University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Artur Jamro
Student no. 370953

Jarosław Ławnicki
Student no. 371162

Igor Matuszewski
Student no. 334777

Marcin Mielniczuk
Student no. 373035

Implementation of safe JavaScript bindings in Servo

Bachelor's thesis
in COMPUTER SCIENCE

Supervisor:
Robert Dąbrowski, PhD
Institute of Informatics

June 2018

Supervisor's statement

Hereby I confirm that the presented thesis was prepared under my supervision and that it fulfils the requirements for the degree of Bachelor of Computer Science.

Date Supervisor's signature

Authors' statements

Hereby I declare that the presented thesis was prepared by me and none of its contents was obtained by means that are against the law.

The thesis has never before been a subject of any procedure of obtaining an academic degree.

Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date Authors' signatures

Abstract

Integrating big codebases is very often a non-trivial task, especially when they are written in languages having different semantics. Here, we explore integration of the SpiderMonkey JavaScript runtime, written in C++, inside the Servo web engine, written in Rust. Due to the Rust type system expresiveness, many implicit invariants and guarantees required by the SpiderMonkey interface were able to be encoded in the Rust type system, resulting in an idiomatic API that is both ergonomic and safe, thanks to static analysis performed by the Rust compiler. Additionally, a compiler plugin providing static analysis has been implemented to detect potential misuse of the API for cases that could not be encoded in the type system.

Keywords

Rust, Servo, JavaScript, SpiderMonkey, borrow checking, static analysis, code generation

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

Software and its engineering — Software organization and properties — Software functional properties — Formal methods — Automated static analysis

Thesis title in Polish

Implementacja bezpiecznych wiązań do JavaScript w Servo

Contents

1. Introduction . . . 5
- 1.1. Servo . . . 5
- 1.2. SpiderMonkey . . . 5
- 1.3. Scope of the work . . . 6

2. Rust . . . 7
- 2.1. Lifetimes . . . 7
- 2.2. Borrow checker . . . 9
- 2.3. Macros . . . 9
- 2.4. Attributes . . . 10

3. Architecture overview . . . 11
- 3.1. Servo . . . 11
- 3.2. Codegen . . . 11
- 3.3. Compiler and compiler plugins . . . 12
  - 3.3.1. Compiler plugins . . . 12
  - 3.3.2. Compilation process overview . . . 13

4. Implemented checks and improvements . . . 15
- 4.1. Rooting . . . 15
  - 4.1.1. CustomAutoRooter . . . 16
  - 4.1.2. Adding lifetimes to handles . . . 19
- 4.2. Heap references . . . 20
  - 4.2.1. `Heap::mut_handle` fiasco . . . 21
  - 4.2.2. Unsafe `Heap::new` constructor . . . 21
- 4.3. WebIDL wrapper types . . . 23
  - 4.3.1. `TypedArray` types . . . 23
  - 4.3.2. `Record` types . . . 25
- 4.4. Compiler plugin . . . 26
  - 4.4.1. How the plugin works . . . 26
  - 4.4.2. Improvements . . . 27
  - 4.4.3. Problems . . . 28
- 4.5. Other minor improvements . . . 29
  - 4.5.1. Capturing JS stack traces . . . 29
  - 4.5.2. Build system improvements . . . 29

5. Development organization 31
5.1. Workflow 31
5.2. Team members' contribution 31
5.2.1. Artur Jamro 31
5.2.2. Jarosław Ławnicki 31
5.2.3. Igor Matuszewski 31
5.2.4. Marcin Mielniczuk 31

6. Summary 33

Bibliography 35

Chapter 1

Introduction

1.1. Servo

Servo is a research project of a next generation web engine written from scratch using Rust programming language, both of which are funded by Mozilla Research. One of the project goals is to gradually replace browser components used in Firefox one at a time.

Starting from November 2017 both share parallel CSS layout engine, Quantum CSS (previously known as Stylo), originally written for Servo in Rust. As benchmarks show, the new engine managed to speed up page rendering in Firefox on average by 30% or even as much as 80% in some cases in comparison to the previous CSS engine used [21]. With benchmarks confirming real benefits coming from using the new components rewritten in Rust, Servo team now aims to bring yet another one, WebRender, which is a hardware-accelerated rendering engine written in Rust [22], to Firefox in 2018.

1.2. SpiderMonkey

SpiderMonkey is a JavaScript engine developed in C++ by Mozilla Foundation. [52] It is used in various Mozilla products, including Servo as well as Gecko, the web browser engine currently used in Firefox.

Communication between a web browser engine and the JavaScript runtime is bidirectional. The runtime is responsible for evaluating the JavaScript code provided by the web engine and, on the other hand, the web engine implements and exposes a common, standarized web API, called the DOM interface, used by the JS scripts to interact with the browser. [30] A slightly more detailed overview of the communication between both engines is illustrated by figure 1.1.

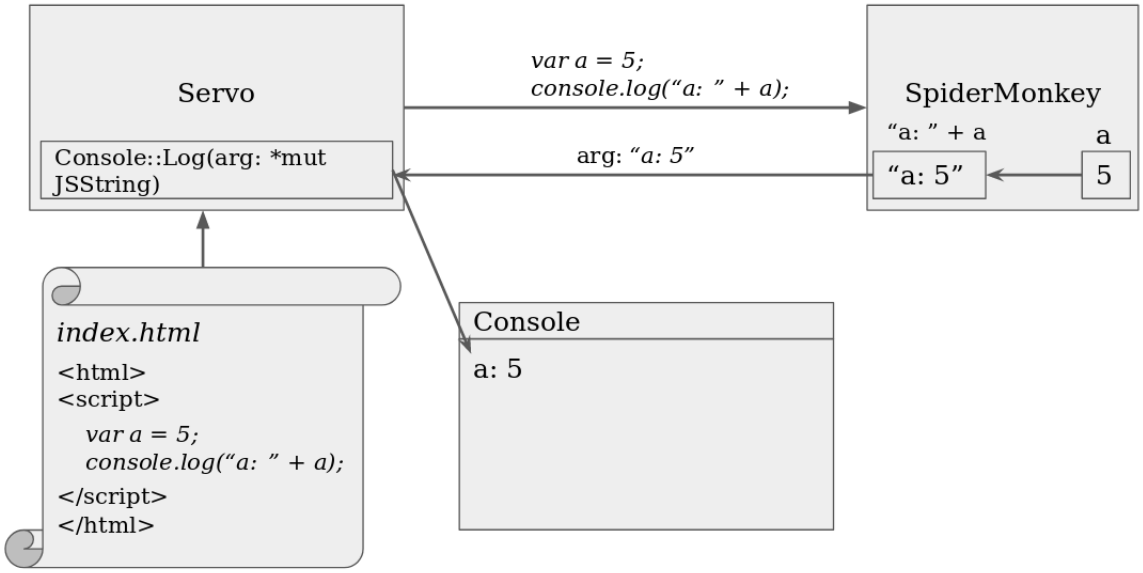

Figure 1.1: Interpretation of a simple JavaScript script. First, Servo extracts the JavaScript code from incoming HTML file and passes it to SpiderMonkey. SpiderMonkey interprets the command, computes the function argument and calls Servo back to write some text to the browser's console.

The main challenge of the Servo - SpiderMonkey integration is ensuring that SpiderMonkey is aware that certain JavaScript objects are needed by Servo and should not be deleted during garbage collection.

1.3. Scope of the work

Before the group work began, Servo already had a partially implemented API, encapsulating communication with the SpiderMonkey JavaScript engine. However, the API was mostly a straightforward reimplementation of the interfacing code used by the Gecko engine, written in C++. Therefore, the above-mentioned wrappers did not use special features of the Rust programming language (described at length in chapter 2). The aim of the project was to finish the implementation of the Servo – SpiderMonkey interface and rewrite the existing code to use features of the Rust language. More specifically, the work focused on the following 4 areas:

1. Ensuring that JavaScript objects will not be deleted by SpiderMonkey's garbage collector mechanism while Servo is using them.
2. Updating references to JavaScript objects being moved by SpiderMonkey's GC.
3. Developing a Rust compiler plugin to statically verify during compilation if the API is used correctly.
4. Adapting any existing code to use the new API.

Chapter 2

Rust

Given the fact that there are many well-established and battle-tested programming languages in the industry, what benefits should another language have to be considered when starting such big of a project, that is a modern web browser engine? This chapter will try to answer that question.

