Horned-OWL: Flying Further and Faster with Ontologies

Phillip Lord
School of Computing, Newcastle University, United Kingdom

Björn Gehrke
Institute for Implementation Science in Health Care, Faculty of Medicine, University of Zurich, Switzerland

Martin Larralde
Leiden University Medical Center, The Netherlands
Structural and Computational Biology Unit, EMBL, Heidelberg, Germany

Janna Hastings
Institute for Implementation Science in Health Care, Faculty of Medicine, University of Zurich, Switzerland
School of Medicine, University of St. Gallen, Switzerland
Swiss Institute of Bioinformatics, Switzerland

Filippo De Bortoli
TU Dresden, Germany
Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI), Dresden/Leipzig, Germany

James A. Overton
Knocean Inc., Toronto, Canada

James P. Balhoff
Renaissance Computing Institute, University of North Carolina, Chapel Hill, NC, USA

Jennifer Warrender
School of Computing, Newcastle University, United Kingdom

Abstract

Horned-OWL is a library implementing the OWL2 specification in the Rust language. As a library, it is aimed at processes and manipulation of ontologies, rather than supporting GUI development; this is reflected heavily in its design, which is for performance and pluggability; it builds on the Rust idiom, treating an ontology as a standard Rust collection, meaning it can take direct advantage of the data manipulation capabilities of the Rust standard library. The core library consists of a data model implementation as well as an IO framework supporting many common formats for OWL: RDF, XML and the OWL functional syntax; there is an extensive test library to ensure compliance to the specification. In addition to the core library, Horned-OWL now supports a growing ecosystem: the py-horned-owl library provides a Python front-end for Horned-OWL, ideal for scripting ontology manipulation; whelk-rs provides reasoning services; and horned-bin provides a number of command line tools.

The library itself is now mature, supporting the entire OWL2 specification, in addition to SWRL rules, and the ecosystem is emerging into one of the most extensive for manipulation of OWL ontologies.

2012 ACM Subject Classification Applied computing → Life and medical sciences; Software and its engineering → Software libraries and repositories; Information systems → Web Ontology Language (OWL)

Keywords and phrases Web Ontology Language, OWL, Semantic Web

Digital Object Identifier 10.4230/TGDK.2.2.9

Category Resource Paper

Supplementary Material Version 1.0.0 of the source code for Horned-OWL was used for the performance results. Version 1.0.1 of the source code for py-horned-owl were used for the performance results.
Software (Source Code): `https://github.com/phillord/horned-owl` [12]
Software (Documentation): `https://docs.rs/horned-owl/1.0.0/horned_owl/` [10]
Software (Source Code): `https://github.com/ontology-tools/py-horned-owl`
Software (Source Code): `https://github.com/INCATools/whelk-rs`

Funding *Björn Gehrke*: Funded by the Wellcome Trust in the GALENOS project.
Filippo De Bortoli: Supported by the German Federal Ministry of Education and Research (BMBF, SCADS22B) and the Saxon State Ministry for Science, Culture and Tourism (SMWK) by funding the competence center for Big Data and AI "ScaDS.AI Dresden/Leipzig".

Acknowledgements During the course of this work Ignazio Palmisano provided extensive help supporting our understanding of the OWL2 specification, and how the OWL API implemented it. We are very grateful; Horned-OWL would not have been possible without him.

Received 2024-06-27 **Accepted** 2024-11-19 **Published** 2024-12-18

Editors Aidan Hogan, Ian Horrocks, Andreas Hotho, Lalana Kagal, and Uli Sattler

Special Issue Resources for Graph Data and Knowledge

1 Introduction

The Web Ontology Language (OWL) has been in existence since 2003, moving to OWL2 in 2009 [5, 4]. It provides a specification for developing ontologies, which provides computationally amenable, logical descriptions of the world. These ontologies are applicable to any domain of knowledge, but have become embedded in biomedicine in particular, where ontologies have got both large individually (in biology, the widely used Gene Ontology - GO [14] - contains $\sim$ 50,000 terms, while in the medical domain SNOMED CT contains roughly 300,000 terms), and spread over an enormous range of biological domains, with the BioPortal [13] containing over 1000 ontologies, containing 14 million terms.

OWL consists of a set of different specifications that describe a data model, a formal semantics that can be used to determine computational entailment, and a number of different syntaxes for serialization, including a mapping to RDF which enables OWL to play its part in the semantic web.

The majority of the current infrastructure for OWL is built using Java; at the time of the inception of OWL, Java was in its prime, and was one of the most common languages in scientific computing. As a result, much of the infrastructure for OWL is implemented in Java, including the OWL API [6] and Apache Jena.

The use of Java brings with it a number of issues. While it is possible to write fast Java, neither the language nor the standard idioms for its use are designed for performance; for example, while the Gene Ontology is large in ontological terms it is only 50,000 terms and 500,000 axioms or when serialized as RDF 5,000,000 triples; in computational terms this is not large, but using the OWL API, GO can still take minutes to read into memory. Second, Java is less and less widely used in scientific computing having been largely displaced by Python; this limits Java's practical utility, especially within Jupyter notebooks which have become a major tool for reproducible science. Finally, Java poorly integrates with machine learning and AI tooling which again is mainly implemented in Python, which limits the ease with which knowledge-rich OWL ontologies can be used with or by this massive growth area in computing.

In this paper, we introduce and describe Horned-OWL, a novel library that provides a Rust API to process OWL ontologies. Compared to the OWL API [6], this library is aimed at scalability and performance; analytical capabilities, rather than GUI development; and interoperability with Python.

