Expressing ontologies using a functional language

J.S. Posthuma
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente, Enschede, The Netherlands
j.s.posthuma@student.utwente.nl

ABSTRACT
Moving towards the next generation of information technology, computers take over more and more human tasks. A limitation of the computer compared to a human is that it has no ‘world knowledge’. An ontology is an abstract representation of data that can be used to define this knowledge. Because of that abstract quality, expressing an ontology in an abstract programming language provides several benefits. In this paper, three approaches are proposed and discussed for expressing an ontology using the abstract paradigm of functional programming.

Categories and Subject Descriptors
I.2.4 [Artificial intelligence]: Knowledge Representation Formalisms and Methods; D.1.1 [Programming techniques]: Applicative (Functional) Programming—Haskell

Keywords
Ontology, reasoning, functional programming

1. INTRODUCTION
Both the human and computer capabilities of processing information are astonishing, of course, both in their own way. People are excellent in creating relations between parts of information, deriving additional information and are able to ‘understand’ it based on semantics. On the other hand, computers cannot understand the information, but can process the information at incredible speed. To fill the gap, different solutions were created to eliminate the limitations of computer information processing, approaching human capabilities.

One of the proposed solutions to solve the gap between the human and computer is an ontology. In general, an ontology is a “formal, explicit specification of a shared conceptualization” [4]. It tries to bring the advantage of a human having ‘world knowledge’ to the computer side, by specifying every ‘thing’ and their relations in the digital and the real world. For people familiar with the Object-Oriented programming paradigm, the definition of ‘thing’ can be compared to the definition of ‘class’.

In this paper, three possible approaches are presented that can be used when expressing an ontology using the functional programming paradigm. The question that leads to this research, the method of research and the outcome are described in section 2. Section 3 shows current usage of ontologies within computer science and describes popular solutions capable of working with ontologies. To be able to understand the different approaches presented by this paper, section 4 explains the basics of an ontology and presents the running example that is used to discuss the approaches. The results of the research, the three approaches, are described and discussed in sections 5, 6 and 7. In additional to the original planned research, section 8 evaluates several limitations of the language used by the different approaches. The most important findings will be stated in section 9. Finally, possible future research based on this paper is given in section 10.

2. RESEARCH QUESTION
An ontology is an abstract representation of knowledge. Expected is that an ontology can be expressed in an abstract programming paradigm in a natural way. Applicative programming, which exists of logical and functional programming, is a paradigm that features this level of abstractness. However, as stated in 3, expressing ontologies using a logical programming language is already done, leaving the functional programming paradigm as other option. Besides the possibility of expressing an ontology in an abstract way, functional languages are also expected to have features that simplify the reasoning that involves an ontology. So, the main research question reads:

- How can an ontology be expressed in a functional programming language?
  - What is the appropriateness of the different approaches on aspects like reasoning, readability, compactness and conversion from and to OWL?
  - What are limitations of the chosen language for expressing an ontology?

2.1 Method of research and results
The research behind this paper results in multiple approaches towards expressing an ontology in a functional language. Haskell is chosen as ‘representative’ for the functional languages. For all approaches written in Haskell, several positions and contentions are evaluated. This includes at least the aspects reasoning, readability, compactness and conversion from and to OWL, as stated in the research question.

The research will not result in a ‘perfect’ approach. The proof of that is outside the scope of this paper. Therefore, the approaches are explained and evaluated on their advantages and disadvantages.
First, an explanation of the approaches and why these approaches are chosen:

- **Direct translation from OWL**
  This is expected to be the most logical choice for most people with little experience in functional programming. Get the most popular implementation in one language and translate it into the other. Therefore it is not necessary to consider the different aspects that need to be covered, assuming that these aspects are already covered during development in the first language.

- **An Object Oriented Programming approach**
  This is the second logical choice. This approach is probably chosen by people who are familiar with functional programming, but are used to work using the OOP approach for a long time. The basic idea is that it uses much of the functional features, but still uses an object based structure for the data.

- **A functional approach**
  Here, it is tried to take a leap, as far away from first approach as possible. The main question is “How would knowledge be represented in a functional language?”. Still by using as many of the functional features as possible.

3. **KNOWN APPLICATIONS**

Currently there are already a number of ontology specification formalisms available, of which OWL (based on DAML + OIL) [18] is most popular. This is because it is one of the foundations on which the next generation of the internet (better known as web 3.0, or the semantic web) is built upon. The main feature of the semantic web, is that computers can reason with the information available on the internet in a semantical way. As shown in Tim Berners-Lee’s article on the semantic web [1], computers should be able to work autonomously while processing semantically annotated information. These semantics can be expressed using ontologies.