Rust is a relatively young language. It dates back to 2006, when Graydon Hoare, a Mozilla employee, started it as a personal project. The company began sponsoring it in 2009 but it took quite a long time for the language to stabilize. The 1.0 version was released in 2015. [1]

Rust focuses on zero-cost abstractions, memory safety, fearless concurrency and takes a lot of inspiration from functional languages, all of this without employment of a garbage collector. It is compiled to native code and aims to be a high-performance language with easy and safe parallelism. [2] [3] It is designed to catch and prevent most common programming pitfalls during compile time by using novel compiler mechanisms. The following sections will be devoted to some of them.

2.1. Lifetimes

In languages where references may be stored in data structures, one of the most prevalent memory safety violations is caused by dangling references, which is often called a *use after free* problem. In languages such as C or C++, where the memory management is manual, it can lead to many serious bugs, including security vulerabilities. While the introduction of smart pointers has largely alleviated the issue, there are still many cases when invalid reference access is possible, e.g. due to iterator invalidation or accidentally returning a pointer to a stack variable.

At first, this looks like a problem that cannot be solved statically, e.g. during compilation, but thankfully, this is not the case. The Rust language introduces a concept of lifetimes in order to guarantee that the references accessed by the program will always be valid. Here is an example taken from the official Rust book. [4]

```
{
        let r;

        {
                let x = 5;  // create a stack variable ‘x‘
                r = &x;  // take a reference
        }
```

```
        println!("r: {}", r);
}
```

An attempt at compiling this snippet results in a very self-explanatory error:

```
error: ‘x‘ does not live long enough
   |
6  |          r = &x;
   |                - borrow occurs here
7  |      }
   |      ^ ‘x‘ dropped here while still borrowed
...
10 | }
   | - borrowed value needs to live until here
```

In most cases, the compiler is capable of automatically inferring appropriate lifetimes associated with given references. Otherwise, these need to be specified manually, for example:

```
fn foo<'a>(x: &'a str, y: &'a str) -> &'a str { ... }
```

The `&'a` notation represent a reference with an associated lifetime named `'a`. In the example above, this means that both input and output references to type `str` must be valid across the same lifetime `'a`.

Relation to the type system and polymorphism

Any type can be also further parametrized by its associated lifetime, for instance:

```
struct Foo<'a, T> {
    x: &'a T
}
```

This is a structure declaration that is *generic* over a type parameter `T` and lifetime `'a`.

Additionally, the language also supports ad-hoc polymorphism [34], expressed via the *trait* system. These *traits* can also be compared to constructs found in different languages - most notably to Haskell's type classes, which served as an inspiration for the current system, but also to Scala traits, Java interfaces, C++ pure abstract classes and similar. [54] [56] [55] [35]

With both polymorphism systems combined it is possible to have a *generic trait* implementation, as long as certain type requirements, called *trait bounds*, are met. Here is a simplified implementation of the `PartialEq` *trait* for a *generic* type `Foo<'a, T>`.

```
trait PartialEq { ... }

impl<'a, T> PartialEq for Foo<'a, T> where T: 'a + PartialEq {
    fn eq(&self, other: Foo<'a, T>) -> bool {
        *self.x == *other.x
    }
}
```

Because the *trait bound* `T: 'a + PartialEq` is specified, the trait is implemented only for a `Foo<'a, T>` type whose type parameter `T` is valid for as long as lifetime `'a` and for which the `PartialEq` trait is implemented, as well.[53]

A major takeaway is that *traits* are easily composable across (possibly nested) *generic* types, including lifetimes associated with them.

2.2. Borrow checker

References, and sharing data in general, turn out to be very tricky to get right when writing concurrent code - it is really easy to have race conditions. When directly writing to memory using two separate references, each pointing to the same memory location, the order of writes and the result is undefined. The Rust language extends the concept of lifetimes to ensure that either a single *mutable* reference, through which data modification can occur, or multiple immutable (read-only) references can exist in a given scope. This prevents *aliasing* (accessing data for modification through many references) during compilation, so that no race condition may occur in safe Rust code.

2.3. Macros

C and C++ are well-known for their macros. They give the programmer enormous possibilities to extend the syntax of the language, allowing home-grown extensions to the language syntax. A notable example is the Qt toolkit, which uses macros to represent the Qt concepts in a concise way (actually, the macros produce metacode to be further parsed by a source-to-source utility called `moc`).

On the other hand, the C-family macros are notorious for their possible misuse. Since the C macros work by performing a simple string substitution, they often lead to unexpected bugs. The class of macro-induced problems is broad. A common mistake is passing an expression with side effects to be passed to the macro, which causes the side effects to be evaluated twice. Another common issue is the occurrence of an unexpected order of evaluation.

Most post-C languages have this decided not to allow any kind of macros. Rust took a different path - instead of disallowing this feature, the language developers have enhanced it and ensured they would not lead to pitfalls known from C.

The Rust macros operate at a much later stage. Instead of copy-pasting the invocation arguments into the raw source, they directly insert nodes into the AST. More precisely, the Rust macros accept token trees and output AST nodes. All syntax extensions invocations are marked with a `!` symbol. [5] This prevents from macro abuse and consequent pitfalls known from C or C++.

Macros are extensively used across the standard library and popular crates, with notable examples:

1. the `println` macro combines the sleekness of the C `printf` function with static verification of the argument correction and static type inference
2. the `try` macro introduces monadic error handling (and is so widespread that it even has a syntactic sugar in the language)
3. the `lazy_static` macro allows definition of static variables, evaluated in a lazy way

The power of macros can be demonstrated by this example, which extends the syntax to allow concise definitions of HTML trees. [6]

```
macro_rules! write_html {
    ($w:expr, ) => (());

    ($w:expr, $e:tt) => (write!($w, "{}", $e));

    ($w:expr, $tag:ident [ $($inner:tt)* ] $($rest:tt)*) => {{
```

```
        write!($w, "<{}>", stringify!($tag));
        write_html!($w, $($inner)*);
        write!($w, "</{}>", stringify!($tag));
        write_html!($w, $($rest)*);
    }};
}

fn main() {
    use std::fmt::Write;
    let mut out = String::new();

    write_html!(&mut out,
        html[
            head[title["Macros guide"]]
            body[h1["Macros are the best!"]]
        ]);

    assert_eq!(out,
        "<html><head><title>Macros guide</title></head>\
         <body><h1>Macros are the best!</h1></body></html>");
}
```

During our Bachelor's project we also used the macros mechanism to generate over 300 wrappers for existing functions. It is described in more detail in section 4.1.2.

2.4. Attributes

In Rust it is possible to attach some extra information to nodes in AST (abstract syntax tree [17]), such as function definitions or type parameters. These pieces are called *attributes* and are utilized by the compiler for specific purposes. Here is an example for conditional compilation:

```
fn main() {
    #[test]
    println!("If you see this, then the code \
              was compiled in the test mode!");
    #[cfg(not(target_os = "linux"))]
    println!("This print will be missed on Linux OS.");
}
```

The `println!` macro invocation labeled with `#[test]` will be omitted if the code is not compiled as a test. The second attribute in the example excludes corresponding `println!` if the code is compiled on a Linux-based OS.

The attributes can also be applied to functions and even generic parameters such as below:

```
#[test]
fn test_me<#[must_root] T>() {
  // ...
}
```

The `#[must_root]` attribute is specific for Servo and is described later in section 4.4.

Chapter 3

Architecture overview

3.1. Servo

Major development of the Servo project and Rust can be traced back to around 2012. Since then, both have evolved considerably, one having influenced the other. Rather than a typical, modern Rust project layout that is generated by the Rust package manager, Servo employs a project layout that is more similar to the one used for Firefox.

Mainly, it consists of a top-level `components/` directory that contains essential components, mostly written in, but not limited to, Rust. In addition to that, it also includes a plethora of Python scripts that allow easier integration with external infrastructure, such as web platform tests (WPT), which are shared between popular browser vendors, or even its own `mach` build system. While the project somewhat leans towards the *monorepo* layout that is popular among big IT projects such as Firefox [23] itself or even Google organization as a whole, it does also tries to leverage Rust package ecosystem and registry - crates.io. Many libraries that are shared by both Firefox and Servo are repackaged and published there as Rust packages (called *crates*). Even the whole rendering subsystem to be later included in Firefox, WebRender [22], is distributed solely as a Rust *crate*.

Main area of interest to our group was the `script` component, which implements the DOM interface [30], and how it interacts with SpiderMonkey, the currently used JavaScript runtime. The DOM interface defines every function that is exposed to and can be called from JavaScript, which means that the `script` component can be effectively considered as the heart of interactive capabilities of the Servo web engine.

