From Linearity to Borrowing

ANDREW WAGNER, Northeastern University, USA
OLEK GIERCZAK, Northeastern University, USA
BRIANNA MARSHALL, Northeastern University, USA
JOHN M. LI, Northeastern University, USA
AMAL AHMED, Northeastern University, USA

Linear type systems are powerful because they can statically ensure the correct management of resources like memory, but they can also be cumbersome to work with, since even benign uses of a resource require that it be explicitly threaded through during computation. *Borrowing*, as popularized by Rust, reduces this burden by allowing one to temporarily disable certain resource permissions (e.g., deallocation or mutation) in exchange for enabling certain structural permissions (e.g., weakening or contraction). In particular, this mechanism spares the borrower of a resource from having to explicitly return it to the lender but nevertheless ensures that the lender eventually reclaims ownership of the resource.

In this paper, we elucidate the semantics of borrowing by starting with a standard linear type system for ensuring safe manual memory management in an untyped lambda calculus and gradually augmenting it with immutable borrows, lexical lifetimes, reborrowing, and finally mutable borrows. We prove semantic type soundness for our Borrow Calculus (`BoCa`) using Borrow Logic (*BoLo*), a novel domain-specific separation logic for borrowing. We establish the soundness of this logic using a semantic model that additionally guarantees that our calculus is terminating and free of memory leaks. We also show that our Borrow Logic is robust enough to establish the semantic safety of some syntactically ill-typed programs that temporarily break but reestablish invariants.

CCS Concepts: • **Theory of computation → Semantics and reasoning**; **Separation logic**.

Additional Key Words and Phrases: borrowing, linear types, semantics, logical relations, separation logic

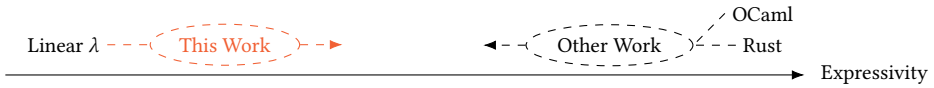
ACM Reference Format:
Andrew Wagner, Olek Gierczak, Brianna Marshall, John M. Li, and Amal Ahmed. 2025. From Linearity to Borrowing. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 415 (October 2025), 27 pages. https://doi.org/10.1145/3764117

1 Introduction

Linear typing is a powerful tool that can rule out a large class of memory bugs, such as use-after-free and memory leaks. But as with many type systems, it is negatively expressive [6, 22] and rejects many programs that, for example, alias pointers in a completely safe way. Moreover, traditional linear type systems require the use of a verbose capability-passing style that pervades APIs. Early developments of linear type systems recognized these limitations and devised ad-hoc ways to temporarily relax linearity [20, 30]. Decades later, the Rust programming language [17] has refined and popularized a particular strategy called *borrowing*. A *borrow* temporarily disables certain resource permissions (e.g., deallocation or mutation) in exchange for enabling certain structural permissions (e.g., weakening and contraction).

Authors' Contact Information: Andrew Wagner, Northeastern University, Boston, USA, ahwagner@ccs.neu.edu; Olek Gierczak, Northeastern University, Boston, USA, gierczak.o@northeastern.edu; Brianna Marshall, Northeastern University, Boston, USA, marshall.br@northeastern.edu; John M. Li, Northeastern University, Boston, USA, li.john@northeastern.edu; Amal Ahmed, Northeastern University, Boston, USA, amal@ccs.neu.edu.

Existing formalizations of borrowing either start from Rust and simplify toward a core calculus (e.g., [11, 32]), or start from another language and retrofit it with borrows (e.g., OCaml [7, 15], ML [25], Granule [16]). A primary objective of this paper is to instead start from the well-understood discipline of linear typing and show how to ease its restrictions via extensions for borrowing.



Type System. We start with an untyped lambda calculus with primitives for manual memory management and review how linear typing layers a discipline on top that guarantees safety but has limitations. We then gradually extend the system with borrow features that relax those limitations. Our extensions are *lightweight* in the sense that they do not add new borrowing primitives or operational semantics to the underlying lambda calculus and the standard linear typing rules have a straightforward embedding into our type system. The type system is presented in a pedagogical way, beginning from the limitations of the linear lambda calculus and motivating different borrowing features one at a time: immutable borrows, then lexical lifetimes, then reborrowing, then mutable borrows. To ensure that our extensions remain grounded in a familiar linear typing discipline while also avoiding capability-passing style for borrows, we restrict our attention to lexical lifetimes, which do not require threading facts about live lifetimes, neither implicitly in the type system (as in [11, 32]) nor explicitly by the user.

Separation Logic. Since our extensions do not add new primitives for borrowing, all borrowing operations must be expressed in terms of existing constructs of our untyped lambda calculus with memory management. However, these derived borrowing operations cannot be syntactically typed—they are typed axiomatically, which means that we ascribe them types. Therefore, we opt to prove semantic type soundness, so we can validate that such axiomatically-typed code is nonetheless well-behaved at the ascribed types, by interpreting types into a new variation of separation logic with *logical borrowing*. Just as borrowing alleviates capability-passing style in the type system, our Borrow Logic (*BoLo*) abstractions mitigate it in proofs as well; there is no threading of lifetime tokens as in RustBelt's Lifetime Logic [11]. Inspired by Charguéraud and Pottier [4], *BoLo*'s distinguishing rule is a variation of the frame rule called the *borrow frame rule*, which borrows a framing proposition instead of hiding it completely. It also supports a dual rule for accessing and updating mutable borrows which we call the *borrow anti-frame* rule, inspired by Pottier [24]. *BoLo* is a linear separation logic so that it naturally establishes memory reclamation; an affine separation logic (e.g., Iris [12]) would require more bookkeeping (e.g., as in [3]).

Semantic Model. We prove the soundness of our Borrow Logic using a semantic model with which adequacy implies termination and memory reclamation, both novel results for a system with mutable borrowing. Despite their apparent similarity to type-preserving, ML-style mutable references, a surprising feature of this model is that it is not stratified by step indices. Traditionally, semantic models for type-preserving updates associate each location with a semantic type to ensure that the contents of the location continue to behave like that type. Specifically, an ML-style mutable reference must always contain values of that type, while a mutable borrow must contain a value of that type when the borrow ends. Tracking semantic types in the model leads to a circularity that is typically circumvented by stratifying the model using a step-index [2] to ensure well-foundedness. However, step-indexing is traditionally not suitable for establishing liveness properties like termination and memory reclamation. Instead, our model uses a different measure, stratifying by the *lifetimes* of borrows. While there are variations of step-indexing that do

$$
\begin{array}{llll}
\textsc{Var} & \ni & x, y, \ldots & \\
\textsc{Loc} & \ni & \ell & \\
\textsc{Val} & \ni & v & ::= \ () \mid (v_1, v_2) \mid \mathtt{inj}_1\, v \mid \mathtt{inj}_2\, v \mid \lambda x.\, e \mid \ell \mid p \\
\textsc{Prim} & \ni & p & ::= \ \mathtt{alloc} \mid \mathtt{free} \mid \mathtt{load} \mid \mathtt{store} \\
\textsc{Expr} & \ni & e & ::= \ x \mid v \mid (e_1, e_2) \mid \mathtt{inj}_1\, e \mid \mathtt{inj}_2\, e \mid e_1; e_2 \mid \mathtt{let}(x, y) = e_1; e_2 \\
 & & & \quad \mid \mathtt{case}\, e \{\mathtt{inj}_1\, x_1.\, e_1, \mathtt{inj}_2\, x_2.\, e_2\} \mid e_2\, e_1 \\
\textsc{Mem} & \ni & \mu & : \ \textsc{Loc} \rightharpoonup \textsc{Val}
\end{array}
$$

$$
\boxed{(\mu, e) \rightarrowtail (\mu', e')} \qquad (\mu, \mathtt{alloc}\ v) \rightarrowtail (\mu \uplus [\ell \mapsto v], \ell) \qquad (\mu \uplus [\ell \mapsto v], \mathtt{free}\ \ell) \rightarrowtail (\mu, v)
$$

$$
(\mu \uplus [\ell \mapsto v], \mathtt{load}\ \ell) \rightarrowtail (\mu \uplus [\ell \mapsto v], v) \qquad (\mu \uplus [\ell \mapsto v_1], \mathtt{store}\ \ell\ v_2) \rightarrowtail (\mu \uplus [\ell \mapsto v_2], ())
$$

Fig. 1. Syntax and dynamics (excerpts).

support liveness properties [28], our approach takes advantage of the latent stratification inherent to borrowing with lifetimes.

Contributions. We make the following contributions.

- We present BoCa, a lightweight borrowing extension to a linear lambda calculus for safe manual memory management. By lightweight, we mean that the expression syntax and operational semantics of the language are not augmented with new cases for borrowing, and the standard linear type system has a straightforward embedding into the extended BoCa type system. BoCa supports immutable and mutable borrows along with lexical lifetimes, reborrowing, and lifetime polymorphism.
- We develop *BoLo*, a separation logic with new abstractions for *logical borrowing*, which is characterized by two distinguishing rules: the borrow frame rule for introducing logical borrows and the borrow anti-frame rule for updating logical mutable borrows. We use *BoLo* to define a semantic model of types for BoCa, but the logic is useful independent of the type system for verifying that untyped code is well behaved, even if it temporarily breaks borrowing invariants.
- We prove semantic type soundness for BoCa and show termination and memory reclamation using a semantic model that must account for the subtle interaction of owned, immutably borrowed, and mutably borrowed memory. The semantic model uses a novel stratification measure, based on lifetimes.

Complete definitions and proofs may be found in our supplementary material [31].

2 Borrowing from the Ground Up

In this section, we develop a core calculus for borrowing from first principles, beginning with a review of linear typing and its motivations (§ 2.1). We introduce borrowing feature by feature, starting with immutable borrows (§ 2.2), then explicit lifetimes (§ 2.3), then reborrowing (§ 2.4), and finally, mutable borrows (§ 2.5). The final version of the Borrow Calculus (BoCa) developed by the end of this section can be found in our supplementary material [31].

As specified in Fig. 1, we will work with a standard call-by-value variant of the untyped lambda calculus with unit, products, sums, and manually managed memory in the form of four primitives, `alloc`, `free`, `load`, and `store`. For the sake of exposition, some example programs may use additional features whose meaning will be clarified as needed.

```
let x = alloc v;
free x;
load x
```

(a) Buggy: use after free.

```
let x = alloc v;
load x
```

(b) Buggy: memory leak.

```
let x = alloc v;
let(x′, y) = swap x ();
free x′;
y
```

(c) Safe: linearly typed.

Fig. 2. Memory safety.

$$\textsc{Type} \quad \ni \quad \mathsf{T} \quad ::= \quad \mathbf{1} \mid \mathsf{T}_1 \oplus \mathsf{T}_2 \mid \mathsf{T}_1 \otimes \mathsf{T}_2 \mid \mathsf{T}_1 \multimap \mathsf{T}_2 \mid \mathsf{Ref}\ \mathsf{T}$$

$$\boxed{\Gamma \vdash \mathsf{e} : \mathsf{T}} \qquad \otimes\mathrm{I}\ \frac{\Gamma_1 \vdash \mathsf{e}_1 : \mathsf{T}_1 \qquad \Gamma_2 \vdash \mathsf{e}_2 : \mathsf{T}_2}{\Gamma_1, \Gamma_2 \vdash (\mathsf{e}_1, \mathsf{e}_2) : \mathsf{T}_1 \otimes \mathsf{T}_2} \qquad \textsc{var}\ \frac{}{\mathsf{x} : \mathsf{T} \vdash \mathsf{x} : \mathsf{T}} \qquad \mathbf{1}\mathrm{I}\ \frac{}{\vdash () : \mathbf{1}}$$

(a) Linearly typed lambda calculus.

$$\vdash \mathtt{alloc} : \mathsf{T} \multimap \mathsf{Ref}\ \mathsf{T} \qquad \vdash \mathtt{free} : \mathsf{Ref}\ \mathsf{T} \multimap \mathsf{T} \qquad \not\vdash \mathtt{load} : \mathsf{Ref}\ \mathsf{T} \multimap \mathsf{T} \qquad \not\vdash \mathtt{store} : \mathsf{Ref}\ \mathsf{T} \multimap \mathsf{T} \multimap \mathbf{1}$$

$$\frac{\mathtt{swap} \triangleq \lambda \mathsf{x} \lambda \mathsf{y}_2.\, \mathtt{let}\ \mathsf{y}_1 = \mathtt{load}\ \mathsf{x}; \mathtt{store}\ \mathsf{x}\ \mathsf{y}_2; (\mathsf{x}, \mathsf{y}_1)}{\vdash \mathtt{swap} : \mathsf{Ref}\ \mathsf{T}_1 \multimap \mathsf{T}_2 \multimap (\mathsf{Ref}\ \mathsf{T}_2) \otimes \mathsf{T}_1}$$

(b) Linearly typed memory operations (see ¶ Memory Operations).

Fig. 3. Statics (excerpts).

2.1 Review: Linear Typing

In a language with manual memory management, the presence of `free` can easily introduce undefined behavior in the form of *use after free*, as demonstrated in Fig. 2a. It is just as easy to introduce *memory leaks*, as demonstrated in Fig. 2b, which eliminates the use-after-free bug from the previous example but fails to clean up its memory. Both of these issues can be mitigated by a *linear typing* discipline, which ensures the correct management of a resource like memory by enforcing that all data is consumed exactly once. This discipline would have statically rejected the buggy programs in Figs. 2a and 2b: in the first case, the reference is used again after it is consumed by `free`, and in the second case, the reference is never consumed by `free`.

Representative rules from the type system for the linear lambda calculus are given in Fig. 3a. Its distinguishing feature—highlighted via shading—is that it does not support the usual structural rules of weakening and contraction, which correspond to forgetting and duplicating a variable, respectively. Instead, typing contexts are split between sub-derivations, and the leaves in the derivation require minimal typing contexts (e.g., a singleton context for variables and an empty context for base values).