While the semantic web still covers a relatively small portion of the internet, several semantic reasoners already exist. To name a few popular OWL/semantic web reasoners: Pellet [16], RacerPro [5] and FaCT++ [17]. However, most popular semantic reasoners are written using the imperative programming paradigm. This is not a problem, in contrary, all the above mentioned reasoners have proven to be reliable and reasonably fast. However, as stated in the Research Question (2), an ontology is an abstract representation of the world. So, it is expected that expressing an ontology is less complex using more abstract programming paradigms.

Using a logical programming language for expressing an ontology usable for reasoning is already done. DLog [10] is a Prolog based semantic reasoner, which converts an OWL ontology into a prolog program. This program can be used to reason based on the given ontology by using Prolog’s natural reasoning capabilities. However a logical programming language does not have to be restricted to only one reasoning algorithm, Prolog, which is one of the most popular logical languages, currently is. This means that reasoning could be more effective when the expressiveness of an ontology falls within languages like SHON, SHIN, SHON’ and SHN [16] for which more efficient algorithms are available [16]. Although not the reason of this research, a functional language is not limited in the choice of reasoning algorithm.

4. **ASPECTS OF AN ONTOLOGY**

4.1 **Ontologies in short**

The definition of ontology within computer science is as follows: “An ontology defines a set of representational primitives with which to model a domain of knowledge or discourse.” [3]. In computer science, the representational primitives are the earlier mentioned classes, properties, and relationships. An ontology is popular in the Artificial Intelligence field, where it is used to create computational models to enable certain kinds of automated reasoning [7]. This is also one of the main reasons ontologies are introduced in the semantic web. As stated by Tim Berners-Lee [1], the idea of the next generation of the internet is that not only people can understand the information available on the World Wide Web, but that computers (up to a certain level) can understand it. So, you can say that the current internet is extended using Artificial Intelligence.

4.2 **Elements**

As previously stated, an ontology is used for sharing “world knowledge”. Like in the real world, there are definitions and instances. For example, when someone tells about a car, it is known what that person means, because the same concept of a car is shared. This is called the definition. Of a definition, there (usually) are instances. In case of the car, every car in the world ‘is’ an instance of the concept car.

This is the basics of an ontology. An ontology contains a ‘Terminological Box’ (also known as the box), which holds all definitions, and the properties which will be explained later on. The ‘Assertional Box’ (abox), holds all instances.

Again, within the real world, almost everything is related. For example, when the concept ‘car’ is discussed, it is obvious that the vehicle that uses an engine and wheels to move, is meant. So a car is a vehicle, but a vehicle does not necessarily have to be a car. The relation between car and vehicle is called subclassing, where car is a subclass of vehicle, and is one of the many different logical relations possible in an ontology. Another example: an amphibious vehicle is the intersection of a land and water vehicle. In OWL, the most common logical class relations are: Subclass, Equivalence, Union, Complement and Intersection [18].

As stated in the introduction, there is another type of relations, called properties. Properties are user-defined relations between instances. Because they are relations between instances, they don’t apply to all instances, which is the case for class-relations. For example, a isMarried-With relation does not count for all person objects. With properties, it is also possible to have logical relations between properties. Because, when a person is a mother, it automatically is a parent. So, isMotherOf is a sub-property of isParentOf.

All these relations, both logical and user-defined, are part of the Terminological Box, because they only describe the world. In addition, the properties are applied in the Assertional Box.

4.3 **Running example**

To evaluate the different aspects, an example ontology is needed, which will be expressed in all approaches. However, most ontologies are too complex and too large for the scope of this paper/research. So a small example is developed trying to cover several features of the different approaches. The ontology is given in abstract syntax format, recommended by W3C [13]:
4.3.1 Terminological Box

ObjectProperty(isParentOf)
ObjectProperty(isFatherOf domain(Male))
ObjectProperty(isMotherOf domain(Female))
ObjectProperty(isChildOf inverseOf(isParentOf))
ObjectProperty(isMarriedTo inverseOf(isMarriedTo)
domain(Person) range(Person))
ObjectProperty(sportsWith inverseOf(sportsWith)
domain(Person) range(Person))
SubPropertyOf(isFatherOf isParentOf)
SubPropertyOf(isMotherOf isParentOf)
Class(Person)
Class(Female partial Person
restriction(isMarriedTo allValuesFrom(Male)))
Class(Male partial Person
restriction(isMarriedTo allValuesFrom(Female)))

4.3.2 Assertional Box

Individual(Sarah type(Female) value(isParentOf Sam))
Individual(Chris type(Male) value(isMarriedTo Sam))
Individual(Bill type(Male)
value(isChildOf Chris)
value(isChildOf Sam))

This ontology contains the following elements: Properties, Classes and Individuals. In addition, it demonstrates sub-properties, inverse properties and partial/subclasses.