As previously mentioned, the engine also shares a JavaScript runtime with Firefox - SpiderMonkey. To reuse it in Rust environment, the entire engine is repackaged and published as the `mozjs-sys` *crate*, adopting the widely used convention in the Rust community to use `*-sys` packages for native (non-Rust) dependencies. To facilitate exposing the original C++ interface to Rust, a tool called `rust-bindgen` [31] is used - it can automatically generate C/C++-compatible data definitions, in addition to exposing and linking appropriate functions, to be later directly used in Rust. In turn, these raw bindings are used by the `rust-mozjs` package, which aims to provide safe Rust bindings to the SpiderMonkey engine.

3.2. Codegen

The Servo as well as other web engines use WebIDL, which is the interface description language used to describe Web application programming interface i.e. data types, interfaces, methods, properties, and other components. [15][16] WebIDL uses stylized syntax independent of any

specific programming language. The advantage of using such an abstract description is that web browser vendors can ensure that the browsers are compatible with each other.

The potential drawback of such abstract definition of interface is the tendency of description and actual interface to diverge. In order to make sure that Servo API is compatible with provided description the actual declaration of numerous methods and data types in Servo are generated automatically from the WebIDL files in conjunction with Rust implementation. The actual code generation is handled by Python program, henceforth called Codegen, which parses the WebIDL file and Rust implementation and produces corresponding output Rust file.

Apart from ensuring the compatibility between interface description and actual implementation the Codegen serves another purpose - it generates the boilerplate code needed at the boundary of Servo/SpiderMonkey interface. This involves converting the parameters of methods from SpiderMonkey's types to Rust wrappers defined in other parts of Servo and unwrapping the results of functions back to corresponding SpiderMonkey types, as well.

The work done included modifying the Codegen to add safety mechanisms related to SpiderMonkey's rooting mechanism (described in 4.1) or to use newly implemented or improved wrapper types.

3.3. Compiler and compiler plugins

Apart from the work on Servo, some changes to the compiler were also required. Designers of the language wanted to have an extensible compiler and it has been achieved by introducing plugins. The Rust compiler, `rustc`, is written in the Rust language. It makes implementation of compiler plugins much easier and well integrated with Rust ecosystem. Compiler plugins usually reside within their own crates, being isolated from project logic, and are easy to manage via the Rust package manager.

3.3.1. Compiler plugins

To use a plugin, the compiler has to compile and load it first, before compiling the actual source code. The `rustc` compiler allows, among others, to introduce syntax extensions, lints and macros that are called *during compilation*, which means the call results are directly stored in the compiled code instead of repeatedly computing it at run time. [8] [9] Here are some notable examples:

1. DOM API is described using inheritance, but the Rust language does not support it, so Servo developers have implemented it explicitly and have written lint pass that checks if inheritance is implemented correctly. [10]

2. Rocket is a web framework written in Rust. [11] It utilizes compiler plugins to generate boilerplate code [12], so that developers can focus on writing actual logic with minimal amount of code without sacrificing any guarantees provided by the language.

3. Clippy is a collection of lints to catch common mistakes. Anyone can run these lints against their code and improve it. [13]

In order to understand how compiler behaviour can be extended, it may be helpful to first understand how the compilation process looks like.

3.3.2. Compilation process overview

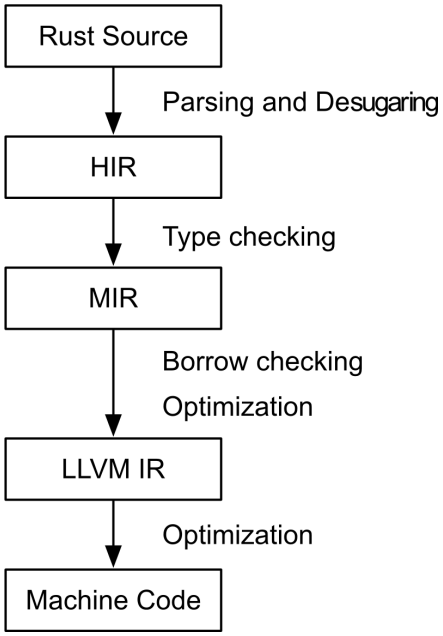

Figure 3.1: Compilation process overview. Image comes from [18].

Subsequent stages of compilation are presented in figure 3.1. As a first stage to most of the compilers, the source code is initially parsed into the *abstract syntax tree* (AST) representation. [17] Then, it is processed and lowered into two Rust-specific intermediate representations, first of them called high-level (HIR) and the second one called middle-level (MIR). The HIR is obtained by desugaring (normalizing convenience syntax) input AST representation, after which the type checking is performed by the compiler. During subsequent lowering to MIR, any compound expressions are broken down down to single assignments, after which the borrow-checking is performed and the representation is further optimized (see also [18]).

Since the `rustc` compiler uses LLVM (widely used, reusable compiler toolchain [14]) itself to generate machine code at the end of the compilation pipeline, the optimized MIR is lowered into an intermediate representation (LLVM IR) to be used by LLVM, and passed down to it. Finally, as the last compilation step, LLVM compiles the intermediate representation into machine code, specific for the targeted platform.

Chapter 4

Implemented checks and improvements

4.1. Rooting

In constrast to Rust, JavaScript is a dynamic language that needs a garbage collector to maintain reasonable performance. Because the lifetime of JavaScript objects are managed separately in SpiderMonkey, to safely use them inside of Servo we need to provide a guarantee that the JavaScript object will not be freed while it is being used in Servo. To provide that, a notion of *rooting* is introduced.

The *rooting* refers to garbage collection terminology. The algorithm used in SpiderMonkey is a mark-sweep GC. To analyze which objects are not used anymore and can be safely collected, the algorithm recursively descends and marks any reachable objects, starting from top-level nodes called *roots*. After marking is finished, the objects that were unreachable are collected. By adding a JavaScript object to the *root* set we can guarantee that it will not be collected while it remains in the set. This means that references to JS objects can be safely handed out to external systems, like Servo, and correct access to aforementioned objects can be guaranteed while they remain in the *root* set.

For certain objects that are managed by the garbage collector, the SpiderMonkey engine distinguishes certain types that are referred to as *GC things*. These can represent different types that are used inside of a JavaScript runtime. For example, `string` or `object` JavaScript types are represented respectively by the `JSString` and `JSObject` types in C++. For FFI (foreign function interface) purposes, SpiderMonkey exports these types to C interface and uses raw pointers to them (e.g. `JSObject*`).

To root GC-managed objects, SpiderMonkey internally uses a generic `Rooted<T>` wrapper type to be used with *GC thing* pointers (like `Rooted<JSObject*>`). It is also exposed in an FFI for every possible associated *GC thing* pointer type. Internally, it adds a pointer to the wrapped *GC thing* pointer to the *root* set on construction and removes it on destruction (using C++ automatic resource management technique called RAII [25]). This also ensures that SpiderMonkey knows about every rooted object in memory, but also implies that the rooted objects must not be moved, so that the stored pointer to it remains valid while it is in the *root* set.

However, it does not cover one important use case - when *GC things* are used as a part of other aggregate. A complete guide to rooting and the governing rules regarding rooting in SpiderMonkey, which we referred to while working on the project, can be found in [24].

4.1.1. CustomAutoRooter

To root *GC things* that are a part of an aggregate, an another type is introduced - `CustomAutoRooter`. It uses a similar mechanism to `Rooted<T>`, where it adds and removes a pointer to the rooted object to the *root* set. However, an aggregate can be of any shape, so the marking of which objects are reachable (also called *tracing*, hence why the algorithm is sometimes called a *tracing GC*) is implemented using a technique called a *visitor pattern*.

In SpiderMonkey it is implemented using multiple inheritance and virtual dispatch in C++. Every class that is to be used with `CustomAutoRooter` infrastructure needs to derive from the `CustomAutoRooter` class (which also derives from helper base `AutoGCRooter` class, which we will not expand upon) and implement the virtual `void trace(JSTracer*)` method. Then, during the GC object marking phase, an appropriate method is called on every rooted object, which is responsible to trace every inner, contained *GC thing*.

However, while the mechanism is very flexible, due to C++ semantics used, such as (delegate) polymorphic object constructor guarantees and implicit dynamic dispatch mechanism, it is hard to expose it directly to a C FFI. In general, Rust also does not support C++ style object-oriented programming paradigm at the language level, which means that the necessary features used had to be emulated manually when implementing SpiderMonkey's `CustomAutoRooter` infrastructure support within Servo.