Memory Operations. We must also assign typing rules to the memory operations, given in Fig. 3b. Whereas `alloc` and `free` are straightforward, there is no sensible linear type to assign to `load` and `store`. The natural candidates, $\mathtt{load} : \mathsf{Ref}\ \mathsf{T} \multimap \mathsf{T}$ and $\mathtt{store} : \mathsf{Ref}\ \mathsf{T} \multimap \mathsf{T} \multimap \mathbf{1}$, are not sound for *linear* references because in both cases, the reference is leaked—it is neither freed by the operation nor returned to the caller to be freed elsewhere. Instead, a common approach is to use a `swap` operator, which operationally combines a `load` and a `store`. There are three important features of `swap`. First, it not only returns the payload, but also the input reference itself, threading it through

```
words, lines, avg : Ref File ⊸ ℕ
avg f  = words f / lines f

main () =
  let f = open "file";
  let a = avg f;
  close f;
  a
```

(a) Correct but not linear.

```
words, lines, avg : Ref File ⊸ Ref File ⊗ ℕ
avg f =
  let(f′, w) = words f;
  let(f″, l) = lines f′;
  (f″, w / l)
main () =
  let f = open "file";
  let(f′, a) = avg f;
  close f′;
  a
```

(b) Linear capability-passing style.

Fig. 4. Linearizing an API, where open : Str ⊸ Ref File and close : Ref File ⊸ 1.

the operation so that it can (and indeed, will) be used again. Second, the rule permits changing the payload type of the reference, which is known as a *strong update*. Whereas strong updates would not generally be sound in a language like ML in which references may alias, linearity ensures that the reference is *unique*, so the context cannot impose any type invariant on the payload. Third, note that the implementation of swap, shown in the premise of the typing rule, is ill-typed since it uses its linear argument x three times. Nonetheless, we will show that this implementation is semantically type sound at the axiomatic type ascribed to swap. Using these types, we can confirm that the analogous versions of the buggy programs from earlier do not type check (Figs. 2a and 2b) and that we can construct a correct version (Fig. 2c).

Capbility-Passing Style. Notice that in the correct program, the reference must be explicitly threaded through every operation, which is sometimes referred to *capability-passing style*. Verbosity can always be hidden by macros, like Idris' !-notation [5], but this is *not* just a matter of local syntax—this pattern ends up polluting APIs globally throughout one's program. Consider the program in Fig. 4a, which uses a hypothetical file library to compute the average number of words per line in a file. Even though this program is perfectly safe and cleans up its resources, it is not linearly typed. Making it so, as shown in Fig. 4b, requires not only performing capability passing locally within each function, but also *across* functions (see shaded type), requiring the entire API to be rewritten, including the file library. Not only is this new API more verbose, it also does not obviously capture the intended behavior, since any of these functions can now replace the provided file handle with an entirely different one.

Linear type systems are often presented with an exponential modality, ! T, which allows "plain old data" to be used in an unrestricted manner. While this is a helpful feature, it does *not* alleviate the use of capability-passing style above, since the data of interest (references) is resource-like, not plain-old data, and we want the safety benefits of treating such data linearly. Exponentials are therefore an orthogonal extension that we will not develop in this paper, though they are completely compatible with our type system.

2.2 Immutable Borrows

In the previous section, we saw two operations that assume references do not alias: free, in order to avoid use after free, and swap, in order to allow strong updates. If we were to disable the use of these operations, then it would be safe to duplicate references, which is the principle of *mutability-xor-aliasing*. However, permanently disabling the use of free would reintroduce

```
words, lines, avg : Imm File ⊸ ℕ

main () =
  let f = open "file";
  let a = withbor f (λf'. avg f');
  close f;
  a

avg f =
  let(f1, f2) = dupl f;
  words f1 / lines f2
```

Fig. 5. Rewriting Fig. 4 with immutable borrows.

memory leaks. Instead, we are interested in a way to *temporarily* disable `free` and `swap`, during which time references can be duplicated and forgotten, but after which time references are treated linearly again. A *borrow* temporarily disables certain resource permissions (e.g., deallocation or mutation) in exchange for enabling certain structural permissions (e.g., duplication and forgetting). For the next several sections, we will focus on *immutable borrows*, written `Imm T`, which sacrifice both `free` and `swap` in exchange for `dupl` and `forget`.

$$\texttt{dupl} : \texttt{Imm T} \multimap \texttt{Imm T} \otimes \texttt{Imm T} \qquad \texttt{forget} : \texttt{Imm T} \multimap \mathbf{1}$$

Fig. 5 shows how the example from Fig. 4 can be rewritten in almost the intended way, except that `Ref` is replaced with `Imm`. This rewrite makes use of the `withbor` operation, which temporarily turns a linear reference (`f`) into an immutable borrow (`f'`) for the lexical extent of the function argument, which we call the *borrower*. Like `swap`, we implement the `withbor` operation using the existing constructs of the untyped language, as follows.

$$\texttt{withbor} \triangleq \lambda x \lambda f.\ (x, f\ x)$$

In § 3, we will see how to establish that syntactically ill-typed terms like `withbor` are *semantically* well-typed. For now, it suffices to ask what type we would even want to assign to `withbor`. One might infer from Fig. 5 that it should satisfy the following typing rule.

$$\texttt{withbor} \stackrel{?}{:} \texttt{Ref T}_1 \multimap (\texttt{Imm T}_1 \multimap \texttt{T}_2) \multimap (\texttt{Ref T}_1) \otimes \texttt{T}_2$$

This rule has the right shape, but it is too lenient. In particular, it would allow the identity function to be the borrower, which could be exploited to cause the following use-after-free bug.

$$\texttt{let}(f_1, f_2) = \texttt{withbor (open "file") } (\lambda f.\ f); \texttt{close } f_1; \texttt{ words } f_2$$

A natural strengthening of the rule is to forbid an `Imm` from appearing in the return type of the borrower. Unfortunately, this restriction is necessary but not sufficient. This rule still permits a borrower that captures the borrow in a closure and returns it, which delays use of the data until the closure is called. Since the environment of the closure is opaque, the type system permits a call after a `free`, triggering a use-after-free bug, as the following example demonstrates.

$$\texttt{let}(f, w) = \texttt{withbor (open "file") } (\lambda f'.\ \lambda\_.\ \texttt{words } f'); \texttt{close } f;\ w\ ()$$

Pushing the previous restriction one step further, what is actually required is that the borrow not be allowed to *escape* the borrower, neither explicitly nor by capture. In effect, the rule needs to ensure that the borrow is *temporary*, as was originally proposed. A straightforward way to accomplish this would be to forbid the return type of the borrower from containing `Imm`s and functions. Indeed, Wadler made a similar proposal long before the advent of modern borrowing systems [30]. Returning closures will be useful, so we generalize the intuition behind this restriction

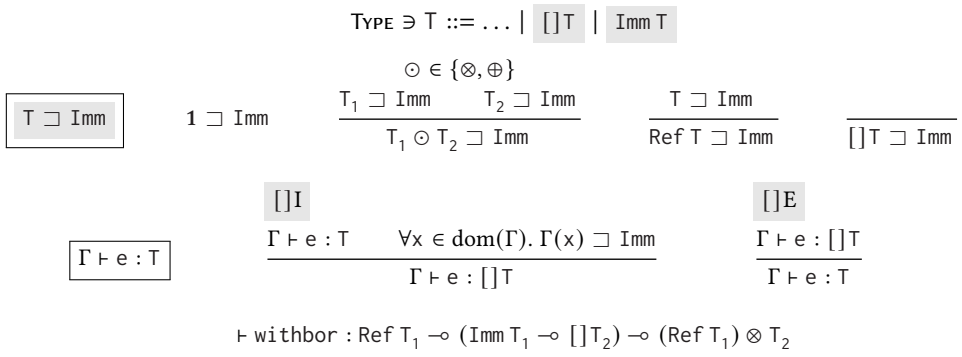


Fig. 6. Typing immutable borrows.

and tag the borrower's return type with a new type constructor, the *outlives modality*, $[]\mathtt{T}$. An inhabitant of $[]\mathtt{T}$ is an inhabitant of $\mathtt{T}$ that *outlives* (i.e., does not use) any borrow.

The introduction form for the outlives modality, given in Fig. 6, ensures that an expression does not use any borrows by scanning the typing context. It uses an auxiliary judgment, $\mathtt{T} \sqsupset \mathtt{Imm}$, which only holds of types $\mathtt{T}$ whose inhabitants definitively *outlive* all `Imm` borrows. In particular, there is no such rule for $\mathtt{Imm}\ \mathtt{T}$, which certainly holds a borrow, nor for functions, which *might* hold borrows and must be guarded by a $[]$. The purpose of the outlives modality is to constrain how terms are *constructed*, not how they are *used*, so it is simply erased in its elimination form. The modality is only significant in the statics and carries no operational content, so the typing rules are applied implicitly (i.e., they are not syntax-directed).

The outlives modality can be used to assign a safe type to `withbor` by ensuring that the borrower returns a value that outlives all borrows, as shown at the bottom of Fig. 6. Doing so ensures that the borrower's return value of type $\mathtt{T}_2$ does not contain any aliases to the newest borrow $\mathtt{Imm}\ \mathtt{T}_1$, which justifies reinstating full ownership of that reference at type $\mathtt{Ref}\ \mathtt{T}_1$. Note that these rules do not yet show how to directly *access* an immutable borrow, which is the subject of § 2.4.

2.3 Lifetimes

The typing rule for `withbor` in the previous section asserts that the borrower must not return *any* borrows. Consider the following example, which nests multiple borrows and attempts to return the outermost borrow from the innermost borrower.

```
login (name, pass: Str) =
  let(ok, ()) = withbor (open "ok.html") (λok′.
    let(err, ()) = withbor (open "err.html") (λerr′.
      let(users, page) = withbor (open "users.csv") (λusers′.
        if (has? users′ name pass) (forget err′; ok′) (forget ok′; err′));
      close users;
      render page)
    close err)
  close ok
```

Even though this program is completely safe, it is rejected by the rule from the previous section. A more precise variant of the outlives restriction would ensure that the borrow created for a *particular* borrower not escape *that* borrower. In order to create such an association between borrows and their borrowers, Fig. 7 extends the type system to associate each borrow with a fresh name, called

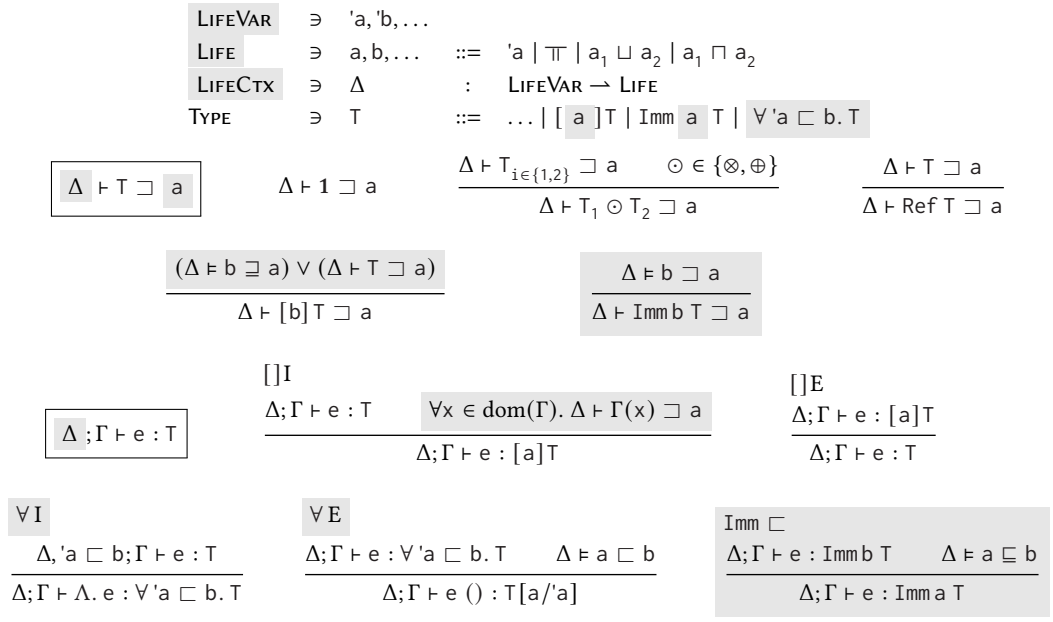


Fig. 7. Typing immutable borrows with explicit lifetimes.

its *lifetime*. Lifetime variables 'a are introduced by a universal quantifier ∀ 'a ⊏ b. T upper bounded by a lifetime b. To track lifetime variables and their bounds, the typing judgment is extended to include an unrestricted lifetime context, Δ. In addition to lifetime variables 'a, lifetimes b include a longest lifetime ⊤ as well as meets ⊓ and joins ⊔, which represent the shortest and longest of their operands, respectively. The length of a lifetime roughly corresponds to the lexical extent of its borrower, and when the scope of a longer lifetime b subsumes the scope of a shorter lifetime a, we say that b outlives a, denoted Δ ⊨ a ⊏ b. All of the rules from the standard linear lambda calculus can be directly embedded into this system by threading this context along, just as one would a type variable context for linear System F. The only operational significance of the universal quantifier is that it is thunked, so its introduction form Λ. e is just shorthand for λ_. e—that is, we reuse λ instead of introducing a new term former.

To associate borrows with their borrowers, we index borrows Imm a T and the outlives modality [a]T by a lifetime. The new introduction form for the outlives modality uses a more granular outlives judgment Δ ⊢ T ⊐ a, which now only forbids the context from holding borrows at lifetime a or shorter. In particular, the judgment now includes a rule for Imm b T when b outlives a. Whereas the new judgment is more permissive for borrows, it is more restrictive for the outlives modality [b]T, which now only outlives a if b or T does. Type well-formedness works more or less the same way as in System F, except that variables here do not stand for types and therefore only appear as indices to other types.

In the revised rule for withbor, the borrower is given a fresh lifetime 'a, which is shorter than all existing lifetimes in Δ, and a borrow Imm 'a T_1 at that lifetime. The result of the borrower, ['a]T_2, outlives 'a. On its own, this rule is not enough to type the example that began this section—the

borrows ok′ and err′ have different lifetimes and therefore different types that cannot be unified. But since ok′ is borrowed before err′, it certainly lives at least as long as err′, so we allow its lifetime to be weakened to match with a coercion rule for borrows (Imm ⊏). This means that given a borrow Imm a T, the lifetime a is really only a lower bound on the "true" lifetime that the borrow was originally assigned. To solve lifetime constraints like Δ ⊨ a ⊏ b, we appeal to the semantic interpretation of lifetimes (§ 3.1), which may be thought of informally as a call to an external solver.