Currently, Horned-OWL implements the core OWL data model, which we discuss in detail in Section 2.2, a pluggable system for indexing which is the focus of Section 2.3 and support for (de)serialization in RDF/XML as well as some other formats detailed by the W3C standard, detailed in Section 2.4.

Aside from the library, Horned-OWL provides a suite of tools to operate with ontologies akin to ROBOT [7] that showcase the usage of the library itself. This suite, named horned-bin, is presented in Section 3, where we additionally describe existing tools that rely on the library, such as the whelk-rs reasoner (Section 3.2) and the py-horned-owl interface (Section 3.3). An evaluation of the performance of the library appears in Section 4 and shows that Horned-OWL substantially outperforms the OWL API particularly in memory usage.

2 Horned-OWL

2.1 Background on OWL2

We start with a brief discussion of the OWL2 specification; an overview is available directly from W3C [5] which we summarize here.

OWL2 is part of the Semantic Web stack which, as the name suggests, brings explicit semantics, while integrating with the Web. It is widely used as a language for the representation of ontologies, which are models of a domain shared between a community of users. In OWL2, the explicit semantics comes from an underlying description logic, which itself maps to first order logic; the link to the web comes from an alignment with the syntax and semantics of RDF/S which is the semantic web representation of a knowledge graph, the use of XML Schema datatypes and IRIs as its primary identifier.

Although at heart, OWL2 simply describes a set of individuals and the relationships between them, it is fairly complex: it has six different types of named entity and 37 different types of axiom. This complexity is such that OWL also supports profiles; simpler subsets with different computational properties. As well as this semantics, OWL2 has three different syntaxes, plus a mapping to RDF which itself has at least five different syntaxes in common use. On top of this, we also have SWRL, which, although not strictly part of OWL2, is sometimes used in OWL ontologies.

It is this complexity which makes OWL2 rather challenging to implement; it was a design aim to support all of OWL2 (including SWRL), however, because we wanted it to operate over the many biological ontologies that already exist; this is also a complex ecosystem and most of the parts of OWL2 are used somewhere in that ecosystem.

2.2 Design

The core of Horned-OWL is the **model** namespace. As the name suggests, this implements the core data model of OWL. As a part of the semantic web, OWL is built largely on top of the IRI (Internationalized Resource Identifiers). This is defined in Horned-OWL simply as follows:

```
  pub struct IRI<A>(pub (crate) A);
```

The type `A` is generic and could be any type; we will cover the reason for introducing this genericity in a later section describing the Python implementation. While the type of `A` is not constrained on the `IRI` struct, in practice the vast majority of methods in Horned-OWL do constrain it to the `ForIRI` trait, which allows using the contents as a string. This design pattern of types unconstrained on struct, but constraints on methods is common in Rust and considered best practice. In practice, the most common type of `A` is `Rc<str>` which is a reference counted pointer, meaning that multiple instances of the same `IRI` do not each instantiate their own string.

```
  impl<T: ?Sized> ForIRI for T where
    T: AsRef<str>
        + Borrow<str>
        // Other traits omitted for length
{
}
```

The generation of new IRI instances is handled by a single entity, called Build. This also handles the generation of AnonymousIndividual instances which, by definition, do not have IRI identifiers. The Build object allows caching and sharing of IRI and, in addition, operates as an "Arena" object, meaning that all IRI instances in an ontology will share the same lifetime.

```
pub struct Build<A: ForIRI>(g
    RefCell<BTreeSet<IRI<A>>>,
    RefCell<BTreeSet<AnonymousIndividual<A>>>,
);
```

While OWL uses IRIs to identify most entities, in Horned-OWL, we use a struct for all OWL named entities; this is known as the newtype pattern in Rust and adds type safety to most of the methods in Horned-OWL; it is not possible to pass an IRI identifying a class to a function requiring an object property, for instance. The Class struct is defined as follows:

```
  pub struct Class<A>(IRI<A>)
```

An OWL ontology consists of a number of components; in Horned-OWL these are all modelled as a single large **enum**. This enum includes all OWL axioms, the ontology IRI and version IRI; and, although not strictly part of OWL, we also support SWRL rules through this mechanism. This varies slightly from the formal definition of OWL; the main advantage of this approach is described later.

Any component in OWL can also support a set of annotations which we support through the use of the following struct:

```
pub struct AnnotatedComponent<A> {
    pub component: Component<A>,
    pub ann: BTreeSet<Annotation<A>>,
}
```

Ontology components themselves have individual representations. For example, a DeclareClass axiom is defined using a "tuple struct" as follows:

```
pub struct DeclareClass<A>(Class<A>)
```

Conversely, the SubClassOf axiom uses named fields, trading concision for readability.

```
 pub struct SubClassOf<A> {
   sup: ClassExpression<A>,
   sub: ClassExpression<A>
 }
```

The complexity of OWL is such that Horned-OWL uses a system of rules to determine naming, with as much consistency as possible, to ensure that in use it maintains the principle of least surprise; it should be possible to guess most field names for anyone familiar with OWL.

An ontology itself is represented by an empty "tagging" trait in Horned-OWL as follows:

```
pub trait Ontology<A> {
}
```

This seems rather perverse given that Horned-OWL is an API for ontologies, but the reason is because an OWL ontology is treated as a set of components; the ontology needs no methods because all information about that Ontology object is available from one of its components. In practice, every type that implements `Ontology` also supports conversion to a Rust `Iterator`; this means that `Ontology` is simply a specialisation of a Rust collection[1].

