Portable Reasoning in Web Assembly

David Pojunas

December 2019

Abstract

David Pojunas, "Portable Reasoning in Web Assembly" Institut National Des Sciences Appliquées, Lyon, December 2019.

This paper demonstrates how to write Rust code for Web Assembly using the wasm-bindgen tool. We create a proof of concept allowing the Rust toolkit Sophia to be run on the web. Sophia is a crate that has a comprehensive toolkit for working with RDF and Linked Data in Rust. Using this crate, we show how it can be compiled for WebAssembly to then be ported for the web. We discuss the power and limitations of the wasm-bindgen client for this process. It allowed us to define Rust types to be used in JavaScript. The wasm-bindgen toolkit facilitated the translation. For Sophia to be fully useable in the web, we follow the JavaScript RDF interface for creating an implementation using Web Assembly as the medium between Sophia in Rust and the RDF/JS interface. In the end, we have a final code source that shows how it is it possible to implement the RDF/JS interface from Rust through Web Assembly to JavaScript. A full implementation of the interface would allow for the performance of Sophia to be prevalent, while not exposing the generosity and power of the Sophia toolkit.

Introduction

The goal of the semantic web is to use linked data in order to make any machine able to understand and interpret the internet. Every page on the internet is an HTML document that only tells a machine, raw information. The semantic web expands on the current web structure by adding new frameworks to make the internet more intelligible. It allows web pages to define semantics based on human-readable information. The machine can then interpret it to be more useful when searching and using the web. The semantic web is an extension of the World Wide Web through standards set by the World Wide Web Consortium. The W3C attempts to maintain common protocol and exchanges on the Web based on well known frameworks like RDF (Resource Description Framework). In this project, we followed the standards of RDF and did not use other implementations.

RDF is the standard model for all data on the Web. It supports the change of schemas over time since it is semantically structured. RDF uses IRIs instead of URLs. An IRI is an Internationalized Resource Identifier, and all URLs are IRIs but not all IRIs are URLs. The actual specifications of this structure can be found at the website [1]. This model is quite simple but powerful enough to allow structured and semi-semantic data to be used for the web. It does this through the use of a graph structure based on triples. Which are defined to have a subject, predicate, and object called an RDF triple. An example of a triple is shown below.

```
<http://schema.org/Person>
<http://www.w3.org/2002/07/owl\#equivalentClass>
<http://xmlns.com/foaf/0.1/Person>.
```

The rules about the semantics of each component of the triple allow for relations to build graphs. Any more information about RDF can be found here [1]. In this project, we follow this scheme of RDF very closely.

The scope and use of RDF on the web right now is quite small. Having the whole internet use the framework defined by W3C is no small task. As of right now, there is a small list of use cases for RDF [2]. In its growing state, lots of APIs comply with W3C in an attempt to build a real semantic web [3].

In this project, we followed a toolkit for RDF and Linked Data in Rust called Sophia written by professor Champin. In Sophia, each statement in RDF is represented as a triple which is made of three terms. A set of triples forms an RDF graph. Each graph can be grouped in a collection called a dataset, where each graph is identified by a unique name. With this specification, we can build full RDF graphs with Sophia. The data model supported by Sophia is a generalization of the strict RDF model defined by W3C.

Sophia leverages Rust's type system in order to generalize almost every aspect of the toolkit. To start creating graphs with Sophia you have a choice of different graphs to make, FastGraph, GenericGraph, and LightGraph. Each graph is well defined in the documentation [4], for building a graph we will use the inmem FastGraph. Below is an example of what creating a graph looks like in Rust.

```
let mut g = FastGraph::new();
let schema = Namespace::new("http://schema.org/").unwrap();
let s_name = schema.get("name").unwrap();
g.insert(&s_name, &rdf::type_, &rdf::Property);
g.insert(&s_name, &rdfs::range, &xsd::string);
```

This is the simplest way to build a graph in Sophia with triples. Not only does Sophia use triples but it also supports quads. A quad is just like a triple except with an optional graph name. Quads in Sophia follow the W3C standard found in the documentation [5]. With both triples and quads, we can build any dataset in Sophia. Sophia is extremely generous and allows for flexibility for building graphs, defining triples, creating terms and searching the graph.

