Extending PROVA for Ontologies Definition and Exchange

Yoan David Ribeiro[1] and Damien da Silva Vaz[1]
(pg25340,pg25300)@alunos.uminho.pt

[1] University of Minho
[2] Educed

Abstract. On the modeling world, the Ontology notion can be very useful, since, we have a humongous number of definitions and relations at our fingertips.So the main goal of this project is to integrate the Ontology notion in PROVA, to have all those definitions and relations available, and thus simplify, in some way, the modeling process.

1 Introduction

When we think of *model*, what's the first thing that pops in our minds? The word *abstraction* is one of the first words that we can remember when we talk about model, because a model is an abstraction of the Real World which ultimately leads, when well executed, to a better understanding of reality. In computer science, a model is exactly an application of the definition. Thus, when we have a problem/situation that we want to model, it is normally a mess, so the first step is to extract the *requirements* of the problem and after those requirement going through a modeling process we'll have our model, which, hopefully, will be simpler than the problem that we had. *PROVA* is an emerging modeling platform which has, as main objective, to model high assurance systems. On that note, PROVA wants to integrate the concept of *Ontology* in its system. And why you may ask? Let's put it this way, imagine that we could have a *bag of terms and relation* about an *area of knowledge* that could be used to model complex systems. It would be really helpful for modeling purposes. So our goal is to make an useful translation between an *Ontology language and PROVA's Boilerplates*. To accomplish the main goal of the project, we explored various points, them being:

- The study of PROVA application and Boilerplates in section 2.
- The investigation of the concept of *Ontology* and the Web Ontology Language in section 3.
- We built an Haskell Data Type for the OWL Language which is described in section 4.
- With the help of the HXT libary, we've built a parser for a subset of OWL and also made a useful translation of our OWL Haskell Data Type to PROVA's Boilerplates, more in depth in section 5.

There are still a lot of objectives to accomplish in this project, so in section 6 we wrap this up with what are our thoughts about the overall project and what's left to be done.
This report results due to a cohesive project of the Master's degree in Computer Engineering's major of Formal Methods in Software Engineering, which theme was proposed by Educed under the supervision of Engineer José Faria[2] and the local tutelage of the professor Manuel Alcino Cunha[1].

2 PROVA in a nutshell

PROVA is an emerging modeling platform for development of *high assurance systems*. And you may say "Yet another modeling platform? What does it do that the others don't?", well PROVA is capable of analysing textual requirements, with a well defined syntax. As any modeling tool, PROVA identifies possible conflicts between requirements, generates simulations of the systems behaviour, proves that a given property is respected by our model, if not, it will give us a counterexample. So PROVA's purpose is to make the modeling process visually less mathematical, and construct a model which describes the system that we are modeling, without any redundant requirement, and, basically, help to make a high assurance systems correct inside.

2.1 Structure of PROVA

PROVA is a webservice, which means that, the platform is divided in *three* parts:

FronEnd What is seen by the user. Provides a friendly way to model a system through the use of boilerplates, as seen in figure 1, which are sentences with a specific syntax.

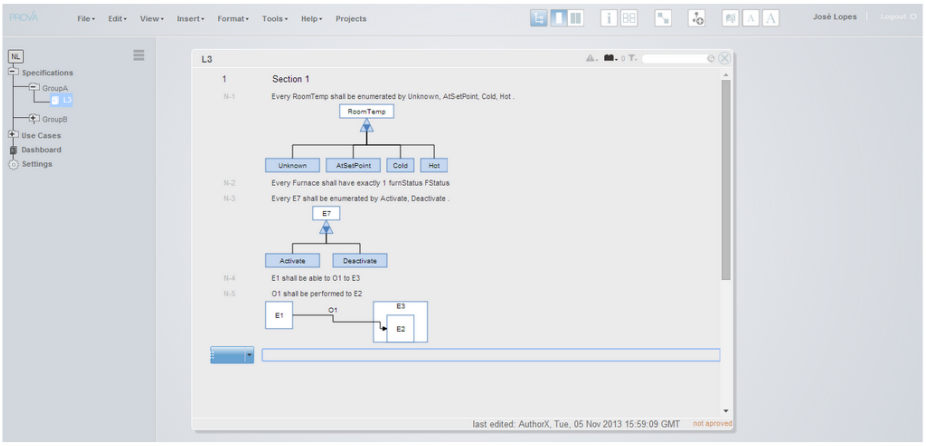

Fig. 1. FrontEnd as it is now.