In addition, a `MutableOntology` type has been defined which provides a generic mechanism for adding and removing entities from an ontology.

2.3 Indexing

To make practical use of the ontology as a collection that Horned-OWL implements, we need to support one or more indexes – mechanisms for efficient look up and querying of ontology components. Horned-OWL does not take the route of providing sensible defaults for its "standard" ontology implementation. Instead, it enables a series of indexes which can be plugged in; this ensures that in use we pay only for the cost of indexes that we need to use.

Currently, Horned-OWL provides data structures for ontologies with four or less indexes; there is no fundamental limitation here, and higher numbers would be possible. Within Horned-OWL itself, three indexes are the most that are used at one time, for the RDF reader.

The simplest ontology index is the `SetIndex` with associated `SetOntology`. This simply stores all ontology components in a memory-backed set. It is useful on its own, but also in conjunction with other indexes, because it guarantees to record all components added to it. However, access to components of the ontology requires iteration.

Perhaps the most useful index is the `ComponentMappedOntology`; this allows rapid access to ontology components based on their type. For example, the following code, which is a unit test, shows how to access to all the `DeclareClass` axioms. As with the `SetOntology`, this also stores all components added to it.

```
let mut o = ComponentMappedOntology::new_rc();
let b = Build::new_rc();
o.declare(b.class("http://www.example.com/a"));
assert_eq!(o.i().declare_class().count(), 1);
```

Other ontology index types include:

Declaration Mapped: Look up the type of a named entity given an IRI
IRI Mapped: Look up components containing a given IRI
Logically Mapped: Look up components logically equal (i.e. ignoring annotations) to an existing component

The importance of these ontology indexes cannot be overstated. For instance, the RDF reader constructs and returns an `RDFOntology` which uses a `SetIndex`, `DeclarationMappedIndex` and a `LogicallyMappedIndex`. This is critical because RDF parsing requires regularly looking up triples that have been already parsed to understand and decode later triples; in particular, it must understand the declared type of many IRIs. Without use of the `DeclarationMappedIndex` this requires a full iteration of the ontology for each lookup which, in practice, means the RDF parser would operate in cubic or worse time. This would result in catastrophically poor performance; our first naive implementation of RDF parsing took well over an hour to parse the Gene Ontology for example. The equivalent XML parser does not need these indexes as the type of an IRI is always given at point of use; the plugging indexing system means, that is does not have to pay the cost of building them.

[1] Restrictions in its type system means that it was not possible to represent this notion directly in Rust. The recent addition of "Generic Associated Types" may make this representation possible; however, GATs are currently quite limited and their availability came well after Horned-OWL was developed.

2.4 Input/Output Framework

In its current version, Horned-OWL supports several different syntaxes of OWL: the OWL/XML syntax defined by part of the OWL 2 specification [2], the OWL/RDF syntax and the OWL functional syntax. The `io` module contains a submodule for each syntax, which in turn consists of a `reader` and a `writer` module. Each of these different formats has some idiosyncrasies.

We first offered support for the OWL/XML syntax, as it was relatively straightforward to implement. The reader works on a single file at a time and returns both an ontology (currently, a `SetOntology`) and a `PrefixMapping` supplying the IRI prefixes. In this case, both the reader and writer heavily rely on the quick-xml crate.

Supporting the OWL/RDF syntax is rather more complex. One of the main challenges is that, to correctly parse an ontology written using this syntax, we must also recursively parse all the imported RDF graphs to determine the kind of entity associated to an IRI, which may be declared in one of the imported graphs or later in the ontology that is currently being parsed. In contrast for OWL/XML this is not needed, as the type of each IRI is given at the point of use. For this reason, the OWL/RDF reader, which reads a single ontology, comes with a companion `closure_reader` which attempts to parse the full import closure. This has been optimised so that the `closure_reader` only parses those parts of the ontology that are absolutely necessary – imported ontologies are only parsed until all the IRIs encountered previously are associated to a declared entity.

Another issue is that no ordering of the triples in the RDF graph is guaranteed, and so the parser may have to traverse the list of RDF triples multiple times; in some more perverse cases, this could result in poor performance; for example, an RDF list where the triples appear from the last item to the first would parse in quadratic time; in practice, all the ontologies we have found in RDF use a first to last appearance which parses in linear time. As was noted in Section 2.3, Horned-OWL indexes are used to avoid other cases which would result in quadratic or worse performance. The reader, therefore, returns a highly indexed ontology type. To read RDF triples, the reader relies on the rio crate.

The OWL/RDF writer in Horned-OWL uses the pretty_rdf crate, which was implemented specifically to support Horned-OWL. The rio crate, which is used in the reader, is focused on serialisation of RDF as a set of triples; pretty_rdf, conversely, supports many of the RDF shorthand syntaxes, something that the OWL API also supports. This makes the RDF output more readable and concise, at the cost of increased time and complexity in serialisation.

Support for the OWL functional syntax is a recent addition to Horned-OWL; this uses the Pest crate to generate the parser from a parsing expression grammar. The writer is implemented through a Rust trait and supports writing most types of the `horned-owl` crate without copying data or requiring a dedicated writer type.

Although not yet complete, we plan to fully support the Manchester syntax for ontologies, which will bring Horned-OWL into parity with the OWL API in terms of the syntaxes it supports [3].

Furthermore, the fact that rio supports other RDF syntaxes such as N3 and Turtle means that Horned-OWL could be further expanded to offer readers for these formats; with a little more effort, we could also support writing in these formats as pretty_rdf uses a data model very similar to rio, and rio can write to these syntaxes. However, we lack a good use case for doing this work. One current limitation, however, is that Rio does not return an IRI prefix mapping for RDF which makes roundtripping hard.