2.4 Reborrowing

So far, we have only seen how to use borrows to program against an API that can inspect the payload of a borrow, but we have not seen how to inspect the payload directly; for example, we cannot case on a boolean under an Imm a 𝔹. Because linear references can be nested under borrows, as in Imm a (Ref 1), using a standard load would not be sound in general, as the following example demonstrates that it could be exploited to free a nested linear reference twice.

```
load : Imm a T ⊸ T
let(x, ()) = withbor (alloc (alloc ())) (Λ. λx′. free(load x′)); free (free x)
```

To prevent such issues, Fig. 8 defines a variant of the load operation called withload, which forbids nested linear references from being accessed at full strength; instead, they will only be accessible as new immutable borrows. As in withbor, withload takes in a function argument that quantifies over fresh lifetimes 'b, which we call the *loader*. Unlike withbor, withload starts with a borrow Imm a T instead of a linear reference, and the loader is given a potentially limited *view* of the payload; in the case of the example above, it provides a new borrow of the nested linear reference. The view of the payload depends on its type; for example, if the payload is "plain old data", like a boolean, then it can be loaded as is.

To capture this dependency, the typing rule for withload uses the metafunction Imm 'b T_1, which maps the payload type T_1 to a view that is safe to load, possibly by *reborrowing* internal references at the fresh lifetime 'b. In some cases, like for computations with capture environments that may hold linear references, there is *no* view at which it would be safe to access the payload, so we map these types to a new, distinguished unknown type, Unk, for which the only operation that is defined is forget. The majority of the clauses for Imm proceed structurally over the type, but two cases warrant particular attention. For reborrowing from the outlives modality Imm 'b ([a] T), the fresh lifetime 'b is shorter than any extant lifetime, including the a indexing the modality, so the modality must be removed, leaving Imm 'b T. For reborrowing immutable borrows Imm 'b (Imm a T), recall that the purpose of the lifetime index a is to prevent it from escaping its borrower, which would enable inconsistent aliasing situations. But in the case of a nested borrow, aliases to the inner borrow Imm a T are allowed to exist anyway; for example, the inner borrow may have been duplicated before the outer borrow was taken, as shown below.

```
x: Imm a T
let(x₁, x₂) = dupl x;
let(y, ()) = withbor (alloc x₁) (Λ. λx₁′.
  withload x₁′ (Λ. λx₁. // x₁ and x₂ both accessible, x₂ allowed to escape ···));
free y
```

For types T such that Imm 'b T = T, one may derive a more traditional load rule, as follows.

```
load : Imm a T ⊸ T                    (∀'b. Imm 'b T = T)
load ≜ λx. withload x (λx′. x′)
```

$$\textsc{Type} \ni \mathsf{T} ::= \ldots \mid \mathtt{Unk}$$

$$\Delta \vdash \mathtt{withload} : \mathtt{Imm\,a\,T_1} \multimap (\forall\,\mathtt{'b} \sqsubset \sqcap\Delta.\ \underline{\mathtt{Imm}}\,\mathtt{'b\,T_1} \multimap [\mathtt{'b}]\,\mathtt{T_2}) \multimap \mathtt{T_2}$$

$$\underline{\mathtt{Imm}}\,\mathtt{'b}\ \mathbf{1} \triangleq \mathbf{1} \qquad \underline{\mathtt{Imm}}\,\mathtt{'b\,(T_1 \oplus T_2)} \triangleq \underline{\mathtt{Imm}}\,\mathtt{'b\,T_1} \oplus \underline{\mathtt{Imm}}\,\mathtt{'b\,T_2} \qquad \underline{\mathtt{Imm}}\,\mathtt{'b\,(T_1 \otimes T_2)} \triangleq \underline{\mathtt{Imm}}\,\mathtt{'b\,T_1} \otimes \underline{\mathtt{Imm}}\,\mathtt{'b\,T_2}$$

$$\underline{\mathtt{Imm}}\,\mathtt{'b\,(T_1 \multimap T_2)} \triangleq \mathtt{Unk} \qquad \underline{\mathtt{Imm}}\,\mathtt{'b\,(Ref\ T)} \triangleq \mathtt{Imm\,'b\,T} \qquad \underline{\mathtt{Imm}}\,\mathtt{'b\,(Imm\,a\,T)} \triangleq \mathtt{Imm\,a\,T}$$

$$\underline{\mathtt{Imm}}\,\mathtt{'b\,([a]T)} \triangleq \underline{\mathtt{Imm}}\,\mathtt{'b\,T} \qquad \underline{\mathtt{Imm}}\,\mathtt{'b\,(\forall\,'a} \sqsubset \mathtt{a.\ T)} \triangleq \mathtt{Unk}$$

Fig. 8. Loading from immutable borrows.

$$\textsc{Type} \ni \mathsf{T} ::= \ldots \mid \mathtt{Mut\,a\,T} \qquad \underline{\mathtt{Imm}}\,\mathtt{'b\,(Mut\,a\,T)} \triangleq \mathtt{Imm\,'b\,T}$$

$$\Delta \vdash \mathtt{forget} : \mathtt{Mut\,a\,T} \multimap \mathbf{1} \qquad \frac{\Delta; \Gamma \vdash \mathtt{e} : \mathtt{Mut\,b\,T} \qquad \Delta \vDash \mathtt{a} \sqsubseteq \mathtt{b}}{\Delta; \Gamma \vdash \mathtt{e} : \mathtt{Mut\,a\,T}}$$

$$\frac{\Delta \vdash \mathtt{T_1} \sqsupset \mathtt{'b}}{\Delta \vdash \mathtt{withbor} : \mathtt{Ref\ T_1} \multimap (\forall\,\mathtt{'a} \sqsubset \sqcap\Delta.\ \mathtt{Mut\,'a\ T_1} \multimap [\mathtt{'a}]\,\mathtt{T_2}) \multimap (\mathtt{Ref\ T_1}) \otimes \mathtt{T_2}}$$

$$\Delta \vdash \mathtt{withswap} : \mathtt{Mut\,a\ T_1} \multimap (\mathtt{T_1} \multimap \mathtt{T_1} \otimes \mathtt{T_2}) \multimap (\mathtt{Mut\,a\ T_1}) \otimes \mathtt{T_2}$$

$$\Delta \vdash \mathtt{withbor} : \mathtt{Mut\,a\ T_1} \multimap (\forall\,\mathtt{'b} \sqsubset \sqcap\Delta.\ \mathtt{Mut\,'b\ T_1} \multimap [\mathtt{'b}]\,\mathtt{T_2}) \multimap (\mathtt{Mut\,a\ T_1}) \otimes \mathtt{T_2}$$

Fig. 9. Typing mutable borrows.

2.5 Mutable Borrows

Earlier, we said that a borrow temporarily disables certain resource permissions in exchange for enabling certain structural permissions. With immutable borrows, we forbid both deallocation and mutation in exchange for duplication and forgetting. If we keep mutation, we can still keep `forget` as long as we drop `dupl`, a combination that is called a *mutable borrow*, whose typing rules are given in Fig. 9. Unlike immutable borrows, the payload type for a mutable borrow is required to have an unambiguous lifetime bound, a restriction that is related to the invariance of mutable references and which will be revisited in depth in § 4.2.

Since a mutable borrow cannot be duplicated, it is *exclusive* in the sense that it cannot coexist with any other aliases. Therefore, it is safe to manipulate its payload directly, so long as the payload is replaced with an inhabitant of the original payload type by the time the borrow ends, as is expected by the lender in `withbor`. One may think of this operation as a delayed, type-preserving variant of `swap`, which we call `withswap`. The following program demonstrates how `withswap` relies on the exclusivity invariant—if the two mutable borrows aliased, there would be a use-after-free bug, but the program is safe if they are distinct.

$$\begin{array}{l}
\mathtt{x_1} : \mathtt{Mut\,'a_1\ Ref\ 1}, \mathtt{x_2} : \mathtt{Mut\,'a_2\ Ref\ 1}, \\
\mathtt{let}(\mathtt{x_1}, ()) = \mathtt{withswap\ x_1}\ (\lambda \mathtt{x_1'}. \\
\quad \mathtt{let}(\mathtt{x_2}, ()) = \mathtt{withswap\ x_2}\ (\lambda \mathtt{x_2'}. \\
\quad\quad \mathtt{free\ x_1'}; \mathtt{free\ x_2'}; (\mathtt{alloc}\ (), ()))); \\
\quad \mathtt{forget\ x_2}; \\
\quad (\mathtt{alloc}\ (), ())); \\
\mathtt{forget\ x_1}
\end{array}$$

```
writeln : Ref File ⊸ Str ⊸ Ref File
greet : Ref File ⊸ Ref File
greet f =
 let f = writeln () f "hello";
 writeln f "world"

//well-typed but unintended
//↳greet f = close f; open "other"
```

(a) Writing with linear capability passing.

```
writeln : ∀ 'a. Mut 'a File ⊸ Str ⊸ 1
greet : ∀ 'a. Mut 'a File ⊸ 1
greet f =
 let (f, ()) = withbor f (Λ. λf'. writeln () f' "hello");
 writeln () f "world"
```

(b) Writing with mutable borrows.

Fig. 10. Turning a linear API into a mutably borrowing API.

We must also extend Imm to account for mutable borrows, Imm 'b (Mut a T). In this regard, mutable borrows are closer to full references than they are to immutable borrows—both assume exclusivity, which would not hold if they were loaded directly, as demonstrated in the previous section. Instead, an immutable borrow of a mutable borrow may be reborrowed immutably as Imm 'b T. Importantly, the lifetime on this reborrow is the fresh lifetime 'b introduced for the load, and not the original lifetime a on the mutable borrow, which would be allowed to escape the scope of the load.

Turning linear capability passing into mutable borrowing. As described in § 2.1, a key benefit of borrowing is that it alleviates the use of capability-passing style across functions. Most commonly, a borrow that is taken as input to a function will not be returned as output, unless it is being bundled into or extracted out of a data structure. For example, Fig. 10b uses mutable borrows to simplify the linear API in Fig. 10a by removing the references from the return types of the functions. The borrowing version not only has the advantage of being closer to what one would write in a traditional language with references (e.g., `int fputs(const char *s, FILE *stream)` in C), but it also rules out unintended inhabitants of the linear type, such as the commented out `greet` implementation that returns a completely different reference.

Whereas one can straightforwardly pass an immutable borrow to a sequence of functions by duplicating it, as shown in Fig. 5, a mutable borrow cannot be duplicated. Instead, to facilitate calling a sequence of functions on mutable borrows, we allow `withbor` to reborrow from a mutable borrow at a new lifetime. Then, one may insert such a reborrow before each call, as demonstrated in the body of `greet`.

3 A Separation Logic for Borrowing

Soundness of the BoCa type system must guarantee not only type and memory safety (i.e., the absence of undefined behavior), but also memory reclamation and termination. Instead of a syntactic progress and preservation proof, this section establishes a more general *semantic type soundness* result (§ 3.3), which facilitates linking with well-behaved code that cannot be syntactically typed. In fact, we have already seen several examples of such code: all of the borrowing operations from the previous section are actually defined internal to the language, but their types—which were specified as axioms in the statics—may only be validated semantically. To facilitate proofs about such programs, we develop *BoLo*, a separation logic with new abstractions for borrowing (§ 3.2).

3.1 From Types to Propositions

The semantics of our Borrow Logic (*BoLo*) will be developed incrementally in § 4, culminating with its full definition in Fig. 19. For now, this section introduces the propositions of the logic by way of example, showing how they are employed to interpret types as separation-logic predicates over

$$
\begin{array}{lll}
\alpha, \beta & \in & \textsc{Life} \qquad \triangleq \quad (\mathbb{N}, \sqcup \triangleq \min, \sqcap \triangleq \max, \dot{\top} \triangleq 0, \sqsubset \triangleq >) \\
\delta & \in & \textsc{LifeVar} \rightharpoonup \textsc{Life}
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{a}\delta & \triangleq & \begin{cases} \delta(\mathsf{'a}), & \mathsf{a} = \mathsf{'a} \\ \dot{\top}, & \mathsf{'a} = \dot{\top} \\ \mathsf{a}_1\delta \sqcup \mathsf{a}_2\delta, & \mathsf{a} = \mathsf{a}_1 \sqcup \mathsf{a}_2 \\ \mathsf{a}_1\delta \sqcap \mathsf{a}_2\delta, & \mathsf{a} = \mathsf{a}_1 \sqcap \mathsf{a}_2 \end{cases} \\
[\![\Delta]\!] & \triangleq & \{\delta \mid \mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\delta) \wedge \forall \mathsf{'a} \in \mathrm{dom}(\Delta).\ \delta(\mathsf{'a}) \sqsubset \Delta(\mathsf{'a})\delta\} \\
\Delta \vDash \mathsf{'a}_1 \sqsubset \mathsf{'a}_2 & \triangleq & \forall \delta \in [\![\Delta]\!]\ .\ \mathsf{'a}_1\delta \sqsubset \mathsf{'a}_2\delta \quad \text{(and similarly for } \sqsubseteq\text{)}
\end{array}
$$

Fig. 11. Semantic lifetimes.

terms. For clarity, propositions of the logic will be styled in red Roman. To begin, each type T is assigned a separation logic predicate on values $\mathcal{V}\,[\![\mathsf{T}]\!]$. The conclusion of this subsection will show how this predicate may be lifted to a predicate $\mathcal{E}\,[\![\mathsf{T}]\!]$ that characterizes expressions that behave like type T. The basic linear logic types map to their separation logic counterparts, up to a change of syntax and some pattern matching.

$$
\begin{array}{lll}
\mathcal{V}\,[\![\mathbf{1}]\!]\,(\mathsf{v}) & \triangleq & \ulcorner \mathsf{v} = () \urcorner \\
\mathcal{V}\,[\![\mathsf{T}_1 \oplus \mathsf{T}_2]\!]\,(\mathsf{v}) & \triangleq & (\exists\, \mathsf{v}_1.\ \ulcorner \mathsf{v} = \mathsf{inj}_1\, \mathsf{v}_1 \urcorner \ast \mathcal{V}\,[\![\mathsf{T}_1]\!]\,(\mathsf{v}_1)) \vee (\exists\, \mathsf{v}_2.\ \ulcorner \mathsf{v} = \mathsf{inj}_2\, \mathsf{v}_2 \urcorner \ast \mathcal{V}\,[\![\mathsf{T}_2]\!]\,(\mathsf{v}_2)) \\
\mathcal{V}\,[\![\mathsf{T}_1 \otimes \mathsf{T}_2]\!]\,(\mathsf{v}) & \triangleq & \exists\, \mathsf{v}_1, \mathsf{v}_2.\ \ulcorner \mathsf{v} = (\mathsf{v}_1, \mathsf{v}_2) \urcorner \ast \mathcal{V}\,[\![\mathsf{T}_1]\!]\,(\mathsf{v}_1) \ast \mathcal{V}\,[\![\mathsf{T}_2]\!]\,(\mathsf{v}_2) \\
\mathcal{V}\,[\![\mathsf{T}_1 \multimap \mathsf{T}_2]\!]\,(\mathsf{v}_2) & \triangleq & \forall\, \mathsf{v}_1.\ \mathcal{V}\,[\![\mathsf{T}_1]\!]\,(\mathsf{v}_1) \mathrel{-\!\!\ast} \mathcal{E}\,[\![\mathsf{T}_2]\!]\,(\mathsf{v}_2\ \mathsf{v}_1)
\end{array}
$$

The separating conjunction $\ast$ *asserts* a division of resources between its components in much the same way as the tensor $\otimes$ does in the statics. Likewise, the magic wand $\mathrel{-\!\!\ast}$ *assumes* a division of resources between the caller and the callee in much the same was as the lolli $\multimap$ does in the statics. Note that functions are interpreted *extensionally*, according to how they behave when supplied semantically well-typed inputs. Pure facts, such as the equalities above, are surrounded by $\ulcorner \_ \urcorner$ and cannot hold resources as this is a *linear* separation logic.