First bit that needed to be taken care of, was C++ dynamic dispatch using virtual function tables (vftables). As mentioned before, `rust-bindgen` generates C++-compatible data definitions in Rust, including proper alignment and vftable types. Remaining piece of work was to emulate the C++ dynamic dispatch mechanism in Rust and write linking glue code. To do that, we used Rust features called *traits*, which can be considered similar to Java *interfaces*, as they are meant to define a set of available operations for a given interface implementer. More specifically, we defined a `CustomAutoTraceable` trait, which also contains a static vftable, akin to how C++ implements it, and necessary glue code:

```
unsafe trait CustomAutoTraceable: Sized {
    const vftable: CustomAutoRooterVFTable =
        CustomAutoRooterVFTable {
            trace: Self::trace,
        };

    unsafe extern "C" fn trace(
        this: *mut ::std::os::raw::c_void,
        trc: *mut JSTracer
    ) {
        let this = this as *const Self;
        let this = this.as_ref().unwrap();
        Self::do_trace(this, trc);
    }

    /// Corresponds to virtual `trace` call
    /// in a `CustomAutoRooter` subclass (C++).
    fn do_trace(&self, trc: *mut JSTracer);
}
```

Thanks to how well Rust *traits* (which happen to be inspired by Haskell's *typeclasses*) compose, we can write:

```
unsafe impl<T> CustomAutoTraceable for CustomAutoRooter<T>
where T: CustomTrace {
    fn do_trace(&self, trc: *mut JSTracer) {
        // call trace on a CustomAutoRooter `data` member
        self.data.trace(trc);
    }
}
```

With this, the compiler will allow us to use any generic instance of `CustomAutoRooter<T>`, as long as the type `T` implements our `CustomTrace` trait. One benefit is that due to composability we are not required to write numerous different C++ templates (e.g. Gecko requires 12 different ones only to trace a `Vec<T>` type, where `T` is traceable), which simplifies the code and makes it easier to maintain.

Furthermore, the compiler will also enforce that the generic type `T` implements the trait we desire. This is in contrast to what generic programming in C++ permits, which only checks whether a type has a function with a compatible signature (same name, compatible arguments). Existence of such functions for a certain type does not mean that they can be used safely with certain traits. In this case, only allowing types implementing our `CustomTrace` trace provides us with additional compile-time guarantees, effectively disallowing any misuse of our `CustomAutoRooter` type.

Second bit that needed addressing was that the objects registered in the *root* set cannot be moved during that time, not to invalidate pointers stored inside SpiderMonkey to the rooted objects. At the time of writing, Rust language does not support *immovable* types, neither at language or standard library level (although an RFC [26] supporting that is being worked on). To work around that - we used Rust features - *ownership semantics* and *lifetimes* (along with RAII [25], a feature it shares with C++).

We modeled our type after another that is a part of the standard library - `Mutex<T>`. It acts as a wrapper type that is meant to *own* wrapped data of type `T`, which can only by accessed using an associated type `MutexGuard<'a, T>`. This is a helper type, which is constructed by calling `Mutex::lock()` function, that is only valid for *lifetime* `'a` and can only access underlying data during that time (after which it releases the corresponding mutex lock on `MutexGuard` destruction using RAII).

Analogous to `Mutex<T>`, our `CustomAutoRooter<T>` is meant to be used as a wrapper that owns the data, while we use an additional helper type `CustomAutoRooterGuard<'a, T>` to provide required *immovable* guarantees about our rooted object. Our guard object is defined as follows:

```
pub struct CustomAutoRooterGuard<'a, T: 'a + CustomTrace> {
    rooter: &'a mut CustomAutoRooter<T>
}
```

As before, we only allow to use this with a `T` type that implements `CustomTrace` trait. Internally, this type *borrows mutably* an instance of `CustomAutoRooter`. While a value is *borrowed*, the compiler checks at compile-time that it will not be moved, hence providing us with the immovable guarantee we originally required. Additionally, Rust also prevents *aliasing*, which means that the compiler allows for *mutable borrows* only when it can prove that the value cannot be accessed from two different places. This also means that our `CustomAutoRooterGuard` object has an exclusive access to the rooted data for the specified *lifetime* `'a`. This is important because the compiler guarantees that the same object cannot be put in the *root* set twice, nor can user misuse the API in any way, leading to an inconsistent internal state.

The end result is a pretty ergonomic API, which can be used in a following way:

```
// Traits below are implemented in rust-mozjs crate:

// Implementing Deref<Target=T> trait means that
// the type can also be treated as of type T
impl<'a, T: 'a + CustomTrace> Deref<Target=T>
    for CustomAutoRooterGuard<''a, T> { ... }

unsafe impl CustomTrace for *mut JSObject { ... }
// If we know how to trace type T, we know how to trace Vec<T>
unsafe impl CustomTrace for Vec<T> where T: CustomTrace {
    fn trace(&self, trc: *mut JSTracer) {
        for elem in self {
            elem.trace(trc);
        }
    }
}

// How user can use the API:

let my_vec: Vec<*mut JSObject> = ...;
// Now rootable owns data inside of my_vec
let mut rootable = CustomAutoRooter::new(my_vec);
{
    // rooted is of type
    // CustomAutoRooterGuard<'a, Vec<*mut JSObject>>
    let rooted = rootable.root();
    // Because we mutably borrow rootable for the entire
    // scope, we are guaranteed that it won't be moved,
    // which in turn also guarantees that SpiderMonkey
    // won't collect and invalidate inner JSObjects

    // let rooted2 = rootable.root();
    //  ^^^ - We can't misuse the API; compiler disallows it,
    // since this would create a second mutable borrow of rootable

    for obj in &rooted {
        // Despite rooted being of a different (wrapper) type,
        // due to the fact that it implements a Deref trait
        // we can treat &rooted as if it were of an actual
        // Vec<*mut JSObject> type
        do_sth_with_raw_pointer(obj);
    }

    // mem::drop(rooted); <- compiler automatically injects
    // correct drop statements (equivalent to C++ destructor calls),
    // thanks to Rust's RAII feature
}
```

Implementing this safe API enabled us to implement other, previously lacking, bindings, which

we will explore later in the following subsections.

4.1.2. Adding lifetimes to handles

The handles in SpiderMonkey

The rooting mechanism was solely introduced to be able to safely access and modify the JavaScript variables. The main mechanism serving this purpose are handles - more precisely `Handle`s and `MutableHandle`s. [29]

These structures are actually wrapper references (or pointers) to rooted instances. They have their origins in SpiderMonkey and were introduced around 2012 as a part of a major GC refactor. [36] It was needed by the C++ implementation of the JavaScript engine and made its way to the Rust bindings in 2015. [38] A question naturally arises when one makes first change to the handle mechanism - why not use the native pointer/reference mechanism?

The main reason is flexibility.

Handles abstract from the intricacies of the type hierarchy in C++ or the trait hierarchy in Rust. Usually we take a handle to an instance of `Rooted`, but this is not always the case. Furthermore, the `Rooted` type was initially not present and the relevant type was named `Root`. One could call for using inheritance, but in fact the common base did not exist from the start either. The handle API survived all the changes in the code structure without major changes, and as a consequence - all the code using it avoided numerous refactors.

Rusty handles

The whole `mozjs` API, and consequently the `script` crate, make extensive use of handles. Due to the complexity involved, initially the handles used throughout Servo were the ones generated by rust-bindgen. Therefore, they took no advantage of the novel security mechanism in Rust as they were bit by bit identical to the C++ types.

The need for lifetimes has been raised as soon as the bindings were introduced to `rust-mozjs`. [37] The Servo developers quickly attempted to fix this defect, first with a compiler bug getting in the way, later encountering problems on the Servo side. [37] [40]

The issue was left untackled for a longer time. As the core developers have stated, it was even unclear if the problem can be solved in the current design of the Servo architecture. They have stressed, that even such outcome would be extremely valuable to Servo. This made the issue a proper research project. [41]

Implementation and challenges

The change consisted of two parts. One of them was to improve the `rust-mozjs` bindings, the other - to adapt Servo to take advantage of the modified API. The latter appeared to be a real challenge.

The `rust-mozjs` part still required care - one could not just blindly add lifetimes wherever a *missing lifetime* error was raised. In case of the Servo side, the complexity of the needed change was substantial. The handle API is used almost everywhere in the `script` crate - the biggest one in the whole `components` subdirectory. So widespread was their use, that we had to migrate entire call patterns instead of fixing single compilation errors.

Still, it was not an automatic job. Some of the variables and, in particular, functions are passed back to `mozjs`, which does not know about the new handles and needs to be given a raw C++ handle. While the new handles support conversion to the raw handles, it needs to be done by an explicit call of the `into()` method. Due to the number of existing mozjs API

calls it was impossible to replace all of them. The solution was to create wrappers taking care of all the needed conversions.