4.4 Instance

Individual(Sam type(Person) value(sportsWith Chris))
Individual(Chris type(Male) value(isMarriedTo Sam))

Finally the properties are specified.

This is where the actual ontology definition begins. It first defines the three (person, male and female) classes, and then defines the different relations between the classes. Finally, there is the box rule. This only collects all previously defined elements, so they can be accessed easily later on.

data TerminologicalBox = TBox [Class | Property]
data AssertionBox = ABox [Instance]
data Class = Class [CDescription]
| ABox [Instance]
data CDescription = Id String
| Restriction Constraint
| Intersection Class
=data Constraint = AllValues [Class]
| SomeValues [Class]
data Axiom = SubClass Class
| Domain [Class]
| Range [Class]
data Property = Property [PDescription]
data PDescription = SubProperty Property
| Inverse Property
| Domain [Class]
| Range [Class]
data Instance = Instance Class
| (Property, Instance)

Just like the original definition of the OWL language, there is a clear separation between the basic elements of an ontology, namely, for the Terminological Box, the Classes and the Properties, and for the Assertion Box, the Instances. A Class can contain multiple class-descriptions and axioms [18]. These descriptions specify a possible id, can add Class constraints, and specify the logical relations. An axiom can describe a subclass and/or an equivalent class-relation. The Property is a collection of multiple property-descriptions. Finally, an Instance is created by specifying the Class and possible adding the object properties.

5. DIRECT TRANSLATION FROM OWL

OWL is the proposed standard for defining ontologies in the next generation of the internet [18]. For that reason, a direct translation from OWL to a functional language is an logical approach. Because the complete specification of the OWL Full language is too complex and too long for this paper, only the elements needed by the example ontology are used.

5.1 Predefined structure

The OWL language exists only to specify ontologies. A functional language exists to fulfill many goals. That’s why it is logical that the basic structure that exists in the OWL language does not (yet) exists in a functional language. So, a predefined structure is needed to “host” all OWL elements in the functional language. Therefore, the W3C recommendation is used as the basic guide of all OWL elements [18]. Note that the structure is completely ontology independent. It does not need to be included in the ontology itself. The structure is given as follows:

- person = Class [(Id "person")
| Domain [person], Range [person]]
- male = Class [(Id "male")
| hasParent = Property [Domain [person]]
| isMarriedTo = Property [Inverse isMarriedTo
| Domain [person], Range [person]]
| isMotherOf = Property [Domain [female]
| hasParent = Property [Domain [female]]
| isFatherOf = Property [Domain [male]
| hasParent = Property [Domain [male]]
| isParentOf = Property [Inverse isParentOf]
| isChildOf = Property [IsChildOf]
| isMarriedTo, isChildOf]
5.3 Assertional Box

Like in the Terminological Box, this part of the ontology is defined using the predefined structure. Again, the axol rule is used to collect all Assertional Box elements for later access.

\[
\text{sarah} = \text{Instance female [isParentOf, sam]} \\
\text{chris} = \text{Instance male [isMarriedTo, sam]} \\
\text{sam} = \text{Instance person [(isChildOf, chris)]} \\
\text{bill} = \text{Instance male [(isChildOf, chris), (isChildOf, sam)]} \\
\text{abox} = \text{ABox [sarah, chris, sam, bill]}
\]

5.4 Discussion

The first noticeable thing is that an ontology requires a predefined structure. This is needed because the elements that are used by the OWL language are not part of a functional language. However, this shouldn’t be a real problem, because when someone wants use the ontology, it still needs additional logic. This might be to check the ontology, or derive information from it using reasoning. The predefined structure could then be a part of the libraries that would be included to use the ontology.

**OWL conversion**

A downside of the self defined structure is that the person that creates the ontology needs to know exactly which structures are available. It cannot use the basic language structures because they might add extra limitations on how to specify an ontology. On the other hand, that is also the biggest advantage of defining a specialized structure. The given OWL ontology can be written in exactly the same structure. The basic data structures available by the functional language won’t be of any restriction when creating an ontology.