[2] `https://www.w3.org/TR/owl2-xml-serialization/`
[3] We remain a little conflicted as to whether this is a good thing; while each of the syntaxes have their advantages, it is not so clear why so many syntaxes are needed

3 Ecosystem

3.1 Command Line

As well as providing a library, Horned-OWL provides an increasing number of command line tools. These support a git-style subcommand command line and provide tools for operating on OWL files in batch. For the current release, these tools are rather biased toward the sort of functionality needed for debugging Horned-OWL, but we expect these to expand into a full suite for OWL manipulation in batch in due course. Currently available tools include:

horned-big: Generates OWL files of arbitrary size, useful for performance testing
horned-compare: Compares the statistics of two ontologies
horned-materialize: Downloads the OWL import closure
horned-parse: Parses an OWL file for errors only
horned-summary: Provides summary statistics of an ontology

3.2 Reasoning interface and whelk-rs

Horned-OWL includes a preliminary reasoner interface defining functions for classifying ontologies, checking entailments and consistency, and retrieving inferred superclasses and subclasses. This interface is implemented by whelk-rs, a port to Rust of the Whelk reasoner, which targets the Java OWL API, which is an adaptation of the reasoning rules defined by Kazakov et al. [8]. whelk-rs supports the OWL EL profile, a subset of the OWL language targeted to scalable reasoning on large, structured terminologies such as biomedical ontologies. Initial testing suggests that whelk-rs is approximately twice as fast as the original Scala-based Whelk reasoner. Our expectation is that performance will improve further with plans to adopt a more idiomatic Rust coding style in the future.

3.3 Python bindings and py-horned-owl

One of the shortcomings of Rust is its learning curve, which is notoriously steeper than that of many other programming languages. In particular, the ownership and borrowing mechanisms of Rust and the fact that it is strongly typed result in this language being not very well suited for prototyping and short development cycles. On the other hand, a language like Python is rather easy to use and widely employed in many industrial and research areas, including scientific computing; however, it tends to be rather slow, especially for highly computational tasks.

The goal of the py-horned-owl[4] library is to make the power and functionality of Horned-OWL available to Python programmers. In this, we are following a frequent design pattern, used by libraries such as NumPy or SciPy, implementing a front-end for most use in Python with a backend written in a more performant native language (C, C++ or Fortran in the case of NumPy and SciPy).

Under the hood, py-horned-owl uses the PyO3 bindings to map the data structures, enums and functions defined in Horned-OWL to Python classes, union types and methods. This allows for the creation and manipulation of ontology components within the Python environment, similarly to what is done in Horned-OWL for Rust. For example, py-horned-owl provides the `PyIndexedOntology` class, representing an ontology with helper methods to query its components based on one or more indexes. One of these indexes is `IRIMappedIndex`, which allows quick access to components by their IRI.

[4] `https://github.com/ontology-tools/py-horned-owl`

We can load existing ontologies using the `open_ontology` method, which supports all OWL serialisations available in Horned-OWL. The axioms and components of the ontology can be queried. The following illustrates a typical scenario of usage:

```python
import pyhornedowl
ontology = pyhornedowl.open_ontology("<path/to/ontology>")
# Get all components
components = ontology.get_components()
# Construct an axiom
from pyhornedowl.model import *
axiom = SubClassOf(
 ontology.clazz(":Child"),
 ObjectSomeValuesFrom(
     ontology.object_property(":has_parent"),
     ontology.clazz(":Human"))
 )
# Add the axiom
ontology.add_axiom(axiom)
```

Since PyO3 outputs native Python modules, the static type information asserted in Rust is lost in the conversion process. However, py-horned-owl provides stubs that encode type information and provide hints which, for example, allow IDEs or other tools to do static type checking.

The development of py-horned-owl has directly influenced that of Horned-OWL. In earlier versions of the crate, `IRI`s were implemented as a newtype wrapper around `Rc<str>`; however, this type cannot be used across threads which is required for PyO3, necessitating instead the use of `Arc<str>`; this type is fully synchronized which, according to the documentation, comes at a 20-30% performance cost. To avoid paying unnecessary allocation and performance costs and retain flexibility, the types used in Horned-OWL have been then made fully generic, leading to the current version of the crate.

4 Evaluation

4.1 Testing

Horned-OWL contains an extensive series of tests to ensure consistency and compliance to the OWL2 specification: in total there are 928 unit tests, 46 doc tests (which are unit tests but visible as examples in code documentation) and 7 integration tests. The test set takes around 3s to run; the majority of this time comes from the doc tests, which is quite normal for Rust.

The bulk of these tests come from the IO framework. All of the readers and writers use a common series of tests. Test files are written using Tawny-OWL [11] which provides a clean, declarative and documentable representation of OWL, backed by the OWL API [11]. These files are then used to generate all the other required formats. For the OWL reader, tests are written by hand and compared to pre-determined semantics hard-coded in Rust; for example we generate ontology serialisations containing a single class declaration:

```clojure
(defclass C)
```

The generated OWX file is parsed and resultant Horned-OWL ontology is checked as follows:

```rust
assert_eq!(ont.i().declare_class().count(), 1);
assert_eq!(
String::from(&ont.i().declare_class().next().unwrap().0),
    "http://www.example.com/iri#C"
);
```

Other readers are checked by parsing and comparing the ontology to that generated by parsing the OWX files. Writers are tested by roundtripping: reading, writing and reading again. These tests are predominantly defined parametrically: addition of a new ontology defined in Tawny-OWL will automatically result in new tests for all serialisations; similarly, we will be able to test any new serialisations against these ontologies.

