.

Es por esto que consideramos que hacer una ontología de lenguajes de programación puede ser muy útil, y puede ayudar al programador en la tarea de elegir el lenguaje más apropiado para el proyecto que tenga que llevar a cabo.

2. Armado del Modelo Ontológico

Inicialmente, volcamos el modelo de nuestro trabajo de Prolog en un esquema ontológico. Para esto, fue necesario tomar ciertas decisiones en cuanto a qué elementos tenía sentido que fueran clases independientes y cuáles solo serían propiedades de dichas clases.

Tomamos la decisión de dividir los lenguajes de programación en lenguajes tipados y no tipados, y de esta manera establecer una relación entre los lenguajes tipados y los sistemas de tipos.

Luego de obtener el primer bosquejo, resultó evidente que el modelo obtenido era muy simple. Esto llevó a ampliar el dominio agregando una nueva clase que representa a los programas.

Tomando ese bosquejo, comenzamos a modelar las restricciones. Durante esta tarea tuvimos que replantearnos algunos aspectos:

- Dada la particularidad de los sistemas embebidos y lo complejo que resulta caracterizarlos, decidimos restringir nuestra ontología a sistemas no embebidos.
- Pese a que no sabíamos si era posible representarlo en Protègè, decidimos emplear un tipo “choice” para representar algunas de las restricciones de los lenguajes, por ejemplo, su nivel (alto | medio | bajo).
- Para representar qué tan libre es un software, decidimos emplear un modelo usando enteros, ya que cuando hablamos de software libre hablamos de 4 libertades, cada una de las cuales incluye a las anteriores.

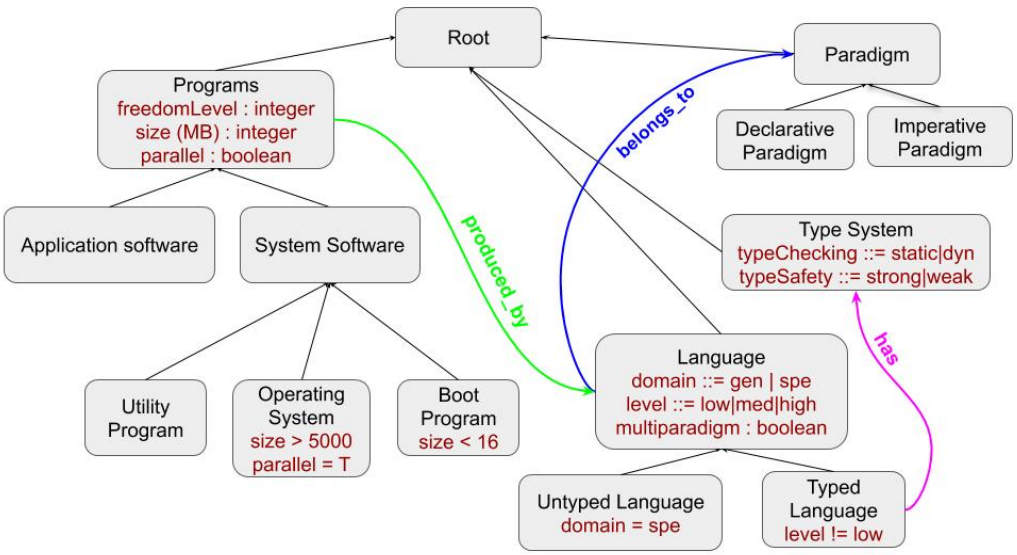


Figura 1: Modelo ontológico obtenido

3. Pasaje del esquema a Protègè

3.1. Tipos de variables

En el pasaje del esquema, nos encontramos con algunas dificultades que llevaron a realizar cambios en los tipos de las variables.

Un primer cambio inicial fue modificar Size a **tipo flotante**, esto nos permitía no aproximar a enteros, sobre todo en los tamaños menores a un MB.

Tuvimos un debate sobre cómo definir la libertad del software. En el modelo adoptado inicialmente notamos dos problemas.

- Ese modelo no nos permitía representar el software no libre.
- El software opensource tampoco tenía representación, pues la segunda libertad de software libre es más potente que simplemente código abierto y la primera es más débil.

Dados esos problemas, terminamos con una nueva representación que agrega la opción "software privativo" y parte la segunda libertad en dos sub-libertades:

- 0: Software privativo (ninguna libertad).
- 1: La libertad de ejecutar un programa de cualquier forma.
- 2: La libertad de estudiar cómo funciona el programa.
- 3: La libertad de cambiar el funcionamiento del programa como uno desee.
- 4: La libertad de redistribuir y hacer copias de ese programa.
- 5: La libertad de modificar el programa y mejorarlo, y además distribuir las copias modificadas del programa.

3.1.1. Variables tipo choice

Las variables del tipo choice nos representaron uno de los mayores conflictos del TP. Nuestra primer aproximación fue utilizar enteros para representar niveles y booleanos para representar las otras opciones, dado que resultaría sencillo.