**Readability**

The predefined structure also effects readability. When a person that is familiar with a functional language (which is assumed), an ontology written purely in that functional language can easily be read by that person, because it does not need to know any (possible) unknown structures. This however is not the case. The person reading the ontology must be familiar with both the OWL elements as the predefined structure, to be able to understand the ontology.

**Reasoning**

Of course, when the basic structures aren’t used, it is impossible to benefit from it. So every possible structure that would facilitate the reasoning process cannot be used. Of course this does not mean that the reasoning itself cannot benefit from a functional language, generally speaking.

**Compactness**

At last, the compactness of the ontology. Because this method defines a structure ‘on top’ of the basic structure it also adds additional effort writing the ontology. While it might not be of great importance for an ontology to be compact in times where computers can process information at lightning speeds, for a person to read, it is still favourable when information has as little unnecessary data around it as possible.

6. AN OBJECT ORIENTED APPROACH

Currently, most programmers are imperative programmers. This is one of the reasons why most popular software, i.e. operating systems, databases or normal desktop software, is made in an imperative programming language. Nowadays imperative programming languages often support the Object Oriented programming approach [9]. However, Object Oriented Programming (OOP) does not necessarily have to be done in an imperative programming language. It is a design pattern that uses the concept of Objects to create its data structures [14]. So it is possible to use OOP for specifying the ontology in a functional language. This approach, however, is used in a slightly different way. Because most programmer are imperative programmers, it is for many people a logic choice to use that experience when creating an ontology, even if this is in a functional language. Because this imperative experience often comes with the OOP pattern, the Object Oriented approach is a logic choice. This section tries to show how an ontology mainly would like when it is designed by a person with an imperative and Object Oriented background.

6.1 Note about the functional language

As previously stated, the functional language chosen to express the different functional ontologies, is Haskell. Although Haskell is a purely functional language [6], it does not cover all structures possible in a functional language. Some obvious limitations will be ignored to enable more elegant approaches in some places. However, this will only be if it is easy to work around the limitation of the language, but uses extra unnecessary space. This topic will be discussed in section 8.

6.2 Terminological Box

With this approach, the direct line between the Terminological Box and the Assertional Box is already much more unclear than the OWL approach. However, for clarity, the ontology is still divided into the two separate parts.

\[
\text{data Class} = \text{Type String} \\
\text{data Type} = \text{Person} \\
\text{data Person} = \text{Male | Female} \\
\text{objects} = \text{person} \\
\text{person} = \text{male ++ female}
\]

6.3 Assertional Box

\[
\text{sarah} = \text{Female *Sarah*} \\
\text{chris} = \text{Male *Chris*} \\
\text{sam} = \text{Female *Sam*} \\
\text{bill} = \text{Male *Bill*} \\
\text{male} = \text{[chris, bill]} \\
\text{female} = \text{[sarah, sam]} \\
\text{isMarried :: Person -> Person} \\
\text{isMarried p | p == chris = sam} \\
\text{isMarried p = inverse isMarried person p} \\
\text{isMotherOf :: Female -> [Person]} \\
\text{isMotherOf p = inverse isParentOf female p} \\
\text{isFatherOf :: Male -> [Person]} \\
\text{isMotherOf p = inverse isParentOf male p}
\]
have no experience with expressing an ontology but are for extra structure. This also means that people, who properties are defined using functions makes the ontology Readability & Compactness

The fact that Classes can be defined using datatypes, structures define the class structure, and the Ontology structures are used as possible. This means (like in approach data define, they should first be converted into normal classes, where the superclass gets converted into the union of those classes. Thus, conversion requires additional logic to solve these limitations.

7. A FUNCTIONAL APPROACH

"How would an ontology be expressed in a functional language, trying to benefit as much of the functional language as possible?", is the starting point of the third approach. Like with every choice or implementation, there is no perfect approach possible. If there is, the proof of such an approach lies beyond this paper. Because the datatype of Haskell is limited and the Classes and Instances only exist to contain the data, the focus of this approach lies in the extended use of properties as functions. On that point there is more improvement available to facilitate the reasoning without heavily compromising compactness and readability.

7.1 Terminological Box

As with the step from the first to the second approach, with the step from the second to the third approach, the direct line between the Terminological Box and the Assertional Box fades even more. This is not necessarily a bad thing, because in this case, it only means that the classes and implementations get more closely related.