One proposition that explicitly claims a resource is the points-to connective $\mapsto$, indicating full and *exclusive* ownership over a location, as is the case for references. Below, the separating conjunction between the points-to and the payload interpretation implies that the payload cannot claim any aliases to the reference.

$$\mathcal{V}\,[\![\mathsf{Ref}\ \mathsf{T}]\!]\,(\mathsf{v}) \triangleq \exists\, \ell, \mathsf{v}'.\ \ulcorner \mathsf{v} = \ell \urcorner \ast \ell \mapsto \mathsf{v}' \ast \mathcal{V}\,[\![\mathsf{T}]\!]\,(\mathsf{v}')$$

The outlives modality and the universal lifetime quantifier have counterparts in the logic, $[\,]$ and $\forall$. Instead of syntactic lifetime variables, these logical operators manipulate *semantic lifetimes*, defined in Fig. 11. As alluded to in § 2.3, every borrow is (logically) assigned a fresh lifetime α, which we model using a semi-bounded lattice of natural numbers. Note that the outlives relation $\beta \sqsupset \alpha$ is the converse of the natural ordering, where 0 is the top lifetime $\dot{\top}$. The lifetime context Δ is interpreted as a substitution δ from syntactic lifetime variables $\mathsf{'a}$ to semantic lifetimes α such that all of the outlives constraints are satisfied. As is typical in models of type polymorphism [27], the type interpretations track a lifetime substitution δ, which is extended at each quantifier and used to close off lifetimes, written $\mathsf{b}\delta$. In the clauses above, it is tacitly threaded along unchanged.

$$
\begin{array}{lll}
\mathcal{V}\,[\![\forall\, \mathsf{'a} \sqsubset \mathsf{b}.\ \mathsf{T}]\!]_\delta\,(\mathsf{v}) & \triangleq & \forall\, \alpha.\ \ulcorner \alpha \sqsupset \mathsf{b}\delta \urcorner \mathrel{-\!\!\ast} \mathcal{E}\,[\![\mathsf{T}]\!]_{\delta[\mathsf{'a} \mapsto \alpha]}\,(\mathsf{v}\ ()) \\
\mathcal{V}\,[\![[\mathsf{a}]\mathsf{T}]\!]_\delta\,(\mathsf{v}) & \triangleq & [\mathsf{a}\delta]\mathcal{V}\,[\![\mathsf{T}]\!]_\delta\,(\mathsf{v})
\end{array}
$$

$\mapsto$ Excl
$\ell \mapsto v \star \ell \mapsto \_ \vDash \bot$

M Excl
$\ell \mapsto \mathrm{M}_\alpha \widehat{P} \star \ell \mapsto \_ \vDash \bot$

I Agree
$\ell \mapsto \mathrm{I}_{\alpha_1} \widehat{P}_1 \star \ell \mapsto \mathrm{I}_{\alpha_2} \widehat{P}_2 \vDash \ell \mapsto \mathrm{I}_{\alpha_1 \sqcup \alpha_2} (\widehat{P}_1 \wedge \widehat{P}_2)$

I Vary
$$\frac{\alpha_1 \sqsupseteq \alpha_2 \qquad \forall v.\, \widehat{P}_1(v) \vDash \widehat{P}_2(v)}{\ell \mapsto \mathrm{I}_{\alpha_1} \widehat{P}_1 \vDash \ell \mapsto \mathrm{I}_{\alpha_2} \widehat{P}_2}$$

M Vary
$$\frac{\alpha_1 \sqsupseteq \alpha_2 \qquad \forall v.\, \widehat{P}_1(v) \nvDash \widehat{P}_2(v)}{\ell \mapsto \mathrm{M}_{\alpha_1} \widehat{P}_1 \vDash \ell \mapsto \mathrm{M}_{\alpha_2} \widehat{P}_2}$$

(a) Borrows.

[] Vary
$$\frac{\alpha_1 \sqsupseteq \alpha_2 \qquad P_1 \vDash P_2}{[\alpha_1] P_1 \vDash [\alpha_2] P_2}$$

[] T
$[\alpha] P \vDash P$

[] 4
$[\alpha] P \vDash [\alpha][\alpha] P$

[] $\star$
$[\alpha](P_1 \star P_2) \nvDash [\alpha] P_1 \star [\alpha] P_2$

$\forall$ Mono
$$\frac{\forall \alpha \sqsubseteq \beta.\, \widehat{P}_1(\alpha) \vDash \widehat{P}_2(\alpha)}{\forall \widehat{P}_1 \vDash \forall \widehat{P}_2}$$

$\forall$ R
$P \vDash \forall \alpha.\, P$

$\forall$ L
$$\frac{\forall \alpha \sqsubseteq \beta.\, \widehat{P}_1(\alpha) \vDash P_2}{\forall \widehat{P}_1 \vDash P_2}$$

$\forall \star$
$\forall \widehat{P}_1 \star \forall \widehat{P}_2 \vDash \forall (\widehat{P}_1 \star \widehat{P}_2)$

(b) Outlives and freshness.

Fig. 12. *BoLo* entailments (excerpts).

Finally, borrows are interpreted logically using variations of the points-to connective that restrict access to the location, $\ell \mapsto \mathrm{I}_\alpha \widehat{P}$ for immutable and $\ell \mapsto \mathrm{M}_\alpha \widehat{P}$ for mutable. Whereas the points-to connective $\ell \mapsto v$ is indexed by a single value v representing full ownership of a single cell, the borrow variations are indexed by a separation predicate $\widehat{P}$ over the payload that characterizes *what the payload value owns*. This means that while a borrow may inhibit access to the resources owned by the payload, the borrow still holds those resources, if indirectly.

$$\begin{aligned} \mathcal{V}[\![\mathtt{Imm\ a\ T}]\!]_\delta(v) &\triangleq \exists \ell.\, \ulcorner v = \ell \urcorner \star \ell \mapsto \mathrm{I}_{\mathtt{a}\delta} \mathcal{V}[\![\mathtt{T}]\!] \\ \mathcal{V}[\![\mathtt{Mut\ a\ T}]\!]_\delta(v) &\triangleq \exists \ell.\, \ulcorner v = \ell \urcorner \star \ell \mapsto \mathrm{M}_{\mathtt{a}\delta} \mathcal{V}[\![\mathtt{T}]\!] \end{aligned}$$

The next several sections will focus on the rules governing the use of these more bespoke connectives. Before moving on, we need to introduce one more proposition—the weakest precondition, $\mathrm{wp}(e)\{\widehat{Q}\}$. Its formal definition will be clarified in § 4.3, but for now, an informal characterization is that it holds in any pre-state sufficient to run e such that it will safely terminate with a value v and a post-state that satisfy the value predicate $\widehat{Q}$. Alternatively, it can be helpful to view it in relation to the more familiar Hoare triple, as shown below. Finally, we can define $\mathcal{E}$ using the weakest precondition with $\mathcal{V}$ as the postcondition.

$$P \vDash \mathrm{wp}(e)\{\widehat{Q}\} \quad \Leftrightarrow \quad \vDash \{P\}\ e\ \{\widehat{Q}\} \qquad\qquad \mathcal{E}[\![\mathtt{T}]\!]_\delta(e) \triangleq \mathrm{wp}(e)\{\mathcal{V}[\![\mathtt{T}]\!]_\delta\}$$

3.2 Logical Borrowing

As demonstrated by the entailments in Fig. 12a, the three variations of the points-to connective reflect the structural permissions of their counterparts in the statics: the owned ($\mapsto$) and mutable borrow (M) variants are *exclusive*, in the sense that it is contradictory for any compatible resource to hold an alias ($\mapsto$ Excl and M Excl), whereas the immutable borrow (I) variant may have aliases as long as they all *agree* on the lifetime and the predicate on the payload (I Agree). Both forms of borrows are antitone in the lifetime ordering and immutable borrows are additionally monotone with respect to entailment in the payload predicate (I Vary). Mutable borrows, on the other hand,

are *invariant* in the payload predicate, as is usually the case for subtyping and mutable references (M Vary).

3.2.1 Outlives and Freshness. The logic justifies calling outlives a *modality* in the sense that it is monotone with respect to entailment ([] Vary), as shown in Fig. 12b. It satisfies the □ axioms of S4: it can always be removed ([] T) and replicated ([] 4). These two rules give the modality a comonadic flavor, as they can be used to derive a co-bind/extend rule. Just like borrows, the modality is antitone with respect to lifetime inclusion: outliving longer lifetimes entails outliving shorter ones. The modality commutes with most other connectives (e.g., [] ⋆), subject to lifetime or domain restrictions.

In the statics, lifetime *freshness* is captured using a constraint on the lifetime variables in scope, but no such context exists explicitly at the level of the logic. Inspired by nominal logic [23], we introduce a freshness quantifier $\curlywedge\, \widehat{P}$ where $\widehat{P}$ is a predicate over lifetimes. Moving under a freshness quantifier requires choosing an upper bound and then the predicate is instantiated with an arbitrary shorter lifetime ($\curlywedge$ Mono). The logic ensures that a fresh lifetime can always be found, and that separate resources that individually require fresh lifetimes may be composed; i.e., that a lifetime "fresh enough" for each may be found ($\curlywedge \star$). The freshness quantifier is effectively monotone with respect to entailment, except that its operand is a predicate over lifetimes.

3.2.2 The Borrow Frame Rule. A distinguishing feature of separation logic is the *frame rule*, shown below. For Hoare triples, it allows one to temporarily ignore a fragment (P_f) of the precondition, validate that the program establishes a postcondition (P_2) using what remains (P_1), and then restore the ignored fragment in the final postcondition.

$$\text{HT-Frame}\quad \frac{\{P_1\}\ e\ \{P_2\}}{\{P_1 \star P_f\}\ e\ \{P_2 \star P_f\}}$$

The appeal of the frame rule is that the verification of e may be conducted using only the resources that it requires, without threading irrelevant information along, but the logic ensures that the temporarily ignored resources are not forgotten by restoring them in the postcondition. Notice that the motivation is almost the same motivation as for borrowing, except that in the case of borrowing the resource is not *completely* ignored; instead, a limited view—the borrow—is preserved. This observation leads us to the *borrow frame rule*, shown for immutable borrows below. To emphasize the similarity to the typing rule for `withbor`, we annotate the related components of the two rules using the same subscripts.

$$\Delta \vdash \texttt{withbor} : \texttt{Ref}\ T_f \multimap (\forall\, \texttt{'a} \sqsubset \sqcap\Delta.\ \texttt{Imm}\ T_f \multimap [\texttt{'a}]T_2)_1 \multimap (\texttt{Ref}\ T_f) \otimes T_2$$

$$\text{HT-Brw-Frame}\quad \frac{\curlywedge\, \alpha.\ \left\{P_1 \star \ell \mapsto I_\alpha\, \widehat{P}_f\right\}\ e\ \{[\alpha]P_2\}}{\left\{P_1 \star \ell \mapsto v \star \widehat{P}_f(v)\right\}\ e\ \left\{P_2 \star \ell \mapsto v \star \widehat{P}_f(v)\right\}}$$

The weakest-precondition variants of the borrow-frame rules are shown in Fig. 13a. The postconditions are effectively written in continuation-passing style, as is standard for weakest-precondition rules, but they otherwise resemble their counterparts in the statics. However, at the level of the logic, we have strictly more information—we know both the particular location of the reference and its payload. For immutable borrows, the payload value cannot change during computation and is persisted by the borrow frame rule. Mutable borrows only guarantee that the payload satisfies the borrowed predicate, so the new payload is quantified over in the postcondition. Note that the

I Frame
$$\ell \mapsto v_p \star \widehat{P}(v_p) \star (\mathcal{N}\,\alpha.\, \ell \mapsto I_\alpha\, \widehat{P} \mathrel{-\!\!\star} \mathsf{wp}(e)\{v_q.\, [\alpha](\ell \mapsto v_p \mathrel{-\!\!\star} \widehat{P}(v_p) \mathrel{-\!\!\star} \widehat{Q}(v_q))\}) \vDash \mathsf{wp}(e)\{\widehat{Q}\}$$

M Frame
$$\frac{\forall v_p.\, \widehat{P}(v_p) \vDash [\beta]\widehat{P}(v_p)}{\ell \mapsto v_p \star \widehat{P}(v_p) \star (\mathcal{N}\,\alpha.\, \ell \mapsto M_\alpha\, \widehat{P} \mathrel{-\!\!\star} \mathsf{wp}(e)\{v_q.\, [\alpha]\, \forall\, v_p'.\, (\ell \mapsto v_p' \mathrel{-\!\!\star} \widehat{P}(v_p') \mathrel{-\!\!\star} \widehat{Q}(v_q))\}) \vDash \mathsf{wp}(e)\{\widehat{Q}\}}$$

(a) Borrow frame rules.