MiddleEnd This part will receive the boilerplates and translate them to be sent to the BackEnd, this is where we focused our attention to do the translation between OWL and PROVA's boilerplates (section 5) and it is written in *Haskell*.

BackEnd Where is located the SMT Solver which will provide us the models based on the requirement received from the MiddleEnd. (Written in Haskell)

2.2 Boilerplates

A *boilerplate* is a well known *pattern* used to describe, represent something. On the computer science world, a *boilerplate* is an *abstraction* to simplify the way we program/model. In PROVA's particular case, boilerplates are well defined sentences used to define a requirement in the analyses of Software Systems, and there are three distinct types of boilerplates.

Boilerplates of the Entity Model These boilerplates are also known as the structural boilerplates, and they are used to describe the *static part* of a system, for instance, the entities of system, the relations between those entities and the attributes/properties of the entities. These are the boilerplates that are needed to do the translation.

Definition 1 (Types of Boilerplates).

- *MULT: Cardinality of an entity/relation, example: "There are* m *A".*
- *ASSOC: Relation between entities, example: "Every* A *shall have* m fixed? r B*", where* A *and* B *are entities,* r *the name of the relation, and* m *is the multiplicity of the relation.*
- *ATTR: Declaration of an attribute or scalar for an entity, example: "Every* A *shall have a(n)* fixed? type?*".*

- *GEN: The In (contains) relation between two relations/entities, example: "Every* A *is a* B*".*
- *DISJOINT: Declaration that two entities are disjoint, example: "*A, B*, . . . , and* Z *are [pairwise] disjoint".*
- *ABSTRACT: Declaration that an entity is abstract (has to be extended), example: "*A *is abstract".*
- *EXTENDS: Partial specialization of a relation/entity, in other words, a simple inheritance, example: "*A *extends* B*".*
- *EXTENDSABS: Total Specialization, example: Every* A *is either a* B *or* C*.*
- *ENUM: Creation of singletons from a class, example: Every* A *is enumerated by* B*,* C*.*
- *INV: Declaration of an invariant about an entity, example: "every* x *in* A *shall satisfy* F*", F being a formula.*
- *INIT: Specify the initial conditions about a set of entities, example: "every* x *in* A *is initialized such that* F*".*
- *ACYCLIC: Relation cannot contain cycles, example: "r is acyclic".*

Boilerplates of the Operational Model On this case, these boilerplates describe the *operations* that the different agents can do in the systems and the effects which those operations can cause in the universe. They, also, describe the condition that will allow an operation to be executed; If it's a necessary conditions then it's a precondition, if it's a sufficient condition then it's a trigger. And they describe the dependencies between operations too.

Boilerplates of the Behavioral Model These are still under development, but they will describe the *dynamics* of the system like a state machine. The operations are seen as messages between objects or internal events, and those operations can trigger or not an evolution of the system.

3 What is an Ontology?

Definition 2 (Ontology - Etymologicaly).
"Onto" (Being) + "logia" (Study) = Study of Being

Formally, an Ontology is a representation of knowledge, which is represented as a set of concepts of a specific domain, with a specific vocabulary to denote relationships between the concepts, properties and types.
We can also see an Ontology as a way to model the data's and the relations between them, giving a semantics to the concepts. To be clear, an Ontology is not a database schema, in anyway[1].
Thus, we can extract, from the definition, the essential characteristics of an Ontology.