7.2 Assertional Box

As with the step from the first to the second approach, the direct line between the Terminological Box and the Assertional Box fades even more. This is not necessarily a bad thing, because in this case, it only means that the classes and implementations get more closely related.

7.3 Discussion

Because this approach tries to benefit as much as possible from the functional aspect, there are as much 'basic structures' used as possible. This means (like in approach two), that the classes and instances are created using the datatypes in Haskell.

OWL Conversion

The conversion from an OWL ontology to this approach requires an extra step. Because subclasses are difficult to define, they should first be converted into normal classes, where the superclass gets converted into the union of those classes. Thus, conversion requires additional logic to solve these limitations.

6.4 Discussion

The biggest difference with the previous approach, is the direct use of the basic data structures, in this case, the data construct. With this structure, it is possible to predefine a datatype that can be used later on. In this case, these data structures define the class structure, and the Ontology classes (person, male and female) itself. There is just one (relatively unimportant) predefined structure needed.

That is, the Class data structure. It is used to hold the type of the class and the unique identifier together. This is partially because otherwise it is (at least for Haskell) not possible to distinguish the different instances. Then the instances itself. The objects are captured in a list, and because of that, every logical operation is possible by using the list functions. Haskell itself has built-in support for the union, intersection and so on. The reason why these relations aren’t defined using the data structures is one of the limitations of Haskell. Its lack of subtypes of other logical operations on datatypes forced us to use a different approach.

Reasoning

Now the basic datatype is used, an interesting feature is available. Where the properties can be defined in the same way as the instances, it is also possible to model them as functions. This approach brings some limitations but mainly add features that simplify the task of reasoning a lot. An example is the inverse action on a property. Where a normal reasoner has to implement this by themselves, in a functional language a simple solution arises. This is demonstrated for the ‘One to One’ (1 → 1) and ‘One to Many’ (1 → *) relations:

\[
\text{isParentOf} :: \text{Person} \rightarrow [\text{Person}] \\
\text{isParentOf} p \quad | \quad p = \text{ sara h } = [\text{ s a m } ] \\
\text{isParentOf} p = \text{ inverse isChildOf } p \\
\quad + \quad \text{ isFatherOf } p \quad + \quad \text{ isMotherOf } p
\]

\[
\text{sportsWith} :: \text{Person} \rightarrow \text{Person} \\
\text{sportsWith} p \quad | \quad p = \text{ s a m } = [\text{ chris, sam }] \\
\text{inverse} \quad \text{sportsWith } p
\]

\[
\text{isChildOf} :: \text{Person} \rightarrow [\text{Person}] \\
\text{isChildOf} p \quad | \quad p = \text{ bill } = [\text{ chris, sam }] \\
\text{inverse} \quad \text{isParentOf } p
\]

6.4 Discussion

The biggest difference with the previous approach, is the direct use of the basic data structures, in this case, the data construct. With this structure, it is possible to predefine a datatype that can be used later on. In this case, these data structures define the class structure, and the Ontology classes (person, male and female) itself. There is just one (relatively unimportant) predefined structure needed.

That is, the Class data structure. It is used to hold the type of the class and the unique identifier together. This is partially because otherwise it is (at least for Haskell) not possible to distinguish the different instances. Then the instances itself. The objects are captured in a list, and because of that, every logical operation is possible by using the list functions. Haskell itself has built-in support for the union, intersection and so on. The reason why these relations aren’t defined using the data structures is one of the limitations of Haskell. Its lack of subtypes of other logical operations on datatypes forced us to use a different approach.

Reasoning

Now the basic datatype is used, an interesting feature is available. Where the properties can be defined in the same way as the instances, it is also possible to model them as functions. This approach brings some limitations but mainly add features that simplify the task of reasoning a lot. An example is the inverse action on a property. Where a normal reasoner has to implement this by themselves, in a functional language a simple solution arises. This is demonstrated for the ‘One to One’ (1 → 1) and ‘One to Many’ (1 → *) relations:

\[
\text{inverse} f o i = [x \mid x \leftarrow o, \text{elem } i (f x)] \quad \rightarrow \quad 1 \rightarrow 1 \\
\text{inverse} f o i = [x \mid x \leftarrow o, i == (f x)] \quad \rightarrow \quad 1 \rightarrow *
\]

This function simply takes another function (property), then maps all possibilities over that function, and gathers all valid answers. In the example, f is the variable for the function, o contains the list of possible answers, and i is the ‘searched for’ variable. Because the properties are defined as functions, the Range and Domain limitations can automatically be written using the function-guides of Haskell. So can the \text{isMotherOf} property be restricted to be used only with females as the domain and persons as range by stating that the function needs to comply to \text{isMotherOf} :: \text{Female} \rightarrow [\text{Person}].