Sin embargo, encontramos una forma de realizar un **tipo enumerado** que consistía en crear un nuevo tipo de datos de Protègè. Elegimos esta opción, sabiendo que era potencialmente más compleja, porque nos parecía correcto emplear un tipo de datos que representara realmente las opciones y no fuera una "dicotomía true/false".

Finalmente, luego de semanas de intentos y frustraciones, en determinado momento teníamos todo resuelto excepto que el razonador de Protègè nos marcaba un error con las restricciones. No encontramos ninguna solución a esto en ningún tutorial ni consultas en internet, lo cual nos frustró bastante y decidimos volver a la idea inicial de enteros y booleanos.

De esta forma, si tomamos los tipos iniciales que utilizamos en el diseño de la ontología, estas fueron las representaciones finales:

- `TypeSafety` → `strongTypeSafety`, que es un booleano que representa si el tipado es fuerte (`true`), o débil (`false`).
- `TypeChecking` → `staticTypeChecking`, que es un booleano que representa si el tipado es estático (`true`), o dinámico (`false`).
- `Domain` → `specificDomain`, que es un booleano que representa si el dominio es específico (`true`), o general (`false`).
- `Level` → `Level`(entero), donde 0 representa nivel bajo, 1 representa medio y 2 representa alto.

3.2. Restricciones

Finalizando el trabajo, nos enfrentamos al problema de que el modo de marcar las restricciones matenía al razonador prendido una gran cantidad de tiempo, llegando a tildar el programa, teniendo que forzar su terminación. Para resolver esto, tuvimos que cambiar la forma de diseñar las restricciones a modo de subclases.

Cuando resolvimos esto nos dimos cuenta de que, si bien el razonador corría sin problemas, no marcaba errores cuando un individuo no cumplía con las restricciones de la clase a la que pertenecía. Luego de probar diferentes alternativas, nos dimos cuenta de que la razón era que las restricciones estaban definidas con `some` y esto hacía que no "obligue" a los individuos a cumplirlas. Cambiamos las restricciones poniendo la cardinalidad en `exactly 1` y comenzó a funcionar correctamente.

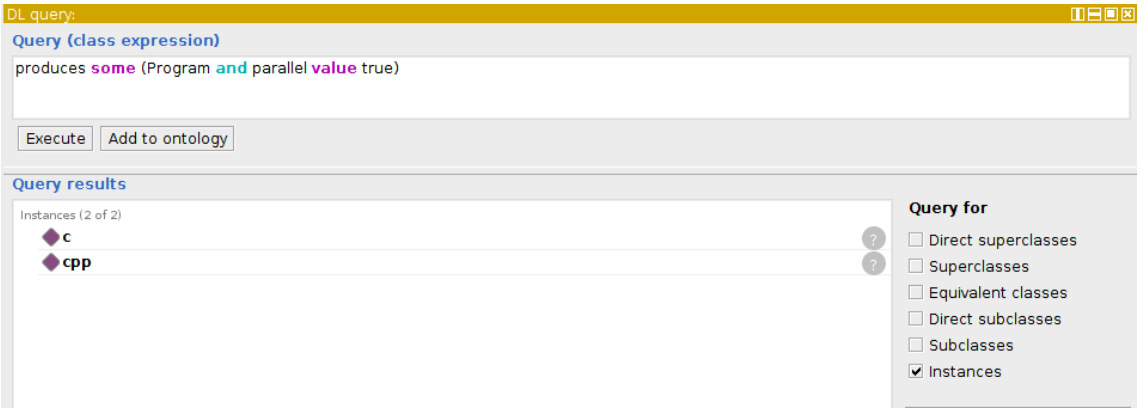
4. Querys

4.1. Lenguajes que generen programas paralelizables

Decidimos hacer esta consulta porque nos parecía interesante desde el punto de vista del programador saber si un lenguaje es útil para hacer programas paralelizables. Esta característica no estaba en nuestra clasificación original y surgió de la nueva clase `Program` y su relación con los lenguajes.

Para realizar esta consulta, necesitábamos la propiedad inversa a `produced_by` que, dado un lenguaje, nos diga qué programas están hechos usándolo. Por eso, agregamos la propiedad `produces`, que va de la clase `Language` a la clase `Program`.

```
produces some (Program and parallel value true)
```

