Experience Report: Josephine

Using JavaScript to safely manage the lifetimes of Rust data

ALAN JEFFREY, Mozilla Research

This paper is about the interface between languages which use a garbage collector and those which use fancy types for safe manual memory management. It uses existing techniques for using linear capabilities to provide safe access to copyable references, but the application to languages with a tracing garbage collector is new. This work is in the area of mixed linear/non-linear languages, but the linear language is Rust, and the non-linear language is JavaScript.

CCS Concepts: • **Software and its engineering → Functional languages**; *Imperative languages*;

Additional Key Words and Phrases: JavaScript, Rust, interoperability, memory safety, affine types

1 INTRODUCTION

This paper is about the interface between languages which use a garbage collector and those which use fancy types for safe manual memory management.

Garbage collection is the most common memory management technique for functional programming languages, dating back to LISP [12]. Having a garbage collector guarantees memory safety, but at the cost of having a required runtime system.

Imperative languages often require the programmer to perform manual memory management, such as the `malloc` and `free` functions provided by C [11]. The safety of a program (in particular the absence of *use-after-free* errors) is considered the programmer's problem. More recently, laguages such as Cyclone [10] and Rust [5] have used fancy type systems such as substructural types [4, 8, 16] and region analysis [15] to guarantee memory safety without garbage collection.

This paper discusses the Josephine API [9] for using the garbage collector provided by the Spidermonkey [13] JavaScript runtime to safely manage the lifetime of Rust [5] data. It uses techniques from L^3 [1] and its application to regions [7], but the application to languages with a tracing garbage collector is new.

1.1 Rust

Rust is a systems programming language which uses fancy types to ensure memory safety even in the presence of mutable update, and manual memory management. Rust has an affine type system, which allows data to be discarded but does not allow data to be arbitrarily copied. For example, the Rust program:

```
let hello = String::from("hello");
let moved = hello;
println!("Oh look {} is hello", moved);
```

is fine, but the program:

```
let hello = String::from("hello");
let copied = hello;
println!("Oh look {} is {}", hello, copied);
```

is not, since `hello` and `copied` are simultaneously live. Trying to compile this program produces:

Author's address: Alan Jeffrey, Mozilla Research, ajeffrey@mozilla.com.

2018. XXXX-XXXX/2018/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
use of moved value: `hello`
 --> src/main.rs:4:32
  |
3 |   let copied = hello;
  |       ------ value moved here
4 |   println!("Oh look {} is {}", hello, copied);
  |                                ^^^^^ value used here after move
```

The use of affine types allows aliasing to be tracked. For example, a classic problem with aliasing is appending a string to itself. In Rust, an example of appending a string is:

```
let mut hello = String::from("hello");
let ref world = String::from("world");
hello.push_str(world);
println!("Oh look hello is {}", hello);
```

The important operation is `hello.push_str(world)`, which mutates the string `hello` (hence the `mut` annotation on the declaration of `hello`). The appended string `world` is passed by reference, (hence the `ref` annotation on the declaration of `world`).

A problem with mutably appending strings is ensuring that the string is not appended to itself, for example the documentation for C `strcat` states "Source and destination may not overlap", but C does not check aliasing and relies on the programmer to ensure correctness. In contrast, attempting to append a string to itself in Rust:

```
let ref mut hello = String::from("hello");
hello.push_str(hello);
```

produces an error:

```
cannot borrow `*hello` as immutable because it is also borrowed as mutable
 --> src/main.rs:3:18
  |
3 |   hello.push_str(hello);
  |   -----          ^^^^^- mutable borrow ends here
  |   |              |
  |   |              immutable borrow occurs here
  |   mutable borrow occurs here
```

In Rust, the crucial invariant maintained by affine types is:

Any memory that can be reached simultaneously by two different paths is immutable.

For example in `hello.push(hello)` there are two occurrences of `hello` that are live simultaneously, the first of which is mutating the string, so this is outlawed.

In order to track statically which variables are live simultaneously, Rust uses a lifetime system similar to that used by region-based memory [15]. Each allocation of memory has a lifetime α, and lifetimes are ordered $\alpha \subseteq \beta$. Each code block introduces a lifetime, and for data which does not escape from its scope, the nesting of blocks determines the ordering of lifetimes.

For example in the program:

```
let ref x = String::from("hi");
{
  let ref y = x;
  println!("y is {}", y);
}
println!("x is {}", x);
```

the variable x has a lifetime α given by the outer block, and the variable y has a lifetime $\beta \subseteq \alpha$ given by the inner block.

These lifetimes are mentioned in the types of references: the type $\&\alpha\ T$ is a reference giving immutable access to data of type T, which will live at least as long as α. Similarly, the type $\&\alpha$ mut T gives mutable access to the data: the crucial difference is that $\&\alpha\ T$ is a copyable type, but $\&\alpha$ mut T is not. For example the type of x is $\&\alpha$ String and the type of y is $\&\beta$ ($\&\alpha$ String), which is well-formed because $\beta \subseteq \alpha$.

Lifetimes are used to prevent another class of memory safety issues: use-after-free. For example, consider the program:

```
let hi = String::from("hi");
let ref mut handle = &hi;
{
  let lo = String::from("lo");
  *handle = &lo;
} // lo is deallocated here
println!("handle is {}", **handle);
```

If this program were to execute, its behaviour would be undefined, since `**handle` is used after `lo` (which `handle` points to) is deallocated. Fortunately, this program does not pass Rust's borrow-checker:

```
`lo` does not live long enough
 --> src/main.rs:6:11
  |
6 |     *handle = &lo;
  |               ^^ borrowed value does not live long enough
7 |   } // lo is deallocated here
  |   - `lo` dropped here while still borrowed
8 |   println!("handle is {}", **handle);
9 | }
  | - borrowed value needs to live until here
```

This use-after-free error can be detected because (naming the outer lifetime to be α and the inner lifetime to be $\beta \subseteq \alpha$) `handle` has type $\&\alpha$ mut $\&\alpha$ String, but `&lo` only has type $\&\beta$ String, no $\&\alpha$ String as required by the assignment.

Lifetimes avoid use-after-free by maintaining two invariants:

> *Any dereference happens during the lifetime of the reference,*
> *and deallocation happens after the lifetime of all references.*

There is more to the Rust type system than described here (higher-order functions, traits, variance, concurrency, …) but the important features are *affine types* and *lifetimes* for ensuring memory safety, even in the presence of manual memory management.

1.2 Spidermonkey

Spidermonkey is Mozilla's JavaScript runtime, used in the Firefox browser, and the Servo [14] next-generation web engine. This is a full-featured JS implementation, but the focus of this paper is its automatic memory management.

Inside a web engine, there are often native implementations of HTML features, which are exposed to JavaScript using DOM interfaces. For example, an HTML image is exposed to JavaScript as a

DOM object representing an <img> element, but behind the scenes there is native code responsible for loading and rendering images.

When a JavaScript object is garbage collected, a destructor is called to deallocate any attached native memory. In the case that the native code is implemented in Rust, this leads to a situation where Rust relies on affine types and lifetimes for memory safety, but JavaScript respects neither of these. As a result, the raw Spidermonkey interface to Rust is very unsafe, for example there are nearly 400 instances of unsafe code in the Servo DOM implementation:

```
$ grep "unsafe_code" components/script/dom/*.rs | wc
    393     734   25514
```

Since JavaScript does not respect Rust's affine types, Servo's DOM implementation makes use of Rust [5, §3.11] *interior mutability* which replaces the compile-time type checks with run-time dynamic checks. This carries run-time overhead, and the possibility of checks failing, and Servo panicking.

Moreover, Spidermonkey has its own invariants, and if an embedding application does not respect these invariants, then runtime errors can occur. One of these invariants is the division of JavaScript memory into *compartments*, which can be garbage collected separately. The runtime has a notion of "current compartment", and the embedding application is asked to maintain two invariants:

- whenever an object is used, the object is in the current compartment, and
- there are no references between objects which cross compartments.

In order for native code to interact well with the Spidermonkey garbage collector, it has to provide two functions:

- a *trace* function, that given an object, iterates over all of the JavaScript objects which are reachable from it, and
- a *roots* function, which iterates over all of the JavaScript objects that are live on the call stack.

From these two functions, the garbage collector can find all of the reachable JavaScript objects, including those reachable from JavaScript directly, and those reached via native code.

Automatically generating the trace function is reasonably straightforward metaprogramming, but rooting safely turns out to be quite tricky. Servo provides an approximate analysis for safe rooting using an ad-hoc static analysis (the *rooting lint*), but this is problematic because a) the lint is syntax-driven, so does not understand about Rust features such as generics, and b) even if it could be made sound it is disabled more than 200 times:

```
$ grep "unrooted_must_root" components/script/dom/*.rs | wc
    213     456   15961
```

1.3 Josephine

Josephine [9] is intended to act as a safe bridge between Spidermonkey and Rust. Its goals are:

- to use JavaScript to manage the lifetime of Rust data, and to allow JavaScript to garbage collect unreachable data,
- to allow references to JavaScript-managed data to be freely copied and discarded, relying on Spidermonkey's garbage collector for safety,
- to maintain Rust memory safety via affine types and lifetimes, without requiring additional static analysis such as the rooting lint,
- to allow mutable and immutable access to Rust data via JavaScript-managed references, so avoiding interior mutability, and
- to provide a rooting API to ensure that JavaScript-managed data is not garbage collected while it is being used.

Josephine is intended to be safe, in that any programs built using Josephine's API do not introduce undefined behaviour or runtime errors. Josephine achieves this by providing controlled access to Spidermonkey's *JavaScript context*, and maintaining invariants about it:

- immutable access to JavaScript-managed data requires immutable access to the JavaScript context,
- mutable access to JavaScript-managed data requires mutable access to the JavaScript context, and
- any action that can trigger garbage collection (for example allocating new objects) requires mutable access to the JavaScript context.

As a result, since access to the JavaScript context does respect Rust's affine types, mutation or garbage collection cannot occur simultaneously with accessing JavaScript-managed data.

In other words, Josephine treats the JavaScript context as an affine access token, or capability, which controls access to the JavaScript-managed data. The data accesses respect affine types, even though the JavaScript objects themselves do not.

This use of an access token to safely access data in a substructural type system is *not* new, it is the heart of Ahmed, Fluet and Morrisett's L^3 Linear Language with Locations [1] and its application to linear regions [15].

Moreover, type systems for mixed linear/non-linear programming have been known for more than 20 years [3]. The aspects that are novel are:

- the linear language is Rust, and the non-linear language is JavaScript, which are both industrial-strength languages,
- the treatment of garbage collection requires a different treatment than regions in L^3, which have a stack discipline, and
- the token contains more state in its type, carrying more than just read/write access, but the current compartment, and the capability to trigger garbage collection.

Acknowledgments

This work benefited greatly from conversations with Amal Ahmed, Nick Benton, Josh Bowman-Matthews, Manish Goregaokar, Bobby Holly, and Anthony Ramine.

2 THE JOSEPHINE API

There are two important concepts in Josephine's API: *JS-managed* data, and the JS *context*. For readers familiar with the region-based variant [7] of L^3 [1], JS-managed data corresponds to L^3 references, and JS contexts to L^3 capabilities.

2.1 JS-managed data

JS-managed data has the type JSManaged$\langle \alpha, C, T \rangle$, which represents a reference to data whose lifetime is managed by JS, which:

- is guaranteed to live at least as long as α,
- is allocated in JS compartment C,
- points to native data of type T.

This type is copyable, so not subject to the affine type discipline, even though it can be used to gain mutable access to the native data. We shall see later that this is safe for the same reason as L^3: we are using the JS context as a capability, and it is not copyable.

In examples, we make use of Rust's *lifetime elision* [6, §3.4], and just write JSManaged$\langle C, T \rangle$ where the lifetime α can be inferred.

In the simplest case, T is a base type like String, but in more complex cases, T might itself contain JS-managed data, for example a type of cells in a doubly-linked list can be defined:

```
type Cell<'a, C> = JSManaged<'a, C, NativeCell<'a, C>>;
```

where:

```
struct NativeCell<'a, C> {
  data: String,
  prev: Option<Cell<'a, C>>,
  next: Option<Cell<'a, C>>,
}
```

This pattern is a common idiom, in that there are two types:

- NativeCell$\langle\alpha, C\rangle$ containing the native representation of a cell, including the prev and next references, and
- Cell$\langle\alpha, C\rangle$ containing a reference to a native cell, whose lifetime is managed by JS.

These types are both parameterized by a lower bound α on the lifetime of the cell, and the compartment C that the cell lives in.

Doubly-linked lists are an interesting example of programming in Rust, and indeed there is an introductory text *Learning Rust With Entirely Too Many Linked Lists* [2], in which safe implementations of doubly-linked lists require interior mutability (and hence dynamic checks) and reference counting.

2.2 The JS context

By itself, JS-managed references are not much use: there has to be an API for creating and dereferencing them: this is the role of the JS *context*, which acts as a capability for manipulating JS-managed data.

There is only one JS context per thread (and JS contexts cannot be shared or sent between threads) so unique access to the JS context implies unique access to all JS-managed data. We can use this to give safe mutable access to JS-managed data, since the JS context is a unique capability.

The JS context has a state, notably keeping track of the current compartment, but also permissions such as "allowed to create new references" or "allowed to dereference". This state is tracked in the type system using phantom types, so the JS context has type JSContext$\langle S\rangle$, where S is the current state.

For example, a program to allocate a new JS-managed reference is:

```
let x: JSManaged<C, String> = cx.manage(String::from("hello"));
```

and a program to access a JS-managed reference is:

```
let msg: &String = x.borrow(cx);
```

These programs make use of the JS context `cx`. In order for the first example to typecheck:

- `cx` must have type & mut JSContext$\langle S\rangle$, where
- S (the state of the context) must have permission to allocate references in C, and
- C must be a compartment.

The second example is similar, except:

- we do not need mutable access to the context, and
- S must have permission to access compartment C.

Fortunately, Rust has a *trait* system (similar to Haskell's class system), which allows us to express these constraints. In the same way that C and S are phantom types, these are *marker* traits with no computational value. The typing for the first example is:

```
(cx: &mut JSContext<S>) where
  S: CanAlloc + InCompartment<C>,
  C: Compartment,
```

and for the second:

```
(cx: &JSContext<S>) where
  S: CanAccess,
  C: Compartment,
```

A program to mutably access a JS-managed reference is:

```
let msg: &mut String = x.borrow_mut(cx);
```

at which point the fact that the JS context is an affine capability becomes important. The typing required for this is:

```
(cx: &mut JSContext<S>) where
  S: CanAccess,
  C: Compartment,
```

That is *unique access to JS-managed data requires unique access to the JS context*, and so we cannot simultaneously have mutable access to two different JS-managed references. This is the same safety condition that region-based L^3 uses.

For example, we can use this (together with Rust's built-in replace function which swaps the contents of a mutable reference) to replace the contents of a cell with a new value:

```
fn replace<S>(self, cx: &'a mut JSContext<S>, new_data: String) -> String where
  S: CanAccess,
  C: Compartment,
{
  let ref mut old_data = self.0.borrow_mut(cx).data;
  replace(old_data, new_data)
}
```

2.3 Typing access

A first-cut type rule for accessing data in a typing context in which S : CanAccess and C : Compartment is:

if cx : & JSContext$\langle S \rangle$ and p : JSManaged$\langle C, T \rangle$ then p.borrow(cx) : & T

(and similarly for mutable access) which is fine, but does not mention the lifetimes. Adding these in gives:

if cx : &α JSContext$\langle S \rangle$ and p : JSManaged$\langle \alpha, C, T \rangle$ then p.borrow(cx) : &α T

which is correct, but assumes that the lifetime that the JS context has been borrowed for is exactly the same as the lifetime of the reference. Separating these gives (when $\alpha \subseteq \beta$):

if cx : &α JSContext$\langle S \rangle$ and p : JSManaged$\langle \beta, C, T \rangle$ then p.borrow(cx) : &α T

This rule is still incorrect, but for a slightly subtle reason. It *is* correct when T is a base type, but fails in the case of a type which includes nested JS-managed references. If that were the rule, then we could write programs such as