It was impossible to write these by hand, too. SpiderMonkey exposes over 300 functions which accept handles. Such wrappers needed to be generated automatically. Here, we took advantage of the powerful macro system Rust provides.

First, we used a simple script, making heavy use of `sed` and `grep` to extract the function signatures from the generated `jsapi` bindings. Then a macro was written which analyzed the function signature and the declared types. It was far from trivial - due to the fact that Rust macros need to output full AST nodes, it required exploiting advanced macro techniques. Depending on the type of the arguments it decided which arguments needed an accompanying `into()` call so as to be converted to a C++ type and which should be passed unchanged. Finally, the macro returned an AST node representing the wrapper.

Thanks to the generated wrappers, the definitive majority of the cases where the types mismatched boiled down to changing the import sources. In the remaining cases it was enough to add explicit conversions back and forth. All of this required careful examination of the call sites.

Aftermath

After the main PR being merged, two smaller improvements were done.

First of all, the pull request uncovered that not all type aliases were migrated to the new, more idiomatic Rust API, which was fixed by a subsequent pull request.

Moreover, the stage was set for a new set of wrappers superseding the current ones. This is due to the fact that the current set of wrappers requires the handles to implement the `Copy` trait.[39] Otherwise, they would consume the handle, effectively breaking existing code which took advantage of the API. The implementation of `Copy` for immutable handles is trivial - it can be provided by the compiler, since they pose no threat to the borrow checker invariants. On the other hand, it is difficult for **mutable** handles. The sheer possibility of mutable handles being cloned violates the borrow checker rules, which means that the current implementation had to effectively bypass the borrow checker and make use of `unsafe` code.

The requirement for implementing `Copy` could be eliminated if all the wrappers *borrowed* mutable handles instead of *copying* them. This would require massive changed at the call sites, though. Therefore, the decision was made to generate a second set of wrappers and let the code gradually migrate to the new set, so that the *copying* wrappers can be removed at some point.

4.2. Heap references

SpiderMonkey uses an additional `Heap<T>` wrapper type, whenever a *GC thing* pointer needs to be stored on the heap. Main reason for that is that its garbage collection algorithm can be ran in *incremental* mode. While the traditional mark-and-sweep GC algorithm often needs to *stop-the-world* (a name, which also stuck for the algorithm) to mark the living nodes and collect unreachable ones, an *incremental* GC is capable of interleaving the program execution and the object collection phase [42]. This can be considered beneficial, as it reduces *GC pause* time between regular program execution, but also requires more work during the marking phase to correctly mark living nodes. *Incremental* GCs do so by maintaining a mechanism called a *write barrier*. It is responsible for tracking every object store operation to ensure correct object livingness status. This is necessary, since the GC needs to be keep track of

all pointers into the *nursery* (a separate GC-managed heap, where JS objects are initially allocated).

The `Heap<T>` wrapper type is responsible for encapsulating the *write barrier* concerns. As such, it only permits modification of the wrapped value via a `set()` method, which in turn notifies the GC of the underlying value change. This restriction, along with how read access was additionally implemented and the safety implications thereof, will be further explored in the following subsections.

4.2.1. `Heap::mut_handle` fiasco

Firstly, it is worth emphasizing that the `Handle` and `MutableHandle` types are orthogonal to the `Heap` type. However, as often happens in software engineering, at some point a set of convenience functions were created that could create those two handle types out of a `Heap` value. The reason for that was that often SpiderMonkey C API accepts function arguments of types (`Mutable`)`Handle` (which in C++ can only be obtained via `Rooted` types), which implictly required that the used JS objects needed to be rooted prior to their usage.

It turned out that in the name of convenience, the safety of the resulting API was sacrificed and the end result completely side-stepped the *write barrier* whenever it handed out a `MutableHandle` to a `Heap` value. Indeed, a user code could freely modify underlying object through a handle, however it was not done via the `Heap` type, thus not correctly informing the GC of underlying object change. This surfaced as a series of weird, hard to reproduce crashes.

After investigating and tracking down the culprit, our team then managed to sanitize the previously used API. Firstly, any usage of the `Heap::handle_mut` function, which creates a `MutableHandle` to the wrapped *GC thing*, was removed from the codebase, along with the function definition itself. Furthermore, the `Heap::handle` function, which returned a shared, immutable `Handle` was marked as `unsafe`. By doing so, for every call of this function, the Rust compiler would require an explicit `unsafe {...}` block in which it could then be called, effectively leaving it to user to acknowledge the possible safety hazard and to prove the soundness of the function call in a given context. However, that in itself would not be very ergonomic. In addition to that, our team also implemented a safe `RootedTraceableBox<Heap<T>>::handle` function, for valid `T` types, successfully retaining necessary guarantees required by the GC. Not only we can safely only create `Handle` values for rooted JS objects, this allowed to reuse some of existing calls to `handle()` function and prevented another safety hazard related to moving `Heap` values, which will be expanded upon in the next section.

4.2.2. Unsafe `Heap::new` constructor

As explained before, `Heap` values encapsulate the *write barrier* concerns of the GC. Whenever a value is changed via the `Heap` wrapper, the GC is internally informed of the change to keep track of pointers into the *nursery*. Similarly to `CustomAutoRooter`, necessary functions are called via FFI with pointers to `Heap` wrappers themselves, so that the SpiderMonkey engine can be aware of its use. Previously, `Heap::new` constructor was defined as follows:

```
impl<T: GCMethods + Copy> Heap<T> {
    pub fn new(v: T) -> Heap<T>
        where Heap<T>: Default
    {
        let ptr = Heap::default();
        ptr.set(v);
```

```
        ptr
    }
    // ...
    pub fn set(&self, v: T) {
        unsafe {
            let ptr = self.ptr.get();
            let prev = *ptr;
            *ptr = v;
            T::post_barrier(ptr, prev, v);
        }
    }
}
```

Here, a `GCMethods` trait is implemented only for types that can be safely used with `Heap` wrappers and implement associated FFI *write barrier* calls (using `T::post_barrier` calls). An attentive reader may spot the mistake here: first, a temporary, uninitialized `Heap` value is created on stack (which itself is safe), then its location is passed through an FFI by calling `set()`, only to be finally moved and returned from the function, thus invalidating the previous memory location that was passed during `set()` function to SpiderMonkey. This was originally possible because the code inside the `set` function operated under the assumption that the `Heap` value is *immovable*, and is inside of an `unsafe` block, which shifts the responsibility to prove the soundness of underlying operations from the compiler to the programmer.

To remedy this, the `Heap::new` function was removed and replaced with a safe `Heap::boxed` variant:

```
impl<T: GCMethods + Copy> Heap<T> {
    /// Using boxed Heap value guarantees that the underlying
    /// Heap value will not be moved when constructed.
    pub fn boxed(v: T) -> Box<Heap<T>>
        where Heap<T>: Default
    {
        let boxed = Box::new(Heap::default());
        boxed.set(v);
        boxed
    }
}
```

While this carries an overhead of a heap allocation for `Box`, this API is safe to use. Initially it constructs an uninitialized `Heap` value on heap, using `Box`. Then it safely calls `set` (which triggers a *write barrier*), since the memory location of a heap allocated value is stable and does not change. Returning a `Box` wrapper type from the function is safe because it only acts as a handle to a value that is actually allocated somewhere else.

The new API is also easily composable, thanks to Rust *ownership* semantics. A new `RootedTraceableBox::from_box` function was implemented that accepts an argument of type `Box<T>`, where `T` is traceable. Because a `Box<T>` type *owns* the inner value of type `T`, the compiler can guarantee that an object constructed via `RootedTraceableBox::from_box` will have the ownership of the underlying data passed to it via the function argument. In practice this means that previous, unsafe API calls like:

```
RootedTraceableBox::new(Heap::new(value))
```

could be rewritten to a safe variant:

```
RootedTraceableBox::from_box(Heap::boxed(value))
```

avoiding any additional runtime cost. Due to *ownership* semantics, the new variant can still perform only a single heap allocation. Once `RootedTraceableBox::from_box` receives ownership over passed `Box` wrapper, it can directly reuse the heap-allocated data, since the type system guarantees that the data is not owned or referred to elsewhere.

4.3. WebIDL wrapper types

4.3.1. `TypedArray` types

Initially, when the scope of the group project was only emerging, full coverage of the WebGL 1.0 [27] specification in Servo was considered. However, we found the existing WebIDL ⇔ Rust bindings code generation capabilities to be insufficient for us to faithfully implement the remaining API. Instead, we focused on improving the code generation, among other things, which also meant implementing the Khronos Group TypedArray specification, needed for the remaining WebGL 1.0 implementation, and also later to be merged into the actual ECMAScript 2015 standard [28]. This meant that improving the support in Servo for TypedArrays would bring substantial benefits all across the board, not only to the existing partial WebGL implementation.

