A Functional API for OWL

Aaron Eberhart[0000−0003−3007−5460] and Pascal Hitzler[0000−0001−6192−3472]

Data Semantics Lab, Kansas State University, USA
{aaroneberhart, hitzler}@ksu.edu

Abstract. We present (f OWL), a minimalistic, functional programming style ontology editor that is based directly on the OWL 2 Structural Specification. (f OWL) is written from scratch, entirely in Clojure, having no other dependencies. Ontologies in (f OWL) are implemented as standalone and homogeneous data structures, which means that the same exact functions written for single axioms or expressions often work identically on any part of an ontology, even the entire ontology itself. The lazy functional style of Clojure also allows for intuitive and simple ontology creation and modification with a minimal memory footprint. All of this is possible without ever needing to use a single class, except of course in the Ontologies one creates!

1 Motivation

The semantic web community has widely adopted the OWL API [2] for ontology creation and development. Many researchers and programmers find this software very useful and enjoy its Object-Oriented style. Other APIs for OWL have also been written in imperative languages like Owlready [3] in Python. However, some developers prefer to program in functional languages and find that the imperative style APIs are more of a hindrance than a help when dealing with ontologies. There has been work towards providing functional style programming for OWL, like Tawny-OWL [4], but this is fundamentally dependent on the OWL API so it is cosmetically functional but not obviously compatible with many of the unique features and optimizations of the functional style. In order to support other styles of programming for ontologies we have developed (f OWL), a functional perspective on ontology development.

2 (f OWL)

As an original standalone piece of software, (f OWL) is not a wrapper for the OWL API. It starts again at the beginning and is truly functional from the ground up. (f OWL) was written with Clojure because it is a functional language that compiles with the Java Virtual Machine, and thus will have a broad compatibility with any machine that can run Java. Clojure has the added benefit

Functional	Lisp-like syntax
No dependencies	Requires only Clojure
No classes	Objects can be created and manipulated directly
Ontology data structure	Homogeneous and uniform: can be traversed
Lightweight	Supports OWL 2, no unnecessary baggage
Clojure Efficiency	Lazy functions and optimizable recursion
Concise function names	Avoids verbose unhelpful Java conventions

Table 1. Features of (f OWL)

of intuitive native laziness, so writing efficient programs to create and manipulate ontologies is not a challenge. To ensure that the semantics are correct, (f OWL) directly implements the grammar of the OWL 2 Structural Specification [5]. And because it is entirely original, (f OWL) is divorced from any historical dependencies and commitments that clutter other software.

(f OWL) includes many novel optimizations that streamline ontology development. True to the functional paradigm (f OWL) uses functions and shuns classes. Indeed all OWL ontology objects can be created directly with (f OWL) functions. And since these functions are not bound up in an arbitrary class hierarchy, they can operate on independent data structures that are internally typed to represent OWL semantics. A (f OWL) ontology itself is simply another data structure made of collections of smaller similar structures. This means that in most cases an expression, an axiom, even an ontology can be traversed recursively as-is without writing more functions. (f OWL) also simplifies many of the unhelpful, highly verbose Java-style function names that occur in other programs and instead adopts a concise functional naming scheme.

(f OWL) is built and tested using Leiningen[1] and the source code can be found on GitHub.[2] During testing, it efficiently reads and writes copies of over 260 different functional syntax ontologies from various domains, including the massive Gene ontology [1], with no errors being detected. It is available on the Clojars[3] repository so that users can easily import it into their own Clojure projects. ClojureDoc[4] for (f OWL) is available, and documentation is also accessible in the read–eval–print loop (REPL) for individual functions while writing programs.

2.1 Examples from the REPL

– Create an axiom

```
fowl.core=> (ont/implies (ont/exists "r" "a") "b")

SubClassOf(ObjectSomeValuesFrom(r a) b)
```

[1] https://leiningen.org/
[2] https://github.com/aaronEberhart/fOWL
[3] https://clojars.org/onto.aaroneberhart/fowl
[4] https://cljdoc.org/d/onto.aaroneberhart/fowl/0.0.1-SNAPSHOT/doc/readme

– Show the documentation for a function

```
fowl.core=> (doc ont/makeOWLFile)

-------------------------
ontology.core/makeOWLFile
([ontology filename & fileType])
  Writes an owl file of the ontology with the supplied file name.
  Optional argument allows choice of file type. No option defaults to functional syntax.
  (Currently only functional syntax defined)
```

– Make an ontology with a Clojure threading expression

```
fowl.core=> (-> ont/emptyOntologyFile
                (ont/setOntologyIRI "http://www.test.iri")
                (ont/addAnnotations (ont/annotation "annotations" "are fun"))
                (ont/addPrefixes (ont/prefix "" "http://www.test.iri/")
                                 (ont/prefix "prefix" "http://www.prefix.iri/"))
                (ont/addAxioms (ont/implies "a" (ont/IRI "prefix" "b"))
                               (ont/implies (ont/or "d" "g") "a")
                               (ont/implies (ont/inverseRole "r") "s")
                               (ont/fact "a" "i")
                               (ont/fact "r" "j" "i")))

Prefix(rdf:=<http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
Prefix(rdfs:=<http://www.w3.org/2000/01/rdf-schema#>)
Prefix(prefix:=<http://www.prefix.iri/>)
Prefix(:=<http://www.test.iri/>)
Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix(owl:=<http://www.w3.org/2002/07/owl#>)

Ontology(<http://www.test.iri>

Annotation(:annotations "are fun")

ObjectPropertyAssertion(:r :j :i)
SubObjectPropertyOf(ObjectInverseOf(:r) :s)
SubClassOf(ObjectUnionOf(:d :g) :a)
SubClassOf(:a prefix:b)
ClassAssertion(:a :i)
)
```

