Data Race Freedom à la Mode

AÏNA LINN GEORGES, MPI-SWS, Germany
BENJAMIN PETERS, MPI-SWS, Germany
LAILA ELBEHEIRY, MPI-SWS, Germany
LEO WHITE, Jane Street, UK
STEPHEN DOLAN, Jane Street, UK
RICHARD A. EISENBERG, Jane Street, USA
CHRIS CASINGHINO, Jane Street, USA
FRANÇOIS POTTIER, Inria, France
DEREK DREYER, MPI-SWS, Germany

We present DRFCaml, an extension of OCaml's type system that guarantees data race freedom for multi-threaded OCaml programs while retaining backward compatibility with existing sequential OCaml code. We build on recent work of Lorenzen et al., who extend OCaml with *modes* that keep track of locality, uniqueness, and affinity. We introduce two new mode axes, *contention* and *portability*, which record whether data has been shared or can be shared between multiple threads. Although this basic type-and-mode system has limited expressive power by itself, it does let us express APIs for *capsules*, regions of memory whose access is controlled by a unique ghost key, and *reader-writer locks*, which allow a thread to safely acquire partial or full ownership of a key. We show that this allows complex data structures (which may involve aliasing and mutable state) to be safely shared between threads. We formalize the complete system and establish its soundness by building a semantic model of it in the Iris program logic on top of the Coq proof assistant.

CCS Concepts: • **Computing methodologies → Concurrent programming languages**; • **Theory of computation → Type theory**; **Separation logic**.

Additional Key Words and Phrases: Concurrency, data races, type systems, OCaml, separation logic, Iris, Coq

ACM Reference Format:
Aïna Linn Georges, Benjamin Peters, Laila Elbeheiry, Leo White, Stephen Dolan, Richard A. Eisenberg, Chris Casinghino, François Pottier, and Derek Dreyer. 2024. Data Race Freedom à la Mode. 1, 1 (October 2024), 39 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

A central challenge of multi-threaded programming is ensuring the absence of *data races*, in which one thread accesses some shared non-atomic data while another thread is simultaneously mutating it. Data races lead programs to behave in ways that are unexpected, difficult to explain, or (in languages like C/C++) completely undefined. Consequently, there has been a great deal of work on static prevention of data races. Among the most promising techniques is that of the Rust programming language, which employs a substructural (or "ownership-based") type system to guarantee absence of data races at compile time. In particular, it uses ownership to enforce the discipline of *aliasing XOR mutability* (or AXM): data can be *aliased* (*i.e.,* have multiple references to

Authors' addresses: Aïna Linn Georges, algeorges@mpi-sws.org, MPI-SWS, Saarland Informatics Campus, Germany; Benjamin Peters, bpeters@mpi-sws.org, MPI-SWS, Saarland Informatics Campus, Germany; Laila Elbeheiry, lelbehei@mpi-sws.org, MPI-SWS, Saarland Informatics Campus, Germany; Leo White, lwhite@janestreet.com, Jane Street, London, UK; Stephen Dolan, sdolan@janestreet.com, Jane Street, London, UK; Richard A. Eisenberg, reisenberg@janestreet.com, Jane Street, New York, USA; Chris Casinghino, ccasinghino@janestreet.com, Jane Street, New York, USA; François Pottier, francois.pottier@inria.fr, Inria, France; Derek Dreyer, dreyer@mpi-sws.org, MPI-SWS, Saarland Informatics Campus, Germany.

2024. ACM XXXX-XXXX/2024/10-ART
https://doi.org/XXXXXXX.XXXXXXX

it) or it can be *mutable*, but it cannot be both at the same time. This discipline in turn ensures that if two threads can access some shared data at the same time, then neither can have mutable access to it, thus ruling out the possibility of data races.

The increasing industry adoption of Rust is remarkable: it demonstrates the power and flexibility of substructural/ownership type systems, and is the most widely deployed example of such a system in practice. However, its success also comes at a cost [11, 1]: the Rust programmer must think about ownership of data at a fine granularity, and take care of how it evolves (flow-sensitively) throughout the program. This cost is arguably unavoidable and even desirable in the context of low-level systems programming with manual memory management, since the same AXM discipline that Rust uses to prevent data races also helps to prevent other dangerous anomalies (such as memory safety violations) which have long plagued C/C++ programs. But in the context of higher-level programming languages with automatic memory management, programmers are accustomed to much simpler and less restrictive type systems than Rust's—type systems which permit arbitrary aliasing of mutable data structures without sacrificing safety. Having to adhere to Rust's AXM discipline throughout one's program may seem a steep price to pay just for data race freedom.

In much the same spirit as recent work by Xu et al. [30], we therefore ask: is it possible to guarantee absence of data races in a high-level programming language without giving up on the "comfort" of its type system? More concretely, can we incorporate some of Rust's core ideas into an existing, high-level, garbage-collected programming language in such a way that

1. the design is *backward-compatible* with the existing language, *i.e.,* legacy sequential code continues to type-check and function as is, but
2. when writing multi-threaded programs, to ensure the absence of data races, one can employ a lightweight form of ownership tracking when needed, in a "pay as you go" manner?

1.1 DRFCaml

In this paper, we explore the above question in the context of OCaml 5, the recent release of OCaml supporting multi-threading. As in Java, data races in OCaml have well-defined semantics [25], but may result in surprising (and incorrect!) behaviors.[1] To avoid these bugs, the programmer is responsible for ensuring that programs are well-synchronized. However, as it stands, OCaml offers no help to the programmer in checking that they have done so.

We propose DRFCaml, a type system extending OCaml's in order to guarantee data race freedom for multi-threaded OCaml programs while remaining backward compatible with existing OCaml code. DRFCaml takes as its starting point recent work by Lorenzen et al. [21], which extends OCaml's type system with *modes* for tracking locality, uniqueness, and affinity of data. Lorenzen et al. use these modes to safely support stack allocation, memory reuse, and a syntactically scoped form of Rust-style "borrowing" for code that wishes to use these features, without requiring changes to existing OCaml code. Their type system has been implemented and deployed at Jane Street, where it has been widely adopted [21]. This suggests that their approach to mode inference is backward-compatible with a large legacy code base. However, their system focuses on the sequential fragment of OCaml.

DRFCaml extends Lorenzen et al.'s mode system with additional mode axes for safe concurrent programming, which we call *contention* and *portability*. The contention axis tracks how data can be safely accessed in the presence of multi-threading: immutable data is always safe to access, but mutable data can be accessed safely only if it is *uncontended*, *i.e.,* guaranteed not to be accessed simultaneously from another thread. The portability axis tracks whether values are safe to be

---

[1] The fact that Java and OCaml have weak memory models increases the range of surprising behaviors that can be caused by data races. However, it is usually desirable to detect and rule out data races, under any memory model.

```
let tbl = RwHashtbl.create () in
(* tbl is contended and can thus be used in a portable closure *)
let t1 = Thread.create (fun () -> RwHashtbl.add tbl 1 "string1") () in
let t2 = Thread.create (fun () -> RwHashtbl.add tbl 2 "string2";
                                  assert(RwHashtbl.find tbl 2 = "string2")) () in ...
```

Fig. 1. A simple example client of `RwHashtbl`

shared between threads, the most interesting case being closures: a closure is *portable* (safe to share between threads) so long as it does not capture any uncontended references in its environment, as such capture would indirectly cause those references to become contended.

The contention and portability modes work jointly to enforce a variant of Rust's AXM discipline: uncontended data can be mutated freely; but once data is shared between threads, it can no longer be mutated. As in Rust, this discipline guarantees data race freedom, but it comes at the expense of disallowing any sharing of mutable state across threads—a significant restriction, since *some* form of shared mutable state is needed to implement communication between threads. Fortunately—also as in Rust—the basic discipline can be safely relaxed by extending the core type system of DRFCaml via *APIs with interior mutability*, *i.e.*, APIs which allow shared data to be mutated in a carefully controlled manner, ensuring that sufficient synchronization is used to avoid data races.

1.2 Modal APIs with Interior Mutability: Capsules and Reader-Writer Locks

In this paper, in addition to presenting the modal type system of DRFCaml, we show how to extend its power with several interior-mutable APIs. We demonstrate the utility of these APIs on a representative example: we take a sequential hash table, written in vanilla OCaml, and make it thread-safe (that is, safely shareable between several threads) by protecting access to it with a reader-writer lock, and adding a few annotations on function signatures and reference allocations. Concretely, we present two APIs:

Capsules enable uncontended data—*with arbitrary internal aliasing*—to be safely shared between threads through the use of a *ghost key* (or "capability", a zero-sized value used to enforce synchronization) whose ownership is statically tracked by the type system. If a thread has unique ownership of the key, it can mutate the shared data stored in the capsule. If a thread merely possesses an aliased key, it can obtain only read access to the shared data. Capsules are inspired by the `GhostCell` API proposed for Rust [31] (see §7 for a comparison).

Reader-writer locks synchronize access to a resource (such as a key) using standard concurrency primitives (*e.g.*, compare-and-swap) under the hood. In particular, we use reader-writer locks to safely transfer unique or shared ownership of a key between threads.

With the above APIs in hand, we can take, for example, `Hashtbl`, a pre-existing sequential implementation of a hash table data type in OCaml, and transform it into a thread-safe version, `RwHashtbl`. Fig. 1 shows a client of the thread-safe `RwHashtbl`. It creates a hash table, forks two threads, and uses the operations of `RwHashtbl` to safely perform concurrent reads and writes to the hash table without fear of data races. Crucially: (1) the implementation of `RwHashtbl` can reuse the original sequential implementation of `Hashtbl` essentially as is (modulo annotations on reference allocations), and (2) the client of `RwHashtbl` need not know anything about DRFCaml's mode system except for the fact that the type `RwHashtbl.t` is contended and portable (meaning that all the operations accept and produce contended and portable values of type `RwHashtbl.t`), so that hash tables can be safely shared across threads. (The implementer of `RwHashtbl`, on the other hand, must have a deeper understanding of modes.)

As the Capsule and Reader-Writer Lock APIs fundamentally extend the power of the core DRFCaml type system, their implementations require the use of unsafe escape hatches, such as OCaml's `Obj.magic`, and unsafe mode casts. To establish that these APIs are nonetheless safe and do not allow data races, we employ a now-standard approach: we build a semantic model of the DRFCaml type system in the Iris separation logic [19], and use this model to establish semantic soundness of the typing rules of DRFCaml along with the Capsule and Reader-Writer Lock APIs. This "logical approach to type soundness", exemplified by the work on RustBelt [18] and documented in a pedagogical fashion by Timany et al. [26], provides a solid foundation for DRFCaml, and lets us imagine that its basic design can be extended with other useful APIs in the future.

1.3 Contributions

In summary, we make the following contributions:

- We present DRFCaml, an extension of a core subset of OCaml that uses *modes* to statically rule out data races without sacrificing backward compatibility or automatic memory management. Because we build directly on the modal framework of Lorenzen et al. [21], we believe that a design based on DRFCaml has the potential to be deployed at scale in the near future.
- We present a modal API for *capsules*, which allows mutable data—constructed in vanilla OCaml with no tracking of aliasing—to be safely shared between threads by protecting it with a *key*. We also present a modal API for *reader-writer locks*, which enables ownership of keys to be properly synchronized between threads.
- We illustrate the power of these APIs, by showing how to use them to convert a sequential OCaml hash table into a thread-safe one with minimal effort.
- We formalize the static and dynamic semantics of DRFCaml and the aforementioned APIs in Coq, and build a semantic model in Coq/Iris in order to verify the soundness of the entire system. All results in this paper have been mechanized in Coq (see supplementary material).

The rest of the paper is structured as follows. In §2, we give a tour of DRFCaml, as well as the Capsule and Reader-Writer Lock APIs, by example. In §3 and §4, we present formal details of DRFCaml and its type system. In §5 and §6, we discuss the proof of semantic soundness of the type system and the two APIs. Finally, in §7, we provide an extensive comparison with related work.

2 A TOUR OF MODAL PROGRAMMING IN DRFCAML

In DRFCaml, a *mode* is a tuple of several pieces of information. Each component of this tuple concerns a specific aspect, or *axis*. For instance, on the *locality* axis, a tuple component can be either **local** or **global**; on the *uniqueness* axis, a tuple component can be either **unique** or **aliased**; and so on. In this section, we recall the three axes introduced in previous work by Lorenzen et al. [21], namely *locality* (§2.1), *uniqueness*, and *affinity* (§2.2). We recall that the effect of a mode is *deep* but can be stopped by an explicit *modality* (§2.3). Then, we reach the contributions of this paper. To forbid data races, we introduce two new axes, namely *contention* and *portability* (§2.4). We point out that all legacy (sequential) OCaml code remains well-typed (§2.5), and describe the mode at which all legacy OCaml code type checks: the **legacy mode**. Next, we discuss the interaction of modes and mutable references (§2.6). Then, we propose two original APIs, namely the Capsule API (§2.7) and the Reader-Writer Lock API (§2.8), which allow multiple threads to safely access shared mutable data structures. These APIs have special status: although the *type* of each operation can be expressed using our type-and-mode system, the *implementations* of these operations do not satisfy the strict rules imposed by our type-and-mode checker. Thus, to prove that these APIs are safe, we must verify that these implementations are *semantically well-typed.* This is the topic of §5 and §6.

2.1 Locality Axis

The locality axis allows users to express the *lifetime* of a value. A mode, projected onto this axis, is either **local** or **global**. The lifetime of a **local** value is restricted to the current *region*.[2] A **global** value, on the other hand, has indefinite (permanent) lifetime. Legacy OCaml values behave like global values. As such, the legacy mode will be **global** in the locality axis (see §2.5). This means that if no annotation is given, a value is considered **global** by default.

