Rusty Links in Local Chains[⋆]

James Noble[1][0000−0001−9036−5692], Julian Mackay[2][0000−0003−3098−3901], and Tobias Wrigstad[3][0000−0002−4269−5408]

[1] Creative Research & Programming, Darkest Karori, `kjx@acm.org`
[2] Julian Mackay, Victoria University of Wellington, `Julian.Mackay@ecs.vuw.ac.nz`
[3] Tobias Wrigstad, Uppsala University, `tobias.wrigstad@it.uu.se`

Abstract. Rust successfully applies ownership types to control memory allocation. This restricts the programs' topologies to the point where doubly-linked lists cannot be programmed in Safe Rust. We sketch how more flexible "local" ownership could be added to Rust, permitting multiple mutable references to objects, provided each reference is bounded by the object's lifetime. To maintain thread-safety, locally owned objects must remain thread-local; to maintain memory safety, local objects can be deallocated when their owner's lifetime expires.

Keywords: Rust · Ownership · Linked Lists.

1 Rusty Links

Rust [23, 19, 24] is well-known as a language that combines control of memory use, safe concurrency, and excellent compiler error messages. Rust achieves this balance thanks to a version of *ownership types* [13, 33, 12] (also known in the literature as "*ownership types*" [22, 29]) which statically track the lifetime (or owner) of each allocated object; when an object goes out of scope, all the memory owned by that object is deallocated. So far, so C++ [32], but Rust's ownership types ensure that programs remain memory safe, so really not C++. Rust then incorporates borrowing [7] and fractional permissions [8] to support an integral multiple-reader/single-writer concurrency model [25]: at any time, an object may either be accessed by multiple read-only aliases, or by a single read-write reference.

Many programmers find Rust hard to learn and to use correctly [1, 4, 30, 22, 31]. This is because Rust's ownership types are necessarily conservative, banning not just all concurrent programs that are *actually* unsafe, but a large number of safe programs as well. Rust's version of ownership types [23] bans common idioms such as circular or doubly-linked lists, to the point where the difficulty of implementing a data structure often taught at first year has now become an Internet trope [3, 27, 17, 9]. To programmers, this manifests as a large number of *false positive* errors or warnings about problems that will never arise in practice. A number of solutions have been proposed for these problems, including incorporating a garbage collector [15], careful library design [2], phantom types [34], or proving unsafe Rust code correct [21, 20].

⋆ Supported by the Royal Society of New Zealand Marsden Grant, and Agoric Inc..

2 Local Chains

We propose to solve this problem with *thread local* ownership. Rust's type system currently supports two kinds of borrowing of a variable `v`. Writing "`& foo`" gains readonly access to `v`, which allows multiple aliasing; while writing "`&mut v`" grants read/write access to only one active alias. For example, we can establish two active readonly references to a variable `v` but we cannot write through either reference, even though the underlying variable is mutable:

```
let mut v : i32 = 12;

let a = &v;
let b = &v;
println!("{:#?}", a);  //read a
println!("{:#?}", b);  //read b
//*a = 45; //a is not mutable, cannot write
```

Alternatively, we can establish one mutable reference to `v` through which we can change `v`'s value:

```
let c = &mut v;
//let d = &mut v; //cannot borrow ‘v‘ as mutable more than once
*c = 45;
println!("{:#?}", c);
```

We propose to add a third kind of borrowing — local ownership — which permits both aliases and mutability. We can establish multiple local references to `v` by writing "`&loc v`" and can change `v`'s value through all of them:

```
let loc v : i32 = 12;

let e = &loc v;
let f = &loc v; // two local read/write borrows
*e = 67;
*f = 76;
println!("{:#?}", e);   println!("{:#?}", f);
```

These local aliases should be enough to support chains of mutable objects. To be safe, local objects can only be accessed locally: they cannot be shared or moved, and must remain within one thread. Rust's ownership deallocates objects whenever they go out of scope. Because local objects can be internally aliased, we cannot deallocate them individually: rather we must deallocate all the local objects in one operation at the end of their *owner's* scope. We can explore per scope memory allocation patterns: fixed size and extensible arenas, reference counting, and even garbage collection, as e.g. in Real-Time Java [28, 6], with extensions to finer-grained scopes, alias analysis, and safe *manual* memory management. Finally, we hope this approach could inform (and be informed by) formal techniques for other "Rust-like" languages such as Pony [14], Encore [10], Deterministic Parallel Java [5], Obsidian [16], Dala [18], and Verona [11].