Implementation overview

At the time of writing, before the work was done, the `rust-mozjs` crate contained mostly low-to-intermediate-level bindings to the SpiderMonkey engine, with `TypedArray` being a notable exception, since it also aimed to encapsulate low-level concerns and expose fairly idiomatic Rust API to the user code. Internally, a `TypedArray` is in fact a single `*mut JSObject`, which also happens to points to an allocated contiguous typed buffer associated with each variant (e.g. an array of unsized 8-bit integers for `Uint8Array`).

As previously mentioned, to correctly use JS values (including `TypedArray`) in Servo, they need to be rooted prior to their usage. However, previous rooting API was neither safe nor ergonomic - user code needed to manually root a bare `*mut JSObject` value, which then could be used to construct a `TypedArray` wrapper, which additionally needed to serve as a root guard itself to prevent cases where the wrapper was used, but the outer root guard went out of scope (thus unrooting the object). Furthermore, this API did not support underlying objects of `Heap<*mut JSObject>` type, when the wrapped JS object was allocated not on the stack, but on the heap. To separate concerns, the rooting functionality was extracted from `TypedArray` and additional support for `Heap<*mut JSObject>` was introduced.

Firstly, we introduced a following trait:

```
pub trait JSObjectStorage {
    fn as_raw(&self) -> *mut JSObject;
    fn from_raw(raw: *mut JSObject) -> Self;
}
```

which we implemented for `*mut JSObject` and `Box<Heap<*mut JSObject>>`. As explained in the 4.2 section, a `Heap` type's location must be stable, hence why it is safe only to implement it when it is `Box`-wrapped. Additionally, the `TypedArray` wrapper type implementation was changed to be generic over a type implementing the aforementioned trait, also including a wrapped member object value of this type. This introduced two variants of the wrapper type:

- `TypedArray<_, *mut JSObject>` (to be used for on-stack values),

- `TypedArray<_, Box<Heap<*mut JSObject>>>` (to be used for on-heap values).

As explained in section 4.1, since the second variant holds wrapped value using a `Box<Heap<*mut JSObject>>`, the wrapper type can be safely used when stored on heap.

With second heap-enabled variant implemented, what also required changing was the rooting implementation. The rooting logic was separated from the wrapper type and so existing stack rooting implementation needed to be changed. Having implemented `CustomAutoRooter` infrastructure, this turned out to be fairly straightforward. As previously explained in section 4.1.1, in this case an implementation of `CustomTrace` trait for `TypedArray<_, *mut JSObject>` (stack variant!) was required:

```
unsafe impl<T> CustomTrace for TypedArray<T, *mut JSObject>
where T: TypedArrayElement {
    fn trace(&self, trc: *mut JSTracer) {
        self.object.trace(trc);
    }
}
```

Similarly, rooting the heap variant is also fairly straightforward:

```
pub type HeapTypedArray<T> =
    TypedArray<T, Box<Heap<*mut JSObject>>>;

// JSTraceable is the equivalent trait for heap-stored values
unsafe impl<T> JSTraceable for HeapTypedArray<T>
where T: TypedArrayElement {
    unsafe fn trace(&self, trc: *mut JSTracer) {
        self.underlying_object().trace(trc);
    }
}
```

These split implementations are simple, yet very powerful. They mix very well with the current tracing infrastructure.

At the moment of writing, WebIDL ⇔ Rust binding code generation currently emits only an on-heap variant for WebIDL union types. One example might be a following generated Rust enum from the `typedef (ArrayBuffer or ArrayBufferView) BufferDataSource;` WebIDL union definition:

```
#[derive(JSTraceable)]
pub enum ArrayBufferOrArrayBufferView {
    ArrayBuffer(RootedTraceableBox<HeapArrayBuffer>),
    ArrayBufferView(RootedTraceableBox<HeapArrayBufferView>),
}
```

Since the `RootedTraceableBox` can hold and root any `T` type that happens to implement the `JSTraceable` trait, the typed-arrays-in-unions part of binding code generation did not require any further redesigning and could benefit from the existing infrastructure.

Measurable improvements

As it turns out, the WebGL 1.0 implementation was not fully blocked on the lack of binding generation support for typed arrays. However, since Typed Array specification was merged

into the ECMAScript 2015 language standard, many APIs started supporting or even requiring those.

After the support for on-heap variant and binding code generation for `TypedArray` finally landed in the codebase, numerous API were unblocked and available for implementation. This change alone managed to fix (number in brackets is a Servo Pull Request ID):

- DOMMatrix (#20405) - 56 WPT tests
- Blob (#20370) - 30 WPT tests
- XMLHttpRequest.send (#20434) - 31 WPT tests
- WebSocket.Send (#20426) - 18 WPT tests
- TextDecoder.Decode (#20413) - 4 WPT tests

Not only it enabled some APIs, it also helped clean up current implementation and with this, many APIs were synced with their WebIDL counterparts:

- WebGL 1.0 (#20396) - 37 items
- DOMMatrix (#20405) - 6 items
- TextDecoder.Decode (#20413) - 5 items
- Bluetooth (#20422) - 4 items
- WebSocket.Send (#20426) - 2 items
- Blob (#20370) - 1 item
- XMLHttpRequest.Send (#20434) - 1 item

4.3.2. `Record` types

The `Record` type is WebIDL name for standard dictionary/map type capable of holding keys of 3 different types: `DOMString`, `ByteString` and `USVString` which are basically 3 different string-like interpretation of byte sequences. The `Record` type is meant to be Servo-side representation of the properties of JavaScript object. [47]

Conversion to `Record` according to WebIDL

WebIDL standard defines conversion of JavaScript object to `Record<K, V>` as follows. For each property of the object we check if it is enumerable and if the property value is not undefined. If so we convert the property key to WebIDL type `K` and value of the property is converted to WebIDL type `V`. Finally we add the resulting key - value pair to the record being constructed and proceed to the next property. [48]

`Record` in Servo and our changes

Before our modifications the implementation of `Record` types in Servo did not conform to the WebIDL specification. In particular, there was only one type of key possible - more specifically the key type parameter was silently ignored and the key was always assumed to be `DOMString`. For example these two WebIDL function declarations:

```
record<DOMString, long> foo();
record<ByteString, long> bar();
```

were interpreted as if both functions accepted no parameters and returned type `record<DOMString, long>`.

Moreover, the conversion of `Record` type was handled in the incorrect way: there was no checking if particular property in JavaScript object is enumerable, so e.g. the following JavaScript test given at the WebIDL `Record` type definition was not passing:

```
Object.defineProperty(obj, "a", {value: 7, enumerable: false});
let result = identity(obj);
console.assert(result.a === undefined); // result.a was 7 in this case
```

These inconsistencies with specification caused some further problems in code that used the `Record` types as well as rendered it impossible to implement some of the other types.

Our work involved fixing above problems. It consisted of writing correct conversions of these string types and basically rewriting the conversion of *Record* type to and from JavaScript value.

4.4. Compiler plugin

Servo and SpiderMonkey cooperate on DOM objects. In Servo, handles to these objects are realized by the `Dom<T>` type and its rooted equivalent of type `DomRoot<T>`. Safety of the integration between Servo and SpiderMonkey is achieved with two major parts:

1. Compiler plugin automatically generates implementations of `JSTraceable` trait that ensures SpiderMonkey is aware of all references between DOM objects.
2. The implementation of `DomRoot<T>` makes sure that SpiderMonkey knows about all DOM objects used from Rust.

There is a problem here. We could copy an unrooted pointer - a `Dom<T>` - to a local variable on the stack, and later - root it and use the DOM object. SpiderMonkey's garbage collector will not be aware of our new reference and could move the object elsewhere or free it. It would result in an *use after free* vulnerability which is really bad for web browser security. So, to be safe, we need to ensure that `Dom<T>` only appears in places where it will be traced and never in local variables, function arguments, and so forth. [43]

Servo developers have already implemented a compiler plugin which performs a lint check for the said case. It is worth mentioning that such a plugin is just an attempt to catch most common mistakes at a low cost with no guarantee to find all mistakes. We have improved the check.

4.4.1. How the plugin works

The plugin provides, among others, `#[must_root]` and `#[allow_unrooted_interior]` attributes. The former one is used to mark DOM objects types that must be rooted whilst the latter one is meant to be used on DOM structures types that manipulate on DOM objects and are allowed to have unrooted local references.

