An Incremental Approach to the Semantics of Borrowing

Brianna Marshall, Andrew Wagner, John Li, Olek Gierczak, Amal Ahmed
Northeastern University
Boston, USA

1 Introduction

Linear type systems are useful for managing ownership of resources, but they can be cumbersome to use. Oftentimes, a developer will temporarily give a resource to a function, and expect it back when the function is done using it. In a linear type system, the resource must be explicitly threaded through and returned by the function.

Rust [6] provides a compromise by allowing a developer to *borrow* a resource before passing it to a function. The function has access to the resource as if it was given it directly, but there is no need for it to return it when it's done. The original owner simply gets it back automatically. This is enforced by a *lifetime*, which is part of the type of the borrowed resource. The lifetime prevents the borrow from being used past a certain point, after which the original owner regains full ownership of the resource.

The semantics and typing of borrowing in Rust are complicated, with a subtle interplay between ownership, immutable and mutable borrows, and lifetimes. In this work, to elucidate the semantics of borrowing, we build core features of borrowing from the ground up. We use a linear variant of call-by-push-value [5] as a starting point, in which we only support ownership of resources. We increase the flexibility of the language in stages, adding different forms of borrows, and then adding lifetimes.

We present a semantic model, a domain-specific separation logic, and a type system for each stage of our borrowing system, and prove semantic type soundness of each using the reasoning power of the logics. As a corollary of our semantic soundness result, we have that every well typed program in each of the borrow calculi is terminating and does not leak memory. Finally, we also show how our separation logic can be used to reason beyond typed programs and validate programs that safely break and reestablish invariants.

2 A Spectrum of Borrowing Features

The core objective of borrowing is to relax the restrictions of linear type systems without losing the benefits of precise resource management. We show how this may be accomplished gradually, by distilling borrowing into a handful of incremental modifications to a traditional linear type system.

HOPE, September 2, 2024, Milan, Italy
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM

Linear Types. The two characteristic features of linear type systems are (1) that axioms (e.g., for variables) are precise about the variables in context; and (2) that contexts are split among premises in rules with multiple sub-terms (e.g., let).

$$\textsc{id}\ \ x : V \vdash x : V \qquad \textsc{let}\ \ \frac{\Gamma_1 \vdash e_1 : \mathsf{Ret}\, V \quad \Gamma_2, x : V \vdash e_2 : C}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\, x = e_1;\ e_2 : C}$$

Taken together, these features can prevent aliasing bugs and the leakage of resources. However, they can also be cumbersome. For example, from a file library, we might expect to find the function $\mathsf{size} : \mathsf{File} \multimap \mathsf{Ret}\,\mathsf{Nat}$, but instead we would find $\mathsf{size} : \mathsf{File} \multimap \mathsf{Ret}\,(\mathsf{Nat} \otimes \mathsf{File})$. Not only does this signature require the use of an awkward *capability-passing style* [1, 2], it also obscures the intended behavior: as a client, we cannot assume that the input file is unmodified, nor that the output file is actually the same as the input file.

Downward-Only, Immutable Borrows. Instead, we would like to be able to temporarily mark a variable as immutable, during which time it can be freely aliased and discarded, but after which ownership is reinstated. While this idea has had various incarnations [4, 7, 8, 10, 12], we take inspiration from Perceus [11] and use an unrestricted *borrowable context* Δ to mark such variables. In the let rule below, since e_1 runs before e_2, it can borrow the variables that e_2 owns. To ensure that e_2 regains ownership of its variables before it runs, the return type of e_1 may not contain any borrows. Moreover, thunks cannot close over any borrows, since it may be forced after the borrows have ended.

$$\textsc{let}\ \ \frac{\Delta, \Gamma_2; \Gamma_1 \vdash e_1 : \mathsf{Ret}\, V \quad \Delta; \Gamma_2, x : V \vdash e_2 : C \quad V\ \text{owned}}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathsf{let}\, x = e_1;\ e_2 : C}$$

$$\textsc{brw}\ \ \frac{\Delta \ni x : V}{\Delta; \varnothing \vdash \mathsf{brw}\, x : \mathsf{Brw}\ V} \qquad \textsc{thunk}\ \ \frac{\varnothing; \Gamma \vdash e : C}{\Delta; \Gamma \vdash \mathsf{thunk}\, e : \mathsf{Thunk}\ C}$$

Using borrows, we can improve the earlier signature to $\mathsf{size} : \mathsf{Brw}\ \mathsf{File} \multimap \mathsf{Ret}\,\mathsf{Nat}$. However, borrows are effectively second-class values, since they may not be returned from a sequenced computation nor closed over in thunks.