The distinction between **local** and **global** is coarse-grained. Our system is less expressive than Rust's, which allows the lifetime of a value to be tied to a *specific* region (not just the *current* region) via so-called lifetime variables. Our approach makes our system a simple, non-intrusive addition to the OCaml type system. While Lorenzen et al. [21] describe how this facility allows stack allocation of local values, our interest is that this axis allows granting *temporary access* to a value. For example, consider the following program fragment:

```
(* Suppose f : int ref @ local -> unit *)
let x : int ref = ref 1 in let y : int ref = ref 2 in
f x; x := 42; f y; assert (!x = 42)
```

Here, the unknown function f takes an integer reference as a parameter, and returns nothing. In the type of f, this parameter is annotated with **local**. This means that f *promises* to treat its parameter as a value whose lifetime is limited to this invocation of f. In other words, f promises *not to retain access* to this parameter after it returns, for example by storing it to a location that survives the function call. In this example, thanks to this promise, one can reason that, once the call f x ends, f has lost access to x, so the call f y cannot affect x. Therefore, the final `assert` statement must succeed.

The locality feature both powers optimizations, such as stack allocation, and also helps to reason about programs. In fact, locality plays a crucial role in our system, and is exploited in the Capsule and Reader-Writer Lock APIs (§2.7 and 2.8).

Let us now offer two concrete examples where a function f accepts a **local** parameter and attempts to let it escape. In these examples, we assume that `t` is an arbitrary type; `t` could be, say, `int ref`, but its definition is irrelevant. Here is the first example:

```
let sm @ global : t ref = ref (...)
let f : t @ local -> unit = fun x -> sm := x
          Error: value escapes its region ^
```

In this example, f attempts to store the value x, which it has received as a **local** parameter, into the **global** reference sm. Since sm has a permanent lifetime, such a store would allow x to outlive this invocation of f. Thus, the type system forbids the store instruction `sm := x`.

The next example displays a slightly more subtle violation of the type discipline:

```
let sm @ global : (unit -> t) ref = ...
let f : t @ local -> unit = fun x -> sm := (fun () -> x)
          Error: value escapes its region ^
```

In this example, f tries to smuggle x through a *closure*: that is, it attempts to store a closure, which captures the value x, into the **global** reference sm. To prevent this, the type system imposes a restriction on closures: a closure that captures a **local** variable must itself be **local**. As a result, the store instruction is again forbidden.

---

[2] In short, each function body forms a region. For more details, see Lorenzen et al. [21, §6.2, §6.3].

2.2 Uniqueness and Affinity Axes

The *uniqueness* axis supports a form of *ownership* reasoning. Lorenzen et al. use uniqueness to achieve memory reuse and allow in-place updates. We need uniqueness for a different reason: our Capsule API (§2.7) introduces a notion of *keys*, which serve as capabilities to access a data structure. These keys must be unique.

A **unique** value is a value that has not been duplicated in the past, so the copy that we have is the unique copy. In particular, if this value is a pointer, then we have unique access to—or *ownership* of—the data structure at this address. **aliased**[3] is the negation of **unique**: an **aliased** value may have been duplicated in the past; there may exist several copies of it, so we cannot assume that we have unique access. If no annotation is given, a value is considered **aliased**. This will be the default for all legacy OCaml values.

It is worth noting that uniqueness is not required in order to mutate a reference. Unlike Rust, we do *not* enforce an AXM discipline. In fact, our goal is precisely to allow a reference to become **aliased**, since this enables us to type-check legacy OCaml code. Instead, we use uniqueness to characterize a value as a capability. For example, consider this program fragment:

```
(* Suppose delete : key @ unique -> unit *)
let x @ unique : key = ... in delete x; delete x
          Error: x cannot be treated as unique ^
```

The function delete expects a key, and returns nothing. Because the key is marked **unique**, it is consumed by delete. Thus, the second call to delete is illegal.

The uniqueness axis provides information about the past: it tells us whether a value has been duplicated. It does not forbid duplicating this value in the future. For example, if x is passed to a function that expects an **aliased** key, x may be (implicitly) downgraded from **unique** to **aliased** via submoding, and can no longer be used as a capability. Limiting future use of a value is the role of the *affinity* axis. Along this axis, **once** indicates that a value must be used at most once, whereas **many** allows a value to be used as many times as one wishes. The uniqueness and affinity axes interact via a simple rule: a closure that captures a **unique** variable must be **once**. To see why this rule is necessary, consider the following program:

```
(* Suppose delete : key @ unique -> unit *)
let x @ unique : key = ... in
let f = (fun () -> delete x) in List.iter f l
   Error: f cannot be used multiple times ^
```

Each call to `f()` causes a call to `delete x`. We have just explained that, because the key `x` is **unique**, calling `delete x` twice in succession is disallowed. Thus, the function `f` must not be called twice: it must be **once**. In the above example, `List.iter` may call `f` several times, so it requires `f` to be **many**. As a result, this example is ill-typed.

We end this subsection with a remark on *borrowing*. While a **unique** value can be downgraded to an **aliased** one, this change cannot be undone: modes can only be weakened. This is a severe restriction: if one wishes to use a **unique** value several times, then its uniqueness must be given up and cannot be recovered. To alleviate this limitation, Lorenzen et al. [21] use a form of borrowing, a construct that transforms a possibly **unique** value v into an **aliased** and **local** value during the execution of a subexpression e, and thereafter reestablishes the original mode of this value. Their notion of borrowing is simpler but more restricted than Rust's, due to the coarse-grained nature of locality.

---

[3] In previous work [21], **aliased** was named **shared**. In this paper, though, we reserve the name **shared** for a different purpose (§2.4).

2.3 Deep Modes and Modalities

So far, we have illustrated the meaning of modes by examining simple "atomic" values, such as an integer reference. New questions arise when one wishes to work with composite values, such as tuples. For instance, consider the following program:

```
let f : int ref @ aliased -> int ref @ unique -> int ref * int ref @ ?
= fun x y -> (x, y)
```

The function `f` expects an **aliased** parameter `x` and a **unique** parameter `y` and returns the pair `(x, y)`. The question is: what mode should this pair carry?

By convention [21, §2.1], modes are *deep*. That is, mode annotations take effect in depth: if a tuple has mode `m` then it is understood that each component has mode `m` as well. Thus, in the above example, the question mark cannot be replaced with **unique**: that would require converting `x` from **aliased** to **unique**, which is forbidden. The question mark *can* be replaced with **aliased**, as it is safe to convert `y` from **unique** to **aliased**. However, doing so would cause a loss of information: the uniqueness of `y` would be forgotten. To circumvent this limitation, a type can be decorated with a mode: the type `'a @@ m` denotes a value of type `'a` at mode `m`. This construct is known as a *modality*.[4] Taking advantage of this feature, in the previous example, one can treat the pair as **unique**, yet with the caveat that its first component is **aliased**. The return type and mode of `f` are then `((int ref @@ aliased) * int ref) @ unique`.

2.4 Contention and Portability Axes

We now reach the first contribution of this paper: we introduce two new axes, namely *contention* and *portability*, whose purpose is to keep track of (and to restrict) the way in which mutable data is shared between threads (immutable data can never cause a data race, and is thus unaffected by these axes).

Many previous type systems and program logics (such as Rust and Concurrent Separation Logic with fractional points-to assertions) prevent data races by ensuring that a value is never at the same time mutable and aliased. However, because we want all legacy (sequential) OCaml code to be well-typed, we do not wish to impose such a strong restriction.

Thus, we introduce a new axis, *contention*, with the following three modes and submoding relation: **uncontended** $\leq$ **shared** $\leq$ **contended**. In short, a value is **uncontended** if mutable fields within this value are accessible for reading and writing by the current thread (and inaccessible to other threads), **shared** if mutable fields within this value are accessible only for reading by the current thread (and possibly accessible for reading to other threads as well), and **contended** if mutable fields within this value are not accessible at all to the current thread.

A reference can be written only if it is **uncontended**, and can be read only if it is **shared** or **uncontended**. For example, the following program is ill-typed, as it attempts to update a **contended** reference:

```
let f : int ref @ contended -> unit = fun x -> x := 42
                         Error: potential data race ^
```

While the contention axis is on the one hand prescriptive (it restricts future read and write accesses), it is also descriptive: it expresses information about the past, namely whether a value has been transmitted to other threads. It is natural (and in fact necessary) to introduce a dual axis, *portability*, which determines whether a value may be transmitted to another thread in the future.

---

[4] Not every mode has a corresponding modality: for instance, the modality `'a @@ aliased` exists, but the modality `'a @@ unique` does not. For further details, see §4.

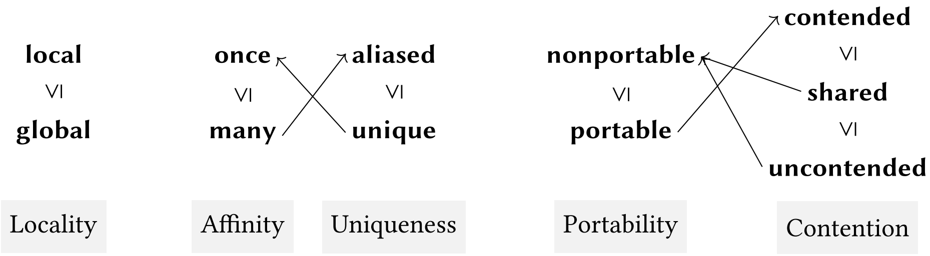

Fig. 2. The full collection of modes

Along this axis, we introduce two points: a **portable** value can safely be transmitted to another thread; a **nonportable** value cannot. The submoding relation is **portable** ≤ **nonportable**.

The contention and portability axes interact through the following rule: if a closure captures an **uncontended** or **shared** value, then this closure must be **nonportable**. In the case of an **uncontended** value, it is easy to see why this rule is necessary: if a closure has read-write access to a mutable value then allowing this closure to be invoked by multiple threads would cause a data race. In the case of a **shared** value, the reason is more subtle; we come back to this point shortly.

As a result of this rule, the following program is ill-typed. Because the reference x is declared **uncontended**, the function f must be **nonportable**. Because f is **nonportable**, invoking f in a new thread is forbidden.

```
let x @ uncontended : int ref = ref 42 in
let f @ nonportable : unit -> int = fun _ -> !x in
Thread.create f ()
              ^ Error: can't cross threads
```

If x was instead declared **contended** then f could be **portable**, but it would then be impossible to use the reference x, thus still rejecting the program.

We now come back to the question: why cannot a **portable** closure refer to a **shared** variable After all, one might think that multiple threads can safely read from the same reference. The reason is illustrated by this example, which must be rejected:

```
let x @ uncontended : int ref = ref 42 in
let y @ shared = x in
Thread.create (fun () -> !y) ()
              ^ Error: can't cross threads
```

Here, an **uncontended** reference x is copied under the name y, and y is weakened to **shared**. As a result, even though access to y is restricted in the child thread, the parent thread might still write to this reference under the name x, causing a data race. An alternative solution would be to allow downgrading **uncontended** to **shared** only if the reference is **unique**; then, in the above example, an error would be detected at the second line. We do not pursue this approach because it would complicate the submoding relation.

Thus, re-iterating what has been said above, **portable** closures are seriously restricted: they cannot have any access to mutable references from their environment. In § 2.7 and 2.8, we will show how to work around this limitation by placing mutable data structures inside capsules.

2.5 Summary of Modes; the Legacy Mode

Fig. 2 offers a summary of all modes, organized along our five axes. In each axis, modes are organized vertically along the submoding relation ($\leq$): the strongest mode appears at the bottom, while the weakest mode appears at the top. For instance, in the "locality" axis, the submoding relation is **global** $\leq$ **local**, because a **global** value can safely be viewed as **local** (this restricts its lifetime), whereas a **local** value cannot be viewed as **global** (that would allow it to escape its scope).

The oriented edges depict the implications that connect distinct axes. Between uniqueness and affinity, we have the following implication: "a closure that captures a **unique** variable must be **once**"; therefore, in the contrapositive form, "the free variables of a **many** closure must be **aliased**". Between contention and portability, the implications are: "a closure that captures an **uncontended** or **shared** variable must be **nonportable**" and "the free variables of a **portable** closure must be **contended**".

Along each axis, we have shown only the points that exist on this axis. A *mode* is a 5-tuple of one point along each axis. Naturally, we do not require users to systematically annotate their code with 5-tuples; that would be heavy. Instead, along each axis, we fix a *default point*, and we allow a component of a 5-tuple to be omitted when it is the default point along its axis.

We choose the default points in such a way that the 5-tuple of the five default points is the *legacy mode*, that is, the mode at which all legacy OCaml code[5] can be type-checked. The legacy mode is defined as follows: **legacy** $\triangleq$ (**global**, **many**, **aliased**, **nonportable**, **uncontended**).

The mode annotation "." denotes the legacy mode. Furthermore, we use the following syntactic sugar: if the declaration of a type `t` is followed by, for example, `default portable contended` then, for values of this type only, the default points on their respective axes become **portable** and **contended**. This convention is used in the Capsule and Reader-Writer Lock APIs (Figures 3 and 4).

2.6 Modes and References

Let us now outline more precisely how modes and *mutable references* interact. This aspect is entirely new: the type system of Lorenzen et al. [21] did not include mutable references at all. References can also be used to model OCaml's mutable fields. Two questions arise: what restrictions do modes impose on references? And what is the relation between the mode of a reference and the mode of its content?

Our answer to the first question is guided by soundness constraints. As we have seen earlier in §2.4, the contention axis restricts the ways in which a reference may be used: a **uncontended** reference can read and written, and a **contended** reference cannot be used at all. The other axes do not restrict when references can be used.