Definition 3 (Characteristics of an Ontology).

- *Interrelation: The relations between the concepts.*

- *Instantiation: As the name says, the power of making an instance of a general concept (individual).*
- *Subsumption: The capacity of defining a concept is more specific than the other (subclasses of).*
- *Exclusion: The capacity of defining that a concept is not ,in any way, defined by an other (disjunction of concepts).*
- *Axiomatisation: The capacity of expressing a more complex statement about a concept.*
- *Attribution: The relation between a concept/instance and a value.*

And from the characteristics we can derive the attributes of an ontology:

- Individuals
- Classes
- Attributes
- Relations
- Function terms
- Restrictions
- Rules
- Axioms
- Events

3.1 Web Ontology Language (OWL)

The Web Ontology Language, OWL for shorts, was created by Mike Dean and Guus Schreiberr, and it was designed to represent rich and complex knowledge between things, a collection of things and relations between things, basically an Ontology.
OWL is, also, a very important part of the W3C Semantic Web Stack, as W3C says: "The Semantic Web is about two things. It is about common formats for integration and combination of data drawn from diverse sources, where on the original Web mainly concentrated on the interchange of documents. It is also about language for recording how the data relates to real world objects. That allows a person, or a machine, to start off in one database, and then move through an unending set of databases which are connected not by wires but by being about the same thing."[2].
So the use of an Ontology is extremely important to give the different concepts, a well defined semantics on the Web. OWL, being a standard, has to be the most used language to define an ontology, nowadays.
Is also worth mentioning that the Web Ontology Language is a instance of the Set Theory. This means that the entities, classes/subclasses are Sets. The universe of those sets is the super-set Thing, the properties are relations between sets or between a set and a primitive type, and the individuals are a specific habitant of a set.

OWL Limitations In OWL, there is no way to represent n-ary relation, however it is well know that any n-ary relation can be reduced to a binary relation, that being said, this limitation is not a problem.

OWL Syntax The Web Ontology Language is based on *Resouce Description Framework* (RDF) and RDFS, and those two are based in XML. Thus a OWL ontology can be represented in many way, too many ways to be exact, which is a problem if we want to make a good parser and translate the OWL information. The syntax of OWL is referenced in [3] So the process focused on a subset of the OWL language, which simplifies the parsing process.

Class and SubClass Declaration This is how we declare two entities in the ontology:

```
<owl:Class rdf:ID="A">
    <rdfs:subClassOf rdf:resource="#Thing" />
</owl:Class>

<owl:Class rdf:ID="B">
    <rdfs:subClassOf rdf:resource="#Thing" />
</owl:Class>
```

In this particular case, we are saying that A and B are a subset of the set Thing, which could be omitted since every set in OWL is a subset of Thing.

Object Property Declaration To define an ObjectProperty which is a property that relates a set to another.

```
<owl:ObjectProperty rdf:about="#R">
    <rdfs:domain rdf:resource="#A" />
    <rdfs:range rdf:resource="#B" />
</owl:ObjectProperty>
```

Data Type Property Declaration We can also define an DataTypeProperty which is a property that goes from a set to a primitive type (String, Float, Integer, etc. which will be represented as c).

```
<owl:DataTypeProperty rdf:about="#Q">
    <rdfs:domain rdf:resource="#A" />
    <rdfs:range rdf:resource="#c" />
</owl:DataTypeProperty>
```

Individual declaration And finally we can say that there is an habitant of a specific class:

```
<owl:NamedIndividual rdf:about="#i">
    <rdf:type rdf:resource="#A" />
</owl:NamedIndividual>
```

More Advanced Syntax The OWL language gives use the freedom to declare more complex thing, for instance that an entity is equivalent to an intersection of other entities. As we already said, all entities are Sets it is perfectly normal to declare something around those lines. The Classes as we know can have 3 types of Axioms:

- SubClassOf - Declaring that a set is subset of another, already seen on the example of the declaration of a class.
- DisjointWith - Declaring that a set is disjoint from an other.

```
<owl:Class rdf:ID="#A">
    <owl:disjointWith rdf:Description="#B">
</owl:Class>
```

In this declaration we are saying that A is disjoint of B.