The check is performed as a lint pass. It means that we have an access to the AST and the HIR of code. Having it, the plugin ensures that in every method there is no local variable on the stack of a type that has the `#[must_root]` attribute and that all data structures which contains a field of a type with `#[must_root]` attribute have `#[must_root]` attribute themselves.

4.4.2. Improvements

In the beginning, let's take a look at the difference between HIR and MIR. HIR is a desugarated Rust code, so we have access to logical structure of the code. We can, for an instance, iterate over struct definitions, functions, expressions within them, generic types, access their attributes and so on. MIR, on the other hand, is much simpler entity. It represents functions as flow graphs. More complex beings like struct object initialization or nested expressions are being simplified down to single assignments. This code looks more similar to assembly code. [18]

Our work begun starting with a GitHub issue titled *Write a MIR lint for rooting analysis.* [44] The motivation behind this was that MIR operates on monomorphized types. It means that we know types of everything what is not the case with HIR and generics. It quickly turned out that MIR pass, which iterates over all generated functions and was the intended solution, had been removed from the compiler in the meantime. Thus, we have solved it with another approach. [45]

Having HIR of a function, we can query the compiler for its MIR. But, if it was a generic function, then we know only generic parameters, not the specific ones used somewhere else where the function is called. So, the solution was not so straightforward as we thought in the beginning.

The following cases were initially not detected and we fixed it.

```
fn foo<T>() { /* ... */ }
fn bar<#[must_root] U>() {
    foo::<U>();
}
```

Here, we have a function `bar` with generic parameter `U` that has an attribute `#[must_root]`, so the plugin will not allow to have values of this type on the stack. It means that it is permitted to call `bar` with a `#[must_root]` type **as well as** with any other type since it does not have such restrictions on its usage. However, a function `foo` requires generic parameter `T` with no `#[must_root]` attribute. It implies that the plugin does not restrict `T` and allows to create values of this type on the stack. Taking a look at the example again can show that `bar` calls `foo` with generic parameter that is `#[must_root]`, but `foo` does not restrict its parameter with `#[must_root]` attribute, so the call is potentially dangerous and must be detected. We can compare this case to traits: type which is `#[must_root]` does not require anything, and parameter without this attribute "implements `CanBePutOnTheStack` trait". One important fact here is that `#[must_root]` is not propagated automatically by the compiler across the generic calls; it also does not restrict which types are allowed in the function, even if they are annotated with the `#[must_root]` attribute. This is one of the lints the plugin is responsible for - it checks if a function call with a specific type is valid in a given context.

Moving forward, we have next example:

```
#[must_root] struct Foo(i32);
struct SomeContainer<T>(T);

fn test() {
   SomeContainer(Foo(3));
}
```

Here we instantiate `SomeContainer<Foo>` object which has field of type `Foo` that is `#[must_root]`, but the `SomeContainer<Foo>` is not `#[must_root]` itself, so it is incorrect.

Then, next case:

```rust
#[must_root] struct Foo(i32);
struct SomeContainer<T>(T);

impl<T> SomeContainer<T> {
    fn foo(val: T) {
        SomeContainer(val);
    }
}

fn test() {
    // should require impl<#[must_root] T> SomeContainer<T>
    SomeContainer::foo(Foo(3));
}
```

As the comment states, we should check if the `impl` provides `T` that is `#[must_root]`. Otherwise, we could work around lint check and, for example, take `#[must_root]` type as a function argument, which lands on stack or even pass it further to data structure that does not guarantee special treatment for `#[must_root]` types.

```
error: ADT generic type must be rooted.
   --> components/script/script_thread.rs:828:28
    |
828 |             job_queue_map: Rc::new(JobQueue::new()),
    |                            ^^^^^^^^^^^^^^^^^^^^^^^^

error: ADT generic type must be rooted.
   --> components/script/script_thread.rs:859:40
    |
859 |             topmost_mouse_over_target: MutNullableDom::new(Default::default()),
    |                                        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

error: Callee generic type must be rooted.
   --> components/script/webdriver_handlers.rs:391:20
    |
391 |                 Ok(input_element.Checked())
    |                    ^^^^^^^^^^^^^^^^^^^^^^^

error: Callee generic type must be rooted.
   --> components/script/webdriver_handlers.rs:393:48
    |
393 |             else if let Some(option_element) = node.downcast::<HTMLOptionElement>() {
    |                                                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

error: aborting due to 9106 previous errors

error: Could not compile `script`.
```

Figure 4.1: Many existing violations detected by the plugin. Screenshot was taken at some point of developing the plugin.

4.4.3. Problems

We met some obstacles on our way. One of them was a bug in the compiler. Having the following code:

```rust
fn foo<#[must_root] U>() { }
```

We should expect `U` to have `#[must_root]`, but it did not. During lowering AST to HIR, this information was not passed down. We have fixed it. [46]

4.5. Other minor improvements

4.5.1. Capturing JS stack traces

Developing a big project like Servo is not easy and requires helpful tools to aid the process. One of the cases includes implementing JavaScript API on the Servo side. When someone implements such an API and it behaves in an unexpected way, they would like to know what actually happens. One of the most desired pieces of information is the stack trace of the JavaScript interpreter. SpiderMonkey provides an API for it, but it is really inconvenient to be used from within Rust. So, we have implemented a simple and really straightforward API for it, as demonstrated in the example below. [19] [20]

```
capture_stack!(in(context) let stack);
let str_stack = stack.unwrap().as_string(None).unwrap();
println!("{}", str_stack);
```