Our answer to the second question is guided mainly by ergonomic considerations. References must be backwards compatible, that is, the value stored inside of a reference at legacy mode must itself be at legacy mode. However, we want a somewhat more flexible design. For example, we want to be able to track the portability of values inside of references. This comes up when storing closures in references, and even more so when we discuss the Capsule API (§2.7). In particular, the latter use case requires the portability of a reference to match the portability of its content. That is, while **nonportable** references can contain **nonportable** values (and, thanks to modalities or to mode weakening, also **portable** values), we wish to restrict **portable** references to contain only **portable** values.

---

[5] OCaml up to version 4.x offers a limited form of concurrency, where only one OCaml thread and several C threads can run concurrently; the main application of this feature is asynchronous input/output. True shared-memory concurrency was introduced in OCaml 5. By "legacy code", we refer to the existing body of sequential OCaml code.

```
module Key : sig
  type 'k t default portable contended      (* the abstract type of keys *)
  type packed = Key : 'k t -> packed        (* an existential type of keys *)
  val create : unit -> packed @ unique      (* key & capsule creation *)
end
module Data : sig
  type ('a,'k) t default portable contended (* data of type 'a protected by key 'k *)
  val create :
    (unit @ . -> 'a @ .) @ local once portable ->
    ('a, 'k) t @ .
  val destroy :
    'k Key.t @ unique ->
    ('a, 'k) t @ . ->
    'a @ .
  val both :
    ('a, 'k) t @ . -> ('b, 'k) t @ . -> ('a * 'b, 'k) t @ .
  val map :
    'k Key.t @ unique ->
    ('a @ . -> 'b @ .) @ local once portable ->
    ('a, 'k) t @ . ->
    'k Key.t * ('b, 'k) t @@ aliased @ unique
  val extract :
    'k Key.t @ unique ->
    ('a @ .  -> 'b @ portable contended) @ local once portable ->
    ('a, 'k) t @ . ->
    'k Key.t * 'b @@ aliased @ unique portable contended
  val map_shared :
    'k Key.t @ local ->
    ('a @ portable shared -> 'b @ .) @ once portable ->
    ('a @@ portable, 'k) t @ . ->
    ('b, 'k) t @ .
  val extract_shared :
    'k Key.t @ local ->
    ('a @ portable shared -> 'b @ portable contended) @ once portable ->
    ('a @@ portable, 'k) t @ . ->
    'b @ portable contended
end
```

Fig. 3. The Capsule API

A naive implementation of this would be to let the mode of the reference itself serve also as the mode of the content. This is unfortunately unsound, because mode weakening, applied to the reference, would then also apply to its content. That would effectively give us covariant references, which are unsound.

Instead, we introduce two separate types of references, namely **nonportable** and **portable** references. The annotation carried by a reference's type determines the portability of its content.

Our typing rules for references are formally presented in §4.4. There, we also describe *atomic references*, which our type system also supports.

2.7 The Capsule API

We now reach a second key contribution of this paper, namely the Capsule API. The type system presented so far does not allow accessing mutable data from multiple threads at all, since **contended** references are inaccessible. This API allows a value (or, more generally, a data structure) to become protected by a **unique** key. Unique ownership of the key enables mutation of the contents of the capsule without fear of data races: if the key becomes aliased, then the contents of the capsule become read-only.

The Capsule API is presented in its entirety in Fig. 3. It consists of two modules, `Key` and `Data`. These modules declare two abstract types, `'k Key.t` and `('a, 'k) Data.t`.

- A value of type `'k Key.t` is a *key*. At runtime, such a value is irrelevant; it is a unit value. At type-checking time, the type variable `'k` serves as a type-level name for this key. The type `Key.packed`, an existential type, hides the name `'k`.
- A value of type `('a, 'k) Data.t` represents *encapsulated data* of type `'a` that is protected by the key `'k`. This type does not involve an indirection: a value of type `('a, 'k) Data.t` is represented at runtime in the same way as a value of type `'a`.

In summary, a capsule is a conceptual boundary, and there is a one-to-one correspondence between keys and capsules: the capsule associated with a key `'k` is just the collection of all encapsulated data that are protected by this key.

By default, the types `'k Key.t` and `('a, 'k) Data.t` are **portable** and **contended**. In other words, keys and encapsulated data are safe to share and access across multiple threads. This makes sense, given that ensuring thread safety is the entire *raison d'être* of capsules!

The function `Key.create` creates a fresh key, whose type and mode are `Key.packed @ unique`. Opening this existential package gives rise to a fresh, abstract key name `'k`; then, the new key has type and mode `'k Key.t @ unique`. Because there is a one-to-one correspondence between keys and capsules, one can think of `Key.create` as also creating a new capsule, which is initially empty and is associated with the key `'k`.

A capsule is populated by applying `Data.create` to a constructor function `f` of type `unit -> 'a`. The result of this function, a value of type `'a`, becomes protected by the key `'k`: in other words, it becomes encapsulated by the capsule. As a witness for this fact, `Data.create` returns the same value at type `('a, 'k) Data.t`. A capsule may be populated in several steps: `Data.create` can be applied several times to the same type-level key `'k`.

Crucially, the constructor function `f` that is passed to `Data.create` must be **portable**.[6] This guarantees that `f` cannot access any pre-existing mutable data (§2.4). So, if `f` returns a mutable data structure, then this data structure must be freshly allocated. In other words, the data that enters the capsule must be "self-contained". The purpose of this restriction is to ensure that any mutable data entering the capsule is properly encapsulated by it (*i.e.,* only accessible via the capsule)—were this not so, an external alias of the capsule's mutable data could be used to incur a data race.

The Capsule API offers several ways to access and mutate a capsule: (1) `Data.destroy` (2) `Data.map`, and (3) `Data.extract` require a **unique** key, while (4) `Data.map_shared` and (5) `Data.extract_shared` do not. Therefore, the last two functions can be applied to an **aliased** key. Two elements of the same capsule can be accessed simultaneously by joining them using `Data.both`.

A **unique** key grants full (read-write) access to the data inside a capsule. In `Data.destroy`, the key and capsule are destroyed, and the data in the capsule is converted back to its original type `'a`. In `Data.map` and `Data.extract`, the data in the capsule is temporarily made accessible to a user-supplied function `f` whose OCaml type is `'a -> 'b`. This function must be **portable**, guaranteeing that it does not have access to any mutable state (beside its argument of type `'a`) and thus cannot leak its argument.

---

[6] The constructor function is also marked **local** and **once**, which means that `Data.create` promises to not leak this function and to invoke it at most once.

(1) In `Data.map`, the function `f : 'a -> 'b` is applied to the data in the capsule, and its result *enters* the capsule, so a value of type `('b, 'k) Data.t` is eventually returned, together with the key, which is still unique.

(2) In `Data.extract`, the function `f : 'a -> 'b` is applied to the data in the capsule, and its result *leaves* the capsule, so a value of type `'b` is returned together with the unique key. Unlike for `Data.map`, the result of `f` here must be **portable**; this prevents `f` from returning a closure whose environment contains pointers to mutable capsule data, which could subsequently lead to a data race if that closure were applied. The value of type `'b` that is eventually returned by `Data.extract` is therefore also **portable**, and must be viewed by the caller of `Data.extract` as **contended**, so that any mutable capsule data that might be exposed through this value cannot be accessed by the caller.

In contrast with a **unique** key, an **aliased** key grants *only read access* to the data inside a capsule. Thus, in `Data.map_shared` and `Data.extract_shared`, which accept an **aliased** key, the function `f` receives read-only access to the data of type `'a`. This is expressed via a new mode, **shared**, which lies between **uncontended** and **contended** on the contention axis (Fig. 2). Like **uncontended** references, **shared** references can be read. Like **contended** references, they cannot be written. In `Data.map_shared` and `Data.extract_shared`, because the data can be read by several threads concurrently, we must require it to be **portable**. This is expressed by requiring the encapsulated data to have type `('a @@ portable, 'k) t`.[7]

A critical point about both `Data.map_shared` and `Data.extract_shared` is that they can only be applied to a **local** key. Thus, they promise to merely *temporarily borrow* this **aliased** key. As we will see in the next section, this is essential to ensure that the temporary nature of the read-only access granted by a reader-writer lock is respected.

As with `Data.map`, `Data.map_shared` only accepts **portable** callback functions. As a result, it is not possible to simultaneously access the **shared** parts of two different capsules. Indeed, it is generally unsound to hold any combination of **uncontended** and **shared** references to two different capsules at once. For example, consider the following snippet:

```
let d3 = Data.extract_shared key1 (fun a => Data.map_shared key2 (fun b => a @@ shared) d2) d1
                  Error: this value is contended but expected to be shared ^
```

Here, a value (*e.g.*, a reference) `a` from the capsule `d1` (governed by `key1`) becomes aliased by another capsule (the result `d3`, governed by `key2`). This could subsequently lead to a data race because one could use `key1` to mutably access `d1` while `d3` is concurrently being accessed via `key2`. Thus, it is important that the above code is disallowed, which it is: the innermost **portable** closure cannot refer to the value `a` as **shared**, only as **contended**.

2.8 The Reader-Writer Lock API

We have seen how capsules associate data structures to keys, and how both **unique** and **aliased** keys are used to safely mediate concurrent access to the data within the capsules. However, we have yet to see how the keys themselves are shared across threads. In this section, we present a Reader-Writer Lock API, which we can use to safely share access to keys.

Fig. 4 presents a Reader-Writer Lock API designed specifically for keys. The Reader-Writer Lock is a typical many-readers single-writer lock: only one thread may gain **unique** access to the key

---

[7] This requirement can be a bit inconvenient, as it implies that the user must plan ahead and place a `@@ portable` modality at the root of the data. In the future, this inconvenience might be relieved, to some extent, by allowing this modality to commute with other type constructors.

```
module RwKeyLock : sig
  type 'k t default portable contended

  val create :
    'k Capsule.Key.t @ unique ->
    'k t @ .
  val unique_protect :
    'k t @ . ->
    ('k Capsule.Key.t @ unique -> ('k Capsule.Key.t * 'b) @ unique portable contended)
       @ once portable ->
    'b @ unique portable contended
  val shared_protect :
    'k t @ . ->
    ('k Capsule.Key.t @ local -> 'b @ portable contended) @ once portable ->
    'b @ portable contended
end
```

Fig. 4. The Reader-Writer Lock API

```
module RwHashtbl = struct
  type t = Table :
    { table : (((int, string) Hashtbl.t) @@ portable,'k) Capsule.Data.t;
       lock : 'k RwKeyLock.t } -> t
    default portable contended

  let create () : t  =
    let key = Capsule.Key.create () in
    let table = Capsule.Data.create (fun () -> box (Hashtbl.create ()) in
    let lock = RwKeyLock.create key in
    Table { table; lock }

  let add (Table { table; lock }) (k : int) (v : string) : unit =
      RwKeyLock.unique_protect lock (fun key ->
        unbox (Capsule.Data.extract key (fun table -> Hashtbl.add (unbox table) k v) table))

  let find (Table { table; lock }) (k : int) : string =
    RwKeyLock.shared_protect lock (fun key ->
      Capsule.Data.extract_shared key (fun table -> Hashtbl.find table k) table)
end
```

Fig. 5. A thread-safe hash table. We omit legacy `@ .` mode annotations.

(via `unique_protect`), whereas multiple threads may concurrently gain **aliased** access to the key (via `shared_protect`).

The readers gain only **local** access to the key: this ensures that the key is not captured and stored for later use, outside the callback function of `shared_protect`.

To display the versatility of the Capsule and Reader-Writer Lock APIs, we present a simple client that uses capsules to share hash tables across threads (Fig. 5). This client implements a module for concurrent hash tables, where hash tables are encapsulated in a capsule, and reader-writer locks are used to grant access to the associated key. A new key is created upon allocation; then, the hash table constructor is called *within a capsule*, which requires `Hashtbl.create` to be **portable**. Since we allow many readers to call `RwHashtable.find`, the hash table itself must be **portable** as well, and `Hashtbl.find` must accept a **shared** argument.

These stronger mode requirements mean that we cannot reuse OCaml's existing `Hashtbl` module *completely* as is (as the legacy mode is too weak). But we also do not have to change its implementation in any substantive way—we merely have to annotate it to indicate: (1) that many of its functions (including `Hashtbl.create`) are in fact **portable**; (2) that `Hashtbl.find` is well-typed with a **shared** argument (because it only *reads* from its argument); and (3) that the references it uses in the definition of the data type `Hashtbl.t` should be **portable**, so that `Hashtbl.t` is **portable**.

Finally, the key is protected by a reader-writer lock. Subsequent operations over the hash table are then performed via the reader-writer lock operations `RwKeyLock.unique_protect` and `RwKeyLock.shared_protect`. In both cases, note that the operation passed to the `RwKeyLock` is handled via closures around the hash table capsules. These closures are **portable** since the capsules are themselves **contended** and **portable**. The above example is type-checked in our modal type system, and allows safe concurrent access to OCaml's existing hash tables.

2.9 Limitations of the Capsule API

While capsules can be used to build thread-safe versions of many data types, they are not a panacea. In particular, consider modules that use *static mutable state—i.e.,* mutable state that is "hidden" in the sense that it is not part of the representation of the abstract data type, but is instead implicitly shared between the operations of the module via the environments of their closures. A public operation that has access to this "static" state *cannot* be **portable**, and therefore cannot be invoked by the callbacks that are passed to the capsule and reader-writer lock operations. This limitation is fundamental and intentional: a module with static mutable state could actually cause data races if its operations were invoked concurrently!