There are a few other forms of test: the RDF reader includes some tests with triples in differing orders which Tawny-OWL cannot directly produce; and some tests cannot directly compare RDF or OWX read ontologies as these two formats differ slightly in their expressive capability. The test set is extensive with 130 different ontologies defined in Tawny-OWL, defining 373 logical components, 13 annotations, and 126 meta components (Ontology IRIs). All 48 of the Horned-OWL component types are used in these ontologies (each type of OWL axiom, SWRL rule and meta component) are present in these files, excepting doc IRIs which are implicit.

We have recently adopted the pre-commit framework, meaning that our tests are defined declartively and run before commit or push; additionally, standardized format and use of Rust idiom are enforced through the `rustfmt` and `clippy` tools respectively.

4.2 Performance

In order to test the performance of Horned-OWL we perform a series of tasks, using real world ontologies where possible. This performance testing can be found in the owl-performance repository and was conducted on a Ubuntu (64-bit) VM with 24GB of memory[5].

In Figure 1, we show the results of generating a simple large ontology: in this case, a set of OWL class declarations. This predominately tests for in-memory performance at constructing an ontology, then serialization of a structurally simple ontology; this functionality is built into Horned-OWL directly; equivalent code was written for the OWL and py-horned-owl. Horned-OWL shows approximately linear performance and is faster than the OWL API. py-horned-owl scales similarly.

The use of an artifical form of ontology is useful for testing scalability, because we can generate ontologies with an arbitrary number of axioms. However, the performance against real world ontologies might be quite different, and for this reason, we test against these next. We have chosen a number of well-known ontologies present in BioPortal; these differ in their size, in terms of logical content, annotation and the complexity of their axiomatisation. The largest, the NCBI taxonomy, representing the species taxonomy, is the simplest consisting only of classes, annotations and subclass statements. Details are shown in Table 1.

Table 1 Size and Complexity of Test Ontologies.

	No. Triples	No. Component	Size on Disk
BFO	1221	741	155k
GO	256778	752036	121M
ChEBI	7466140	4281507	772M
NCBI Taxonomy	17648344	16134154	1.5G

We test the parsing performance by simply parsing these ontologies and testing against time, as shown in Figure 2. Horned-OWL is faster in all cases. As can be seen, py-horned-owl shows a small performance penalty. Some of this may be because of the cost of the interface between Python and Rust, or the use of the synchronized `Arc` instead of the single-threaded `Rc`.

[5] The JVM was set to a max heap size of 10G.