Professor Champin has run benchmarks that show the efficiency and performance boost Rust allows when using Sophia. The details of the benchmark are found through his Github repository [6]. We can glean from the benchmark that Sophia has a linear load time for an NT file and the load rate is higher than any other RDF implementation tested. The rest of the benchmark gives more specific performance boosts for using Sophia. This performance is due to the fast execution time. Also, the Rust compiler is very good at building optimized code. Sophia was designed with this in mind and pushes Rust capabilities to its absolute limit. As the language evolves, Sophia will adapt as well to ensure the fast performance is achieved.

We know that Sophia is fast but RDF is made for the web. The goal of the project was to explore how to compile Sophia for WebAssembly to be used on the web. WebAssembly (WASM) is a binary instruction format for a stack-based virtual machine. This means that when using WebAssembly while in a browser, it is possible to get native performance on any task made possible by the binary format .wasm. WebAssembly is still currently being worked on and updated to support multiple languages for more compatibility over the whole internet [7]. There is one distinction that need to be made about how we used WebAssembly in this project. It is possible to compile a program into the binary format .wasm to then be run as an executable directly in the browser. This method was not used because Sophia is an API and is not intended to be used as an executable.

The other option for utilizing WebAssembly is through a Rust library toolkit called wasm-bindgen. Wasm-bindgen facilitates high-level interactions between wasm modules and JavaScript. This crate allows for the interaction and cross compilation from Rust to JS/wasm. There are many supported types for Rust to JS and JS to Rust through the wasm format. After compiling Rust code to a .wasm for JS, we can then run the Rust code as wasm through JS directly in the browser and even use the browser console to call functions and classes. We leveraged this tool to test if it is possible to make Sophia run on the web for fast performance and be used with the RDF/JS interface.

Before we could test Sophia with wasm, it was important to be able to use and understand the wasm-bindgen toolkit [8]. The steps it took to compile a simple rust program for JS/wasm is defined in the next section.

Rust to Wasm

To be able to compile Rust into the wasm format, install the wasm bindgen client to cargo in order to target the compilation for wasm. With Rust's target [9] cross compilation support, it is simple to directly build and execute code to the .wasm format. Below are the commands for installing and setting up the wasm client before ever creating a new project.

```
cargo +nightly install wasm-bindgen-cli
rustup target add wasm32-unknown-unknown
```

An important note, make sure that the wasm bindgen is installed to the cargo version used. For this project cargo nightly was used but it works without nightly as of now. After adding the client to cargo and rustup, a project can be made. Use a lib.rs file for writing Rust to wasm code. In order for this work it is essential to add to the Cargo.toml file the following.

```
[lib]
crate-type = ["cdylib"]

[dependencies]
wasm-bindgen = "0.2.55"
sophia = "0.3.0"
js-sys = "0.3.32"
```

After rebuilding with new dependencies, create a small Rust program. For examples of small functions see the examples here [10]. These examples were used in the beginning. And supported types will be discussed in a later section. After making a small lib.rs file use the following command for creating the wasm format.

```
cargo +nightly build --target wasm32-unknown-unknown
```

This will run and add a target folder in the project which contains a wasm folder where the actual wasm binary lives. However, this is just the binary format of wasm and the wasm-bindgen for cross compilation into JS/wasm has not been done. Since a bundler was not used when compiling, the wasm format will still need to be supported by our browser. No modules were used, which allows for running JS directly with the html document that's rendered to a webpage. The following command will build and add stitching code that is generated automatically. This is what wasm-bindgen was created to do, and be the intermediary between JS and wasm.

```
wasm-bindgen --target no-modules
target/wasm32-unknown-unknown/debug/wasm_example.wasm
--out-dir .
```