Another unavoidable limitation is the need to annotate existing OCaml libraries with **portable** and **shared** modes, as we saw with the `Hashtbl` module. While this limitation is mostly a matter of adding annotations to module signatures and relevant reference allocations, it may still be a challenge to consider all uses of each function in a module signature, where one might need multiple versions of the same signature for each mode use case. We believe this limitation can likely be overcome by introducing a notion of mode polymorphism.

Finally, there are other limitations of capsules that we believe are not fundamental and could be lifted in future work. We foresee the following improvements to the Capsule API:

- We believe an operation `Data.project_shared : 'k Key.t @ . -> ('a @@ portable, 'k) t @ . -> 'a @ portable shared` would be sound. It would enable a **shared** alias to be extracted from encapsulated data, given a **global** and **aliased** key.
- The operations `Data.map_shared` and `Data.extract_shared` require callbacks that are **global** instead of **local**, as opposed to the other functions on `Data`. We think that they can, in fact, also be **local**, thus allowing the callbacks to reuse the same key, or even a different **local** and **aliased** one, to another capsule in a nested call to `Data.*_shared`.
- Similarly, we believe that the callback arguments in the Reader-Writer Lock API could also be **local**, which would reap similar benefits as above. To be more concrete, it would allow programs such as the following, which is currently rejected:

```
RwKeyLock.shared_protect lock1
  (fun key1 => RwKeyLock.shared_protect lock2
    (fun key2 => let x = Capsule.Data.extract key2 fun1 in
                 let y = Capsule.Data.extract key1 fun2 in ...))
```

$$\begin{array}{lcllcl}
l & \in & \text{Locality} & ::= & \textbf{local} \mid \textbf{global} & \\
o & \in & \text{Affinity} & ::= & \textbf{once} \mid \textbf{many} & \pi \in \text{ThreadId} \quad \ell \in \text{Loc} \quad \iota \in \text{Fid} \quad n \in \mathbb{N} \\
u & \in & \text{Uniqueness} & ::= & \textbf{aliased} \mid \textbf{unique} & a \in \text{Addr} ::= \ell \mid (\pi, n) \\
p & \in & \text{Portability} & ::= & \textbf{nonportable} \mid \textbf{portable} & \omega \in \text{Order} ::= \text{NA}_{\{1,2\}} \mid \text{AT} \\
c & \in & \text{Contention} & ::= & \textbf{contended} \mid \textbf{shared} \mid & st \in \text{LockSt} ::= \textsc{wr} \mid \textsc{r}_n \\
 & & & & \textbf{uncontended} &
\end{array}$$

$$\begin{array}{lcll}
m & \in & \text{Mode} & \triangleq \ \text{Locality} \times \text{Affinity} \times \text{Uniqueness} \times \text{Portability} \times \text{Contention} \\
v & \in & \text{Value} & ::= \ () \mid z \mid \mathsf{true} \mid \mathsf{false} \mid a \mid \lambda^{(\iota,a)} f\ x, e \mid (v, v) \mid \mathsf{inl}(v) \mid \mathsf{inr}(v) \\
e & \in & \text{Expression} & ::= \\
\multicolumn{4}{l}{\quad v \mid x \mid \mathsf{let}\ x := e\ \mathsf{in}\ e \mid (e; e) \mid \lambda^{l} f\ x, e \mid e(e) \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid e \oplus e \mid \oplus(e) \mid} \\
\multicolumn{4}{l}{\quad \mathsf{case}\ e\ \{\mathsf{inl}\ x \longrightarrow e; \mathsf{inr}\ x \longrightarrow e\} \mid \mathsf{inl}(e) \mid \mathsf{inr}(e) \mid (e, e) \mid \mathsf{unpair}\ e\ \mathsf{as}\ (x, y)\ \mathsf{in}\ e \mid} \\
\multicolumn{4}{l}{\quad \mathsf{alloc}^{l} \mid {!^{\omega}} e \mid e \leftarrow^{\omega} e \mid \mathsf{cmpXchg}(e, e, e) \mid \mathsf{xchg}(e, e) \mid \mathsf{faa}(e, e) \mid \mathsf{fork}(e) \mid} \\
\multicolumn{4}{l}{\quad \mathsf{borrow}\ x := e\ \mathsf{for}\ y := e\ \mathsf{in}\ e \mid \mathsf{box}(e) \mid \mathsf{unbox}(e) \mid \mathsf{region}(e) \mid \mathsf{end}^{n}(e)}
\end{array}$$

Fig. 6. DRFCamlLang syntax

In the current version of the API, Since `key1` is **local**, it can't be used in the innermost **global** closure.

3 DRFCAMLLANG

In § 2 we presented the modes through examples written in OCaml. In this section, we present the language used to formalize the modal type system, namely an OCaml-like λ-calculus called DRFCamlLang. DRFCamlLang is a typical λ-calculus with recursive functions, higher-order store, and multi-threading. Its distinguishing feature is a store made up of two components: a heap, which behaves like the OCaml heap, and (for each thread) a stack of values. The stacks keep track of the lifetimes of stack-allocated values.

Fig. 6 describes the values and expressions of DRFCamlLang. Values include the unit value, integers, Booleans, λ-abstractions, and addresses. Since the store separates the heap and one stack per thread, an address is either a heap location ℓ or a stack location (π, n), where π is a thread identifier and n is an offset into this thread's stack. A λ-abstraction is labeled with an address a, which can be regarded as its physical address, and may be a heap address or a stack address, and with a function-id ι, which can be regarded as its logical address. Whereas, due to the stack allocation discipline, physical addresses can be reused, logical addresses are never reused.

Expressions include control constructs (conditionals and sequencing), unary and binary operations (collectively denoted $\oplus$), pairs and sums, and function application. On top of this, DRFCamlLang offers a number of operations to allocate, read and write mutable references. There is just one kind of reference, but we distinguish non-atomic and atomic accesses. A fresh mutable reference is allocated by alloc^{l}, where l determines whether the reference is allocated in the heap (**global**) or on the stack (**local**). Closures are also allocated, so the expression $\lambda^{l} f\ x, e$ (binding both the function f itself and its argument x) is tagged with a locality l. Loads and stores are annotated with an order ω, which determines whether the operation is non-atomic or atomic ($\textsc{at}$). A non-atomic operation is further split into two parts: $\textsc{na}_1$ and $\textsc{na}_2$. The former flags the location as "currently being read from or written to", and the latter applies the relevant operation and resets the flag. Both parts check whether a location's flag is compatible with the current operation. Thus, the program gets stuck whenever a non-atomic store occurs at the same time as another non-atomic access.[8]

---

[8] This method for modeling data races was also employed by Jung et al. [18] and is described in detail in Jung's thesis [17].

The three operations cmpXchg (conditional swap), xchg (unconditional swap), and faa (fetch and add) are atomic. Finally, DRFCamlLang introduces several new operations: borrow, which lets a unique value become locally aliased; box and unbox, which introduce and eliminate modalities; region, which creates a new stack region; and end^n, which destroys all stack locations at and above index n.

The semantics of DRFCamlLang is defined as a stateful small-step operational semantics, where the state consists of three components (h, s, fs):

$$\begin{array}{rcll} h & \in & \mathrm{Heap} & \triangleq \ \mathrm{Loc} \hookrightarrow \mathrm{LockSt} \times (\mathrm{Fid} + \mathrm{Value}) \\ s & \in & \mathrm{Stacks} & \triangleq \ \mathrm{ThreadId} \hookrightarrow \mathrm{list}\ (\mathrm{LockSt} \times (\mathrm{Fid} + \mathrm{Value})) \\ fs & \in & \mathrm{Funcs} & \triangleq \ \mathcal{P}_{\mathrm{fin}}(\mathrm{Fid}) \end{array}$$

The heap h is a finite map from locations to “memory slots”, which are pairs of a lock state and either a function-id or a value. The lock state is used to track a thread’s non-atomic access to some location: state $\textsc{wr}$ denotes a write access; state $\textsc{r}_n$ denotes n concurrent read accesses. The collection of stacks s is a finite map from thread-ids to stacks, where each stack is a list of memory slots. Finally, the function set fs is a finite set of all the previously allocated function-ids.

A single step is denoted by $(h, s, fs, e) \rightsquigarrow_\pi (h', s', e', efs)$, where π is the thread-id at which the expression e is executed, and efs — a list of thread-id and expression pairs, which we will refer to as a thread pool — is the list of threads spawned by e. We use $(h, s, fs, tp) \rightsquigarrow (h', s', fs', tp')$ to denote a step within a thread pool tp. By lack of space, we omit the small-step reduction rules. A selection of these rules is given in our technical appendix [13, §A]. The following paragraphs summarize the non-standard aspects of this semantics.

Fork and allocations. Each thread has its own stack. fork allocates a new stack and a fresh thread-id. A local allocation pushes a new memory slot onto the current thread’s stack.

Stack regions. A stack is not explicitly decomposed into stack frames or regions. Instead, the region operation implicitly creates a new region, just by reading the current stack size n; later, this region is destroyed by truncating the stack at size n. More precisely, the expression region(e) reduces in three stages, as follows. First, region(e) reduces to $\mathrm{end}^n(e)$, where n is the current size of the current thread’s stack. Second, $\mathrm{end}^n([])$ is an evaluation context, so the expression e is allowed to reduce, in zero, one or more steps, to a value v. Finally, $\mathrm{end}^n(v)$ deallocates all stack locations at and above the cutoff n, and reduces to v.

Atomic and non-atomic memory accesses; data races. Following standard practice, we distinguish atomic and non-atomic memory accesses. This distinction is necessary because it plays a role in the definition of a data race. By definition, a *data race* is a situation where two threads attempt to access the same location, at least one access is a write, and at least one access is non-atomic. Furthermore, following an established practice [18, 20], we build a data race detector into the dynamic semantics of DRFCamlLang. In other words, we set up the semantics in such a way that a data race can cause a crash, so that crash-freedom of well-typed programs implies data race freedom.

Our data race detector works as follows. First, every memory slot is equipped with a lock state, which is checked and updated by all memory access operations. Second, a non-atomic memory access is executed in two steps, whereas an atomic access is executed in just one step. In between the two steps of a non-atomic memory access, the memory slot is locked, so an independent attempt to access this memory slot causes a crash, unless both accesses are read accesses.

In summary, this operational semantics has the property that “if a machine configuration has a data race, then it can reduce to a configuration where at least one thread is stuck”. As a consequence, we obtain the following (machine-checked) theorem:

Theorem 3.1 (No Crash Implies No Race). *Let (σ, tp) be a well-formed machine configuration, where σ is the store and tp is the thread pool. If, in every configuration (σ', tp') reachable from (σ, tp), every thread either is a value or is able to step, then, in every configuration (σ', tp') reachable from (σ, tp), there is no data race.*

Program logic. In §5, we will present a semantic model of DRFCaml and its type system. This model is defined in the Iris logic [19], and is built on top of a program logic for DRFCaml. We define the program logic in terms of Iris's weakest preconditions, adjusted to work on languages where the thread-id's are visible at the level of the operational semantics (similar adjustments have been made by *e.g.*, Kaiser et al. [20], where thread-id's were paired with expressions; we pair them with steps in the operational semantics instead). Weakest precondition statements are denoted by $\mathsf{wp}\ e\ \{\Phi\}_\pi$, and intuitively express that the expression e may execute in thread π, that it does not get stuck, and if it reduces to a value v then $\Phi(v)$ holds. This intuition is formally proved in an adequacy theorem, which relates weakest preconditions to a pure statement in the meta-logic. Given this adequacy statement, we can prove the following corollary about weakest preconditions:

Corollary 3.1. *If $\vdash \mathsf{wp}\ e\ \{\Phi\}_\pi$ then executing the closed program e (with an initially empty heap and stack, and with thread identifier π) cannot cause a data race.*

Proof. Apply Theorem 3.1 followed by adequacy of the weakest precondition. □

4 MODAL TYPE SYSTEM

The DRFCamlLang types comprise the unit, Boolean, and integer types, sums and products, function types, and modalities (§4.3), as well as non-atomic and atomic references (§4.4):

$$\tau \in \text{Type} ::= \mathbb{1} \mid \mathbb{B} \mid \mathbb{Z} \mid \tau + \tau \mid \tau \times \tau \mid \tau \mathbin{@} m \to \tau \mathbin{@} m \mid \Box^{\eta} \tau \mid \mathsf{ref}_p(\tau) \mid \mathsf{atomic}(\tau)$$

Our typing judgments $\Gamma \vdash e : \tau \mathbin{@} m$ are annotated with a mode m. A context is a list of variables which are either disabled $x : -$ or annotated with a type and mode:

$$\Gamma \in \text{Context} ::= \varnothing \mid \Gamma, x : - \mid \Gamma, x : \tau \mathbin{@} m$$

An order on each mode axis is defined as in Fig. 2; it is then lifted pointwise to modes m. We lift our ordering on modes to contexts, and permit weakening modes in both conclusion and context:

$$\varnothing \leq \varnothing \qquad \frac{\Gamma_1 \leq \Gamma_2}{\Gamma_1, x : \tau \mathbin{@} m \leq \Gamma_2, x : -} \qquad \frac{\Gamma_1 \leq \Gamma_2 \qquad m_1 \leq m_2}{\Gamma_1, x : \tau \mathbin{@} m_1 \leq \Gamma_2, x : \tau \mathbin{@} m_2}$$

$$\frac{\Gamma_2 \leq \Gamma_1 \qquad \Gamma_1 \vdash e : \tau \mathbin{@} m_1 \qquad m_1 \leq m_2}{\Gamma_2 \vdash e : \tau \mathbin{@} m_2}\ \textsc{Sub}$$

All typing rules can be found in our technical appendix [13, §C]. Units, Booleans, and integers can be typed at any mode. Most typing rules are standard, up to simple mode annotations and context joining (§4.1). For example, the rule for products is defined as follows:

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \mathbin{@} m \qquad \Gamma_2 \vdash e_2 : \tau_2 \mathbin{@} m}{\Gamma_1 + \Gamma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2 \mathbin{@} m}\ \textsc{Pair}$$

Here, the contexts that type the two components are joined, as denoted by $\Gamma_1 + \Gamma_2$. Each component must be well-typed at the mode of the product, namely m. Only closures and fork (§4.2), modalities (§4.3), and references (§4.4) interact with modes in interesting ways (Fig. 7).

$$\textsc{NonRecLam}\quad \frac{\Gamma, \text{🔒}_{(l_2,o_2,p_2)}, x : \tau \mathbin{@} m \vdash e : \tau' \mathbin{@} m'}{\Gamma \vdash \lambda^{l_2}\_\; x, e : (\tau \mathbin{@} m \to \tau' \mathbin{@} m') \mathbin{@} (l_2, o_2, u_2, p_2, c_2)} \qquad \textsc{Fork}\quad \frac{\Gamma, \text{🔒}_{(\mathbf{global},o,\mathbf{portable})} \vdash e : \tau_1 \mathbin{@} m_1}{\Gamma \vdash \mathsf{fork}(e) : \mathbb{1} \mathbin{@} m_2}$$

$$\textsc{Box}\quad \frac{\Gamma \vdash e : \tau \mathbin{@} \eta(m)}{\Gamma \vdash \mathsf{box}(e) : \Box^{\eta}\tau \mathbin{@} m} \qquad \textsc{Unbox}\quad \frac{\Gamma \vdash e : \Box^{\eta}\tau \mathbin{@} m}{\Gamma \vdash \mathsf{unbox}(e) : \tau \mathbin{@} \eta(m)} \qquad \textsc{NaAlloc}\quad \frac{\Gamma \vdash e : \tau \mathbin{@} (l, \mathbf{many}, u, p, \mathbf{uncontended})}{\Gamma \vdash \mathsf{alloc}^{l}(e) : \mathsf{ref}_p(\tau) \mathbin{@} (l, o, u', p, c)}$$

$$\textsc{NaLoad}\quad \frac{\Gamma \vdash e : \mathsf{ref}_p(\tau) \mathbin{@} (l, o', u', p', c) \qquad c \neq \mathbf{contended}}{\Gamma \vdash\, !^{\mathrm{NA}} e : \tau \mathbin{@} (l, o, \mathbf{aliased}, p, c)} \qquad \textsc{AtStore}\quad \frac{\begin{array}{c}\Gamma_1 \vdash e_1 : \mathsf{atomic}(\tau) \mathbin{@} m_1 \\ \Gamma_2 \vdash e_2 : \tau \mathbin{@} (\mathbf{global}, \mathbf{many}, u, \mathbf{portable}, c)\end{array}}{\Gamma_1 + \Gamma_2 \vdash e_1 \xleftarrow{\mathrm{AT}} e_2 : \mathbb{1} \mathbin{@} m_2}$$

$$\textsc{AtAlloc}\quad \frac{\Gamma \vdash e : \tau \mathbin{@} (\mathbf{global}, \mathbf{many}, u, \mathbf{portable}, c)}{\Gamma \vdash \mathsf{alloc}^{\mathbf{global}}(e) : \mathsf{atomic}(\tau) \mathbin{@} (l, o, u', p, c')} \qquad \textsc{AtLoad}\quad \frac{\Gamma \vdash e : \mathsf{atomic}(\tau) \mathbin{@} m}{\Gamma \vdash\, !^{\mathrm{AT}} e : \tau \mathbin{@} (l, o, \mathbf{aliased}, p, \mathbf{contended})}$$

$$\textsc{NaStore}\quad \frac{\begin{array}{c}\Gamma_1 \vdash e_1 : \mathsf{ref}_p(\tau) \mathbin{@} (l', o', u', p', \mathbf{uncontended}) \\ \Gamma_2 \vdash e_2 : \tau \mathbin{@} (\mathbf{global}, \mathbf{many}, u, p, \mathbf{uncontended})\end{array}}{\Gamma_1 + \Gamma_2 \vdash e_1 \xleftarrow{\mathrm{NA}} e_2 : \mathbb{1} \mathbin{@} m_2}$$

Fig. 7. Selected typing rules for closures, fork, modalities, and references

4.1 Context Joining

Following Lorenzen et al. [21], the type system enforces the following two rules: (1) if a variable is marked **once** (as opposed to **many**) then it is used at most once; (2) if a variable is used several times then it is marked **aliased** (as opposed to **unique**). This is achieved via a partial context joining operation $\Gamma_1 + \Gamma_2$, which is defined as follows (technically, $\Gamma_1 + \Gamma_2 := \Gamma$ is a relation, since in the last case there are multiple possible Γ's that match the right-hand side of the definition):

$$\begin{array}{rcl}
\varnothing + \varnothing & := & \varnothing \\
(\Gamma_1, x : -) + (\Gamma_2, x : -) & := & (\Gamma_1 + \Gamma_2), x : - \\
(\Gamma_1, x : \tau \mathbin{@} \mu) + (\Gamma_2, x : -) & := & (\Gamma_1 + \Gamma_2), x : \tau \mathbin{@} \mu \\
(\Gamma_1, x : -) + (\Gamma_2, x : \tau \mathbin{@} \mu) & := & (\Gamma_1 + \Gamma_2), x : \tau \mathbin{@} \mu \\
(\Gamma_1, x : \tau \mathbin{@} (l, o_1, \mathbf{aliased}, p, c)) & & \\
+\, (\Gamma_2, x : \tau \mathbin{@} (l, o_2, \mathbf{aliased}, p, c)) & := & (\Gamma_1 + \Gamma_2), x : \tau \mathbin{@} (l, \mathbf{many}, u, p, c)
\end{array}$$

When a variable $x : \tau \mathbin{@} m$ is used in multiple expressions, x is only available to them as **aliased** and is required to be **many** in the ambient context. As a result, a **unique** variable becomes **aliased** if used in both branches of a context join, and **once** variables are never duplicated. Meanwhile, the portability and contention axes introduce no complication; by virtue of the Sub typing rule, the context join operation takes the meet operation (greatest lower bound) for these axes.

4.2 Closures, Locks, and Fork

The type system restricts which variables may be referred to inside a λ-abstraction. For instance, **global** (**many**, **portable**) closures must not capture **local** (**once**, **nonportable**) variables. Analogously, a **many** closure must not capture **unique** variables, as a **unique** reference could become aliased if the closure were copied. Instead, a **unique** variable must be weakened to **aliased** before being captured by a **many** closure. A similar interaction occurs between portability and contention: An **uncontended** or **shared** binding captured by a **portable** closure becomes **contended**.

Again following Lorenzen et al. [21], this is formalized using an operation on contexts, known as a *lock* $\text{🔒}_{(l,o,p)}$. It is used in the typing rule for λ-abstractions (NonRecLam in Fig. 7): Typing a λ-abstraction at mode (l, o, u, p, c) introduces a lock $\text{🔒}_{(l,o,p)}$ on the context. The mode of a variable $y \in \Gamma$, viewed from inside the λ-abstraction, is not necessarily the same as the mode of this variable viewed from the outside; the lock might change the uniqueness and contention modes of bindings. Bindings might also be disabled entirely. The lock operation is defined as follows:

$$\begin{aligned}
\varnothing, \text{🔒}_{(l_2,o_2,p_2)} &\coloneqq \varnothing \\
\Gamma, x : -, \text{🔒}_{(l_2,o_2,p_2)} &\coloneqq \Gamma, \text{🔒}_{(l_2,o_2,p_2)}, x : - \\
\Gamma, x : \tau \mathbin{@} (l_1, o_1, u_1, p_1, c_1), \text{🔒}_{(l_2,o_2,p_2)} &\coloneqq \begin{cases} \Gamma, \text{🔒}_{(l_2,o_2,p_2)}, x : (l_1, o_1, u_1 \vee o_2^{\dagger}, p_1, c_1 \vee p_2^{\dagger}) & \\ & \text{if } l_1 \leq l_2,\ o_1 \leq o_2, \text{ and } p_1 \leq p_2 \\ \Gamma, \text{🔒}_{(l_2,o_2,p_2)}, x : - & \text{otherwise} \end{cases}
\end{aligned}$$

To explain this definition, we introduce the following example. Say we are typing a closure, and introduce a lock at mode (**local**, **many**, **nonportable**) to the context which contains a variable x at mode (**global**, **many**, **unique**, **nonportable**, **uncontended**). The variable remains accessible after taking the lock because **global** $\leq$ **local**, **many** $\leq$ **many**, and **nonportable** $\leq$ **nonportable**.

However, the uniqueness mode of x within the closure must change: it must only be accessible at mode **aliased**. To formalize this, we define a dagger operation † that relates affinity and portability modes to their corresponding dual uniqueness and contention modes:

$$\begin{aligned}
\textbf{once}^{\dagger} &\coloneqq \textbf{unique} & \qquad \textbf{nonportable}^{\dagger} &\coloneqq \textbf{uncontended} \\
\textbf{many}^{\dagger} &\coloneqq \textbf{aliased} & \qquad \textbf{portable}^{\dagger} &\coloneqq \textbf{contended}
\end{aligned}$$

Thus, after applying the lock, x will be typed at uniqueness mode **unique** $\vee$ $\textbf{many}^{\dagger}$ = **aliased** and contention mode **uncontended** $\vee$ $\textbf{nonportable}^{\dagger}$ = **uncontended**.

The construct fork(e) is analogous to `Thread.create (fun () -> e) ()` in OCaml. Its typing rule ensures that the closure $\lambda().e$ is **global** and **portable**. This is enforced using the $\text{🔒}_{(\textbf{global},o,\textbf{portable})}$ lock in the Fork typing rule.

4.3 Boxes and Modalities

A modality η can be interpreted as a function from modes to modes, which maps the mode of a box to the mode of its contents. Thus, in the rules Box and Unbox, the mode of the contents of the box is determined by $\eta(m)$ where m is the mode of the boxed value. DRFCaml supports the following modalities, corresponding to the **global**, **many**, **aliased**, **portable**, **contended**, and **shared** modes, respectively:

$$\begin{aligned}
G(l,o,u,p,c) &\coloneqq (\textbf{global}, o, \textbf{aliased}, p, c) & \qquad P(l,o,u,p,c) &\coloneqq (l, o, u, \textbf{portable}, c) \\
M(l,o,u,p,c) &\coloneqq (l, \textbf{many}, u, p, c) & \qquad C(l,o,u,p,c) &\coloneqq (l, o, u, p, \textbf{contended}) \\
A(l,o,u,p,c) &\coloneqq (l, o, \textbf{aliased}, p, c) & \qquad S(l,o,u,p,c) &\coloneqq (l, o, u, p, c \vee \textbf{shared})
\end{aligned}$$

To improve readability, we use the notation `'a @@ global` to denote the $\square^{G}$`'a` type, `'a @@ many` to denote $\square^{M}$`'a` , and so on. The G modality is somewhat special, as it requires its contents to be not only **global**, but also **aliased**. This interaction between locality and uniqueness is required to ensure that borrowing is sound [21, §2.6].

Not every mode has a corresponding modality: for instance, it would not make sense to have a **local** modality $L(l, o, u, p, c) := (\textbf{local}, o, u, p, c)$, because it would allow a reference from the heap to the stack, breaking the lifetime guarantees of **local**:

```
let x @ local : int ref = ref 0
let y @ global : (int ref @@ local) ref = ref (box x)
```

More generally, **local** state cannot be nested inside of **global** state. Similarly, a **many** value cannot contain anything **once**, an **aliased** value cannot contain anything **unique**, etc. This is also why the S modality only takes a join instead of setting the mode to **shared**: if we defined it as $S(l, o, u, p, c) := (l, o, u, p, \textbf{shared})$, it would be possible to nest **shared** inside of **contended** state, and then to leak it to other threads; see § 2.4 for why this would be unsound.

For readers familiar with monadic vs. comonadic modalities, it may be helpful to observe that M and P are comonadic, while A, C, and S are monadic. G is almost comonadic, save for its interaction with uniqueness. The "polarity" of our modalities coincides with their (co-)monadicity: The comonadic G, M, and P modalities correspond to the bottom mode of their axes, while the monadic A and C modalities correspond to the top mode of their axes. Lastly, the modalities of the three axes that apply to closures and locks (namely, G, M, and P) are precisely the comonadic ones.

4.4 References

The typing rules for non-atomic references $\text{ref}_p(\tau)$ are shown in Fig. 7. We distinguish between **portable** references $\text{ref}_{\textbf{portable}}(\tau)$ and **nonportable** references $\text{ref}_{\textbf{nonportable}}(\tau)$ (see also § 2.6). This annotation influences the mode of the value that is stored inside the reference.

A newly allocated reference is **many**, **unique**, and **uncontended**; The typing rule NaAlloc allows arbitrary o, u, c, but **many**, **unique**, and **uncontended** are the best choices. Its locality reflects whether it is allocated in the heap or on the stack. Its portability matches the portability of the reference type.

Contention influences how a reference can be used. An **uncontended** reference can be read and written; a **shared** reference can only be read (NaStore); and a **contended** reference cannot be accessed at all (NaLoad, NaStore). There are no other restrictions on the use of references.

The relation between the mode of a reference and the mode of its contents is more complex: each axis has its own rules.

On the affinity and uniqueness axes, the rules are as follows. The contents of a reference is always **many** and **aliased**, regardless of the mode of the reference itself. Thus, when a reference is allocated or written, the value that one wishes to store is required to be **many** and **aliased**. Conversely, when a reference is read, the resulting value is guaranteed to be **many** and **aliased**.