WP Load I
$$\ell \mapsto I_\alpha\, \widehat{P} \star (\forall\, v.\, \ell \mapsto I_\alpha(\_.\, \widehat{P}(v)) \mathrel{-\!\!\star} \mathsf{wp}(v)\{\widehat{Q}\}) \vDash \mathsf{wp}(\mathtt{load}\,\ell)\{\widehat{Q}\}$$

I Reborrow
$$\ell \mapsto I_\alpha(\mathcal{N}\,\beta.\, \circlearrowleft_\beta\, \widehat{P}) \star (\mathcal{N}\,\beta.\, \forall\, v.\, \widehat{P}(v) \mathrel{-\!\!\star} \mathsf{wp}(e)\{[\beta]\widehat{Q}\}) \vDash \mathsf{wp}(e)\{\widehat{Q}\}$$

$\circlearrowleft \mapsto$
$$\ell \mapsto v \star [\alpha]\widehat{P}(v) \vDash \circlearrowleft_\alpha\, \ell \mapsto I_\alpha\, \widehat{P}$$

$\circlearrowleft$ I
$$\frac{\beta \sqsupseteq \alpha}{\ell \mapsto I_\beta\, \widehat{P} \vDash \circlearrowleft_\alpha\, \ell \mapsto I_\beta\, \widehat{P}}$$

$\circlearrowleft$ M
$$\frac{\beta \sqsupseteq \alpha}{\ell \mapsto I_\beta\, \widehat{P} \vDash \circlearrowleft_\alpha\, \ell \mapsto I_\alpha\, \widehat{P}}$$

(b) Immutable loading and reborrowing rules.

M Anti-Frame
$$\ell \mapsto M_\alpha\, \widehat{P} \star (\forall\, v_p.\, \ell \mapsto v_p \mathrel{-\!\!\star} \widehat{P}(v_p) \mathrel{-\!\!\star} \mathsf{wp}(e)\{v_q.\, \exists\, v_p'.\, \ell \mapsto v_p' \star \widehat{P}(v_p') \star (\ell \mapsto M_\alpha\, \widehat{P} \mathrel{-\!\!\star} \widehat{Q}(v_q))\}) \vDash \mathsf{wp}(e)\{\widehat{Q}\}$$

(c) The mutable anti-frame rule.

Fig. 13. *BoLo* borrowing rules.

borrowed predicate must have an unambiguous lifetime bound just as in the type system, which is expressed at the logical level using $[\beta]$.

3.2.3 The Immutable Load and Reborrow Rules. Recall that in well-typed programs, immutable borrows are accessed using the `withload` operation (Fig. 8), whose type is parameterized by the metafunction Imm. This metafunction maps a type to its *reborrowed* type, which immutably borrows any linear references or mutable borrows in the payload and obfuscates any closures that may hold onto such resources. To access immutable borrows in the logic, we will use a rule of a similar shape, but the metafunction Imm does not have a clear counterpart for separation logic propositions. Whereas there is a limited grammar of types and overapproximation is commonplace in type systems, the space of propositions is far richer and may describe resources with complex control-flow dependencies that an analogous metafunction would need to address. We instead adopt the more extensional approach of characterizing the full space of safe reborrows, which is captured by the *reborrow modality*, $\circlearrowleft_\alpha P$. It is a $\diamond$-style modality that holds of resources that *could* satisfy P *if* reborrowed at a fresh lifetime α. In terms of its application, the reborrow modality plays an analogous role to the metafunction in the I Reborrow rule in Fig. 13b. In terms of its construction, the modality supports entailments with analogous clauses in Imm, as for references ($\circlearrowleft \mapsto$, $\circlearrowleft$ I, and $\circlearrowleft$ M), but also validates reborrows of propositions that do not correspond to any source types, such as existentials and pure propositions.

Fig. 14 sketches a proof similar to the compatibility lemma for the `withload` operation, but for simplicity, we use arbitrary predicates instead of semantic types (§ 3.1), which require more

withload ≜ λxλf. f () (load x)

(Iᴺ Mono, ∀ R, ∀ L, Refl)

Iᴺ β. ∀ v. P̂(v) −★ wp(v_f () v){Q̂} ⊨ Iᴺ β. ∀ _. P̂(v) −★ wp(v_f () v){Q̂}

(I Reborrow)

ℓ ↦ I_α(_. Iᴺ β. ⟳_β P̂(v)) ★ Iᴺ β. ∀ v. P̂(v) −★ wp(v_f () v){Q̂} ⊨ wp(v_f () v){Q̂}

(wp App, wp Bind, wp Load I)

(ℓ ↦ I_α Iᴺ β. ⟳_β P̂)₁ ★ (Iᴺ β. ∀ v. P̂(v) −★ wp(v_f () v){[β]Q̂})₂ ⊨ wp(withload ℓ v_f){Q̂}₃

Fig. 14. Deriving withload. Changes between proof states are highlighted.

unfolding. The antecedents of the conclusion are (1): an immutable borrow of ℓ for α whose payload can be reborrowed as $\widehat{P}$ for β; and (2) a continuation v_f that can use such a reborrow $\widehat{P}(v)$ for any β with payload value v to establish a postcondition $[\beta]\widehat{Q}$ that outlives β. The consequent of the conclusion is (3) that calling withload with ℓ and v_f establishes $\widehat{Q}$. The proof makes use of the rule wp Load I (Fig. 13b), which ensures that after loading a value v from an immutable borrow of ℓ, we can specialize its borrowed predicate to v by turning it into a constant function, since all future reads from the immutable borrow must produce the same value. After physically loading the payload value v and specializing the borrow predicate, we use the rule I Reborrow to "logically load" a reborrow of the payload resource.

3.2.4 The Mutable Anti-Frame Rule. Recall that in well-typed programs, mutable borrows are accessed using the withswap operation, which temporarily provides ownership to the payload but demands that a suitably typed payload be restored. To access mutable borrows in the logic, we will use a rule with the same shape but which provides *direct access* to the borrowed location. Inspired by [24], we call this the *mutable anti-frame rule*, which is dual to the mutable borrow frame rule above. It strongly resembles the typing rule for withswap except that it provides direct access to the location instead of loading the payload.

$$\Delta \vdash \texttt{withswap} : \texttt{Mut a } T_f \multimap (T_f \multimap T_f \otimes T_2) \multimap (\texttt{Mut a } T_f) \otimes T_2$$

$$\frac{\text{M Anti-Frame}\quad \forall v.\ \left\{P_1 \star \ell \mapsto v \star \widehat{P}_f(v)\right\}\ e\ \left\{\exists v'.\ P_2 \star \ell \mapsto v' \star \widehat{P}_f(v')\right\}}{\left\{P_1 \star \ell \mapsto M_\alpha\, \widehat{P}_f\right\}\ e\ \left\{P_2 \star \ell \mapsto M_\alpha\, \widehat{P}_f\right\}}$$

$$\Delta \nvdash \texttt{withswap} : \texttt{Mut a } T_f \multimap (\texttt{Ref } T_f \multimap \texttt{Ref } T_f \otimes T_2) \multimap (\texttt{Mut a } T_f) \otimes T_2$$

Note that the analogous strengthening of the withswap typing rule, as sketched above, is *not* sound, since unlike the logic, the type system could not prevent the reference from being freed and replaced with an impostor. If one were to enrich reference types to track locations, as in L^3 [1, 18], then perhaps such a variation would be admissible. One immediate use for this additional power

$$\texttt{dupl} \triangleq \lambda \texttt{x}.\,(\texttt{x},\texttt{x}) \qquad \texttt{forget} \triangleq \lambda \texttt{x}.\,() \qquad \texttt{withbor} \triangleq \lambda \texttt{x} \lambda \texttt{f}.\,(\texttt{x},\texttt{f}\ ()\ \texttt{x})$$

$$\texttt{withswap} \triangleq \lambda \texttt{x} \lambda \texttt{f}.\,\texttt{let}(\texttt{y},\texttt{z}) = (\texttt{f}\ (\texttt{load}\ \texttt{x}));\texttt{store}\ \texttt{x}\ \texttt{y};(\texttt{x},\texttt{z})$$

Fig. 15. Implementation of borrowing constructs.

is that it may be used to derive mutable reborrowing rules like the following, which temporarily downgrades a mutable borrow to an immutable one.

$$\frac{\ell \mapsto \mathsf{v_p} \star \widehat{\mathsf{P}}(\mathsf{v_p}) \star (\ell \mapsto \mathrm{M}_\alpha\, \widehat{\mathsf{P}} \mathrel{-\!\!\star} \widehat{\mathsf{Q}}(\mathsf{v_q})) \vDash \exists\, \mathsf{v_p'}.\, \ell \mapsto \mathsf{v_p'} \star \widehat{\mathsf{P}}(\mathsf{v_p'}) \star (\ell \mapsto \mathrm{M}_\alpha\, \widehat{\mathsf{P}} \mathrel{-\!\!\star} \widehat{\mathsf{Q}}(\mathsf{v_q}))}{\text{(}\exists\text{ R, Refl)}}$$

$$\frac{\mathrel{\text{⅄}}\, \beta.\, \ell \mapsto \mathrm{I}_\beta\, \widehat{\mathsf{P}} \mathrel{-\!\!\star} \mathsf{wp}(\mathsf{e})\{[\beta](\ell \mapsto \mathrm{M}_\alpha\, \widehat{\mathsf{P}} \mathrel{-\!\!\star} \widehat{\mathsf{Q}})\} \vDash \mathrel{\text{⅄}}\, \beta.\, \ell \mapsto \mathrm{I}_\beta\, \widehat{\mathsf{P}} \mathrel{-\!\!\star} \mathsf{wp}(\mathsf{e})\{\mathsf{v_q}.\, [\beta](\ell \mapsto \mathsf{v_p} \mathrel{-\!\!\star} \widehat{\mathsf{P}}(\mathsf{v_p}) \mathrel{-\!\!\star} \exists\, \mathsf{v_p'}.\, \ell \mapsto \mathsf{v_p'} \star \widehat{\mathsf{P}}(\mathsf{v_p'}) \star [\ell \mapsto \mathrm{M}_\alpha\, \widehat{\mathsf{P}} \mathrel{-\!\!\star} \widehat{\mathsf{Q}}(\mathsf{v_q})])\}}{\text{(I Frame)}}$$

$$\frac{\ell \mapsto \mathsf{v_p} \star \widehat{\mathsf{P}}(\mathsf{v_p}) \star \mathrel{\text{⅄}}\, \beta.\, \ell \mapsto \mathrm{I}_\beta\, \widehat{\mathsf{P}} \mathrel{-\!\!\star} \mathsf{wp}(\mathsf{e})\{[\beta](\ell \mapsto \mathrm{M}_\alpha\, \widehat{\mathsf{P}} \mathrel{-\!\!\star} \widehat{\mathsf{Q}})\} \vDash \mathsf{wp}(\mathsf{e})\{\mathsf{v_q}.\, \exists\, \mathsf{v_p'}.\, \ell \mapsto \mathsf{v_p'} \star \widehat{\mathsf{P}}(\mathsf{v_p'}) \star (\ell \mapsto \mathrm{M}_\alpha\, \widehat{\mathsf{P}} \mathrel{-\!\!\star} \widehat{\mathsf{Q}}(\mathsf{v_q}))\}}{\text{(M Anti-Frame)}}$$

$$\ell \mapsto \mathrm{M}_\alpha\, \widehat{\mathsf{P}} \star \mathrel{\text{⅄}}\, \beta.\, \ell \mapsto \mathrm{I}_\beta\, \widehat{\mathsf{P}} \mathrel{-\!\!\star} \mathsf{wp}(\mathsf{e})\{[\beta](\ell \mapsto \mathrm{M}_\alpha\, \widehat{\mathsf{P}} \mathrel{-\!\!\star} \widehat{\mathsf{Q}})\} \vDash \mathsf{wp}(\mathsf{e})\{\widehat{\mathsf{Q}}\}$$

3.3 Semantic Type Soundness

The logic is employed to establish semantic type soundness for the type system, which requires that any syntactically well-typed term is also semantically well-typed. The general definition of semantic typing for open terms must quantify over closing substitutions for the syntactic typing contexts. As described in Fig. 7, substitutions for Δ map syntactic lifetime variables to semantic lifetime variables which are collectively consistent with its outlives constraints. Valid substitutions for Γ map typed variables to closed values in the $\mathcal{V}$ interpretation of their types.

$$\begin{array}{lcl} \gamma & : & \textsc{Var} \rightharpoonup \textsc{Val} \\ [\![\Gamma]\!]_\delta\,(\gamma) & \triangleq & \ulcorner \mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\gamma) \urcorner \star (\star_{\mathsf{x} \in \mathrm{dom}(\Gamma)} \mathcal{V}\,[\![\Gamma(\mathsf{x})]\!]_\delta\,(\gamma(\mathsf{x}))) \\ \Delta; \Gamma \vDash \mathsf{e} : \mathsf{T} & \triangleq & \forall \delta \in [\![\Delta]\!], \gamma.\ [\![\Gamma]\!]_\delta\,(\gamma) \vDash \mathcal{E}\,[\![\mathsf{T}]\!]_\delta\,(\mathsf{e}\gamma) \end{array}$$

Lemma 3.1 (Fundamental Property). *If* $\Delta; \Gamma \vdash \mathsf{e} : \mathsf{T}$ *then* $\Delta; \Gamma \vDash \mathsf{e} : \mathsf{T}$.