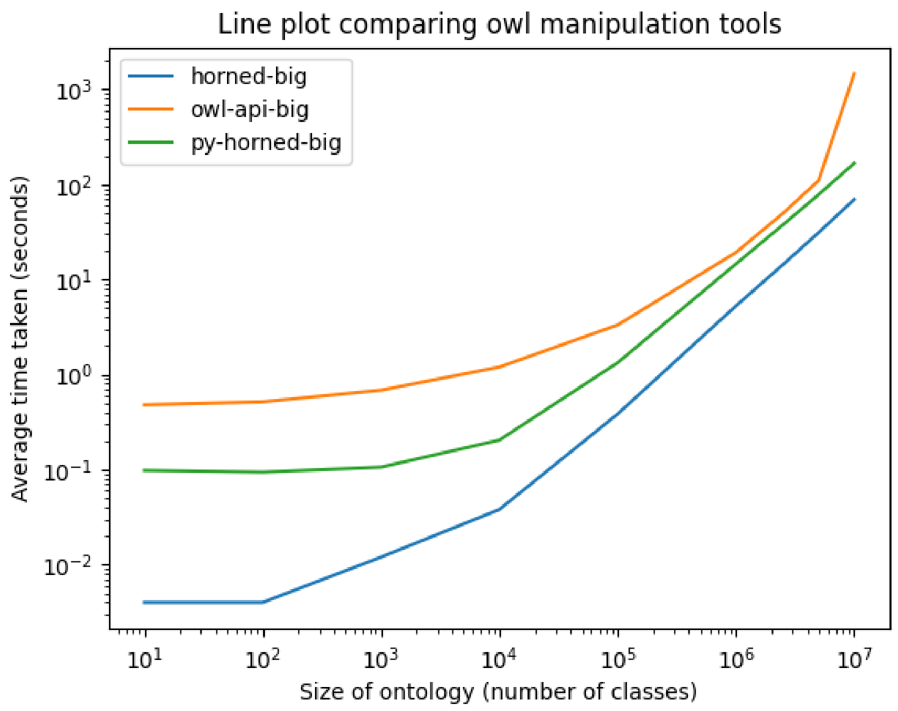


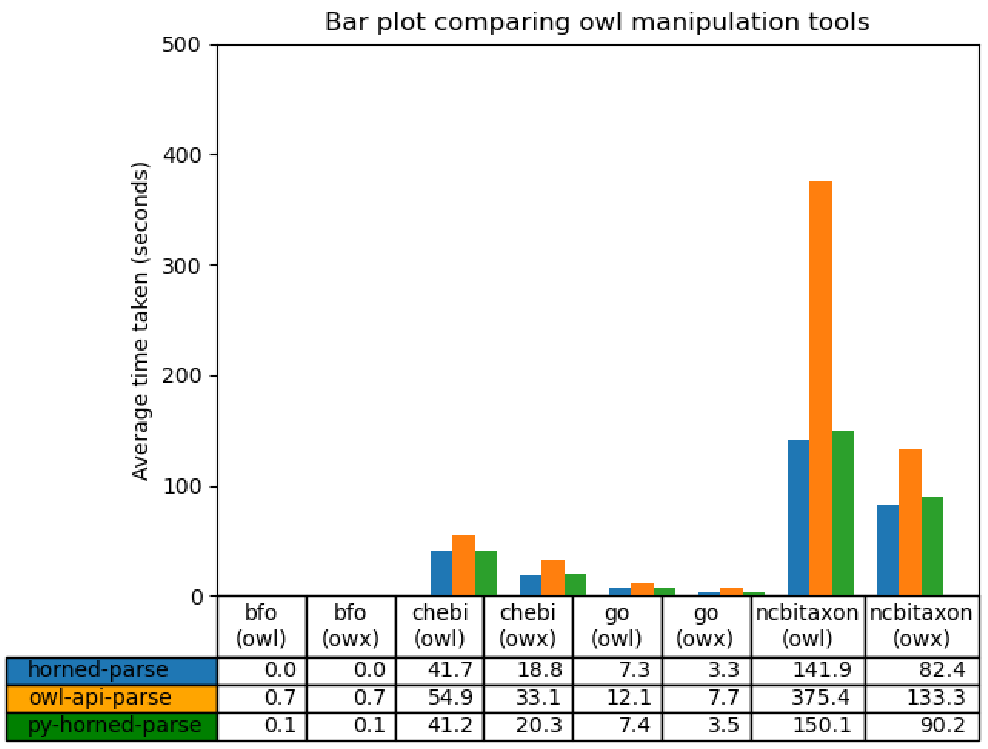
Figure 1 Generating a big ontology.

Finally, we test memory usage. This is a somewhat complex task given the different nature of the environments we are testing. Rust has deterministic memory use, but both Java and Python are garbage collected and will tend to use the memory that is available to them. We took, therefore, the simple approach of running each in restricted memory, we achieved on Ubuntu through the use of the `systemd-run` command. We test only whether the parsing completed or not. As can be seen from Figure 3, Horned-OWL is capable of running in a memory constrained environment. We have additionally tested extreme memory constraints: Horned-OWL is capable of parsing bfo.owl in 2M of memory, which is 20x smaller than the OWL API.

Performance testing is always a difficult task: there are many factors and variables to control and the tasks carried out are often time-consuming making heavy use of CPU. The results are frequently insightful, however; as a result of the work for this paper, we uncovered a performance bug in the associated pretty_rdf crate that resulted in poorly scalable performance while writing RDF[6]; likewise our analysis of py-horned-owl has made us reconsider the use and representation of indexes which resulted in substantial performance improvements. However, we will always be limited in a capacity to make such improvements while the performance testing is hard; therefore, we have now re-implemented a formal benchmarking harness for Horned-OWL; Rust support has considerably advanced since our first effort in this area; this should make our performance test results less emphemeral and will make future decisions on optimisations more possible.

In short, it is clear that Horned-OWL is already highly performant in terms of both CPU and memory usage and has a clear path to becoming more so.

[6] A three-line bugfix restored linear rather than quadratic performance which meant writing NCBI taxonomy became practical; see `https://github.com/phillord/pretty_rdf/commit/66466737`



	bfo (owl)	bfo (owx)	chebi (owl)	chebi (owx)	go (owl)	go (owx)	ncbitaxon (owl)	ncbitaxon (owx)
horned-parse	0.0	0.0	41.7	18.8	7.3	3.3	141.9	82.4
owl-api-parse	0.7	0.7	54.9	33.1	12.1	7.7	375.4	133.3
py-horned-parse	0.1	0.1	41.2	20.3	7.4	3.5	150.1	90.2

Figure 2 Parsing various well known ontologies.