On the portability axis, we distinguish two types of references. The content of a **portable** reference is **portable**; the content of a **nonportable** reference is **nonportable**.

On the locality axis, the rule is: a reference and its contents have the same locality. Allocating a reference at locality l requires a value of locality l, and reading a reference at locality l yields a value of locality l. Unfortunately, we cannot allow *writing* a **local** value into a **local** reference, because the **local** mode does not provide sufficiently precise lifetime information. So, the typing rule NaStore only allows writing a **global** value to a reference (of arbitrary locality).

On the contention axis, the rule is: a reference and its contents have the same contention. Thus, reading an **uncontended** or **shared** reference yields a value with the same contention; writing a reference (which must be **uncontended**) and allocating a reference (which initially is **uncontended**) both require an **uncontended** value.

There is a separate type of atomic references $\text{atomic}(\tau)$ that permit only atomic operations, including compare-and-exchange (cmpXchg), fetch-and-add (faa), atomic loads ($!^{\text{sc}}e$), and atomic stores ($e_1 \leftarrow^{\text{sc}} e_2$). The typing rules of these atomic operations – some of which are shown in Fig. 7 – are simpler. Atomic references are always allocated on the heap, so they are initially **global**. They can be safely shared between threads: that is, they are **portable**. They can be accessed even if they are **contended**. The contents of an atomic reference are always **global**, **many**, **aliased**, **portable**, and **contended**. This is very restrictive, but necessary: Atomic references, by design, can be used to transfer arbitrary values across threads, and so those values must also be safe to share across threads, that is, **portable** and **contended**.

5 SEMANTIC TYPE SOUNDNESS

The type system of DRFCaml guarantees data race freedom by ensuring that mutable data is never accessed simultaneously by different threads. However, this is too restrictive to allow for the implementation of APIs such as the Capsule API, which fundamentally depend on the ability to *carefully* mutate shared state. These implementations circumvent this restriction by using unsafe casts (such as `Obj.magic`) to escape type-and-mode-checking.

In order to formalize and verify the implementation of the capsule API, we define a notion of *semantic type safety*, and manually verify that the `Capsule` module is type-safe. To do this, we interpret each type as a predicate in the program logic that we have defined for DRFCaml (§ 3). A predicate can be thought of, very roughly, as a set of values, so this is a natural way of explaining the meaning of types. Furthermore, a predicate in a modern Iris-based program logic can also describe notions of unique ownership, shared ownership, invariants that all threads agree to obey, and so on; so this is a very powerful way of explaining the meaning of types.

5.1 Overview of the Model

We start off with an overview of the semantic model, which consists of a logical relation defined in the Iris logic, comprising a *value* relation $[\![\tau]\!]$ and an *expression* relation $\mathcal{E}[\![\tau]\!]$. These give a semantic interpretation of a type τ, which can be a standard syntactic type, giving rise to a standard type interpretation, or an abstract type defined by some API, giving rise to a bespoke type interpretation.

We use ghost state and Iris invariants to capture the various features expressed by the modes. In particular, our goal is to express (1) the temporary lifetime of local values, (2) the isolation guarantees of **portable** functions, (3) the read-only restriction of **shared** references and (4) the duplicability of **aliased** references.

The first three properties are expressed by parameterizing the relations by three sets, ε_{mut}, ε_{ro} and Δ, and the fourth property is expressed by using features of the Iris logic (Iris invariants and the persistence modality $\square$). The signature of the logical relation is thus as follows: $[\![\tau]\!]_m^{\varepsilon_{\text{mut}},\varepsilon_{\text{ro}},\Delta}$ where ε_{mut} *reflects the set of aliased non-atomic references that are accessible for reading and writing*; ε_{ro} *reflects the set of aliased non-atomic references that are accessible for reading only*; Δ *reflects the set of locals that are accessible for reading and writing.*

Here, we use the word "accessible" to mean that there is permission to access; we do *not* use it as a synonym for "reachable". We write "a local" to refer to an entity whose lifetime is lexical: at present, a local is either a stack-allocated value or a borrow. (In our operational semantics, borrowing a global value creates a local copy of it, whose lifetime is limited.) We use the word "reflects", as opposed to "is", because these are not exactly sets; the reality is more complex, but we lack space to provide more detail.

The value relation is also parameterized with a mode m, which determines *how* to interpret some type τ. For example, a reference at mode **uncontended** and a reference at mode **contended** will receive different interpretations.

Crucially, none of these parameters are fixed forever. For example, when a **unique** reference is downgraded to **aliased**, the set of accessible read-write references grows; yet, all existing values remain well-typed. Furthermore, the mode at which a type is interpreted may dictate that the interpretation be independent of a particular parameter. For example, the interpretation of a function type at mode **portable** does not depend on the current sets of accessible references; so, when these sets grow or shrink, all existing portable functions remain well-typed. These observations give rise to a collection of *monotonicity requirements*, or *core conditions*, which every semantic type must satisfy. Below, we highlight three of these core conditions; our Coq formalization includes a total of ten.

Definition 5.1 (Excerpt of the Core Conditions of the Logical Relation)**.**

(1) if $(\varepsilon'_{\mathsf{mut}}, \varepsilon'_{\mathsf{ro}}) \sqsupseteq (\varepsilon_{\mathsf{mut}}, \varepsilon_{\mathsf{ro}})$ then $[\![\tau]\!]_m^{\varepsilon_{\mathsf{mut}},\varepsilon_{\mathsf{ro}},\Delta}(v) \mathrel{-\!\!*} [\![\tau]\!]_m^{\varepsilon'_{\mathsf{mut}},\varepsilon'_{\mathsf{ro}},\Delta}(v)$,
where $(\varepsilon'_{\mathsf{mut}}, \varepsilon'_{\mathsf{ro}}) \sqsupseteq (\varepsilon_{\mathsf{mut}}, \varepsilon_{\mathsf{ro}}) \triangleq \varepsilon'_{\mathsf{mut}} \supseteq \varepsilon_{\mathsf{mut}} \wedge \varepsilon'_{\mathsf{mut}} \cup \varepsilon'_{\mathsf{ro}} \supseteq \varepsilon_{\mathsf{ro}}$

(2) if $m.p = \textbf{portable}$ and $m.c = \textbf{contended}$ then $[\![\tau]\!]_m^{\varepsilon_{\mathsf{mut}},\varepsilon_{\mathsf{ro}},\Delta}(v) \mathrel{-\!\!*} [\![\tau]\!]_m^{\varepsilon'_{\mathsf{mut}},\varepsilon'_{\mathsf{ro}},\Delta}(v)$.

(3) if $m \leq m'$ then
$$\textit{resources over}\ \varepsilon_{\mathsf{mut}} * [\![\tau]\!]_m^{\varepsilon_{\mathsf{mut}},\varepsilon_{\mathsf{ro}},\Delta}(v) \Rrightarrow\!\!*_{\top}$$
$$\exists\, \varepsilon'_{\mathsf{mut}}.\ \varepsilon_{\mathsf{mut}} \subseteq \varepsilon'_{\mathsf{mut}} * \textit{resources over}\ \varepsilon'_{\mathsf{mut}} * [\![\tau]\!]_{m'}^{\varepsilon'_{\mathsf{mut}},\varepsilon_{\mathsf{ro}},\Delta}(v)$$

Item 1 is a simple monotonicity requirement: enlarging the sets of accessible read-write and read-only references, or allowing read-write access to previously read-only references, does not invalidate any existing values. Item 2 is more atypical: it states that the interpretation of a type at mode **portable** and **contended** is insensitive to the sets of accessible references. This reflects and combines two facts: (1) a **portable** function cannot access any references; (2) a **contended** reference cannot be accessed. Therefore, regardless of its type, the well-typedness of a **portable** and **contended** value does not depend at all on any reference. Finally, Item 3 reflects mode weakening: if a value is well-typed at mode m then it is also well-typed at a weaker mode m'.[9] This statement is formulated in a way that allows $\varepsilon_{\mathsf{mut}}$ to grow. The reason for this is, when a **unique** reference is turned into an **aliased** reference, $\varepsilon_{\mathsf{mut}}$ must grow, since one more aliased reference becomes accessible. We write "*resources over* $\varepsilon_{\mathsf{mut}}$" to gloss over a number of ghost resources that must evolve together with $\varepsilon_{\mathsf{mut}}$.

5.2 The Logical Relation

In this section, we present the expression relation $\mathcal{E}[\![\tau]\!]$ and part of the definition of the type interpretation $[\![\tau]\!]$. These are shown in Fig. 8. For presentation purposes, we keep the explanation at a high level and refer to the Coq mechanization for the full definition.

As described above, the expression relation is parameterized by the sets $\varepsilon_{\mathsf{mut}}$, $\varepsilon_{\mathsf{ro}}$ and Δ, and the mode m. It is also parameterized by a thread-id π, a stack size n, and a fraction q. The thread-id indicates which thread the expression is running in; the stack size indicates the current size of π's stack; and the fraction governs access to read-only references.

The expression relation is defined in terms of the weakest precondition described in §3, where the postcondition guarantees that the final value satisfies the value relation $[\![\tau]\!]$, at some expanded $\varepsilon'_{\mathsf{mut}}$ and Δ'. Additionally, the postcondition returns three key propositions: $\mathcal{L}(\pi, n', \varepsilon'_{\mathsf{mut}}, \varepsilon_{\mathsf{ro}}, \Delta')$, $\mathcal{M}\textsc{em}(\varepsilon'_{\mathsf{mut}}, \varepsilon_{\mathsf{ro}}, q)$ and $\mathrm{collectFrames}(n, n', \pi, \Delta, \Delta')$. Very roughly,

- $\mathcal{L}(\pi, n, \varepsilon_{\mathsf{mut}}, \varepsilon_{\mathsf{ro}}, \Delta)$ grants full access to the locals in Δ.
- $\mathcal{M}\textsc{em}(\varepsilon_{\mathsf{mut}}, \varepsilon_{\mathsf{ro}}, q)$ grants full access to the read-write references in $\varepsilon_{\mathsf{mut}}$ and partial access (at fraction q) to the read-only references in $\varepsilon_{\mathsf{ro}}$.

---

[9] The funny implication $\Rrightarrow\!\!*_{\top}$ is an Iris ghost update. It lets us allocate new ghost state and invariants.

$$\mathcal{E}\llbracket\tau\rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\pi,n,q,\boldsymbol{m}}(e) \quad\triangleq\quad \mathsf{wp}\ e\left\{v.\ \begin{array}{l} \exists n'\ \Delta'\ \varepsilon'_{\mathrm{mut}}.\ n \leq n' \wedge \Delta \subseteq \Delta' \wedge \varepsilon_{\mathrm{mut}} \subseteq \varepsilon'_{\mathrm{mut}} \\ * \llbracket\tau\rrbracket^{\varepsilon'_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta'}_{\boldsymbol{m}}(v) \\ * \mathcal{L}(\pi, n', \varepsilon'_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}}, \Delta') \\ * \mathcal{M}\textsc{em}(\varepsilon'_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}}, q) \\ * \mathsf{collectFrames}(n, n', \pi, \Delta, \Delta') \end{array}\right\}_{\pi}$$

$$\llbracket\mathbb{1}\rrbracket^{-,-,-}_{-}(v) \triangleq v = () \qquad \llbracket\mathbb{B}\rrbracket^{-,-,-}_{-}(v) \triangleq \exists b.\ v = b \qquad \llbracket\mathbb{Z}\rrbracket^{-,-,-}_{-}(v) \triangleq \exists z.\ v = z$$

$$\begin{aligned}
\llbracket\tau_1 + \tau_2\rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\boldsymbol{m}}(v) \quad &\triangleq\quad (\exists v_1.\ v = \mathsf{inl}(v_1) * \llbracket\tau_1\rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\boldsymbol{m}}(v_1)) \vee \\
&\quad (\exists v_2.\ v = \mathsf{inr}(v_2) * \llbracket\tau_2\rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\boldsymbol{m}}(v_2)) \\
\llbracket\tau_1 \times \tau_2\rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\boldsymbol{m}}(v) \quad &\triangleq\quad \exists v_1\ v_2.\ v = (v_1, v_2) * \llbracket\tau_1\rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\boldsymbol{m}}(v_1) * \llbracket\tau_2\rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\boldsymbol{m}}(v_2) \\
\llbracket\Box^{\eta}\tau\rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\boldsymbol{m}}(v) \quad &\triangleq\quad \llbracket\tau\rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\eta(\boldsymbol{m})}(v)
\end{aligned}$$

$$\begin{array}{l}
\llbracket\tau_1\ @\ m_1 \to \tau_2\ @\ m_2\rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\boldsymbol{m}}(v) \quad \triangleq v = \lambda \ldots\ * \forall \pi\ \varepsilon'_{\mathrm{mut}}\ \varepsilon'_{\mathrm{ro}}\ \Delta'\ q. \\
\quad (\varepsilon'_{\mathrm{mut}}, \varepsilon'_{\mathrm{ro}}) \sqsupseteq^{\boldsymbol{m.p}} (\varepsilon_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}}) \quad\to\quad \Delta' \sqsupseteq^{\boldsymbol{m.l},\boldsymbol{m.p}} \Delta \quad\to\quad \Box^{\boldsymbol{m.o}}\ \forall n\ v_1. \\
\quad \left\{\begin{array}{c} \llbracket\tau_1\rrbracket^{\varepsilon'_{\mathrm{mut}},\varepsilon'_{\mathrm{ro}},\Delta'}_{m_1}(v_1) * \\ \mathcal{L}(\pi, n, \varepsilon'_{\mathrm{mut}}, \varepsilon'_{\mathrm{ro}}, \Delta') * \mathcal{M}\textsc{em}(\varepsilon'_{\mathrm{mut}}, \varepsilon'_{\mathrm{ro}}, q) \end{array}\right\} \mathrel{-\!\!*} \mathcal{E}\llbracket\tau_2\rrbracket^{\varepsilon'_{\mathrm{mut}},\varepsilon'_{\mathrm{ro}},\Delta'}_{\pi,n,q,m_2}(v(v_1))
\end{array}$$

where

$$\Delta' \sqsupseteq^{l,p} \Delta \quad\triangleq\quad \begin{cases} \Delta \subseteq \Delta' & \textit{if } l = \textbf{local} \wedge p = \textbf{nonportable} \\ \mathsf{atomic}(\Delta) \subseteq \Delta' & \textit{if } l = \textbf{local} \wedge p = \textbf{portable} \\ \top & \textit{otherwise} \end{cases}$$

$$(\varepsilon'_{\mathrm{mut}}, \varepsilon'_{\mathrm{ro}}) \sqsupseteq^{p} (\varepsilon_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}}) \quad\triangleq\quad \begin{cases} \varepsilon'_{\mathrm{mut}} \supseteq \varepsilon_{\mathrm{mut}} \wedge \varepsilon'_{\mathrm{mut}} \cup \varepsilon'_{\mathrm{ro}} \supseteq \varepsilon_{\mathrm{ro}} & \textit{if } p = \textbf{portable} \\ \top & \textit{otherwise} \end{cases}$$

Fig. 8. A selection of standard interpretations, where $\Box^{m.o}$ is $\Box$ when $m.o$ = **many** and nothing otherwise

- $\mathsf{collectFrames}(n, n', \pi, \Delta, \Delta')$ grants permission to reclaim all of π's stack locations in the interval $[n, n')$, and guarantees that this does not break the well-typedness of any surviving value. In other words, it guarantees that local (stack-allocated) references do not escape.