– EquivalentClass - Declaring that a set is equivalent to another

```
<owl:Class rdf:about="#A">
<owl:equivalentClass>
    <owl:Class>
        <owl:intersectionOf rdf:parseType="Collection">
            <rdf:Description rdf:about="B"/>
            <owl:Restriction>
                <owl:onProperty rdf:resource="#R"/>
                <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
                    0
                </owl:cardinality>
            </owl:Restriction>
        </owl:intersectionOf>
    </owl:Class>
</owl:equivalentClass>
</owl:Class>
```

It can be seen that, the code above is more complex, and it says that the entity A is equivalent to the intersection between B and the elements of A which applying the property R gives a empty set.
This example, shows the case of an intersection, but we could have other set operation such as, union or complement.

4 OWL - Haskell Representation

The approach to represent the Web Ontology Language is based in the idea of doing a homogeneous representation, in Haskell, of the OWL Ontology structure, knowing that it's possible to represent the same information with different syntax in the OWL language. So it's crucial to find a representation which is generic enough to support all the information that is extracted from the OWL Ontology. This information is divided in:

4.1 Ontology

Ontology The Ontology data type represents the structure of the OWL Ontology in Haskell and it's the main data type. Each field of the Ontology data type represents the main components of an OWL Ontology which are a list of entities, object properties, data properties and individuals as defined below.

```
data Ontology = Ontology {
                            entities :: [Entity],
                            oProperties :: [Property],
                            dProperties :: [Property],
                            individuals :: [Individual]
                } deriving (Show, Eq)
```

4.2 Entities

Axioms As mentioned in Section 3 the classes are composed by axioms and it's necessary to add an `ErrorAxiom` to cover the others axioms which are not valid in the parser. So the type axioms is defined as:

```
data Axiom = SubClass {
                 super :: Entity
             }
             | EquivalentTo {
                 classEqui :: Entity
             }
             | DisjointWith {
                 classes :: Entity
             }
                       | ErrorAxiom
                       deriving (Show, Eq)
```

Entity The Entity shows one of the main components of the OWL Ontology. Them being:

- Class is simply a name and collection of axioms that describe a set of individuals.
- Anonymous class defines an unnamed class which has basically an operation and they can be:
  - Intersection.
  - Union.
  - Complement.
  - OneOf (which is not being considered)
- A more complex set can be represented as a Restriction.
- *Thing* is the super-set which as all the entities of the ontology
- *OWLNothing* represents the empty set.

```
data Entity = Class {
                name :: String,
                axioms :: [Axiom]
              }
              | AnonClass {
                operation :: Entity
              }
              | IntersectionOf {
                setI1 :: Entity,
                setI2 :: Entity
              }
              | UnionOf {
                setU1 :: Entity,
```

```
    setU2 :: Entity
}
| ComplementOf { setC :: Entity }
| Restriction {
    onProperty :: String,
    valuesFrom :: ValuesFrom,
    value :: String
}
| Thing
| OWLNothing
deriving (Show, Eq)
```

ValuesFrom The ValuesFrom is an extra data type which is defined as multiples choices and correspond to the values where the restriction will be valid. The NIL option is evaluated as a NULL option which is an empty value. And the other options defines as it is described by their name. These values were chosen due to the OWL Ontology format for values options.

```
data ValuesFrom = AllValuesFrom
                | SomeValuesFrom
                | HasValue
                | Cardinality
                | NIL
                deriving (Show,Eq)
```

4.3 Properties

Properties Properties define an object or a data type properties of an OWL Ontology.

```
data Property = ObjectProperty {
                        oPName :: String,
                        oPdomain :: String,
                        oPrange :: String,
                        characteristic :: [String]
              }
              | DataTypeProperty {
                        dTName :: String,
                        dTdomain :: String,
                        dTrange :: String
              } deriving (Show, Eq)
```

ObjectProperties

ObjectProperties Object properties links a set to a set with the respective domain and range. The name describes the name of the object properties. And the characteristics is basically a list of extra information about the relation, if it is a functional property, etc, which will not be parsed, for now.