After all of these files are added. Add a package.json file for packaging the wasm and JS files to then be shipped to the browser. The configuration of this file is found in our GitHub repository. Finally, add a standard html

document to the project. Add the generated wasm.js file to the script, then run the Rust code through wasm. Below is an example of what the html document should look like inside the body.

```
<script src="your_project.js"></script>
<script>
async function run(){
    await wasm_bindgen('wasm_example_bg.wasm');
    console.log(wasm_bindgen.your_rust_function);
}
    run();
</script>
```

The async function was used here because the standard API for wasm is asynchronous. The easiest way to use this feature is to embed it directly to the html document. Once all of this is complete, spin up a quick web server to run the project. Any web server will work, in this project yarn was used, but it should not matter. Once the server is running, access the browsers terminal and check to ensure that everything is working. This was the simplest way to compile Rust to wasm and shipped to the web. Wasm-bindgen has a lot of specifications about its capabilities, which will be highlighted in the next section.

Rust to JavaScript

Wasm-bindgen is the glue allowing Rust to be used with JS and running wasm on the browser. The documentation has all the current updated support for the toolkit. However, it is constantly being updated and it's important to check the Github repository for issues and fixes. The main functionality of wasm-bindgen is to convert between JS and Rust both ways. It's command line tool has multiple options available to tweak the JavaScript generated. Allowing for control over how the Rust is being optimized for wasm. To have the Rust code be recognized, the *#[wasm_bindgen]* macro supports some configuration for controlling how imports and exports are handled. It can also change what the generated JavaScript will contain. This is used to convert the Rust code to wasm. It's very useful but has some limitations.

Overall, wasm-bindgen has good support for importing JS types to Rust. However, it has less support for exporting Rust to JS.

To use Sophia in wasm, we first attempted to expose Sophia directly to JS. With the limited types for exporting Rust to JS, we were able to load a graph from JS and extract triples. The purpose of the example was to see how Sophia worked with wasm-bindgen. Due to a lack of support we were only able to define simple functions. These allowed for the creation and manipulation of Sophia in memory graphs. In turn, the types used between Rust and JS were strings. Which are well supported by wasm-bindgen however most of the Sophia crate is completely generic. Wasm-bindgen can't directly interpret Sophia types and structs. Meaning this example was not scalable and would not fully expose the crate. We were not attempting to write any JS code, but allowing Sophia to be accessed through wasm by JS. This initial approach was too broad because of the complexity and generosity of the Sophia crate. There were too many structs, traits and types for wasm-bindgen to export. Clearly, exposing Sophia and unwrapping the architecture of the crate was not an option.

What we propose as the solution, to follow the RDF/JS: Data model specification [11], in a wasm-bindgen project. This specification is a low level interface representing the RDF data independent of a serialized format in a JavaScript environment. This interface defines all the RDF interfaces that need to be implemented for a full implementation. The specification gives us the types of every interface and functions to comply with this RDF/JS data model. Meaning we can write the implementation in Rust using wasm-bindgen to define all interfaces.

For example, the Term interfaces tells us which JS types are used and all functions that are implemented.

```
interface Term {
  attribute string termType;
  attribute string value;
  boolean equals(optional Term? other);
};
```

Since wasm-bindgen can import JS strings we can define this exact interface with Rust. Using the wasm-bindgen macro, we can represent this is exact term with Sophia as the correct Rust type: Term<Rc<str>>. Sophia ac-

cepts this type and will return a string if needed. We can't define a generic term because of wasm-bindgens limitations. If we are direct and use a string for all Sophia types, writing these interfaces will be simple. This means we can expose Sophia to JS with wasm by creating a wasm-bindgen representations of Sophia to be used as this JS interface. Allowing for a total use of Sophia which does not strip the toolkit from its strengths. However, not everything that the JS/RDF interface implements is also in Sophia. These are not major differences and can be worked around on a case by case basis. These are caused by differences in the implementation strategies. Which will require either new code to be added to Sophia, or omitted from the interface implementation. For example, Sophia does have a notion of a Default-Graph, but quads from the Default-Graph have no graph name defined by the Term. This is caused by the specification RDF-JS, where quads from the Default-Graph have a graph name which is a specific variant of the Js Term interface. This is an interface in the specification, but there is no clear solution on how to implement it one to one to Sophia. With this in mind, the major components are translatable for basic RDF graph functionality.