Lifetimes. To overcome this limitation, every borrow is annotated with a *lifetime* during which the borrow is allowed to escape and after which it may not. When variables become borrowable (see LET), they are annotated with an upper bound on how long they may be borrowed: below,

$\Gamma_2[l]$ annotates every binding with l. Subsequently, any borrow of one of these variables (see BRW) is taken at a strictly shorter lifetime, $l' < l$. Additionally, a thunk is annotated with a lower bound on the lifetimes it closes over. Together, these annotations ensure that it is sufficient to examine a value's return type in order to determine whether it is allowed to escape. This demonstrates one advantage of using call-by-push-value: in a call-by-value calculus, an annotation is required on every negative type (e.g., functions, additive/lazy products), and in a call-by-name calculus, an annotation is required on every type.

$$\textsc{let}\ \frac{\Delta, \Gamma_2[l]; \Gamma_1 \vdash e_1 : \mathsf{Ret}\, V \quad \Delta; \Gamma_2, x : V \vdash e_2 : C \quad V > l}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathsf{let}\, x = e_1;\ e_2 : C}$$

$$\textsc{brw}\ \frac{\Delta \ni x : V[l] \quad l' < l}{\Delta; \varnothing \vdash \mathsf{brw}\, x : \mathsf{Brw}_{l'}\, V} \qquad \textsc{thunk}\ \frac{\Delta; \Gamma \vdash e : C \quad \Delta, \Gamma \geq l}{\Delta; \Gamma \vdash \mathsf{thunk}\, e : \mathsf{Thunk}\, l\, C}$$

Mutable Borrows. The borrowing machinery introduced so far works just as well for mutable borrows, or resources that can be temporarily updated but not freed. However, mutable borrows may not be aliased. To resolve this, the only additional restriction needed is that the mutable borrowable context Δ_m is split between sub-terms, as usual. Below we now use Δ_i for the immutable borrowable context.

$$\textsc{let}\ \frac{\begin{array}{c}\Delta_i, \Gamma_2^i[l]; \Delta_m^1, \Gamma_2^m[l]; \Gamma_1 \vdash e_1 : \mathsf{Ret}\, V \\ \Delta_i; \Delta_m^2; \Gamma_2^1, \Gamma_2^2, x : V \vdash e_2 : C \quad V > l\end{array}}{\Delta_i; \Delta_m^1, \Delta_m^2; \Gamma_1, \Gamma_2^i, \Gamma_2^m \vdash \mathsf{let}\, x = e_1;\ e_2 : C}$$

Lifetime Polymorphism. So far, functions over borrows will only work at particular lifetimes, but in many cases the particular lifetimes are less important than their relationship to other lifetimes. To make the system more flexible, we additionally allow bounded quantification over lifetimes, where the bounds are tracked in a constraint context.

3 A Logical Approach to Borrowing

For each of our calculi, we prove that the type system is sound, ensuring both termination and the absence of memory leaks. We do so using semantic type soundness: types are interpreted into a logical relation over domain-specific separation logic predicates, and we prove the fundamental theorem using this logic.