The proof of the Fundamental Property is by induction on the syntactic typing judgment, and it is divided into a collection of *compatibility lemmas*, one per syntactic typing rule, which establishes that its semantic analogue is admissible. For example, the syntactic elimination form for functions is reflected into a semantic elimination form for functions, and the soundness of this rule is established using the program logic.

$$\frac{\Delta; \Gamma_1 \vDash \mathsf{e}_1 : \mathsf{T}_1 \qquad \Delta; \Gamma_2 \vDash \mathsf{e}_2 : \mathsf{T}_1 \multimap \mathsf{T}_2}{\Delta; \Gamma_1, \Gamma_2 \vDash \mathsf{e}_2\ \mathsf{e}_1 : \mathsf{T}_2}$$

Crucially, this version of the rule does not assume that the terms are *syntactically* well-typed. As mentioned at the start of the section, the real payoff of semantic typing is that it enables the validation of terms that behave according to a type but cannot be checked syntactically. In fact, as shown in Fig. 15, every borrowing construct is defined using existing terms of the language, which underscores that borrowing is purely a *discipline* on top of the original language. For example, due to the similarities elucidated above, the semantic typing of `withbor` is almost a direct consequence of the borrow frame rule.

The Fundamental Property is just one part of the overall proof of semantic type soundness. It establishes that the syntactic type system is sound with respect to the logic, but it remains to show the soundness of the logic itself. Additionally, we must show *adequacy* of the weakest precondition—that it really does characterize executions that are safe, terminating, and reclaim memory. The next section develops proofs for these properties, but to close the chapter on semantic type soundness, we show that they are sufficient to show termination and memory reclamation for well-typed programs, which is usually stated for closed programs at base type.

Theorem 3.2 (Adequacy). *If* $\vDash \mathrm{wp}(\mathsf{e})\{\ulcorner \widehat{P} \urcorner\}$ *then* $(\varnothing, \mathsf{e}) \longrightarrow^* (\varnothing, \mathsf{v})$ *and* $\widehat{P}(\mathsf{v})$ *for some* v.

Corollary 3.3. *If* $\vDash \mathsf{e} : \mathbf{1}$ *then* $(\varnothing, \mathsf{e}) \longrightarrow^* (\varnothing, ())$.

4 A Semantic Model of Borrowing

This section develops a model of the logic that validates the entailments from the previous section and the adequacy of the weakest precondition. The most important question that must be addressed is what the inhabitants of this semantic model will be. As emphasized in the previous section, borrows are a purely logical notion—they do not manifest physically in the operational semantics. Indeed, physical memories do not have enough structure to support even the most basic reasoning principles—they are global, whereas the frame rules express locality, and they cannot distinguish between owned and borrowed locations. Instead of physical memories, our model will be inhabited by *logical resources*, which are like memory fragments annotated with extra semantic information.

To establish safety, we must enforce that all borrows are *preserved* by program execution—that every immutable borrow stays exactly the same, and that every mutable borrow preserves its invariant. As in models of mutable references [2], we will record the invariant of a mutable borrow, but here, we must also ensure that is temporally *exclusive*. At the same time, an immutable borrow must prohibit any observable changes to its state, even when it holds data that would otherwise by mutable, like an owned pointer or a mutable borrow. To resolve this tension and ensure that borrows are preserved even when nested arbitrarily, we will record a *witness* for each borrow, which is a first-class sub-resource that can impose its own invariants but which might be further constrained by outer borrows.

We construct the model incrementally. In our first incomplete attempt, shown below, a *logical resource* ρ will be an annotated fragment of memory that decorates each memory cell ψ with its ownership status: own, imm, or mut. The next few subsections will develop the definitions of immutable (**Imm**) and mutable (**Mut**) cells, but for now, we can at least say that only immutable cells should be composable, written $\psi_1 \bowtie \psi_2$. When we eventually define cell composition, resource composition $\rho_1 \bullet \rho_2$ will lift it point-wise; it is defined when all overlapping cells are composable $\psi_1 \bowtie \psi_2$. Non-overlapping cells (e.g., in $\rho_1 \setminus \rho_2$) are always included in composition. Next, we will begin to fill in some of the gaps (marked with ?) below. Finally, separation logic propositions P are predicates on resources, which are lifted to predicates $\widehat{\mathrm{P}}$ over domains like lifetimes or values.

P	$\in$	**SProp**	$\triangleq$ **Res** $\to \mathbb{P}$
$\widehat{\mathrm{P}}, \widehat{\mathrm{Q}}$	$\in$	**SPred**(X)	$\triangleq X \to$ **SProp**
ρ	$\in$	**Res**	$\approx$ Loc $\rightharpoonup$ **Cell**
ψ	$\in$	**Cell**	$\approx$ own(Val) + imm(**Imm**) + mut(**Mut**)
		Imm	$\approx$?
		Mut	$\approx$?

Resource composition	$\rho_1 \bullet \rho_2$	$\triangleq$	$(\rho_1 \setminus \rho_2) \uplus (\rho_2 \setminus \rho_1) \uplus [\ell \mapsto \psi_1 \bullet \psi_2 \mid \rho_1(\ell) = \psi_1 \wedge \rho_2 = \psi_2]$
Cell composition	$\psi_1 \bullet \psi_2$	$\approx$	? if $\psi_1 \bowtie \psi_2$
Cell compatibility	$\psi_1 \bowtie \psi_2$	$\approx$	$\psi_1 = \mathrm{imm}(?) \wedge \psi_2 = \mathrm{imm}(?)$

4.1 Immutable Cells

At a minimum, the definition of the immutable borrow connective $\ell \mapsto \mathrm{I}_\alpha \widehat{\mathrm{P}}$ must ensure that the resource ρ contains an imm cell at the given location ℓ. Moreover, because the separation logic is precise about resources, it must not contain any other locations. We also expect the definition to mention the payload predicate $\widehat{\mathrm{P}}$, though it is not yet clear of what value and resource it should hold.

$$\ell \mapsto \mathrm{I}_\alpha \widehat{\mathrm{P}}(\rho) \overset{?}{\approx} \rho = \ell \mapsto \mathrm{imm}(?) \wedge \widehat{\mathrm{P}}(?)(?)$$

Recall that in the borrow frame rule (§ 3.2), one exchanges an owned pointer $\ell \mapsto \mathsf{v}$ and ownership of the payload $\widehat{\mathrm{P}}(\mathsf{v})$ for a borrow $\ell \mapsto \mathrm{I}_\beta \widehat{\mathrm{P}}$ at a fresh lifetime β. At the moment the borrow is created, there is a particular *witness*—the value v and a resource ρ'—for the payload predicate $\widehat{\mathrm{P}}(\mathsf{v})(\rho')$, which can be used to fill the holes above if recorded in the cell. As alluded to in the interpretation of borrow types (§ 3.1), ownership of this witness ρ' is temporarily moved into the borrow.

$$\begin{array}{lcl} \textbf{Imm} & \overset{?}{\approx} & \textsc{Val} \times \textbf{Res} \\ \ell \mapsto \mathrm{I}_\alpha \widehat{\mathrm{P}}(\rho) & \overset{?}{\approx} & \exists \mathsf{v}, \rho'.\ \rho = \ell \mapsto \mathrm{imm}(\mathsf{v}, \rho') \wedge \widehat{\mathrm{P}}(\mathsf{v})(\rho') \end{array}$$

Suspiciously, the definition does not yet constrain the lifetime α. Since the lifetime indexing a borrow is upper bounded by its original lifetime, a tempting completion of the definition is to record the initial lifetime β and use it to bound α. However, because of reborrowing, immutable borrows of the same witness at the same location may be taken multiple times, each with a fresh lifetime, as in the following example.

```
x : Imm 'a (Ref T)
let (x1, x2) = dupl x;
withload x1 (Λ. λy1. withload x2 (Λ. λy2.
  // y1 : Imm 'b1 T and y2 : Imm 'b2 T are aliases with different lifetimes
  ...))
```

This wrinkle has two important consequences. First, regarding the incomplete definition above, an immutable cell will record the *set* of lifetimes $\widetilde{\beta}$ that it has been borrowed at (written $\mathrm{imm}(\widetilde{\beta}, \mathsf{v}, \rho')$), and the borrow connective will bound its lifetime index α by the longest among them. Second, whereas immutable cells ψ_1, ψ_2 at the same location must all have the same witness, they may have different lifetimes $\widetilde{\beta}_1, \widetilde{\beta}_2$, and so cell composition uses their union.

$$\begin{array}{lcl} \textbf{Imm} & \approx & \wp(\textbf{Life}) \times \textsc{Val} \times \textbf{Res} \\ \ell \mapsto \mathrm{I}_\alpha \widehat{\mathrm{P}}(\rho) & \approx & \exists \widetilde{\beta}, \mathsf{v}, \rho'.\ \rho = \ell \mapsto \mathrm{imm}(\widetilde{\beta}, \mathsf{v}, \rho') \wedge \widehat{\mathrm{P}}(\mathsf{v})(\rho') \wedge \alpha \sqsubseteq \bigsqcup \widetilde{\beta} \\ \psi_1 \bowtie \psi_2 & \triangleq & \exists \widetilde{\beta}_1, \widetilde{\beta}_2, \mathsf{v}, \rho.\ \psi_1 = \mathrm{imm}(\widetilde{\beta}_1, \mathsf{v}, \rho) \wedge \psi_2 = \mathrm{imm}(\widetilde{\beta}_2, \mathsf{v}, \rho) \\ \mathrm{imm}(\widetilde{\beta}_1, \mathsf{v}, \rho) \bullet \mathrm{imm}(\widetilde{\beta}_2, \mathsf{v}, \rho) & \triangleq & \mathrm{imm}(\widetilde{\beta}_1 \cup \widetilde{\beta}_2, \mathsf{v}, \rho) \end{array}$$

4.2 Mutable Cells

Replaying the development of immutables for mutables would lead to the following preliminary definition, where the treatment of lifetimes may be simplified to a single lifetime because it is not possible for mutable borrows to alias at all, let alone with different lifetimes.

$$\begin{array}{lcl} \textbf{Mut} & \overset{?}{\approx} & \textbf{Life} \times \textsc{Val} \times \textbf{Res} \\ \ell \mapsto \mathrm{M}_\alpha \widehat{\mathrm{P}}(\rho) & \overset{?}{\approx} & \exists \beta, \mathsf{v}, \rho'.\ \rho = \ell \mapsto \mathrm{mut}(\beta, \mathsf{v}, \rho') \wedge \widehat{\mathrm{P}}(\mathsf{v})(\rho') \wedge \alpha \sqsubseteq \beta \end{array}$$

$$
\begin{array}{rcl}
\textbf{SProp}_\alpha & \triangleq & \textbf{Res}_\alpha \to \mathbb{P} \\
\textbf{Res}_\alpha & \triangleq & \mathsf{Loc} \rightharpoonup \textbf{Cell}_\alpha \\
\textbf{Cell}_\alpha & \triangleq & \mathsf{own}(\textsc{Val}) + \mathsf{imm}(\textbf{Imm}_\alpha) + \mathsf{mut}(\textbf{Mut}_\alpha) \\
\textbf{Imm}_\alpha & \triangleq & \{(\widetilde{\beta} : \wp^+(\textbf{Life}), \mathsf{v} : \textsc{Val}, \rho : \textbf{Res}_{\lfloor\cdot\rfloor\widetilde{\beta}}) \mid \alpha \sqsubseteq \lceil\cdot\rceil\widetilde{\beta}\} \\
\textbf{Mut}_\alpha & \triangleq & \{(\beta \sqsupset \alpha, \mathsf{v} : \textsc{Val}, \rho : \textbf{Res}_\beta, \widehat{\mathsf{P}} : \textsc{Val} \to \textbf{SProp}_\beta) \mid \widehat{\mathsf{P}}(\mathsf{v})(\rho)\} \\
\mathsf{P} \in \textbf{SProp} & \triangleq & \textbf{Res} \to \mathbb{P} \\
\rho \in \textbf{Res} & \triangleq & \mathsf{Loc} \rightharpoonup \textbf{Cell} \\
\psi \in \textbf{Cell} & \triangleq & \bigcup_\alpha \textbf{Cell}_\alpha
\end{array}
$$

$$
\begin{array}{ll}
\rho_1 \bullet \rho_2 & \triangleq \quad (\rho_1 \setminus \rho_2) \uplus (\rho_2 \setminus \rho_1) \uplus [\ell \mapsto \psi_1 \bullet \psi_2 \mid \rho_1(\ell) = \psi_1 \wedge \rho_2 = \psi_2] \\
\mathsf{imm}(\widetilde{\alpha}_1, \mathsf{v}, \rho) \bullet \mathsf{imm}(\widetilde{\alpha}_2, \mathsf{v}, \rho) & \triangleq \quad \mathsf{imm}(\widetilde{\alpha}_1 \cup \widetilde{\alpha}_2, \mathsf{v}, \rho)
\end{array}
$$

Fig. 16. Semantic structures.

However, this definition does not embody mutation—it only says what the *current* witness (v, ρ') is, but does not say what it is allowed—or indeed, required—to be after updates. As in Ahmed's model of higher-order mutable references [2], the solution is to track an *invariant*—a semantic predicate over values that characterizes the space of allowed updates. In the definition above, we would hope to use the payload predicate $\widehat{\mathsf{P}}$ as the invariant. Unfortunately, just as in Ahmed's model, the most straightforward definition is not well-founded.

$$
\begin{array}{rcl}
\textbf{Mut} & \approx & \textbf{Life} \times \textsc{Val} \times \textbf{Res} \times (\textsc{Val} \to \overbrace{\textbf{Res} \to \mathbb{P}}^{\textbf{SProp}}) \\
\textbf{Res} & \approx & \mathsf{Loc} \rightharpoonup \underbrace{\mathsf{own}(\mathsf{v}) + \mathsf{imm}(\textbf{Imm}) + \mathsf{mut}(\textbf{Mut})}_{\textbf{Cell}}
\end{array}
$$

Since Ahmed [2], the prevailing technique for breaking this kind of circularity is to stratify the definitions by a natural number, the step-index, which is tied to computational steps. Unfortunately, traditional step-indexing cannot be used to show liveness properties like termination. As in step-indexing, we break the circularity using stratification, but using a very different measure: the outlives lifetime ordering, $\beta \sqsupset \alpha$. Since every borrow is assigned a fresh lifetime and mutable cells may not alias, it should not be possible for a mutable cell to hold a resource that uses its own lifetime.