Readability & Compactness

The fact that Classes can be defined using datatypes, instances can be the instances of those datatypes and that properties are defined using functions makes the ontology compact and easier to read for people. There is no need for extra structure. This also means that people, who have no experience with expressing an ontology but are familiar with the functional programming paradigm, are capable of reading the ontology.

OWL Conversion

The conversion from an OWL ontology to this approach requires an extra step. Because subclasses are difficult to define, they should first be converted into normal classes, where the superclass gets converted into the union of those classes. Thus, conversion requires additional logic to solve these limitations.

7. A FUNCTIONAL APPROACH

"How would an ontology be expressed in a functional language, trying to benefit as much of the functional language as possible?", is the starting point of the third approach. Like with every choice or implementation, there is no perfect approach possible. If there is, the proof of such an approach lies beyond this paper. Because the datatype of Haskell is limited and the Classes and Instances only exist to contain the data, the focus of this approach lies in the extended use of properties as functions. On that point there is more improvement available to facilitate the reasoning without heavily compromising compactness and readability.

7.1 Terminological Box

As with the step from the first to the second approach, with the step from the second to the third approach, the direct line between the Terminological Box and the Assertional Box fades even more. This is not necessarily a bad thing, because in this case, it only means that the classes and implementations get more closely related.

\[
data \text{ Person } = \text{ Male | Female } \\
\text{ objects } = \text{ people } \\
\text{ people } = \text{ male } \leftrightarrow \text{ female }
\]

7.2 Assertional Box

\[
data \text{ Male } = \text{ Chris | Bill } \\
data \text{ Female } = \text{ Sarah | Sam } \\
\text{ male } = [\text{ Chris, Bill }] \\
\text{ female } = [\text{ Sarah, Sam }] \\
\text{ isMarried Chris Sam } = \text{ True } \\
\text{ isMarried } \alpha \beta = \text{isMarried } \beta \alpha \\
\text{ isMotherOf } \alpha \beta = \text{isParentOf } \& \text{ elem } \alpha \text{ female } \\
\text{ isFatherOf } \alpha \beta = \text{isParentOf } \& \text{ elem } \alpha \text{ male } \\
\text{ isParentOf Sarah Sam } = \text{ True } \\
\text{ isParentOf } \alpha \beta = \text{isChildOf } \beta \alpha \\
\| \text{ isMotherOf } \alpha \beta \\
\| \text{ isFatherOf } \alpha \beta \\
\text{ isChildOf Bill Chris } = \text{ True } \\
\text{ isChildOf Bill Sam } = \text{ True } \\
\text{ isChildOf } \alpha \beta = \text{isParentOf } \beta \alpha 
\]
However the focus lies on the properties, it is noticeable that the instances are part of the datatype definitions. This in comparison to the second approach, where the still were instances of a datatype. This brings the advantage that they can be used, using pattern matching.

**Reasoning**

In this approach, the properties are defined as boolean functions. They require two parameters, which need to be instances, and return based on those two instances if the relation exists or not, True or False. When modeling the inverseOf element of a property, no extra function is needed any more. The function can directly be mapped by switching the parameters of the function, and forwarding it to the function that should be the inverse. For example, the isParentOf is the inverse property of isChildOf. So, when the property is used, and it does not find a direct relation, it continues to look for the parent by, instead of asking if a person is the parent of another person, asking if that second person is the child of the first. Not only the inverse element can be used in this way. In the case of sub-properties, the parameters can be forwarded to its sub-functions. This is also shown by isParentOf, by forwarding the function to isFatherOf and isMotherOf. Another advantage of using boolean functions as properties, is that it is possible to benefit from higher-order functions. With these higher-order functions, it is possible to create the existential ∃ and universal ∀ quantifiers from predicate logic to work on these property-functions. With these quantifiers, it is possible to eliminate an important step in the reasoning process, namely property-chaining. Property chaining is a common way to derive information from an ontology that is not directly visible. For example: Sarah is the parent of Sam, and Bill is the child of Sam. This means that Sarah, who is a woman, is the grandmother of Bill. Generally spoken, person α is a grandmother of person γ, when that person α is the mother of a person β, who is the parent of person γ. This can be written in predicate logic as follows (where X denotes all objects):

\[ P(\alpha, \gamma) = \exists_{\beta \in X} (\text{isMotherOf}(\alpha, \beta) \land \text{isParentOf}(\beta, \gamma)) \]