DataTypeProperties

DataTypeProperties . Datatype properties links a set to data values. And also the name describes the name of the data type properties.

4.4 Individuals

Individuals In addition to classes, we want to be able to describe their members. We normally think of these as individuals in our universe of things. An individual is minimally introduced by declaring it to be a member of a class.

```
data Individual = Individual {
                      nameI :: String,
                      parentClass :: Entity
                  }deriving (Show,Eq)
```

5 OWL parser

To build a parser for the Web Ontology language, which is basically *XML*, and knowing that the PROVA's middle-End, which is the part of PROVA we are interested in, is Haskell, the approach is fairly simple. To find a complete and simple library to help the parsing process of *XML*, and this library is HTX.

5.1 Haskell Toolkit XML (HTX)

As already seen, the Web Ontology Language syntax is purely XML, we've searched for a good Library that could provide an easy way to translate the OWL to our OWL representation in Haskell. So we used the Haskell Toolkit XML, also known as, HTX. HTX provides us a collection of tools for processing XML, and those tools are based on the concept of Arrows.

5.2 Arrows

An Arrow is an abstraction of computation introduced by John Hughes[4]. With arrows we can describe a computation in a pure and declarative way.

Definition 4 (Arrow a => a b c). *This means that there is an arrow a from b to c*

The basic Arrow data type is defined as follow:

```
class Arrow a where
    arr:: (b -> c) -> a b c
    (>>>):: a b c -> a c d -> a b d
```

With these definitions we can construct arrows from functions and we can compose arrows giving us a new arrow.

5.3 Parsing of OWL

Approach The main idea for our parser was to create the same effect of a tree structure (level by level) considering our data structure from Haskell (which is designed, by the way, with the idea of the OWL Ontology structure).

5.4 Main function - parser

The `main` function is the heart of the program. Basically, it is responsible of the parsing process, the scanning of the error in the OWL file, the creation of boilerplates for PROVA's System from the parsed OWL, and the specification of the last to be sent to the solver and valided them. Also the `main` function output the error list for a given OWL Ontology file and the boilerplates generated. Finally, it checkup the extension of the OWL Ontology.

5.5 An example of a function of the parser

```
getEntities :: ArrowXml cat => cat (XmlTree) Entity
getEntities = atTag "owl:Class" >>>
        proc l -> do
        className <- parseClass  -< l
        lAxioms <- procAxioms -< l
        returnA -< Class { name = (split className),
                           axioms = lAxioms }
```

This is an example of the HXT library used on a function named `getEntities`. The `atTag` function do a search on the OWL Ontology for a tag named <owl:Class> and will be the starting point for the parsing process of the class's axioms, returning the XmlTree (representing the Xml Tree of that class) which will be used by the proc instruction. The proc (λ-function for Arrows) instruction have the following syntax:

- `l` will have the initial XmlTree
- Each line below the proc instruction receives an Arrow to be executed by the `parse` function or the `proc` function and returns a type.
- The last line of proc is always the `returnA` which returns all the information that was parsed.

In this case, we can understand that the function `getEntities` process the Classes by getting the name of the class and the respective list of axioms. The others functions of the parser has the same strategy.