Based on this observation, the semantic structures are stratified by a lifetime α, as shown in Fig. 16. Notice that the type of a borrowed cell in $\textbf{Imm}_\alpha$ and $\textbf{Mut}_\alpha$ is dependent on its lifetime. For an imm cell, the shortest among its lifetimes $\lceil\cdot\rceil\widetilde{\beta}$ must still be longer than the stratification index α, and the witness ρ sits in the stratification at the longest among its lifetimes $\lfloor\cdot\rfloor\widetilde{\beta}$. For a mut cell, its lifetime β must be longer than the stratification index α, its witness ρ sits in the stratification at its lifetime β, and so does its predicate $\widehat{\mathsf{P}}$. Additionally, the witness must actually witness the payload predicate $\widehat{\mathsf{P}}(\mathsf{v})(\rho)$. Note that the witness (v, ρ) in a mut cell does not have an analog in [2], but it will be useful in the next section.

4.3 The Weakest Precondition

Before we define the weakest precondition for *BoLo*, we review the total correctness weakest precondition from ordinary separation logic over memory fragments μ.

$$
\mathsf{wp}(\mathsf{e})\{\widehat{Q}\}(\mu) \triangleq \forall \mu_f \# \mu, \exists \mu' \# \mu_f, \mathsf{v}.\ (\mu_f \uplus \mu, \mathsf{e}) \to^* (\mu_f \uplus \mu', \mathsf{v}) \wedge \widehat{Q}(\mathsf{v})(\mu')
$$

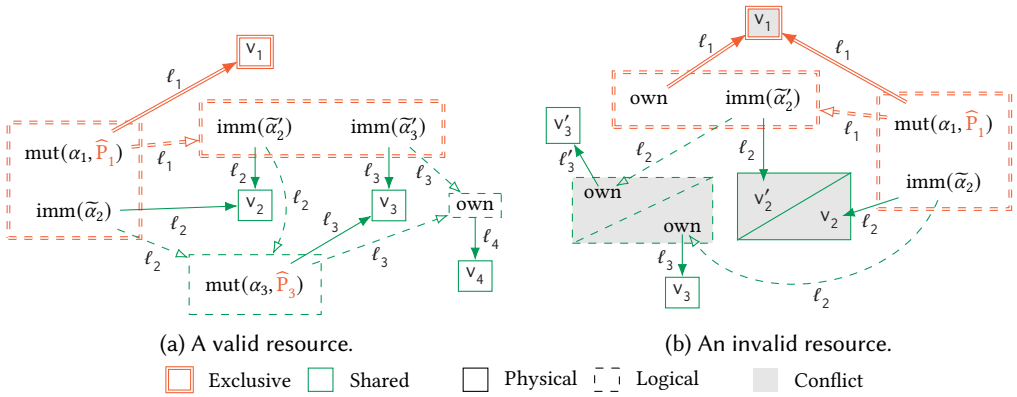


Fig. 17. Visualizing resources.

In general, the weakest precondition is supposed to characterize the memory fragments μ for which e will terminate with a new memory μ' and value v that satisfy the post-condition $\widehat{Q}(\mathsf{v})(\mu')$. In order to validate the frame rule, the definition for separation logic imposes an additional restriction, that arbitrary disjoint *frames* μ_f must be preserved by the execution, which ensures that e only acts *locally* on its memory fragment μ. To define the weakest precondition for *BoLo*, we must close two gaps: lowering logical resources to physical memories and strengthening the frame-preservation condition to support the borrow frame rule (§ 3.2).

Lowering Resources to Memories. Whereas physical memories are flat maps, logical resources are trees with resources for nodes, edges from cells to their contents, and values for leaves. However, at runtime, this logical resource tree must still be representable as a flat physical memory, so it is imperative that every location have an unambiguous value. Therefore, it can be clearer to view the resource as a DAG in which aliases point to the very same object, as shown in Fig. 17a. In this visualization, each node is either a physical value (rendered solid) or a logical resource (rendered dashed) containing cells. Every cell has an edge to its physical value (rendered solid), but a borrow cell has an extra edge to its witness resource (rendered dashed). An edge from a cell is labelled with the location of that cell. A node is considered *shared* (rendered single green) if it is reachable from an imm cell and is considered *exclusive* (rendered double red) otherwise. In the example, notice that aliasing edges, like ℓ_2 or ℓ_3, are incident from imm cells or shared nodes, and that they agree on the nodes they point to. Also, even though ℓ_3 is incident from a mut cell, which would normally be exclusive, it has an imm ancestor, which makes its whole subtree shared and ℓ_3 aliasable.

However, not all resources can be viewed this way—Fig. 17b depicts an example in which aliases point to conflicting nodes (rendered shaded) and cannot be collapsed, like ℓ_2 pointing to both v_2 and v'_2, or having two witnesses own(v_3) and own(v'_3). It also portrays aliasing of exclusive locations, like ℓ_1, that do map to the same cell but, unlike the previous example, are not guarded by immutable cells, which violates the mutability-xor-aliasing restriction. Conceptually, in a *valid* resource, every pair of aliases map to the same object and each has an immutable ancestor.

The weakest precondition assumes validity of the pre-resource and asserts validity of the post-resource, so only valid resources will be mapped to memories. The first step is to *flatten* the resource by composing all the nodes, bottom-up. The advantage of reusing composition is that it already rules out all of the inconsistent aliasing cases mentioned above. In fact, it rules out *too many* cases—composition is never defined on exclusive cells, but they should be allowed to alias if under immutables if their contents are consistent, as with ℓ_2 or ℓ_3 in Fig. 17a. Under immutables, flattening

Validity	$\checkmark\rho$	$\triangleq$	$(\!\lvert\rho\rvert\!)$ defined
Lowering	$[\![\rho]\!]$	$\triangleq$	$[\ell \mapsto v \mid (\!\lvert\rho\rvert\!)(\ell) = v]$
Flattening	$(\!\lvert\rho\rvert\!)$	$\triangleq$	$\mathrm{E}(\!\lvert\rho\rvert\!)_\bullet \bullet \mathrm{A}(\!\lvert\rho\rvert\!)$
↳Exclusive flattening	$\mathrm{E}(\!\lvert\rho\rvert\!)_\bullet$	$\triangleq$	$\rho \mid_{\mathrm{own}} \bullet \rho \mid_{\mathrm{mut}} \bullet \left(\bullet_{\rho' \in \mathrm{cod}(\rho\mid_{\mathrm{mut}})} \mathrm{E}(\!\lvert\rho'\rvert\!)_\bullet\right)$
↳Aliasable flattening	$\mathrm{A}(\!\lvert\rho\rvert\!)$	$\triangleq$	$\rho \mid_{\mathrm{imm}} \circ \left(\circ_{\rho' \in \mathrm{cod}(\rho\mid_{\mathrm{mut}})} \mathrm{A}(\!\lvert\rho'\rvert\!)\right) \circ \left(\circ_{\rho' \in \mathrm{cod}(\rho\mid_{\mathrm{imm}})} \mathrm{E}(\!\lvert\rho'\rvert\!)_\circ \circ \mathrm{A}(\!\lvert\rho'\rvert\!)\right)$
Compatibility	$\rho_1 \# \rho_2$	$\triangleq$	$\rho_1 \bowtie \rho_2 \wedge \checkmark(\rho_1 \bullet \rho_2)$

(a) Lowering resources to memories.

$$\begin{aligned}\rho_1 \leftsquigarrow \rho_2 \quad \triangleq \quad & \forall \ell, \widetilde{\alpha}, \beta, v, \rho, \widehat{P}. \\ & (\!\lvert\rho_1\rvert\!)(\ell) = \mathrm{imm}(\widetilde{\alpha}, v, \rho) \Leftrightarrow (\!\lvert\rho_2\rvert\!)(\ell) = \mathrm{imm}(\widetilde{\alpha}, v, \rho) \qquad (1) \\ & \wedge\ (\!\lvert\rho_1\rvert\!)(\ell) = \mathrm{mut}(\beta, -, -, \widehat{P}) \Leftrightarrow (\!\lvert\rho_2\rvert\!)(\ell) = \mathrm{mut}(\beta, -, -, \widehat{P}) \qquad (2)\end{aligned}$$

(b) Borrow-preserving updates.

Fig. 18. Constructing the weakest precondition.

uses a more permissive variant of the composition operator $\circ$ that is defined only for cells that agree on the payload and have a consistent lifetime. When the cell types differ, this relaxed composition operator takes the more restrictive cell type according to the ordering own < mut < imm, which reflects the legal sequences in which a borrow or reborrow can occur at a location.

The flattening operator $(\!\lvert\rho\rvert\!)$ is defined in Fig. 18a in two parts. The exclusive part of the resource—excluding all immutable cells—is composed bottom-up using the regular composition operator $\bullet$, producing the resource $\mathrm{E}_\bullet(\!\lvert\rho\rvert\!)$. Separately, the aliasable part of the resource is composed using the relaxed composition operator $\circ$, producing the resource $\mathrm{A}(\!\lvert\rho\rvert\!)$. The latter includes would-be exclusive cells that appear under immutables, so it reuses the exclusive walk but with the relaxed operator, $\mathrm{E}_\circ(\!\lvert\rho\rvert\!)$. Finally, the flattening $(\!\lvert\rho\rvert\!)$ is defined to be the strict composition $\mathrm{E}_\bullet(\!\lvert\rho\rvert\!) \bullet \mathrm{A}(\!\lvert\rho\rvert\!)$, since no cell can be both exclusive and aliasable. If $(\!\lvert\rho\rvert\!)$ is defined, the resource is considered to be valid $\checkmark\rho$. After flattening, all reachable cells have been lifted to the same level, meaning the resource is *almost* a flat memory. The final step is to erase the logical cell information, leaving a physical memory $[\![\rho]\!]$. For example, flattening and erasing the resource in Fig. 17a produces the memory $[\ell_i \mapsto v_i \mid i < 4]$. However, flattening the resource in Fig. 17b is not defined—the conflict at ℓ_1 causes $\bullet$ and therefore $\mathrm{E}_\bullet$ to be undefined, while the conflict at ℓ_2 causes $\circ$ and therefore A to be undefined.

Borrow-Preserving Updates. We are still not ready to define the weakest precondition for *BoLo*. Frame preservation alone is not sufficient to support the borrow frame rule, which must be able to restore ownership of the borrowed resource after the borrow has ended. Since a mutable borrow is exclusive, its location cannot be held by any compatible framing resource. Therefore, upgrading a mutable borrow to ownership of its witness is frame-preserving even though it violates the borrowing discipline. We require a stronger condition to ensure that the weakest precondition rejects the following entailment.

$$\ell \mapsto \mathrm{M}_\alpha\, \widehat{P} \not\vdash \mathrm{wp}(())\{\exists v.\ \ell \mapsto v \star \widehat{P}(v)\}$$

In particular, the condition on the post-resource in the weakest precondition must be strengthened to assert that borrows are preserved. The update relation $\leftsquigarrow$, defined in Fig. 18b, requires that all reachable immutable borrows are preserved with their value v witness ρ exactly as is (Clause 1), and that all mutable borrows are preserved up to a change in the witness (Clause 2). Reachability can be expressed as a lookup with the flattened resource $(\!\lvert\rho\rvert\!)$, since it raises all reachable cells to the

$$
\begin{array}{lll}
\ulcorner P \urcorner & (\rho) & \triangleq \rho = \varnothing \wedge P \\
\mathrm{P}_1 \star \mathrm{P}_2 & (\rho) & \triangleq \exists \rho_1, \rho_2 \bowtie \rho_1.\ \rho = \rho_1 \bullet \rho_2 \wedge \mathrm{P}_1(\rho_1) \wedge \mathrm{P}_2(\rho_2) \\
\mathrm{P}_1 \mathrel{-\!\!\star} \mathrm{P}_2 & (\rho_2) & \triangleq \forall \rho_1 \bowtie \rho_2.\ \mathrm{P}_1(\rho_1) \Rrightarrow \mathrm{P}_2(\rho_1 \bullet \rho_2) \\
\ell \mapsto \mathsf{v} & (\rho) & \triangleq \rho = \ell \mapsto \mathrm{own}(\mathsf{v}) \\
\ell \mapsto \mathrm{I}_\alpha\, \widehat{\mathrm{P}} & (\rho) & \triangleq \exists \widetilde{\beta}, \mathsf{v}, \rho'.\ \rho = \ell \mapsto \mathrm{imm}(\widetilde{\beta}, \mathsf{v}, \rho') \wedge \widehat{\mathrm{P}}(\mathsf{v})(\rho') \wedge \alpha \sqsubseteq \lfloor \cdot \rfloor \widetilde{\beta} \\
\ell \mapsto \mathrm{M}_\alpha\, \widehat{\mathrm{P}} & (\rho) & \triangleq \exists \beta \sqsupseteq \alpha, \mathsf{v}, \rho'.\ \rho = \ell \mapsto \mathrm{mut}(\beta, \mathsf{v}, \rho', \widehat{\mathrm{P}}) \\
[\alpha]\mathrm{P} & (\rho) & \triangleq @\rho \sqsupseteq \alpha \wedge \mathrm{P}(\rho) \\
\mathrm{wp}(\mathsf{e})\{\widehat{\mathrm{Q}}\} & (\rho) & \triangleq \forall \rho_f \# \rho, \exists \mathsf{v}, \rho' \# \rho_f, \rho^+ \# (\rho' \bullet \rho^+).\ \rho \leftrightsquigarrow (\rho' \bullet \rho^+) \wedge \rho^+ \mid_{\mathrm{own}} = \varnothing \\
& & \quad \wedge (\llbracket \rho_f \bullet \rho \rrbracket, \mathsf{e}) \to^* (\llbracket \rho_f \bullet \rho' \bullet \rho^+ \rrbracket, \mathsf{v}) \wedge \widehat{\mathrm{Q}}(\mathsf{v})(\rho') \\
\circlearrowleft_\alpha \mathrm{P} & (\rho) & \triangleq @\rho \sqsupseteq \alpha \wedge \exists \rho', \pi : \mathrm{dom}(\rho') \to \mathbf{Res}.\ \rho \geq \bullet_{\ell \in \mathrm{dom}(\rho')} \pi(\ell) \\
& & \quad \wedge \mathrm{P}(\rho') \wedge \forall \ell, \mathsf{v}, \rho_\ell.\ \ell \in \mathrm{dom}(\pi(\ell)) \\
& & \quad\ \wedge \rho(\ell) = \mathrm{own}(\mathsf{v}) \Rrightarrow \rho'(\ell) = \mathrm{imm}(\{\alpha\}, \mathsf{v}, \pi(\ell) \setminus \ell) \qquad (1) \\
& & \quad\ \wedge \rho(\ell) = \mathrm{imm}(-, -, -) \Rrightarrow \rho'(\ell) = \rho(\ell) \wedge \mathrm{dom}(\pi(\ell)) = \{\ell\} \qquad (2) \\
& & \quad\ \wedge \rho(\ell) = \mathrm{mut}(-, \mathsf{v}, \rho_\ell, -) \Rrightarrow \rho'(\ell) = \mathrm{imm}(\{\alpha\}, \mathsf{v}, \rho_\ell) \wedge \mathrm{dom}(\pi(\ell)) = \{\ell\} \qquad (3)
\end{array}
$$

$$
\mathrm{emp} \triangleq \ulcorner \top \urcorner \qquad !\,\mathrm{P} \triangleq \mathrm{emp} \wedge \mathrm{P} \qquad \{\mathrm{P}\}\ \mathsf{e}\ \{\widehat{\mathrm{Q}}\} \triangleq !(\mathrm{P} \mathrel{-\!\!\star} \mathrm{wp}(\mathsf{e})\{\widehat{\mathrm{Q}}\}) \qquad \curlywedge\, \widehat{\mathrm{P}} \triangleq \exists\, \beta, \forall\, \alpha \sqsupseteq \beta.\ \widehat{\mathrm{P}}(\alpha)
$$

Fig. 19. Modelling *BoLo* propositions (excerpts).

root, as mentioned in the previous subsection. Notice that this definition heavily relies on tracking the witness resource in a mutable cell: if only the invariant were recorded, then writing a resource into a mutable cell would not preserve reachability of that resource's borrows.