The definition of ∃ and ∀ result in the following two Haskell functions. They require two partial applied functions, and combines this using the higher-order functions. The all function only exists to generalize the source (objects) of the objects for both functions:

```haskell
forall :: (Object -> Bool) -> (Object -> Bool) -> Bool
forall alpha beta = and . map beta . all $ alpha
exists :: (Object -> Bool) -> (Object -> Bool) -> Bool
exists alpha beta = or . map alpha . all $ alpha
all alpha = filter alpha objects
```

Converting the earlier given predicate to Haskell results in the following code:

```
isGrandmotherOf alpha beta = exists (isMotherOf alpha) (isParentOf beta)
```

Now, when is asked if Sarah is the grandmother of Bill, the answer is given through the following steps:

- All possible objects for isMotherOf are given
- This is done by filtering the objects by isParentOf α, given that the object is a female (elem α female)
- For the remaining objects, it is checked if they match isParentOf to β
- This time, the isParentOf returns the answer by using the inverse of isChildOf
- Finally, the exists function checks if there is a valid object left that matched all criteria

**Readability**

However this approach again only uses the basic structures, it might be confusing why the properties are defined as boolean functions, and don’t just return the valid values. It also isn’t possible to define multiple answers in one line, used by the ‘one-to-many’ properties. Every relation has to be specified on its own (this is the case for isChildOf). Another aspect is the Domain and Range elements of a property. They cannot be specified as function-guides, so must be implemented in the logic of the rule itself. In the case of isMotherOf, adding the following code: `elem α female`.

**Compactness**

The difference in compactness compared to the second approach is the limitation of ‘one-to-many’ properties. This adds a little extra space.

**OWL Conversion**

The conversion from OWL is primarily the same as approach two. It also requires an extra step of processing that works around the limitations of this approach, such as the separation of ‘one-to-many’ properties into multiple ‘one-to-one’ properties.

### 8. HASKELL LIMITATIONS

While the approaches prove that expression an ontology in a functional language has benefits, during research, there appeared some limitations of the language chosen to represent the functional languages. In this section, the most important limitations discovered are discussed.

#### 8.1 Infinite loop detection

When the properties are defined as functions, it is a logical choice to use the ‘passing through’ of data to create the logical relations between the properties. However, doing this so for the inverse- and sub-property relations often results in an infinite loop. This is the case for the isMarried property, which is the inverse of itself:

```
isMarried Chris Sam = True
isMarried α β = isMarried β α
```

A way to work around this, is by using a helper function:

```
isMarried α β = || isMarried β α
_isMarriedChrisSam = True
```

This however is not very desirable. What might be the ideal solution is a form of ‘infinite loop detection’. This means that the language needs to keep track of the order in which the functions, including its parameters, and detect when it ends in an infinite loop. This is possible for a functional language, because there are no shared variables. However, it needs to be specified what the return value is when such a loop is detected.
8.2 Logic relations between datatypes

The way to define data-structures in Java, is by using classes. Following the Object Oriented pattern, it is possible to define a relation between classes, by using inheritance. In Haskell, datatypes are used to define data-structures. It is, however, not (yet) possible to define relations between datatypes. Although, there are some proposals for implementing subtyping [11, 12]. If the datatypes would be seen as a collection of fields, subclassing, union and intersection could be used like this:

\[
\text{data Vehicle} = \text{Vehicle}\{\text{brand ::= String, age ::= Int}\}
\]

\[
\text{data Car} \subseteq \text{Vehicle} = \text{Car}\{\text{numberOfWheels ::= Int, maxSpeed ::= Int}\}
\]

\[
\text{data Boat} \subseteq \text{Vehicle} = \text{Boat}\{\text{numberOfSails :: Int, maxSpeed :: Int}\}
\]

\[
\text{data Amphibian} = (\text{Car} \cap \text{Boat}) = \text{Amphibian}
\]

The subset-relation of Car and Boat to Vehicle adds the brand and age to their definitions. Subsequently, the Amphibian would then contain all fields, including numberOfWheels and numberOfSails.