3 Fearless Symmetries

We can illustrate `&loc` drawing on examples suggested by a recent paper [26]. Singly linked lists can be implemented relatively straightforwardly in Rust: each node's `next` pointer owns the subsequent nodes — modulo `Option` to support potentially null values, and `Box` to ensure heap allocation.

```
struct Node<'d> {
    elem: &'d mut Data,
    next: Link<'d>,
}
type Link<'d> = Option<Box<Node<'d>>>;

pub struct Data { item: i32, }
```

Even a relatively simple operation — here CDRing down the list, incrementing each element by 10, and printing out adjacent pairs of elements — can easily run foul of Rust's ownership type system, aka the fearsome "borrow checker". Compiling something like this:

```
fn cdr_down<'d> (n : &mut Link<'d>) -> i32 {
   let mut sum = 0;
   let mut cursor = n;
   let mut precsr = &None;
   while cursor.is_some() {
       cursor.as_mut().unwrap().elem.item += 10;
       sum += cursor.as_ref().unwrap().elem.item;
       precsr = cursor;
       cursor = &mut cursor.as_mut().unwrap().next;
       if precsr.is_some() {
         println!("precsr: {:#?}  cursor: {:#?}",
           precsr.as_ref().unwrap().elem.item,
           cursor.as_ref().unwrap().elem.item,
       );
    }
   };
   sum
}
```

results in the informative error message below, essentially because we have two cursors going down the list at the same time — `cursor` pointing to the current link, and `precsr` pointing to the previous link.

```
error[E0502]: cannot borrow '*cursor' as mutable
because it is also borrowed as immutable
| precsr = cursor;
|          ------ immutable borrow occurs here
| cursor = &mut cursor.as_mut().unwrap().next;
|              ^^^^^^^^^^^^^^^ mutable borrow occurs here
| if precsr.is_some() {
|    ------ immutable borrow later used here
```

It is possible to fiddle with the types to get different error messages, but there is no way to avoid either a "cannot borrow" error when creating mutable aliases, or a "cannot assign" error when writing through a shared alias. Our `&loc` local references should be able to support these kind of functions. The code below shows how all the various references modes could be changed to `&loc`; this will allow multiple mutable references to be used in the same scope. This should be safe even if a `&mut` reference is passed to the function (c.f. borrowing in $\mathcal{K}$ [10]).

```
fn cdr_down<'d> (n : &loc Link<'d>) -> i32 {
   ...
   while cursor.is_some() {
       cursor.as_loc().unwrap().elem.item += 10;
       sum += cursor.as_loc().unwrap().elem.item;
       precsr = cursor;
       cursor = &loc cursor.as_loc().unwrap().next;
       if precsr.is_some() {
         println!("precsr: {:#?}  cursor: {:#?}",
           precsr.as_loc().unwrap().elem.item,
           cursor.as_loc().unwrap().elem.item,);
```

Safe Rust famously cannot support doubly-linked lists [3]: setting list `head` and `tail` pointers to the same node will necessarily produce "cannot borrow" errors:

```
error[E0499]: cannot borrow `*new_node` as mutable more than once
|         self.head = new_node;
|         --------------------
|         |           |
|         |           first mutable borrow occurs here
|         assignment requires `*new_node` is borrowed for `'owner`
|         self.tail = new_node;
|                     ^^^^^^^^ second mutable borrow occurs here
```

Altering field declarations to use local references with an appropriate lifetime should resolve these issues quite straightforwardly — although without garbage collection, all allocated links will be deallocated at the end of the List's lifetime:

```
pub struct List<'owner> {
    head: &'owner loc Node,
    tail: &'owner loc  Node,
}
```

At this point, local references are nothing more than a proposal. Although the general theory of ownership types is well established [13], integration into Rust will involve dealing with all the fine details of Rust's existing ownership models [29], and ultimately a prototype implementation in the Rust compiler.

Acknowledgements

Thanks to David J. Pearce and Elisa Castegren for discussions, and to Amanda Stjärna and the anonymous reviewers for comments on drafts.

This work is supported in part by the Royal Society of New Zealand Te Apārangi Marsden Fund Te Pūtea Rangahau a Marsden, and Agoric.