```
let cell: Cell<'a, C> = ...;
let next: Cell<'a, C> = cell.borrow(cx).next?; // cell is keeping next alive
cell.borrow_mut(cx).next = None;            // nothing is keeping next alive
cx.gc();                                       // something that triggers GC
next.borrow(cx);                                 // this is a use-after-free
```

The problem in this example is that after setting cell's next pointer to `None`, there is nothing in JS keeping `next` alive, so it is reachable from Rust but not from JS. After a GC, the JS runtime can deallocate `next`, so accessing it is a use-after-free error.

In a language with built-in support for GC, there would be hidden GC root introduced by putting `next` on the stack, but Rust does not have support for such hidden rooting.

The problem in general is that when accessing p : JSManaged$\langle\beta, C, T\rangle$, using a JS context borrowed for lifetime $\alpha \subseteq \beta$, there may be nested JS-managed data, also with lifetime β. These are being kept alive by p, which is fine as long as p is not mutated, but mutating p might cause them to become unreachable in JS, and thus candidates for garbage collection.

The fix used by Josephine is to replace any nested uses of β in T by α, that is the type rule is (when $\alpha \subseteq \beta$):

$$\text{if } cx : \&\alpha\, \text{JSContext}\langle S\rangle \text{ and } p : \text{JSManaged}\langle\beta, C, T\rangle \text{ then } p.\text{borrow}(cx) : \&\alpha\, T[\alpha/\beta]$$

The conjecture that Josephine makes is that this is safe, because GC cannot happen during the lifetime α. In order to ensure this, we maintain an invariant:

Any operation that can trigger garbage collection requires mutable access to the JS context.

This is why `cx.manage(data)` requires `cx` to have type `&mut JSContext<S>`, *not* because we are mutating the JS context itself, but because allocating a new reference might trigger GC.

In Rust, the substitution $T[\alpha/\beta]$ is expressed by T implementing a trait JSLifetime$\langle\alpha\rangle$:

```
pub unsafe trait JSLifetime<'a> {
  type Aged;
  unsafe fn change_lifetime(self) -> Self::Aged { ... }
}
```

This is using an *associated type* T::Aged to represent $T[\alpha/\beta]$. In particular, JSManaged$\langle\beta, C, T\rangle$ implements JSLifetime$\langle\alpha\rangle$ as long as T does, and JSManaged$\langle\beta, C, T\rangle[\alpha/\beta]$ is JSManaged$\langle\alpha, C, T[\alpha/\beta]\rangle$.

The implementation of JSLifetime$\langle\alpha\rangle$ has a lot of boilerplate, but fortunately that boilerplate is amenable to Rust's metaprogramming system, so user-defined types can just mark their type as `#[derive(JSLifetime)]`.

3 INTERFACING TO THE GARBAGE COLLECTOR

Interfacing to the Spidermonkey GC has two parts:

- *tracing*: from a JS-managed reference, find the JS-managed references immediately reachable from it, and
- *rooting*: find the JS-managed referenes which are reachable from the stack.

From these two functions, it is possible to find all of the JS-managed references which are reachable from Rust. Together with Spidermonkey's regular GC, this allows the runtime to find all of the reachable JS objects, and then to reclaim the unreachable ones.

These interfaces are important for safety, since under-approximation can result in use-after-free, and over-approximation can result in space leaks.

In this section, we discuss how Josephine supports these interfaces.

3.1 Tracing

Interfacing to the Spidermonkey tracer via Josephine is achieved by implementing a trait:

```
pub unsafe trait JSTraceable {
  unsafe fn trace(&self, trc: *mut JSTracer);
}
```

Josephine provides an implementation:

```
unsafe impl<'a, C, T> JSTraceable for JSManaged<'a, C, T> where
  T: JSTraceable { ... }
```

User-defined types can then implement the interface by recursively visiting fields, for example:

```
unsafe impl<'a, C, T> JSTraceable for NativeCell<'a, C> {
  unsafe fn trace(&self, trc: *mut JSTracer) {
    self.prev.trace(trc);
    self.next.trace(trc);
  }
}
```

This is a lot of unsafe boilerplate, but fortunately can also be mechanized using meta-programming by marking a type as `#[derive(JSTraceable)]`.

One subtlety is that during tracing data of type T, the JS runtime has a reference of type $\&T$ given by the `self` parameter to `trace`. For this to be safe, we have to ensure that there is no mutable reference to that data. This is maintained by the previously mentioned invariant:

Any operation that can trigger garbage collection requires mutable access to the JS context.

Tracing is triggered by garbage collection, and so had unique access to the JS context, so there cannot be any other live mutable access to any JS-managed data.

3.2 Rooting

In languages with native support for GC, rooting is supported by the compiler, which can provide metadata for each stack frame allowing it to be traced. In languages like Rust that do not have a native GC, this metadata is not present, and instead rooting has to be performed explicitly.

This explicit rooting is needed whenever an object is needed to outlive the borrow of the JS context that produced it. For example, a function to insert a new cell after an existing one is:

```
pub fn insert<C, S>(cell: Cell<C>, data: String, cx: &mut JSContext<S>) where
  S: CanAccess + CanAlloc + InCompartment<C>,
  C: Compartment,
{
  let old_next = cell.borrow(cx).next;
  let new_next = cx.manage(NativeCell {
    data: data,
    prev: Some(cell),
    next: old_next,
  });
  cell.borrow_mut(cx).next = Some(new_next);
  if let Some(old_next) = old_next {
    old_next.borrow_mut(cx).prev = Some(new_next);
  }
}
```

This code is the "code you would first think of" for inserting an element into a doubly-linked list, but is in fact not safe because the local variables `old_next` and `new_next` have not been rooted. If GC were triggered just after `new_next` was created, then it could be reclaimed, causing a later use-after-free.

Fortunately, Josephine will catch these safety problems, and report errors such as:

```
error[E0502]: cannot borrow `*cx` as mutable because
  it is also borrowed as immutable
```

```
    |
    |         let old_next = self.borrow(cx).next;
    |                                   -- immutable borrow occurs here
    |         let new_next = cx.manage(NativeCell {
    |                        ^^ mutable borrow occurs here
...
    |     }
    |     - immutable borrow ends here
```

The fix is to explictly root the local variables. In Josephine this is:

```
let ref mut root1 = cx.new_root();
let ref mut root2 = cx.new_root();
let old_next = (... as before ...).in_root(root1);
let new_next = (... as before ...).in_root(root2);
```

The declaration of a `root` allocates space on the stack for a new root, and `managed.in_root(root)` roots `managed`. Note that it is just the reference that is copied to the stack, the JS-managed data itself doesn't move. Roots have type $\mathsf{JSRoot}\langle\beta, T\rangle$ where β is the lifetime of the root, and T is the type being rooted.

Once the local variables are rooted, the code typecheck, because rooting changes the lifetime of the JS-managed data, for example:

if $p : \mathsf{JSManaged}\langle\alpha, C, T\rangle$
and $r : \&\beta\ \mathsf{mut}\ \mathsf{JSRoot}\langle\beta, \mathsf{JSManaged}\langle\beta, C, T[\beta/\alpha]\rangle\rangle$
then $p.\mathsf{in\_root}(r) : \mathsf{JSManaged}\langle\beta, C, T[\beta/\alpha]\rangle$.

Before rooting, the JS-managed data had lifetime α, which is usually the lifetime of the borrow of the JS context that created or accessed it. After rooting, the JS-managed data has lifetime β, which is the lifetime of the root itself. Since roots are considered reachable by GC, the contents of a root are guaranteed not to be GC'd during its lifetime, so this rule is sound.

4 RELATED WORK

Mixed linear/non-linear.

L3.

https://arxiv.org/abs/1803.02796

Rust+GC

Servo

5 FUTURE WORK

REFERENCES

[1] A. Ahmed, M. Fluet, and G. Morrisett. L3: A linear language with locations. *Fundamenta Informaticae*, 77(4):397–449, 2007.

[2] A. Beingessner et al. *Learning Rust With Entirely Too Many Linked Lists.* http://cglab.ca/ abeinges/blah/too-many-lists/book/.

[3] N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. In *Proc. Computer Science Logic*, pages 121–135, 1995.

[4] P. N. Benton, G. M. Bierman, V. de Paiva, and M. Hyland. A term calculus for intuitionistic linear logic. In *Proc. Int. Conf. Typed Lambda Calculi and Applications*, pages 75–90, 1993.

[5] The Rust Project Developers. *The Rust Programming Language.* 2011. https://doc.rust-lang.org/book.

[6] The Rust Project Developers. *The Rustinomicon.* 2011. https://doc.rust-lang.org/beta/nomicon/.

[7] M. Fluet, G. Morrisett, and A. Ahmed. Linear regions are all you need. In *Proc. European Symp. Programming*, pages 7–21, 2006.

[8] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[9] A.S.A. Jeffrey. Josephine, 2017. https://github.com/asajeffrey/josephine.
[10] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proc. USENIX*, pages 275–288, 2002.
[11] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* 2nd edition, 1988.
[12] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960.
[13] Spidermonkey. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.
[14] Servo, the parallel browser engine project. https://servo.org/.
[15] M. Tofte and J.-P. Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.
[16] D. Walker. Substructural type systems. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 3–43. MIT Press, 2002.