8.3 Reflection

In an ontology, when defining a Class and creating several instances, a list of all instances of that Class is often needed. Gathering all instances of a type is part of reflection, but is not possible in Haskell [8, 2, 15]. As an example of which use this reflection would be, a part of the second approach is used:

\[
\begin{align*}
\text{sarah} &= \text{Female "Sarah"} \\
\text{sam} &= \text{Female "Sam"}
\end{align*}
\]

\[
\text{female} = [\text{sarah, sam}]
\]

When using reflection, it is not needed to gather all instances manual. In this case this is done by female. Assuming that there is a reflection library, a call to get all instance might look like Reflect.InstancesForType Female. Because this ontology has only two instances of the Female datatye, it is not necessary for this case, but as with the ontology grows, keeping the list of objects up-to-date gets unmanageable.

9. CONCLUSION & DISCUSSION

Because of its natural abstract character, it is a natural choice to describe an ontology in an abstract language: the functional programming language. This paper presents three different approaches on how an ontology can be expressed in Haskell. The first approach, as a direct translation from the popular Semantic Web format OWL. Its strong points clearly are the direct translation. Every possible construction or relation that can be made in the original OWL ontology can also be expressed in the functional approach. This however also has its downsides. It needs a predefined data-structure that is build on top of the basic data-structure that Haskell provides. This results in an ontology that is not very compact or easily readable. In addition, when using the ontology for semantic reasoning, it cannot benefit from the functional features that rely on that basic structure, resulting in a ‘normal’ reasoning algorithm.

For the second approach, the angle of Object Oriented programming is used. This results in an ontology where the basic data-structures are used for defining and instantiating the classes and instances. In combination with properties defined as functions, several ‘elements’ of an ontology, like the domain and range restrictions, and the inverse relation of the properties could be implemented using function-guides and a predefined function. This takes over several tasks that earlier would have been done by the reasoner. In readability and compactness, this also is an improvement, because of the use of the basic data-structures. It can be read without any knowledge of a predefined structure. However, the conversion from OWL to this approach is not a simple conversion. Not all elements defined in the OWL can (easily) be translated into Haskell data-structures, and requires additional translation logic that cannot guarantee lossless conversion. It might even require reasoning.

The last approach begins with a question, “How would an ontology be expressed in a functional language, trying to benefit as much of the functional language as possible?”. This again results in the use of datatypes for the classes and instances, and defining properties as function, however, in a different way. With some small changes in the classes and instance, the main difference lies in the definition of the properties. Because they are defined as boolean function, property-relations as inverse and subproperty can be created using only the boolean operations ∨ and ∧. The most interesting feature however is that the it is possible to create the existential ∃ and universal ∀ quantifiers from predicate logic, using only one function each. This results in the possibility of function chaining, without using a reasoner, resulting in a less complex reasoning algorithm. The readability and compactness are almost equal to the second approach. The conversion from OWL to this approach however is slightly more complex than the second approach.

As stated in the Research Question, section 2, there is no ‘perfect’ approach, and even if it were so, the proof of that lies beyond this paper. However, two of the approaches clearly show some advantages. The first approach in the capability of fully expressing the OWL language, and the third approach in taking over multiple features usually done by a reasoner. So, when the ontology is only needed for keeping data in a functional way, the first approach seems the logic choice. However, when the ontology is used to derive and answer semantic queries, using a reasoner, the third approach has the most potential.

During the research, it became clear that the language used for the approaches, Haskell, has its limitations as ‘representative’ of a functional language. While there is no definition of ‘the’ functional language, when solving the limitations listed in 8, Haskell would probably still be a functional language. However, this proof also lies beyond this paper.

10. FUTURE WORK

Because the internet is moving to the semantic web, ontologies become more important as a solution for saving data in a semantic way. With that semantic web, reasoning based on this ontologies also becomes more important. This makes it worthwhile to examine how the different reasoning strategies, developed for ontologies that fall within the expressiveness of SHOIN, SHIQ, SHOIN and SHI [16], would perform using the different approaches. This requires the development of multiple reasoners for the different approaches, but could very well result in a reasoning system that is capable of competing with the currently available imperative reasoners.

Of course, future work can also be done by continuing the development of different approaches that might be
more suitable expressing an ontology. During research, an extra approach was developed, but wasn’t completed, due to a limited time. This approach uses infix boolean functions to define classifications, like subclass, equals and intersection. This approach has potential, because it does not need the functional features presented in 8.2.

11. ACKNOWLEDGMENTS

I would like to thank my fellow-students and mentor of the “Data Application and Integration” track for their valuable input. Also, thanks to Jan Kuper, who supplied me with enough enthusiasm and criticism to continue the research.

12. REFERENCES