Furthermore, using the shown API is much safer than doing it by hand.

```
let javascript = "
    function foo(arg1) {
        var bar = function() {
            print_stack();
        };
        bar();
    }

    foo(\"arg1-value\");
";
rooted!(in(context) let mut rval = UndefinedValue());
let _ = runtime.evaluate_script(global, javascript, "test.js", 0, rval.handle_mut());

// in print_stack():
capture_stack!(in(context) let stack);
let str_stack = stack.unwrap().as_string(None).unwrap();
println!("{}", str_stack);
assert_eq!("foo/bar@test.js:3:21\nfoo@test.js:5:17\n@test.js:8:13\n".to_string(), str_stack);
```

```
     Running target/debug/deps/capture_stack-278d9e43fb861a86

running 1 test
foo/bar@test.js:3:21
foo@test.js:5:17
@test.js:8:13

test capture_stack ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Figure 4.2: Capturing JavaScript stack trace and test result in terminal. The `print_stack()` function is defined in Rust - analogously like `console.log()` in figure 1.1.

4.5.2. Build system improvements

We also improved the robustness of the build system. Some of the C/C++ dependencies of Servo require CMake for building, sometimes - at least version 3. To obtain seamless integration of such libraries into the Rust's package management system, even C/C++ libraries are packaged as crates. The `cmake-rs` crate [61] takes care of the whole configuration of libraries taking advantage of CMake and as a result is a common choice in such cases.

This was an issue on distributions such as CentOS, where the default version of CMake is version 2 and version 3 is provided by `/usr/bin/cmake3`. To work this around, we added support for overriding the default executable using an environmental variable, `CMAKE`. [59] [58]

Another issue on CentOS was the outdated LLVM version, which made it impossible to generate bindings to C++ libraries. A solution to this problem was to execute special preparation steps before commencing the build. [60] https://github.com/alexcrichton/cmake-rs All in all, we have enabled and documented building Servo on CentOS.

Another issue could be observed on rolling, always up-to-date distributions. Python is often chosen as a language of choice for build script, instead of Bash scripts or custom Makefiles. For compatibility reasons, they are often written in Python 2, sometimes in a way incompatible with Python 3.

Unlike most LTS distributions, Arch Linux ships Python 3 as a default [62], which caused a build error while trying to execute a build script written for Python 2. We added a simple mechanism, which tries to detect a dedicated Python 2 binary (most distributions provide a `python2` binary as a *convention-over-configuration*), falling back to the generic `python` executable otherwise.[57] This guaranteed a seamless build on Arch Linux.

Chapter 5

Development organization

5.1. Workflow

Our work was organized in weekly sprints. As a conclusion of each sprint we held a meeting during which we reported the progress we made as well as possible problems that emerged. We ended these appointments with assigments of tasks to be done during next week.

5.2. Team members' contribution

In addition to the following individuals' contribution, everyone was equally engaged in regularly communicating with Servo project developers, coordinating team work as a whole and reviewing other members' work.

5.2.1. Artur Jamro

- Designed and implemented JavaScript stack traces API (section 4.5.1)
- Reimplemented and improved compiler plugin (section 4.4)

5.2.2. Jarosław Ławnicki

- Designed and reimplemented `Record` type (section 4.3.2)

5.2.3. Igor Matuszewski

- Reimplemented CustomAutoRooter infrastructure (section 4.1.1)
- Sanitized `Heap` references API (section 4.2)
- Redesigned `TypedArray` API (section 4.3.1)

5.2.4. Marcin Mielniczuk

- Added lifetimes to the `Handle` types (section 4.1.2)
- Minor fixes in the build system (section 4.5.2)

Chapter 6

Summary

By enhancing the static analysis performed on the Servo project, especially integration with the SpiderMonkey JavaScript runtime, an entire class of errors was eliminated and potential programmer mistakes were prevented. Even the best API documentation does not prevent mistakes from happening, which is why it is so beneficial having the compiler statically enforce additional rules required by an API, instead of relying on best practices or team wisdom alone.

The Rust programming language is incapable of expressing every constraint (e.g. it still lacks support for dependent types [32]) nor can it perform any formal proof of software properties. However, its novel *affine* type system [33] (expressed via *ownership semantics*), coupled with parametric and ad-hoc polymorphism, allow for much bigger expresiveness power among widespread, industry standard programming languages such as C++.

This successfully allowed to express all the relevant, implicit C++ API guarantees in the Rust type system, making the Servo–SpiderMonkey integration safer and more robust in the process.

Bibliography

[1] https://en.wikipedia.org/wiki/Rust_(programming_language)

[2] https://www.rust-lang.org/en-US/

[3] https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html

[4] https://doc.rust-lang.org/book/second-edition/ch10-03-lifetime-syntax.html

[5] https://danielkeep.github.io/tlborm/book/mbe-syn-README.html

[6] https://doc.rust-lang.org/book/first-edition/macros.html

[7] Rustc guide, how to build and run https://rust-lang-nursery.github.io/rustc-guide/how-to-build-and-run.html

[8] Rust unstable book, plugin https://doc.rust-lang.org/unstable-book/language-features/plugin.html

[9] Rustc plugin registry documentation https://manishearth.github.io/rust-internals-docs/rustc_plugin/registry/struct.Registry.html

[10] Servo DOM inheritance https://doc.servo.org/script/dom/index.html#inheritance

[11] Rocket web framework https://rocket.rs

[12] Rocket codegen https://api.rocket.rs/rocket_codegen/

[13] Rust-clippy https://github.com/rust-lang-nursery/rust-clippy

[14] The LLVM Project homepage https://llvm.org/

[15] Web IDL Editor's Draft, 4 March 2018 https://heycam.github.io/webidl/

[16] MDN web docs: WebIDL https://developer.mozilla.org/en-US/docs/Glossary/WebIDL

[17] Abstract syntax tree https://en.wikipedia.org/wiki/Abstract_syntax_tree

[18] Introducing MIR https://blog.rust-lang.org/2016/04/19/MIR.html

[19] JS stack trace issue. https://github.com/servo/servo/issues/14987

[20] JS stack trace pull request. https://github.com/servo/rust-mozjs/pull/381

[21] Firefox Stylo CSS engine improvements benchmark https://bugzilla.mozilla.org/show_bug.cgi?id=1330412#c47

[22] WebRender project page https://github.com/servo/webrender

[23] Mozilla Version Control Tools: Vendoring Projects in a Monorepo http://mozilla-version-control-tools.readthedocs.io/en/latest/vcssync/vendoring.html

[24] GC Rooting Guide https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/GC_Rooting_Guide

[25] http://en.cppreference.com/w/cpp/language/raii

[26] https://github.com/rust-lang/rfcs/blob/master/text/2349-pin.md

[27] Khronos Group, WebGL 1.0 Specification https://www.khronos.org/registry/webgl/specs/1.0.3/

[28] Ecma International, ECMAScript 2015 Language Specification, TypedArray Objects http://www.ecma-international.org/ecma-262/6.0/#sec-typedarray-objects

[29] RootingAPI.h, SpiderMonkey developers,

[30] Mozilla Contributors, Introduction to the DOM https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

[31] rust-bindgen repository https://github.com/rust-lang-nursery/rust-bindgen/

[32] Ticki, RFC #1930: The Π type trilogy https://github.com/rust-lang/rfcs/issues/1930 (accessed April 15th, 2018)

[33] Wikipedia, Affine type systems https://en.wikipedia.org/wiki/Substructural_type_system#Affine_type_systems

[34] Wikipedia, Ad-hoc polymorphism https://en.wikipedia.org/wiki/Ad_hoc_polymorphism

[35] C++ Reference, Abstract class http://en.cppreference.com/w/cpp/language/abstract_class

[36] Brian Hackett, Bug 707049 - Dynamic analysis for identifying moving GC hazards https://bugzilla.mozilla.org/show_bug.cgi?id=707049, access date: 4th Apr 2018

[37] Michael Wu, Make Handle objects take lifetimes, issue #153, https://github.com/servo/rust-mozjs/issues/153, access date: 5th Apr 2018

[38] Michael Wu, Upgrade to SM 39, issue #150, https://github.com/servo/rust-mozjs/pull/150, access date: 13th Apr 2018

[39] Rust developers, std::marker::Copy, https://doc.rust-lang.org/std/marker/trait.Copy.html, access date: 13th Apr 2018

[40] Léo Testard, Lifetime handling doesn't handle function pointers properly, issue #10501 https://github.com/rust-lang/rust/issues/10501, access date: 5th Apr 2018

[41] Josh Matthews et al, An excerpt from the logs of the #servo IRC channel https://mozilla.logbot.info/servo/20171127#c13919577-c13919608, access date: 5th Apr 2018

[42] SpiderMonkey garbage collection: Incremental marking https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals/Garbage_collection#Incremental_marking

[43] Rooting static analysis https://github.com/servo/servo/blob/master/components/script/docs/JS-Servos-only-GC.md#custom-static-analysis

[44] Write a MIR lint for rooting analysis (issue #14902) https://github.com/servo/servo/issues/14902

[45] Write a MIR lint for rooting analysis (pull request #20264) https://github.com/servo/servo/pull/20264

[46] Compiler bug fix (pull request) https://github.com/rust-lang/rust/pull/49242

[47] https://heycam.github.io/webidl/#idl-record

[48] https://heycam.github.io/webidl/#es-record

[49] Marcin Mielniczuk, Use python2 for the build whenever possible and bump the version, issue #132, Usepython2forthebuildwheneverpossibleandbumptheversion.#132, access date 13th Apr 2018

[50] Marcin Mielniczuk, Bump the version of cmake to allow fixing build on CentOS, issue #125, https://github.com/servo/gecko-media/pull/125, access date 13th Apr 2018

[51] Marcin Mielniczuk, Add support for the CMAKE environmental variable, issue #42, https://github.com/alexcrichton/cmake-rs/pull/42, access date: 13th Apr 2018

[52] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey

[53] Advanced Lifetimes, The Rust Programming Language Book, https://doc.rust-lang.org/book/second-edition/ch19-02-advanced-lifetimes.html, access date: 14th Apr 2018

[54] Advanced Traits, The Rust Programming Language Book, https://doc.rust-lang.org/book/second-edition/ch19-03-advanced-traits.html#using-supertraits-to-require-one-traits-functionality-within-another-trait, access date: 15th Apr 2018

[55] A Gentle Introduction to Haskell: Classes, https://www.haskell.org/tutorial/classes.html, access date: 15th Apr 2018

[56] Traits | Scala Documentation, https://docs.scala-lang.org/tour/traits.html, access date: 15th Apr 2018

[57] Marcin Mielniczuk, Use python2 for the build whenever possible and bump the version, issue #132, Usepython2forthebuildwheneverpossibleandbumptheversion.#132, access date 13th Apr 2018

[58] Marcin Mielniczuk, Bump the version of cmake to allow fixing build on CentOS, issue #125, https://github.com/servo/gecko-media/pull/125, access date 13th Apr 2018

[59] Marcin Mielniczuk, Add support for the CMAKE environmental variable, issue #42, https://github.com/alexcrichton/cmake-rs/pull/42, access date: 13th Apr 2018

[60] Marcin Mielniczuk, Fix and document build on CentOS 7.4, issue #19545 https://github.com/servo/servo/pull/19545, access date: 27th Apr 2018

[61] Alex Crichton, cmake-rs on GitHub, https://github.com/alexcrichton/cmake-rs, access date: 27th Apr 2018

[62] Allan McRae, Python is now Python 3 https://www.archlinux.org/news/python-is-now-python-3/, access date: 27th Apr 2018