4.3 Comparison to other libraries

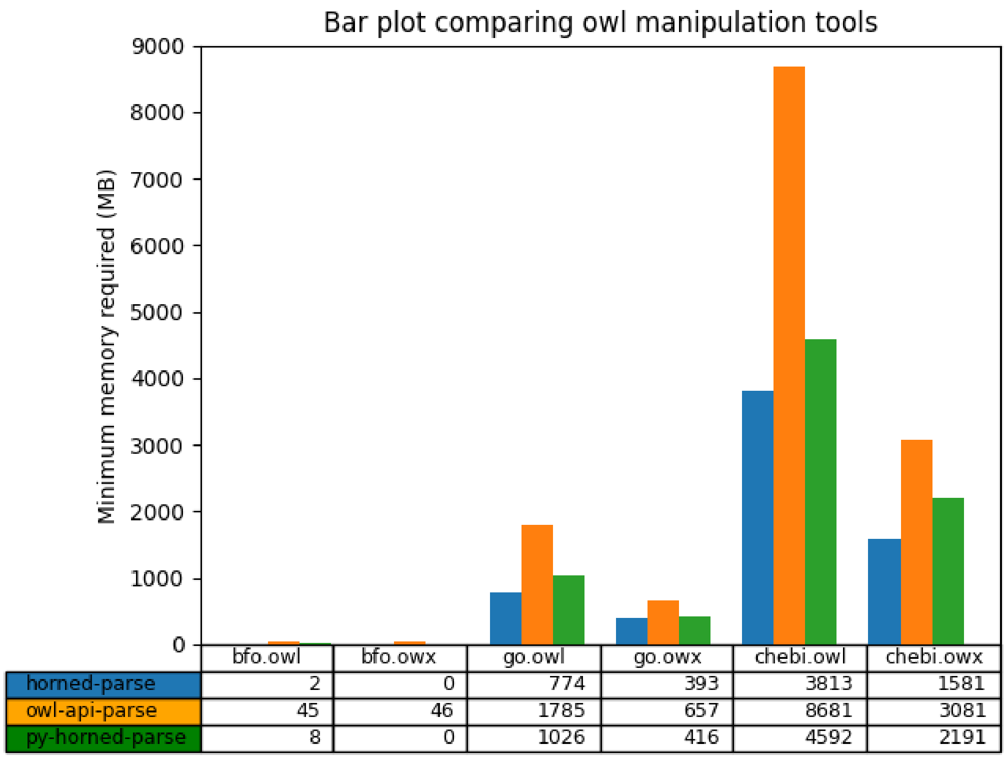
Since the Horned-OWL project started a number of other OWL libraries have been developed, which cover some of the same ground as Horned-OWL. Most notably here is OWLReady (now OWLReady2) which is a Python implementation of OWL2 [9]; its emphasis is on a high-level Object Oriented interface in Python for OWL, somewhat similar to Tawny-OWL [11] which takes a similar approach in Clojure. While this form of interface is very convenient for programming, our experience with both Tawny-OWL and the OWL API is that it is not as convenient for dynamic ontology manipulation and analyses. Additionally, it is likely to come with an overhead; our early analysis tends to confirm this, as Horned-OWL appears to be significantly more performant than OWLReady. A more recent addition to this space is COWL [2]; this is aimed at memory constrained embedded systems; it provides a data model and supports functional syntax only. Currently, Horned-OWL cannot perform well in this space, but we note that Rust does provide strong support for embedded systems through the `no_std` environment; this could be supported in later versions of Horned-OWL, providing a crate that, like COWL, supports only the data model and limited syntax options.

4.4 Limitations

Horned-OWL has a number of limitations.

We are currently testing Horned-OWL against all ontologies in BioPortal [13]. Unfortunately, the complexity of OWL means that comparing results to the OWL API is non-trivial; in addition, there are some known failures.

In the ideal world, given the predominance of the OWL API for the generation of OWL, we would like Horned-OWL to be identical with OWL API serializations. This is extremely challenging for a number of reasons; and this is particularly true for the RDF serialisation of OWL. First,



	bfo.owl	bfo.owx	go.owl	go.owx	chebi.owl	chebi.owx
🟦 horned-parse	2	0	774	393	3813	1581
🟧 owl-api-parse	45	46	1785	657	8681	3081
🟩 py-horned-parse	8	0	1026	416	4592	2191

Figure 3 Bar chart showing the affect restricting memory has on the parsing tools.

for a general OWL ontology it is neither possible to determine unambiguously what the RDF serialisation is nor, in reverse, determine the OWL ontology from a given RDF representation. Second, to add to this complexity, RDF provides a number of "shortcut" syntaxes which are both complex to implement and mean that a single RDF graph can be serialized in many different ways.

Similarly, while the XML representation of OWL is much less ambiguous and should roundtrip cleanly whether produced by the OWL API or Horned-OWL, the two are not currently lexically identical if for no other reason than for the use of whitespace. These lexical differences also impact on RDF/XML representation of OWL. This would be problematic for uses of Horned-OWL where ontologies are stored in tools such as git; switching regularly between the OWL API and Horned-OWL would result in a large number of misleading diffs. This could be circumvented by roundtripping the final ontology in a pipeline using either Horned-OWL or the OWL API. We note that a similar issue is currently caused by different versions of the OWL API which do not produce whitespace identical serialisation; we are sure the same issue will face Horned-OWL as it evolves.

We note that some differences in behaviour between Horned-OWL and the OWL API are pushing at the limitations of the OWL specification; it is not always clear which is correct. For example, the OWL API will produce empty IRI tags (`<IRI></IRI>`) which Horned-OWL refuses to parse; we see the impact of this in the performance testing, as Horned-OWL currently cannot parse `bfo.owx`. Similarly the OWL API adds typing information for built in classes (`<owl:AnnotationProperty rdf:about="http://www.w3.org/2000/01/rdf-schema#comment"/>`), which are probably correct but unnecessary; Horned-OWL benignly reports these as unhandled.

5 Discussion

Horned-OWL itself is now feature complete for OWL2, as well as including SWRL rules. As these specifications are now stable and themselves unlikely to evolve, our hope is that Horned-OWL itself will now show a similarly slow evolution. The core library (i.e. the Horned-OWL crate) now contains the data model and serialisation: the other features that we have added (the indexing described earlier, plus a visitor and some normalisation functionality) were necessary for efficient implementation of this core.

This does not mean that the overall environment of Horned-OWL will not expand; however, we will do so by adding additional crates. We have already gone this route by removing the command line function to its own crate (horned-bin) albeit one managed in the same repository as Horned-OWL; again, this is for reasons of performance; users of the library should not need to bear to the cost of additional dependencies required by these binaries. In the case of the command line library, we already have a good idea of the functionality that it is likely to need: the ROBOT tool [7] which is built on the OWL API, provides a clear exemplar here. For py-horned-owl, there is no real equivalent capability for scripting OWL and it will be interesting to see what functionality will be developed there. Finally, we note a possibility raised by Horned-OWL that we have not yet fully explored: Rust has strong support for WebAssembly which raises the possibility that OWL might yet become usable on the web.