References

1. Abtahi, P., Dietz, G.: Learning Rust: How experienced programmers leverage resources to learn a new programming language. In: CHI Extended Abstracts. pp. 1–8 (2020)
2. Beingessner, A.: You can't spell Trust without Rust. Master's thesis, Computer Science, Carleton University (2015)
3. Beingessner, A.: Learn Rust with entirely too many linked lists. `https://-rust-unofficial.github.io/too-many-lists` (Mar 2019), accessed April Fools Day 2022
4. Blaser, D.: Simple explanation of complex lifetime errors in Rust (2019), ETH Zürich
5. Bocchino, R., Heumann, S., Honarmand, N., Adve, S., Adve, V., Welc, A., Shpeisman, T.: Safe Nondeterminism in a Deterministic-by-Default Parallel Language. In: POPL (2011)
6. Bollella, G., Canham, T., Carson, V., Champlin, V., Dvorak, D.L., Giovannoni, B., Indictor, M.B., Meyer, K., Murray, A., Reinholtz, K.: Programming with non-heap memory in the real time specification for Java. In: OOPSLA Companion. pp. 361–369 (2003)
7. Boyland, J.: Alias burying: Unique variables without destructive reads. Software: Practice & Experience **31**(6) (May 2001)
8. Boyland, J.: Checking interference with fractional permissions. In: Static Analysis Symposium. pp. 55–72 (2003)
9. Cameron, N.: What's the "best" way to implement a doubly-linked list in Rust? `http://featherweightmusi,ngs.blogspot.com/2015/04/graphs-in-rust.html` (Apr 2015), accessed April Fools Day 2022
10. Castegren, E., Tobias Wrigstad: Reference capabilities for concurrency control. In: ECOOP (2016)
11. Chisnall, D., Parkinson, M., Clebsch, S.: Project Verona (2021), `www.microsoft.com/en-us/research/project/project-verona`
12. Clarke, D., Östlund, J., Sergey, I., Tobias Wrigstad: Ownership types: A survey. In: Aliasing in Object-Oriented Programming. Types, Analysis and Verification, LNCS, vol. 7850 (2013)
13. Clarke, D., Potter, J.M., James Noble: Ownership types for flexible alias protection. In: OOPSLA (1998)
14. Clebsch, S., et al.: Deny capabilities for safe, fast actors. In: AGERE. pp. 1–12 (2015)
15. Coblenz, M., Mazurek, M.L., Hicks, M.: Does the bronze garbage collector make Rust easier to use? A controlled experiment. In: ICSE (2022)
16. Coblenz, M.J., Aldrich, J., Myers, B.A., Sunshine, J.: Can advanced type systems be usable? an empirical study of ownership, assets, and typestate in Obsidian. OOPSLA (2020)

17. Cohen, R.: Why writing a linked list in (safe) Rust is so damned hard. `https://rcoh.me/posts/rust-linked-list-basically-impossible/` (Feb 2018), accessed April Fools Day 2022
18. Fernandez-Reyes, K., Gariano, I.O., James Noble, Greenwood-Thessman, E., Homer, M., Tobias Wrigstad: Dala: A simple capability-based dynamic language design for data race-freedom. In: Onward! (2021)
19. Hu, V.: Rust breaks into TIOBE top 20 most popular programming languages (Jun 2020), infoQ
20. Jung, R., Dang, H.H., Kang, J., Dreyer, D.: Stacked borrows: An aliasing model for Rust. In: POPL (2019)
21. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: Rustbelt: Securing the foundations of the rust programming language. PACMPL **2**(POPL), 66:1–66:34 (Jan 2017)
22. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: Safe Systems Programming in Rust: The Promise and the Challenge. Communications of the ACM (2020)
23. Klabnik, S., Nichols, C.: The Rust Programming Language. 2nd edn. (2018)
24. Krill, P.: Microsoft forms Rust language team (Feb 2021), infoWorld
25. Lea, D.: Concurrent Programming in Java. Addison-Wesley, 2nd edn. (Dec 1998)
26. Milano, M., Turcotti, J., Myers, A.C.: A flexible type system for fearless concurrency. In: PLDI (2022)
27. ndrewxie: What's the "best" way to implement a doubly-linked list in Rust? `https://users.rust-lang.org/t/-whats-the-best-way-to-implement-a-doubly-linked-list-in-rust/27899/7` (Mar 2019), accessed April Fools Day 2022
28. Noble, J., Weir, C.: Small Memory Software: Patterns for systems with limited memory. Addison-Wesley (2000)
29. Pearce, D.J.: A lightweight formalism for reference lifetimes and borrowing in Rust. TOPLAS **43**(1) (2021)
30. Qin, B., Chen, Y., Yu, Z., Song, L., Zhang, Y.: Understanding memory and thread safety practices and issues in real-world Rust programs. In: PLDI. pp. 763–779 (2020)
31. Spencer, R.J.: Four ways to avoid the wrath of the borrow checker (2020), justanotherdot.com
32. Stroustrup, B.: The C++ Programming Language (1986)
33. James Noble, Potter, J., Vitek, J.: Flexible alias protection. In: ECOOP (Jul 1998)
34. Yanovski, J., Dang, H., Jung, R., Dreyer, D.: GhostCell: separating permissions from data in Rust. In: ICFP (2021)