The semantic interpretation of the basic types—namely unit, Booleans and integers—is straightforward: it is completely independent of the mode parameter m.

The semantic interpretation of a compound type—that is, a sum or a product—consists of an appropriate combination of the interpretations of its components. The same mode parameter m is used in the semantic interpretation of the components, thus expressing that the modes are (by default) deep. In contrast, in the semantic interpretation of the modality type $\Box^{\eta}$, the mode parameter m is changed to $\eta(m)$ (§4.3) in the semantic interpretation of the content.

Next, we describe the more involved semantic interpretation of function types $\tau_1\ @\ m_1 \to \tau_2\ @\ m_2$, which are inhabited by closures. First, we quantify over a thread-id π, a view $\varepsilon'_{\mathrm{mut}}$ and $\varepsilon'_{\mathrm{ro}}$, a locals context Δ', and a fraction q. These represent the possible state at the time the closure is called.

Crucially, the possible choices for this state depend on the mode m. For example, if a closure is **local**, then it may enclose **local** values, and must therefore be applied to a superset of the current Δ. On the other hand, if a closure is **global**, then it can be applied to any Δ', since it cannot depend on Δ at all. A similar principle appears in Dreyer, Neis and Birkedal's work [9], where a distinction between public and private future worlds is used to distinguish functions and continuations. An analogous kind of reasoning applies to **portable** closures, which can be applied to arbitrary sets $(\varepsilon'_{\mathrm{mut}}, \varepsilon'_{\mathrm{ro}})$ of accessible references, as opposed to **nonportable** closures, which must be applied to future worlds $(\varepsilon'_{\mathrm{mut}}, \varepsilon'_{\mathrm{ro}}) \sqsupseteq (\varepsilon_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}})$ of the current state. Finally, an interesting interaction occurs for **portable** and **local** closures. A priori, a **local** closure ought to depend on the locals context Δ. However, since it is also **portable**, we know that it does not depend on non-atomic references. As such, it may only depend on those parts of Δ not related to non-atomic references. We model this by extracting the relevant parts of Δ using the $\mathrm{atomic}(\Delta)$ operation (here left abstract).

Once the future state has been suitably constrained, we use the affinity of m to determine whether this function may be called at most once or many times. In the latter case, the semantic interpretation of the function type must be persistent: this is expressed by a persistence modality $\Box$.

The final part of the assertion states that v is a valid (well-typed) closure if, for every valid (well-typed) actual argument v_1, for every stack size n, given the access permissions expressed by $\mathcal{L}$ and $\mathcal{M}\textsc{em}$, the function application $v(v_1)$ is safe and produces a valid (well-typed) result.

We leave out a detailed explanation of the interpretation of references. In broad strokes, to model atomic references, we use Iris invariants; this is standard. To model non-atomic references, we use custom-made "fractional invariants": they are a simplified variant of RustBelt's fractured borrows [18], without support for RustBelt's lifetime logic. In order to open a fractional invariant, an auxiliary resource is needed. This auxiliary resource is exactly what can be found in $\mathcal{M}\textsc{em}(\varepsilon_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}}, q)$. The semantic interpretation of references must therefore depend on either $\varepsilon_{\mathrm{mut}}$ (in the case of an **uncontended** value) or $\varepsilon_{\mathrm{ro}}$ (in the case of a **shared** value).

5.3 Semantic Typing

In § 5.2, we outlined the standard semantic interpretation of types. However, more generally, a semantic type is defined as any predicate $[\![\tau]\!]_m^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta} : \mathrm{Value} \longrightarrow \mathit{iProp}$ which satisfies the conditions from Definition 5.1. In fact, all definitions from Fig. 8 are parameterized over semantic types, rather than syntactic types. From this abstract notion of a semantic type, we derive the following definition of semantic typing:

$$\begin{aligned}\Gamma \vDash e : \tau \mathrel{@} m \quad \triangleq \quad & \Box \forall \pi\; n\; \varepsilon_{\mathrm{mut}}\; \varepsilon_{\mathrm{ro}}\; q\; \Delta\; \gamma, \mathcal{G}[\![\Gamma]\!]^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}(\gamma) \mathrel{-\!\!*} \\ & \mathcal{L}(\pi, n, \varepsilon_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}}, \Delta) \mathrel{-\!\!*} \mathcal{M}\textsc{em}(\varepsilon_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}}, q) \mathrel{-\!\!*} \mathcal{E}[\![\tau]\!]_{\pi,n,q,m}^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}(\gamma(e))\end{aligned}$$

In this definition, Γ is a semantic type context, containing declarations from variable names to a semantic type and a mode. The context interpretation $\mathcal{G}[\![\Gamma]\!]^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}(\gamma)$ then asserts that every value in a simultaneous substitution γ satisfies the appropriate semantic type in Γ, at the parameters $\varepsilon_{\mathrm{mut}}$, $\varepsilon_{\mathrm{ro}}$ and Δ.

With semantic typing now defined, we prove the following key soundness theorems. First and foremost, we prove that semantic typing is compatible with every inference rule of the type system.

Theorem 5.1 (Compatibility). *Each inference rule of the syntactic type system is also a valid implication of semantic typing judgments. For example:*

$$\Gamma_1 \vDash e_1 : \tau_1 \mathrel{@} m \mathrel{-\!\!*} \Gamma_2 \vDash e_2 : \tau_2 \mathrel{@} m \mathrel{-\!\!*} \Gamma_1 + \Gamma_2 \vDash (e_1, e_2) : \tau_1 \times \tau_2 \mathrel{@} m$$

where $\tau_{[1,2]}$ *are semantic types, and* $\Gamma_{[1,2]}$ *are semantic contexts.*

This theorem is generic over semantic types. As such, it can be applied to the standard interpretation of syntactic types, as well as exotic interpretations of API types. It is therefore strictly stronger than the following more standard fundamental theorem of logical relations:

Theorem 5.2 (Fundamental Theorem of Logical Relations).
If $\Gamma \vdash e : \tau \mathrel{@} m$*, then* $\Gamma' \vDash e : \tau \mathrel{@} m$*, where* Γ' *is the result of applying the standard type interpretation to each declaration in* Γ*.*

The fundamental theorem shows that our semantic typing definition is sound with respect to the syntactic type system. Perhaps more interesting is the following theorem, which states that semantic typing guarantees the absence of data races:

Theorem 5.3 (Semantically Typed Expressions are Data Race Free). *If* $[\,] \vDash e : \tau \mathbin{@} m$*, then executing the closed program* e *(with an initially empty heap and stack) is safe and cannot cause a data race.*

Proof. The proof instantiates the semantic typing definition to an empty memory and locals context, applies adequacy of the weakest precondition to prove that e is safe, and applies Corollary 3.1 to prove that e does not incur a data race. ☐

5.3.1 Semantic interpretation of locks. When proving the compatibility lemmas from Theorem 5.1, it becomes necessary to consider the semantic interpretation of locks. Our locks act as operations over syntactic contexts. These operations are easily lifted to semantic contexts, because they examine just the "mode" information in the context, and ignore the "type" information. Applying a lock to a context filters out declarations with an incompatible locality, affinity or portability, and weakens the uniqueness and contention of the remaining declarations. By exploiting the mode weakening condition (Definition 5.1), one observes that this operation preserves the semantic interpretation of a context.

lemma 5.4 (Semantic Lock preservation).

$$\mathcal{M}\textit{EM}(\varepsilon_{\text{mut}}, \varepsilon_{\text{ro}}, q) \mathrel{-\!\!*} \mathcal{G}[\![\Gamma]\!]^{\varepsilon_{\text{mut}},\varepsilon_{\text{ro}},\Delta}(\gamma) \mathrel{-\!\!*} \exists \varepsilon'_{\text{mut}}.\ \mathcal{M}\textit{EM}(\varepsilon'_{\text{mut}}, \varepsilon_{\text{ro}}, q) * \mathcal{G}[\![\text{🔒}_{(l,o,p)}\Gamma]\!]^{\varepsilon'_{\text{mut}},\varepsilon_{\text{ro}},\Delta}(\gamma)$$

The application of a lock can change a binding from **unique** to **aliased**. In that case, new fractional invariants must be allocated, which means expanding ε_{mut}.

Our next observation is that once a lock operation has been applied, the context contains bindings at certain modes only. For example, a **portable** lock guarantees that $\text{🔒}_{(l,o,\textbf{portable})}\Gamma$ contains no declarations at mode **nonportable**,**uncontended** or **shared**. As a result, we can lift many of the conditions from Definition 5.1 to the semantic interpretation of locked contexts.

For example, the following lemma lets us arbitrarily change ε_{mut} and ε_{ro} in a semantic context with a **portable** lock:

lemma 5.5. $\mathcal{G}[\![\Gamma]\!]^{\varepsilon_{\text{mut}},\varepsilon_{\text{ro}},\Delta}(\gamma) \mathrel{-\!\!*} \mathcal{G}[\![\text{🔒}_{(l,o,\textbf{portable})}\Gamma]\!]^{\varepsilon'_{\text{mut}},\varepsilon'_{\text{ro}},\Delta}(\gamma)$

These lemmas are crucial for proving the compatibility lemmas for fork and arrow types.

6 SPECIFYING AND VERIFYING THE CAPSULE API

The Capsule API is implemented using unsafe type casts between an inner type `'a` at various modes and `('a, 'k) Data.t`. Hence our soundness proof in § 5 does not yield soundness of the Capsule API (§ 2.7), since there is no compatibility lemma for `Obj.magic`. Instead, we manually verify that the Capsule API (§ 2.7) implementation is semantically sound.

Recall the introduction to capsules in § 2.7. In broad terms, a capsule wraps data which can refer to mutable state, and a key of some existential type is used to regulate thread-safe access to this data. To represent mutable state semantically, the value interpretation defined in § 5 is parameterized by the sets of accessible read-write and read-only references $\varepsilon_{\text{mut}}, \varepsilon_{\text{ro}}$. In § 5, we saw how the memory interpretation $\mathcal{M}\textit{EM}(\varepsilon_{\text{mut}}, \varepsilon_{\text{ro}}, q)$ grants access to these references. The key difficulty to the proof of semantic soundness of the Capsule API is tracking and sharing this memory interpretation across calls to the API from different threads. To be more concrete, when reasoning about the creation of a new `Data.t`, the constructor function yields a fresh instance of $\mathcal{M}\textit{EM}(\varepsilon_{\text{mut}}, \emptyset, 1)$, which is necessary to reason about subsequent calls to `Data.map`, `Data.extract`, *etc.* An important part of the proof is thus to define the right Iris invariant to track and store these propositions. We will refer to this invariant as *keyInv*.

In this section we first describe the semantic interpretations of the Capsule API (§ 2.7) types `('a, 'k) Data.t` and `'k Key.t`, outline the proof of safety of `map`, and state our overall soundness theorem.

As alluded to in § 5.1, we define two bespoke semantic type interpretations for the `Data` and `Key` types. The data interpretation $\llbracket \texttt{('a, 'k) Data.t} \rrbracket_{-}^{-,-,-}(v)$, where we denote unused parameters by –, simply wraps an interpretation $\llbracket \texttt{'a} \rrbracket_{\textbf{legacy}}^{\varepsilon_{\mathrm{mut}},-,-}(v)$ of the value at legacy mode, as well as some auxiliary ghost resources to track which $\varepsilon_{\mathrm{mut}}$ set is required to interpret the value. The key interpretation $\llbracket \texttt{'k Key.t} \rrbracket_{m}^{-,-,\Delta}(w)$ where $m.u$ = **unique** gives full access to the contents of the capsule. To be more precise, together with *keyInv*, it can be used to gain full access to a memory interpretation $\mathcal{M}\textsc{em}(\varepsilon_{\mathrm{mut}}, \emptyset, 1)$ corresponding to the mutable state needed to interpret `'a`. Similarly, if $m.u$ = **aliased**, then it can be used to gain partial access to a memory interpretation with read-only access to $\varepsilon_{\mathrm{mut}}$, namely $\mathcal{M}\textsc{em}(\emptyset, \varepsilon_{\mathrm{mut}}, q)$ at some fraction q. In either case, the key interpretation can only be reestablished if the corresponding memory interpretation is relinquished.