Final Proof of Concept

We have created a proof of concept of how an implementation of RDF might be drafted. The JS/RDF specification was not considered in this example code. However, it does give an attempt at how the wasm-bindgen code in Rust would look like for implementing the structs and traits. Sophia can dictate how the interface is implemented as long as the contract of the interface is maintained. Some complications arise with iterable types between JS and Rust, as well as dealing with lifetimes for Rust objects used by JS. In our documentation we were able to return an iterator from Rust to JS. This does not follow the JS/RDF specification specifically. Which leaves us with the issue of lifetimes. If the interface is followed directly, it's possible for a defined lifetime by Sophia to cause issues for the wasm compilation. No details are known and this is only a warning for using types with lifetimes in the wasm-bindgen code. This could be avoided by not exposing these structures directly to wasm-bindgen. Overall, with this first specification it should be completely possible to expose Sophia to JS in order to wasm to maintain efficiency and performance.

The JS/RDF specification used, only defines the data model for RDF. There are other specifications that JS has and could be implemented. However, these start to stray away from how Sophia is implemented so these interfaces were out of the scope in this project. In it current state nothing has been implemented for the JS/RDF interface. In the example code, it shows how the implementation could be done. It is certain that using the API allows for the use of Sophia directly to wasm. All of these components need to be put together to have a working version of the JS/RDF data model interface that is only using Sophia. Once that is completed then Sophia will be fully functional in the browser.

References

[1] Graham Klyne, Jeremy J. Carroll, Brian McBride. *RDF 1.1 Concepts and Abstract Syntax*. (MIT, ERCIM, Keio, Beihang), 2004-2014 W3C®, *https://www.w3.org/TR/rdf11-concepts/#referents*

[2] Thomas Baker, Natasha Noy, Ralph Swick, Ivan Herman. *Semantic Web Case Studies and Use Cases*. (MIT, ERCIM, Keio), 1994-2012 W3C®, *https://www.w3.org/2001/sw/sweo/public/UseCases/*

[3] Google, Microsoft, Yahoo and Yandex. For more information see Github, *https://github.com/schemaorg/schemaorg*. Source cite, *http://schema.org/Person*

[4] Pierre-Antoine Champin. *https://docs.rs/sophia/0.3.0/sophia/*

[5] Gavin Carothers, Lex Machina, Inc. *RDF 1.1 N-Quads, A line-based syntax for RDF datasets*. (MIT, ERCIM, Keio, Beihang), 2012-2014 W3C®, *https://www.w3.org/TR/n-quads/*

[6] Pierre-Antoine Champin. *https://github.com/pchampin/sophia_benchmark/blob/master/benchmark_results.ipynb*

[7] Mozilla. *Web Assembly*. Mozilla and individual contributors, 2005-2020, *https://webassembly.org*

[8] Rust Wasm Team. *https://rustwasm.github.io/docs/wasm-bindgen/*

[9] Rustc, *The rustc book. https://doc.rust-lang.org/rustc/targets/index.html*

[10] Rust Wasm Team. *https://rustwasm.github.io/docs/wasm-bindgen/examples/index.html*

[11] Thomas Bergwinkl, Michael Luggen, elf Pavlik,Blake Regalia, Piero Savastano, Ruben Verborgh. *RDF/JS: Data model specification*. 2019, *https://rdf.js.org/data-model-spec/#dom-term-termtype*