5.6 Parsing and Error Handling

The Error system analyses syntactically every tag as the parser does. The flow of the error system is to check every tag and every attribute associated with the tag (if available). If the attribute of a tag isn't as it was supposed to be, it will output in an error file and will explain which attribute isn't valid and which line of the ontology (not the line number but the line with the attribute associated).

```
data Error = Error {
                type_ :: String
           } deriving (Show,Eq)
```

The Error data type as mentioned above, has a type property and collects the error as a string.

```
getOwlClass :: ArrowXml cat => cat XmlTree Error
getOwlClass =  atTagE "owl:Class"
               >>>
               proc cl ->
               do
                    attribute <- (getAttrl>>>getAttrName) -< cl
                    line <- xshow (this >>>
                            changeChildren (take 1)) -< cl
                    returnA -< Error { type_ = errorOwlClass
                            (qualifiedName attribute) line }

errorOwlClass :: String -> String -> String
errorOwlClass "rdf:about" _ = []
errorOwlClass "rdf:ID" _ = []
errorOwlClass [] _ = []
errorOwlClass attribute line = "Invalid attribute type : "
                               ++ attribute
                               ++ ", line : "
                               ++ (rmvExtraTag line)
```

The Error system use a basic strategy which is :

- Find the tag which we want to check the errors, in this example it's <owl:Class>
- Get the attributes of that tag and check if the tags are correctly mentioned on `errorOwlClass`.
- If the tag isn't there, it means that the tag doesn't exist and generate a string as mentioned on the last option of the function `errorOwlClass` with the atribute on the node and the line where he find.

And basically, every functions available for the error system uses the same strategy as the one mentioned above which is: find the tag, get the attribute of that

tag and match it with the error function available to the function. If there is no match, it will generate an error string.

5.7 Translation to PROVA's Boilerplates

The translation of OWL Haskell for PROVA's Boilerplates begins with this function:

```
mainBPlateGen :: [Ontology] -> [Boilerplate Name]
mainBPlateGen [(Ontology {
                    entities = e,
                    oProperties = op,
                    dProperties = dp,
                    individuals = ind})] =
                    [Mult mSome (set (Boilerplates.Id.mkName "Thing") [])]
                    ++ (bplateGenEntities e)
                    ++ (bplateGenObjectProperties op)
                    ++ (bplateGenDataTypeProperties dp)
                    ++ (bplateGenIndividuals ind)
```

To sum it up, the function will transform each part of the ontology, individually and will return a list of Boilerplates. It's important to notice that the first boilerplate which is introduced in the list is the *Mult mSome (set (Boilerplates.Id.mkName "Thing") [])* The mind set for each of this function responsable for the translation is to individualize the OWL Statement, to simplify the translation process. To be explicit here is a table which shows the relation between the OWL Haskell and the PROVA's Boilerplate:

Entities Table 1 presents examples of the translation based in the semantic of OWL [5] [6]. Our translation only considers, in the EquivalentTo axiom, the operation IntersectionOf, but it can be switched to UnionOf, and the ComplementOf, with all the modifications necessary. However for this two operations, the translation, in the case of one of the set being a Restriction, the Cardinality of a property is not defined.