In a standard separation logic, the *frame rule* can be used to temporarily ignore a resource R for the duration of an expression:

$$\textsc{Frame}\ \frac{\{P\}\, e\, \{R\}}{\{P \star Q\}\, e\, \{Q \star R\}}$$

As originally developed by Charguéraud and Pottier [3], instead of *ignoring* Q completely, Q can be temporarily "downgraded" to an immutable borrow using the *borrow frame rule*:

$$\textsc{Brw-Frame-IDO}\ \frac{\{P \star \mathsf{Brw}\, Q\}\, e\, \{R\} \quad R \text{ owned}}{\{P \star Q\}\, e\, \{Q \star R\}}$$

Such a rule can be used to validate the soundness of the downward-only, immutable let typing rule: the interpretation of the newly borrowable context will be borrow-framed in the premise. Notice that the post-condition R is required to be owned (i.e., must always be satisfied without borrows), matching the side-condition in the typing rule. A $\mathsf{Brw}\ Q$ proposition is always duplicable even when Q is not, and it only confers immutable access to the resource satisfying Q, as expected. Following the changes to the type system, this rule can be updated to support lifetimes and mutable borrows:

$$\textsc{Brw-Frame}\ \frac{\{P \star \mathsf{Brw}_l^m\, Q\}\, e\, \{R\} \quad Q, R > l}{\{P \star Q\}\, e\, \{Q \star R\}}$$

Taking inspiration from the anti-frame rule of Pottier [9], we also show that a mutable borrow can be temporarily "upgraded" to full ownership using the *mutable borrow anti-frame rule*. Given a mutable borrow of a proposition, which can be thought of as a temporary invariant, we are permitted to "open" the invariant so long as we re-establish it again in the post-condition.

$$\textsc{Mut-Brw-Anti-Frame}\ \frac{\{P \star Q\}\, e\, \{Q \star R\}}{\{P \star \mathsf{Brw}_l^{\mathsf{mut}}\, Q\}\, e\, \{R\}}$$

4 Semantic Model

Soundness of the logic is established using a Kripke resource model that distinguishes between what is owned, immutably borrowed, and mutably borrowed. The owned fragment is simply an exclusive heap, as is standard in separation logic. The immutable fragment contains shareable, individual resources, which represent the state "frozen" at the time of the borrow. The mutable fragment contains exclusive predicates, which represent the invariants that must hold during the borrow. Even though the model supports higher-order store, the substructural discipline prevents cycles in the heap, and we manage to prove termination by using a novel stratification measure in our semantic models.

5 Conclusion

This is work in progress. The languages, type systems, operational semantics, and logics are fully developed. The logical relations are defined and we have proved a large part of the fundamental property using the logic. We are currently working on establishing the soundness of the logic using our semantic model. We expect to have completed all the work and submitted a conference paper prior to HOPE in September. We would like a 30-minute slot for the talk.

References

[1] Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007. L3: a linear language with locations. *Fundamenta Informaticae* 77, 4 (2007), 397–449.

[2] Arthur Charguéraud and François Pottier. 2008. Functional translation of a calculus of capabilities. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. 213–224.

[3] Arthur Charguéraud and François Pottier. 2017. Temporary read-only permissions for separation logic. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings 26*. Springer, 260–286.

[4] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34.

[5] Paul Blain Levy. 1999. Call-by-Push-Value: A Subsuming Paradigm. In *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA '99)*. Springer-Verlag, Berlin, Heidelberg, 228–242.

[6] Nicholas D Matsakis and Felix S Klock. 2014. The rust language. *ACM SIGAda Ada Letters* 34, 3 (2014), 103–104.

[7] Liam O'Connor, Christine Rizkallah, Zilin Chen, Sidney Amani, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Alex Hixon, Gabriele Keller, Toby Murray, et al. 2016. COGENT: certified compilation for a functional systems language. *arXiv preprint arXiv:1601.05520* (2016).

[8] Martin Odersky. 1992. Observers for linear types. In *European Symposium on Programming*. Springer, 390–407.

[9] François Pottier. 2008. Hiding local state in direct style: a higher-order anti-frame rule. In *2008 23rd Annual IEEE Symposium on Logic in Computer Science*. IEEE, 331–340.

[10] Gabriel Radanne, Hannes Saffrich, and Peter Thiemann. 2020. Kindly bent to free us. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–29.

[11] Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: Garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 96–111.

[12] Philip Wadler. 1990. Linear types can change the world!. In *Programming concepts and methods*, Vol. 3. Citeseer, 5.