One key area limitation for the current Horned-OWL ecosystem is in reasoning. Currently, whelk-rs provides support for the EL profile, but there is no DL reasoner available. We note that history has not been kind to many OWL2 reasoners with most abandoned or no longer usable [1]. Nonetheless, it should be possible to either port one of these to Rust, as whelk-rs has been, or use them directly through the Rust C ABI interface. We have not yet begun exploring whether this would be possible, nor whether it would be sensible, since the high worst case complexity of DL means the reasoners might not scale to the size of ontology that Horned-OWL can otherwise handle.

The initial experiments on Horned-OWL started over seven years ago. At this time, Rust was relatively immature, making initial progress quite slow. It has been pleasing to see that both the language and ecosystem has advanced substantially since this time. This has included an increased support for Semantic Web technologies: Horned-OWL for example, makes use of the rio[7] crate which provides RDF parsing support. Other Semantic Web technologies that are supported in Rust include SPARQL through the oxigraph, SHACL and shape expressions through the rudof crate and Linked Data through the sophia framework [3], to mention a few. Horned-OWL fits cleanly into this ecosystem, by providing support for OWL.

We believe that Horned-OWL is essential to the future utility and importance of the OWL specification and ontologies more generally. We have already noted the practical reality that Python now has completed a virtual take over in scientific computing, and that without good support for OWL in this language, scientists will simply move to other technologies. More importantly, however, while ontologies have been enormously successful, particularly in biomedicine, they now feature in much of the same space as newer AI technologies; these have highlighted what has always been a fundamental limitation of scalability. Horned-OWL cannot fully resolve this problem, but with its focus on performance it is a step in the right direction.

[7] `https://github.com/oxigraph/rio`

References

1 Konrad Abicht. OWL reasoners still useable in 2023, 2023. arXiv:2309.06888, doi:10.48550/arXiv.2309.06888.

2 Ivano Bilenchi, Floriano Scioscia, and Michele Ruta. Cowl: A lightweight OWL library for the semantic web of everything. In Giuseppe Agapito, Anna Bernasconi, Cinzia Cappiello, Hasan Ali Khattak, In-Young Ko, Giuseppe Loseto, Michael Mrissa, Luca Nanni, Pietro Pinoli, Azzurra Ragone, Michele Ruta, Floriano Scioscia, and Abhishek Srivastava, editors, *Current Trends in Web Engineering - ICWE 2022 International Workshops, BECS, SWEET and WALS, Bari, Italy, July 5-8, 2022, Revised Selected Papers*, volume 1668 of *Communications in Computer and Information Science*, pages 100–112. Springer, 2022. doi:10.1007/978-3-031-25380-5_8.

3 Pierre-Antoine Champin. Sophia: A Linked Data and Semantic Web toolkit for Rust. The Web Conference 2020: Developers Track, April 2020. URL: https://www2020devtrack.github.io/site/schedule.

4 Bernardo Cuenca Grau, Ian Horrocks, Boris Motik, Bijan Parsia, Peter Patel-Schneider, and Ulrike Sattler. OWL 2: The next step for OWL. *Journal of Web Semantics*, 6(4):309–322, 2008. Semantic Web Challenge 2006/2007. doi:10.1016/j.websem.2008.05.001.

5 W3C OWL Working Group. OWL 2 Web Ontology Language Document Overview (Second Edition), 2012. URL: https://www.w3.org/TR/owl2-overview/.

6 Matthew Horridge and Sean Bechhofer. The OWL API: A java API for OWL ontologies. *Semantic Web*, 2(1):11–21, 2011. doi:10.3233/SW-2011-0025.

7 Rebecca C. Jackson, James P. Balhoff, Eric Douglass, Nomi L. Harris, Christopher J. Mungall, and James A. Overton. ROBOT: A tool for automating ontology workflows. *BMC Bioinform.*, 20(1):407:1–407:10, 2019. doi:10.1186/S12859-019-3002-3.

8 Yevgeny Kazakov, Markus Krötzsch, and Frantisek Simancik. The Incredible ELK - From Polynomial Procedures to Efficient Reasoning with $\mathcal{EL}$ ontologies. *J. Autom. Reason.*, 53(1):1–61, 2014. doi:10.1007/S10817-013-9296-3.

9 Jean-Baptiste Lamy. Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies. *Artificial Intelligence in Medicine*, 80:11–28, 2017. doi:10.1016/j.artmed.2017.07.002.

10 Phillip Lord. Horned-OWL. Software (visited on 2024-11-29). doi:10.4230/artifacts.22531.

11 Phillip Lord. The Semantic Web takes Wing: Programming Ontologies with Tawny-OWL, 2013. arXiv:1303.0213.

12 Phillip Lord, Martin Larralde Björn Gehrke, Janna Hastings, Filippo De Bortoli, James A. Overton, James P. Balhoff, and Jennifer Warrender. Horned-OWL. Software (visited on 2024-11-29). doi:10.4230/artifacts.22530.

13 Natalya F. Noy, Nigam H. Shah, Patricia L. Whetzel, Benjamin Dai, Michael Dorf, Nicholas Griffith, Clement Jonquet, Daniel L. Rubin, Margaret-Anne Storey, Christopher G. Chute, and Mark A. Musen. BioPortal: ontologies and integrated data resources at the click of a mouse. *Nucleic Acids Research*, 2009. doi:10.1093/nar/gkp440.

14 The Gene Ontology Consortium. The Gene Ontology Resource: 20 years and still GOing strong. *Nucleic Acids Research*, 47(D1):D330–D338, January 2019. doi:10.1093/nar/gky1055.