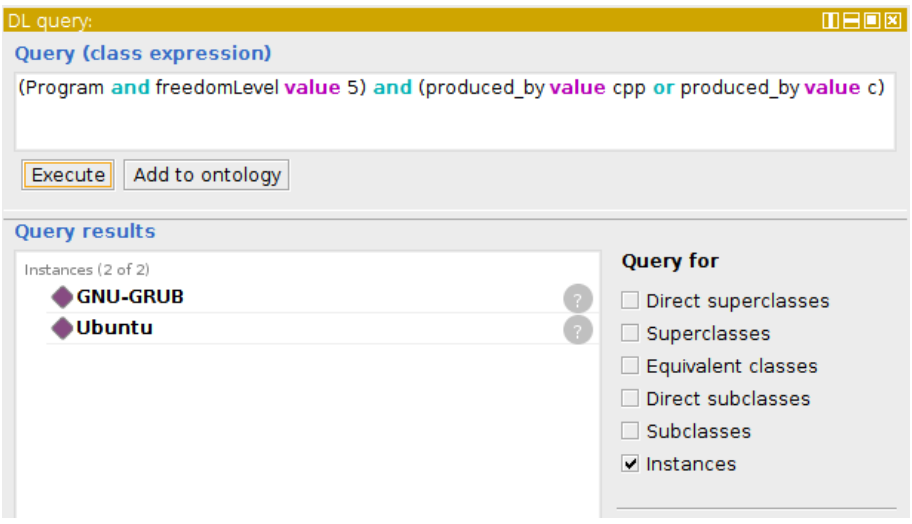


4.2. Programas libres producidos por C o C++

Realizamos esta consulta para dar otro punto de vista al trabajo, ya que nos enfocamos mucho en los lenguajes y creemos que también puede ser muy útil analizar ciertas características de los programas.

Si uno busca programas con ciertas libertades para hacerle modificaciones, puede querer que estén realizados en ciertos lenguajes que le sean de preferencia.

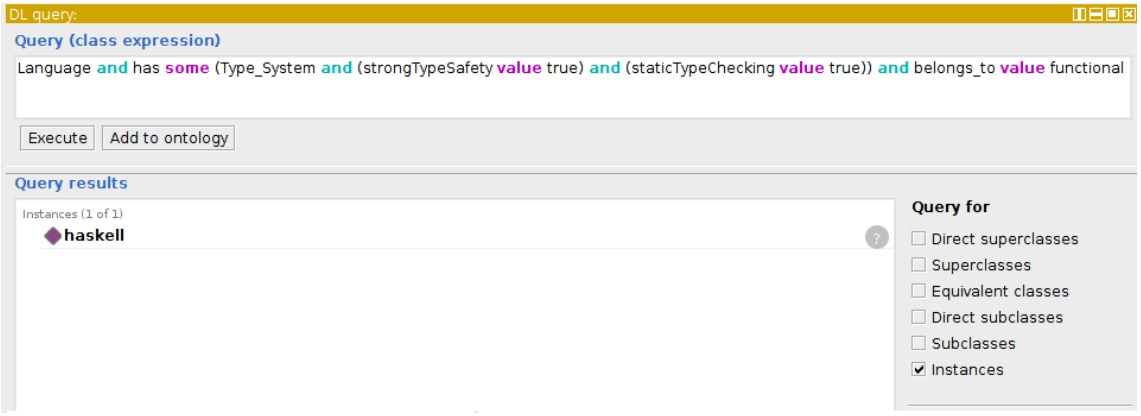
```
(Program and freedomLevel value 5) and
(produced_by value cpp or produced_by value c)
```



4.3. Lenguajes funcionales, que tengan tipado estático y fuerte

Con el objetivo de comparar el comportamiento con el TP de prolog, hicimos esta consulta que pregunta por un lenguaje con ciertas características, desde un lugar similar a lo que se hacía en el trabajo anterior.

```
Language and has some (Type_System and (strongTypeSafety value true)
and (staticTypeChecking value true)) and belongs_to value functional
```



5. Conclusiones

En este trabajo encontramos la mayor dificultad en el traspaso del modelo inicial a la Ontología definida en Protègè. Dichos conflictos, mencionados en el capítulo 3 de este trabajo, requirieron realizar muchas modificaciones que terminaron cambiando la idea inicial del proyecto que teníamos a otro tipo de representación de los datos.

Afortunadamente estos cambios de representación no quitaron poder de expresión a la ontología realizada, con lo cual una vez realizados los cambios nos permitió realizar las consultas que esperábamos sin mayores problemas.

Esta ontología nos permite analizar relaciones existentes entre programas y lenguajes de programación. Las primeras dos consultas que realizamos son ejemplos que relacionan una propiedad de los programas (como funcionar en paralelo o ser software libre) con el lenguaje utilizado para diseñarlos. Esto nos permite sabe qué lenguaje se utiliza para determinada propiedad del programa.

Por otra parte, desde el trabajo anterior teníamos la idea de poder realizar un sistema capaz de recomendar o sugerir qué lenguajes deberíamos utilizar para enfrentar distintos problemas.

En ese contexto, con la tercer consulta observamos que el resultado es similar, por lo que creemos que esta representación abarca a la obtenida en Prolog. Además nos da la posibilidad de hacer preguntas más interesantes y con mayor poder de expresión.

En definitiva, el resultado que obtuvimos no es algo muy potente. Esto se debe fundamentalmente a la poca cantidad de individuos y características que poseen cada una de las clases. Si se incrementaran la cantidad de objetos de la ontología, la capacidad de expresión y de resolución de problemas de esta sería cada vez mayor, lo cual nos permitiría acercarnos a un sistema que permita recomendar lenguajes con mejor precisión y conocimiento.