The final weakest precondition definition, given in Fig. 19, uses the update relation to further constrain the post-resource $(\rho' \bullet \rho^+)$: not only must frames ρ_f be preserved, but also borrows $\rho \leftrightsquigarrow (\rho' \bullet \rho^+)$. Since the update relation constrains the reachable borrows to be exactly the same before and after running an expression, this definition embodies *lexical* borrowing. Inspired by Charguéraud and Pottier [4], in order to support `forget` on borrows, the post-condition is only required to hold of a fragment ρ' of the post-resource, but the discarded fragment ρ^+ must not contain any owned cells. This is essential for the memory reclamation component of adequacy (Theorem 3.2), which insists that owned cells are freed rather than forgotten.

4.4 Outlives, Freshness, and Reborrowing

Excerpts from the rest of the model are given in Fig. 19. The standard intuitionistic and separation logic propositions are defined as usual under a linear interpretation. We include an unrestricted modality $!\,\mathrm{P}$ for propositions that hold with the empty resource, which is similar to persistence modality $\Box P$ in Iris. As with the persistence modality, Hoare triples in our logic are defined in terms of the unrestricted modality and the weakest precondition.

The only bespoke connectives that remain are the outlives modality, the freshness quantifier, and the reborrowing modality. The lifetime of a resource $@\rho$ is defined to be the shortest among the lifetimes in its cells, which the outlives modality $[\alpha]\mathrm{P}$ uses to restrict P to the resources that strictly outlive α. The proof that $\mathrm{P} \vDash \exists\, \alpha.\ [\alpha]\mathrm{P}$ can then pick α to be the next shortest lifetime $\downarrow @\rho$ for any ρ. The freshness quantifier $\curlywedge\, \alpha.\ \mathrm{P}$ is actually a derived form: it states that there exists a lower bound β such that P will outlive any shorter lifetime α. For a particular resource ρ, the choice of β will be no longer than its lifetime $@\rho$, but proving some of the rules in Fig. 12b require it to be potentially shorter; for example, the rule $\curlywedge \star$ uses the join of the component bounds.

The final interesting connective is the reborrow modality $\circlearrowleft_\alpha \mathrm{P}$. To start, it only holds of resources ρ that live longer than α, which is required for lifetime stratification. It is a $\diamond$-style modality; it existentially quantifies over resources ρ' that satisfy the given proposition $\mathrm{P}(\rho')$. Informally, this

resource ρ' must be just like a *subresource* of the original resource ρ, except that its exclusive locations are marked imm at the given lifetime α and its imm locations are preserved at their original lifetimes. Formally, this is stated by constructing a location-indexed partial partition π whose domain matches that of the new resource ρ' and whose codomain composes to a subresource ($\leq^1$) of the original resource ρ. The conditions placed on each partition ℓ in π depends on the type of cell at ℓ in the original resource ρ. In any case, the cell in question must be contained in the partition $\pi(\ell)$. If the cell was owned in ρ (Case 1), then it will be turned into an immutable cell at lifetime α in the reborrowed resource ρ' using the original value v and the remainder of the partition $\pi(\ell) \setminus \ell$ for the witness resource. If the cell was immutable in ρ (Case 2), then it will be preserved as is in the reborrowed resource ρ'. Finally, if the cell was mutable in ρ (Case 3), then it will be turned into an immutable cell at lifetime α in the reborrowed resource ρ' using the original value v and witness resource ρ_ℓ. For a borrow cell (Cases 2–3), the partition $\pi(\ell)$ is additionally constrained to contain only that cell, since it already has a witness resource.

5 Related Work and Discussion

Borrowing Formalisms. Before Rust, the `let!` construct of Wadler [30] is the earliest example we can find of a borrowing-like construct, which temporarily allows linear data to be treated unrestrictedly in a lexical scope if used in an immutable way. Similar constructs can be found in [19, 20], but none of these systems support mutable borrows or lifetimes. The `freeze` construct of Ahmed et al. [1] allows linear references to be turned into shared ML-style mutable references, but the type system does not ensure that this conversion is temporary.

Rust [17] developed and popularized the idea of borrowing and supports features not addressed by our type system, like non-lexical lifetimes. Weiss et al. [32] formalized surface Rust and proved syntactic type soundness, while Jung et al. [11] formalized an intermediate form of Rust and proved semantic type soundness. These efforts cover more advanced features from Rust, like non-lexical lifetimes and interior mutability for shared references, but are far removed from the linear lambda calculus.

Tree Borrows [29] (preceded by Stacked Borrows [10]) is a permission-based augmentation to an operational semantics that dynamically enforces Rust borrows in the presence of unsafe code. While our work is focused on the type system and logic, there are similarities between our borrow resources and Tree Borrows' permissions. In Tree Borrows, references derived from an immutable borrow are all immutable, analogous to how our immutable borrows impose immutability in its sub-resource. But our model enforces deep immutability of the data an immutable borrow points to, while Tree Borrows only maintains a shallow immutability of the reference itself.

Lorenzen et al. [15] imports ideas from Rust into OCaml in the form of modes that indicate permissions and locality constraints that loosely resemble the different borrowing forms. While this initial version uses progress and preservation, the follow-up work by Georges et al. [7] proves semantic type soundness and generalizes to the concurrent setting.

The type system of Radanne et al. [25] is arguably the most similar to ours; it starts from a variant of ML called Affe with an abstract resource API that encodes substructural constraints using kinds. As in our calculus, borrows have lexical lifetimes, but they are incorporated by extending the term language, whereas our borrowing constructs are implemented in the underlying language. Whereas we use the standard notion of context splitting from linear logic, Affe defines a bespoke ordered splitting operation that is type-sensitive.

Marshall and Orchard [16] incorporate a version of lexical borrowing into Granule [21], which is based on graded modal type theory. However, their borrowing construct requires that borrows

[1]Where $\rho_1 \leq \rho_3 \triangleq \exists \rho_2 \bowtie \rho_1.\ \rho_1 \bullet \rho_2 = \rho_3$.

be explicitly returned at the end of their scope, which does not alleviate capability-passing style, and their mutable borrows are not type-preserving in general. Reinking et al. [26] develop a notion of lexical borrowing for reference counting which is used to help optimize away some increment/decrement calls and reuse space. Different than the other type systems, borrowing is mostly obscured from the programmer, and a borrowed binding can always be upgraded by inserting an increment.

Program Logics. The logic that is most similar to ours is that of [4], whose "read-only frame rule" was the inspiration for our borrow frame rule. Their analogue of an immutable borrow is the read-only modality, which distributes over most connectives freely instead of requiring a separate notion of reborrowing, like our logic does. Our logic additionally incorporates lifetimes and mutable borrows.

Jung et al. [11] develop the Lifetime Logic for reasoning about Rust's non-lexical lifetimes in Iris [12]. They employ it to prove semantic type soundness for their Rust formalization and to verify an abstraction of the Rust standard library. Non-lexical lifetimes are supported by explicitly threading around a lifetime token that provides the capability to access borrows at that lifetime. Also in Iris, the Leaf library [8] provides generalized abstractions for temporary sharing of resources, but they have not yet been used to encode borrows and lifetimes.

There would be two main challenges in mechanizing *BoLo* with Iris. First, Iris is affine instead of linear, so one would either need to abandon leak freedom or use a linear fork of [9] or library for [3] Iris. Second, our tree-shaped resource is not directly expressible using Iris' built-in combinators, nor is it compatible with Iris' solver for recursive domain equations, so one would either need to solve the equations manually or devise an alternative representation more suitable for Iris.

Semantic Models. Mutable borrows are like traditional ML-style references in that they only support type-preserving updates, but they are exclusive. Since Ahmed [2], models of mutable references typically use step-indexing to break the kind of circularity observed in § 4.2, but traditional step-indexing is not amenable to proving liveness properties like termination. Transfinite step-indexing [28] is a variation that uses ordinals instead of naturals and can be employed to show liveness properties. In our case, the lifetime stratification inherent to the type system is enough to avoid the use of step-indexing altogether, which is similar to the universe stratification recently proposed by Koronkevich and Bowman [13].

Language Extensions. Our hope is that `BoCa` will be extended, varied, or mixed into other calculi. Including additional pure constructs, like records or variants, would be straightforward since they would not require changes to the model, though one would likely wish to add corresponding reborrowing rules. On the other hand, including non-terminating constructs would require step-indexing the model [2] in the standard way and weaken the leak freedom property to hold only on terminating executions. Adding new memory manipulation constructs may or may not require changes to the model, depending on whether they can use the same—or at least a similar—memory model. For example, in ongoing follow-up work, we are using a variant of `BoCa` with CompCert-style block-based memory [14], which does require redoing the model, but the changes are mostly mechanical, not conceptual.

Acknowledgments

We thank Zack Eisbach and the anonymous reviewers for their careful feedback and suggestions. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. N66001-21-C-4023 and Agreement No. HR00112590130.

Data-Availability Statement

Complete definitions and proofs may be found in our supplementary material [31].

References

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

[2] Amal Jamil Ahmed. 2004. *Semantics of types for mutable state.* Princeton University.

[3] Aleš Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. 2019. Iron: Managing obligations in higher-order concurrent separation logic. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

[4] 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.

[5] The Idris Community. 2020. !-notation. https://docs.idris-lang.org/en/latest/tutorial/interfaces.html#notation

[6] Matthias Felleisen. 1990. On the expressive power of programming languages. In *European Symposium on Programming*. Springer, 134–151.

[7] Aïna Linn Georges, Benjamin Peters, Laila Elbeheiry, Leo White, Stephen Dolan, Richard A Eisenberg, Chris Casinghino, François Pottier, and Derek Dreyer. 2025. Data Race Freedom à la Mode. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 656–686.

[8] Travis Hance, Jon Howell, Oded Padon, and Bryan Parno. 2023. Leaf: Modularity for temporary sharing in separation logic. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 31–58.

[9] Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. 2024. Deadlock-free separation logic: Linearity yields progress for dependent higher-order message passing. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 1385–1417.

[10] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked borrows: an aliasing model for Rust. *Proc. ACM Program. Lang.* 4, POPL, Article 41 (Dec. 2019), 32 pages. doi:10.1145/3371109

[11] 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.

[12] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20.

[13] Paulette Koronkevich and William J Bowman. 2024. Type Universes as Allocation Effects. *arXiv preprint arXiv:2407.06473* (2024).

[14] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

[15] Anton Lorenzen, Leo White, Stephen Dolan, Richard A Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with modal memory management. *Proceedings of the ACM on Programming Languages* 8, ICFP (2024), 485–514.

[16] Daniel Marshall and Dominic Orchard. 2024. Functional Ownership through Fractional Uniqueness. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 1040–1070.

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

[18] Greg Morrisett, Amal J. Ahmed, and Matthew Fluet. 2005. L^3: A Linear Language with Locations. In *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3461)*, Pawel Urzyczyn (Ed.). Springer, 293–307. doi:10.1007/11417170_22

[19] 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).

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

[21] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–30.

[22] Daniel Patterson and Amal Ahmed. 2017. Linking Types for Multi-Language Software: Have Your Cake and Eat It Too. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 71)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 12:1–12:15. doi:10.4230/LIPIcs.SNAPL.2017.12

[23] Andrew M Pitts. 2003. Nominal logic, a first order theory of names and binding. *Information and computation* 186, 2 (2003), 165–193.

[24] 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.

[25] 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.
[26] 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.
[27] John C Reynolds. 1983. Types, abstraction and parametric polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congres*. 513–523.
[28] Simon Spies, Neel Krishnaswami, and Derek Dreyer. 2021. Transfinite step-indexing for termination. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29.
[29] Neven Villani, Johannes Hostert, Derek Dreyer, and Ralf Jung. 2025. Tree Borrows. *Proc. ACM Program. Lang.* 9, PLDI, Article 188 (June 2025), 24 pages. doi:10.1145/3735592
[30] Philip Wadler. 1990. Linear types can change the world!. In *Programming concepts and methods*, Vol. 3. Citeseer, 5.
[31] Andrew Wagner, Olek Gierczak, Brianna Marshall, John M. Li, and Amal Ahmed. 2025. From Linearity to Borrowing (Supplementary Material). *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 415 (Oct. 2025). doi:10.1145/3764117
[32] Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. 2021. Oxide: The Essence of Rust. arXiv:1903.00982 [cs.PL] https://arxiv.org/abs/1903.00982

Received 2025-03-26; accepted 2025-08-12