– Add every third axiom from a vector into a set with a tail recursive loop

```
fowl.core=> (loop [counter 0
                   axiomSet #{}
                   axioms [(ont/implies (ont/exists "r" "a") "b")
                           (ont/implies (ont/or "b" "c") (ont/not (ont/or "d" "e")))
                           (ont/implies (ont/roleChain "r" (ont/inverseRole "s")) "t")
                           (ont/fact (ont/inverseRole "s") "i" "j")
                           (ont/fact "a" "i")
                           (ont/fact "d" "i" (ont/stringLiteral "l"))
                           (ont/implies (ont/roleChain "s" "q") "t")]]

              (if (empty? axioms)                ; are there no axioms left?

               axiomSet                          ; return axiom set

               (recur                            ; else recurse with new values

                (inc counter)                    ; counter + 1

                (if (= 0 (mod counter 3))        ; is counter mod 3 == 0?
                 (conj axiomSet (first axioms))  ; add first axiom to set
                 axiomSet)                       ; else keep current axiom set

                (rest axioms))))                 ; remove first axiom

#{ObjectPropertyAssertion(ObjectInverseOf(s) i j)
  SubClassOf(ObjectSomeValuesFrom(r a) b)
  SubObjectPropertyOf(ObjectPropertyChain(s q) t)}
```

– Use a partial function and a list comprehension to make axioms for a class

```
fowl.core=> (let [aImplies (partial ont/implies "a")]
              (for [b ["c" (ont/all "r" "d") (ont/and "e" "f" "g")]] (aImplies b)))

(SubClassOf(a c)
 SubClassOf(a ObjectAllValuesFrom(r d))
 SubClassOf(a ObjectIntersectionOf(e f g)))
```

More examples are available on the project GitHub page.

3 Evaluation

As a preliminary benchmark, we choose a few operations in (f OWL) that are commonly used and have near equivalent counterparts in the OWL API: read an ontology, write an ontology, get the NNF of all class axioms. Table 2 shows the average time to complete a task, and Table 3 shows the average times when they are each scaled by the number of axioms in the ontology. As expected, the performance of (f OWL) with respect to the OWL API closely parallels the difference between native Java and Clojure. Though it is usually faster when reading files, as indicated by the scaled times, (f OWL) is slowed down while reading large files by the need to construct many immutable objects for the ontology, which is typical for Clojure programs. However, even a large (f OWL) ontology can be accessed and traversed in a very efficient manner. This allows (f OWL) to write files quickly. And it can get the NNF of all class axioms significantly faster than even optimized stream functions.

All testing was done on a computer running Ubuntu 20.04.1 64-bit with an Intel Core i7-9700K CPU@3.60GHz x 8, 47.1 GiB DDR4, and a GeForce GTX 1060 6GB/PCIe/SSE2.

	Read File	Write File	Get NNF
OWL API	167.1696	91.61577	23.737692
(f OWL)	1275.384	68.87327	13.186410

Table 2. Average of times in milliseconds for tested ontologies

	Read File	Write File	Get NNF
OWL API	85.367181	8.82415379	1.8501445
(f OWL)	60.724610	6.01522460	0.6555225

Table 3. Average times in nanoseconds scaled by number of axioms

4 Conclusion

(f OWL) provides a lightweight functional alternative for OWL ontology development. It has numerous benefits that will doubtless increase as development finalizes and it matures. (f OWL) is highly practical for developers who prefer a functional language, it supports all of OWL 2, and provides unique functional methods for structuring and manipulating ontology data.

5 Future Work

There are two major efforts in progress to improve this project. The first is writing and testing new functions to allow the parsing of XML and RDF OWL files. This capability is integrated into the current project framework but it is incomplete and takes a fair amount of time to test and debug. Additionally, because OWL files are frequently in XML and RDF format we have delayed the implementation of ontology imports until after these file types can be read. Once the imports are implemented (f OWL) should support at least as much semantics for standard OWL ontology development as other APIs. A custom reasoner is also in the early stages of development for (f OWL). We hope that this will streamline reasoning over (f OWL) ontologies by running while they are created.

Acknowledgement This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-18-1-0386.

References

1. Gene Ontology Consortium: The Gene Ontology (GO) database and informatics resource. Nucleic Acids Research **32**(Database-Issue), 258–261 (2004)
2. Horridge, M., Bechhofer, S.: The OWL API: A Java API for OWL ontologies. Semantic Web **2**(1), 11–21 (2011)
3. Lamy, J.B.: Owlready: Ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies. Artificial intelligence in medicine **80**, 11–28 (2017)
4. Lord, P.: The semantic web takes wing: Programming ontologies with tawny-owl. In: Rodriguez-Muro, M., Jupp, S., Srinivas, K. (eds.) Proceedings of the 10th International Workshop on OWL: Experiences and Directions (OWLED 2013) co-located with 10th Extended Semantic Web Conference (ESWC 2013), Montpellier, France, May 26-27, 2013. CEUR Workshop Proceedings, vol. 1080. CEUR-WS.org (2013), `http://ceur-ws.org/Vol-1080/owled2013\_16.pdf`
5. Parsia, B., Patel-Schneider, P., Motik, B.: OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition). W3C recommendation, W3C (Dec 2012), http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/