OWL Haskell	**PROVA's Boilerplates**
`Class { name = "A", axioms = [] }`	Gen "A" idenP "Thing" idenP
`Class { name = "A", axioms = [SubClass{ super = Class { name = "B", axioms = []}}] }`	Gen "A" idenP "B" idenP
`Class { name = "A", axioms = [SubClass{ super = Restriction { onProperty = "R", valuesFrom =AllValuesFrom, value = B }] }`	Gen name idenP B ["_R"]
`Class { name = "A", axioms = [DisjointWith{ classes = Class { name = "B", axioms = [] }] }`	Disj ["A","B"]
`Class {name = "A", axioms=[EquivalentTo{ classEqui = Class { name = "B", axioms = []}}] }`	Gen "B" idenP "A" idenP, Gen "A" idenP "B" idenP
`Class {name = "A", axioms=[EquivalentTo{ classEqui = AnonClass { operation = IntersectionOf { setI1 = Class {name= "B", axioms = [] }, setI2 = Class {name = "C", axioms = []} }}] }`	Inv "A" (SCmpF Is (Intersect (set "B" []) (set "C" [])) (set "A" []))
`Class {name = "A", axioms=[EquivalentTo{ classEqui = AnonClass { operation = IntersectionOf { setI1 = Class {name= "B", axioms = [] }, setI2 = Restriction {onProperty = "R", valuesFrom = AllValuesFrom, value = "C"} }}] }`	Inv "A" (SCmpF Is (Intersect (set "R" (["C"])) (set "B" [])) (set "A" []))
`Class {name = "A", axioms=[EquivalentTo{ classEqui = AnonClass { operation = IntersectionOf { setI1 = Class {name= "B", axioms = [] }, setI2 = Restriction {onProperty = "R", valuesFrom = SomeValuesFrom, value = "C"} }}] }`	Inv "A" (MultF mSome (Intersect (set "R" (["C"])) (set "B" []))))
`Class {name = "A", axioms=[EquivalentTo{ classEqui = AnonClass { operation = IntersectionOf { setI1 = Class {name= "B", axioms = [] }, setI2 = Restriction {onProperty = "R", valuesFrom = SomeValuesFrom, value = "C"} }}] }`	Gen "A" idenP "B" idenP, (Inv "A" (MultF (mJust (v)) (the "A" ["R"])))

Table 1. Examples of the translation of entities

Properties Table 2 shows the translation of properties. To notice, the translation of the DataTypeProperties is not complete. As it was already stated the DataTypeProperties are atributtes, and it was also said that PROVA has a Boilerplate named ATTR, however it only supports atributtes of type Integer, so it was decide that the translation of DataTypeProperties is the same as the ObjectProperties.

OWL Haskell	**PROVA's Boilerplates**
`ObjectProperty { oPName = "R", oPdomain = "A", oPrange = "B", characteristic = [] }`	has "A" (From 0) NotFixed "R" "B"
`DataTypeProperty { oPName = "R", oPdomain = "A", oPrange = a, characteristic = [] }`	has "A" (From 0) NotFixed "R" a (Not Correct)

Table 2. Examples of the translation of Properties

Individuals Table 3 shows the translation of individuals, which is pretty straight forward.

OWL Haskell	**PROVA's Boilerplates**
`Individual { nameI = "a", parentClass = Class { name = "A", axioms = []} }`	Mult (mOne) (set ("a") []), Gen "b" idenP "A" idenP

Table 3. Examples of the translation of Individuals

6 Conclusions and Future Works

The Haskell OWL parser will be *open-source*, however it only parses a little subset of OWL, the next step would be to parse the remaining statements, and try to cover the majority of the Web Ontology Language, even though OWL can be written in many ways. Also another work would be to complete the translation of OWL to PROVA's Boilerplates and basically make the PROVA front-end able to import the OWL file which is simple, since the parser is already made and it's just to import the file and inject the content in the parser. PROVA's concept is innovative, but it lacks documentation to build a strong code based in the core of the application. It's an new modelling platform however documentation has to be done one day or another. To conclude, this work is not finished. We've learnt a lot about Web Ontology Language and we deepened our Haskell knowledge, but it would requires further investigation of the OWL semantics to build a strong and correct parser, and much better understanding about PROVA's functionalities.

References

1. Coral Calero, Francisco Ruiz, M.P.: Ontologies for Software Engineering and Software Technology. Springer (2006)
2. : W3c semantic web activity. http://www.w3.org/2001/sw/
3. : Owl web ontology language guide. http://www.w3.org/TR/owl-guide/
4. Hughes, J.: Generalising monads to arrows. (1998)
5. : Owl web ontology language semantics and abstract syntax. http://www.w3.org/TR/owl-semantics/
6. : Owl dl semantics. http://www.obitko.com/tutorials/ontologies-semantic-web/owl-dl-semantics.html