To give an idea of how the Capsule API (§ 2.7) is verified, we outline the proof of `Data.map`, which is implemented as follows:

```
let Data.map key f v = (key, Obj.magic (f (Obj.magic v)))
```

Given `key @ unique : 'k Key.t`, some data `v @ . : ('a, 'k) Data.t` protected by that key, and a function `f`, it first casts `v` to `v @ . : 'a`, and then executes `f v`. Our goal is to show:

$$\forall \texttt{'k 'a 'b}.\ [\,] \vDash \text{Data.map} : \tau_{map}(\texttt{'k}, \texttt{'a}, \texttt{'b}) \mathbin{@} (\textbf{global}, \textbf{many}, \textbf{aliased}, \textbf{portable}, \textbf{contended})$$ [10]

where $\tau_{map}(\texttt{'k}, \texttt{'a}, \texttt{'b})$ is the type of `Data.map`, and the semantic interpretation of $\tau_{map}(\texttt{'k}, \texttt{'a}, \texttt{'b})$ corresponds to the arrow interpretation applied to $\llbracket \texttt{'k Key.t} \rrbracket$, $\llbracket \texttt{('a, 'k) Data.t} \rrbracket$ and the universally quantified semantic type interpretations $\llbracket \texttt{'a} \rrbracket$ and $\llbracket \texttt{'b} \rrbracket$.

We prove this goal by going step-by-step through the implementation. To verify the cast we need to show that we can reproduce it semantically, *i.e.*, As discussed above, from $\llbracket \texttt{('a, 'k) Data.t} \rrbracket_{-}^{-,-,-}(\texttt{v})$ we obtain $\llbracket \texttt{'a} \rrbracket_{\textbf{legacy}}^{\varepsilon_{\mathrm{mut}},-,-}(\texttt{v})$, for some $\varepsilon_{\mathrm{mut}}$. To execute `f v`, however, we need a matching memory interpretation $\mathcal{M}\textsc{em}(\varepsilon_{\mathrm{mut}}, \emptyset, 1)$. It is obtained by temporarily giving up ownership of the semantic interpretation of the key `k`, which is restored by returning an updated view after the execution of `f v`.

The verification of all Capsule API (§ 2.7) functions is similar, in spirit, to what we just explained; Although more complex interactions between keys, data, and memory interpretations need to be handled for read-only access. We have also verified an implementation of the reader-writer lock. Overall, we prove the following theorems:

THEOREM 6.1. *The Capsule API (Fig. 3) is semantically sound.*

THEOREM 6.2. *The Reader-Writer Lock API (Fig. 4) is semantically sound.*

7 RELATED WORK

There is a vast literature on using types to soundly (but conservatively) enforce absence of data races, dating back at least to Abadi and Flanagan's early and influential paper [12]. There are also a number of well-known approaches to static race detection for Java and C [10, 23, 28], which rely on whole-program call-graph information, sacrificing soundness for scalability and error detection with fewer false positives. In the interest of space, we compare here with the most closely related work on type-based approaches, focusing attention on the goals we set out in the introduction.

---

[10] Since the module is shared across threads, we want its mode to be **portable** and **contended**.

Capsules bear a close resemblance to the `GhostCell` API proposed for Rust by Yanovski et al. [31]. The two approaches tackle a similar problem, but come at it from opposite directions. Rust natively supports thread-safe sharing of mutable data, but has only limited support for safely programming mutable data types with internal aliasing. The aim of `GhostCell` is to overcome that limitation. OCaml has the reverse challenge: safe mutable state with internal aliasing is no problem—thanks to garbage collection—but the language does not guarantee data race freedom when state is shared across threads. The aim of capsules is to overcome *that* limitation. Hence, a key design goal of capsules, not met by `GhostCell`, is to allow existing sequential OCaml code to be easily made thread-safe, even if that code constructs data structures with internal aliasing.

The goals of the capsule API also align closely with those of Haller and Loiko's work on LaCasa [16]. LaCasa extends Scala with aliasing control, guaranteeing thread safety in a backwards-compatible way by separating data from the (affine) permission to access it. A box `Box[T]` in LaCasa encapsulates some mutable data of type `T`, and roughly corresponds to `('k, ref 'a) Data.t` in our system. LaCasa's `CanAccess` type plays a role similar to our keys, in that it provides the permission necessary to access some box.

`Box[T]` only supports classes `T` that follow the *object-capability discipline (ocap)*, which ensures for example that `T` does not access global state. LaCasa adds an annotation to classes to track whether they are ocap. This is similar to our restriction that the Capsule API callbacks can only call **portable** functions, since those cannot access shared state, either. The default portability mode is **nonportable**, so as discussed in §2.8, we need to annotate **portable** functions explicitly in order to allow them to be invoked on capsules.

There are, however, some major differences between LaCasa and DRFCaml. Firstly, although LaCasa does provide simple locality and affinity tracking for the `Box` and `CanAccess` types, its approach to affinity tracking relies on integration with its message-passing concurrency primitives. As such, it is not clear if it can be generalized to handle unstructured concurrency. DRFCaml, on the other hand, tracks locality and affinity of all types. Consequently, capsules are easier to integrate with other APIs that use modes, like the reader-writer lock. Our system also supports sharing or borrowing keys, which we use to allow shared read-only access to encapsulated data. Secondly, in LaCasa, an access permission is tied to the *unique* box that it protects (and with which it was created). Thanks to the combination of Scala's path-dependent types and implicit parameters, the tracking of this access permission is mostly automated. In contrast, the Capsule API allows multiple encapsulated pieces of data to be protected by a single key, but these keys have to be passed around explicitly.

Capturing Types [5, 30] and Reachability Types [4, 29] attack a high-level problem that is very similar to ours: to develop a mechanism that keeps track of aliasing, thereby allowing data races to be statically forbidden, without imposing *a priori* restrictions on the shape of the heap.

The key idea behind *capturing types* is to decorate closures with *sets of variables* to keep track of which capabilities each closure has access to. To make such a system tractable, Boruch-Gruszecki et al. [5] define a subtyping discipline—similar to DRFCaml's submoding discipline—and a new boxing type to prevent the unnecessary propagation of annotations whenever a variable is not directly used. They then define a *pure* closure as one that captures no capabilities, and an *impure* function as one that can capture any capability, expressed using the universal capability **cap** (similar to $\top$ in our locality, portability, and affinity axes). While DRFCaml does not express purity (portable closures may still atomically mutate data), the overall methodology is similar: a closure marked as **portable** may not mutate enclosed non-atomic data. Likewise, the mode of a function's argument does not determine the mode of the function—*e.g.*, one can define a signature for `map` which is itself **portable**, while taking a **nonportable** function as argument.

Xu et al. [30] go on to show how capturing types can be used to prevent data races. They extend the capturing types design [5] with fork-join parallelism and static prevention of data races. The calculus performs *descriptive alias tracking* (closures can capture arbitrary variables and get adequately labeled), and imposes restrictions when closures are invoked in parallel: namely, closures can run in parallel only if their capturing types are "separate". Note that separation here does not mean disjointness: to allow for multiple simultaneous readers, the calculus introduces two new root capability types, **ref** for general mutation, and **rdr** for general reading, where **rdr** is separate from itself, but not from **ref**. The calculus thus depends on a structured fork-join to regain mutable access to some temporarily shared data structure. In contrast, DRFCaml prevents data races even in the presence of unstructured concurrency, and is compatible with nondeterministic concurrency mechanisms such as reader-writer locks.

Reachability types [4, 29] are similar to capturing types, but track the reachable set of a function's free variables rather than tracking the effect of using them. Their system allows one to express a unique access restriction and a use-once policy, similar to DRFCaml's uniqueness and affinity axes. They also support programming patterns such as "non-escaping function arguments", which DRFCaml accounts for using **local** arguments. As with capturing types, reachability types can be used to guarantee safe parallel computations, by asserting that reachable variables are either disjoint or read-only on both sides. But also as with capturing types, Bao et al. [4] restrict attention to structured parallelism.

Both reachability and capturing types guarantee data race freedom. However, it is unclear whether a similar methodology can be applied to a language such as OCaml. Boruch-Gruszecki et al. [5] describe various language requirements to make such systems usable, several of which do not apply to OCaml. In particular, the language should have support for reference-dependent typing (similar to path-dependent typing in DOT [2]) as well as subtyping. Furthermore, without a language feature such as Scala's implicits, capability parameters would need to be added to all existing signatures in legacy code.

There have been a number of other type-based approaches to data race freedom which, like DRFCaml, (a) use some form of (often *region*-based [27]) encapsulation to separate chunks of mutable data from one another, and (b) annotate pointer types with *capabilities* [6] to track uniqueness and aliasing and to ensure safe mutation [8, 15, 14, 24, 22]. We will focus here on the most recent such approaches.

Milano et al. [22] use so-called `isolated` (`iso`) pointers, which "dominate" (*i.e.,* control access to) a region of the heap, in order to achieve "fearless concurrency". The flexibility of their type system comes from two key features: (1) the ability to type check programs with a minimal need for user-level annotations beyond the `iso` keyword, and (2) a property called "tempered domination", which allows for domination to be *locally* broken, and eventually repaired, sometimes requiring a dynamic disconnectedness test on regions. Thanks to tempered domination, it becomes trivial to implement doubly-linked lists (notoriously difficult in languages such as Rust). The same flexibility can be observed in DRFCaml, which allows for arbitrary legacy data structures to be encapsulated in a capsule. The disconnectedness test also enables isolated regions to be dynamically separated, a feature that is not supported by DRFCaml. Milano et al. [22] establish data race freedom by proving a stronger global isolation property of the language. Unlike DRFCaml, they do not yet support shared read-only access, and consider only a `send` primitive to share `iso` pointers across threads. Finally, unlike DRFCaml, their primary goal is to design a new language with the same guarantees as existing work but with more flexibility and minimal annotations, whereas the goal of DRFCaml is to safely port an existing language (and its legacy code) to a concurrent setting.

Arvidsson et al. [3] present Reggio, a region-based type system design applied to the Verona language, whose notion of reference capabilities and "view adaptations" bears resemblance to DRFCaml's modes and context locks 🔒$_{(l,o,p)}$. Regions in Reggio are isolated, and can only be mutated while *active*. This is done using a lexically scoped construct, **enter**, which takes a unique designated reference—called the “bridge object”—as its argument and activates the associated region. The bridge object functions analogously to a key in a capsule, but offers a bit more flexibility. Notably, bridge objects only need to be externally unique (a single incoming reference from another region), and may be an arbitrary object from that region. To maintain region isolation, programs may only mutate one region at a time: the so-called “window of mutability”. An active region is marked as *suspended* (accessible, but immutable) whenever another region is entered, and *closed* (inaccessible except for its unique bridge object) when its lexical scope ends. In general, no references may point to non-bridge objects from other regions. An exception is made for temporary references, which can point to the temporary objects of a suspended region. This functionality is not fully supported by DRFCaml, for which the lifetime information of **local** is too coarse-grained. An interesting direction for future work would be to generalize DRFCaml with similar techniques as in Reggio, *i.e.,* distinguishing between “**local** to current region” and “**local** to some parent region”.

Cheeseman et al. [7] build on the Reggio design [3], and outline exactly how regions (and their bridge objects) can be synchronized across threads, akin to how access to capsules are shared by wrapping keys in a synchronization primitive. Reggio’s guiding principle to achieve data race freedom is similar to DRFCaml: programs that run in parallel may only mutate one isolated region at a time. Regions, like capsules, can be nested and merged (capsules can be merged by destroying a capsule in another capsule). However, Reggio’s notion of isolation is somewhat rigid: once an object crosses into another region, it may never be mutated, even if it would be safe to do so (for example an atomic store operation). In contrast, DRFCaml enables the extraction of data from a capsule so long as it is **contended**, thus allowing for a more flexible notion of isolation.

DRFCaml is motivated in large part by the goal of ensuring data race freedom in a well-established high-level language with a large legacy code base, namely OCaml. Consequently, we have designed DRFCaml as an extension of the type-and-mode system proposed by Lorenzen et al. [21]. Their design supports global type-and-mode inference in a Hindley-Milner style system with higher-order functions—an important criterion for adoption in the functional programming community—and an implementation of such an inference system has been successfully deployed at Jane Street. Since DRFCaml’s typing rules are similar to Lorenzen et al.’s, we expect it to enjoy similar type-and-mode inference, though that remains to be demonstrated and evaluated in future work. Moreover, our design illustrates that, despite their coarse-grained simplicity, Lorenzen et al.’s locality and uniqueness modes have uses above and beyond their original intended purposes. As we have shown, locality is useful not only for stack allocation but also for implementing temporary borrowing of shared resources (*e.g.,* when acquiring a reader lock), and uniqueness is useful not only for memory reuse but also for tracking ownership of capsule keys.

REFERENCES

[1] Javad Abdi, Gilead Posluns, Guozheng Zhang, Boxuan Wang, and Mark C. Jeffrey. 2024. When Is Parallelism Fearless and Zero-Cost with Rust?. In *Symposium on Parallelism in Algorithms and Architectures.* 27–40. https://doi.org/10.1145/3626183.3659966

[2] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 9600)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.). Springer, 249–272. https://doi.org/10.1007/978-3-319-30936-1_14

[3] Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. 2023. Reference Capabilities for Flexible Memory Management. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 1363–1393. https://doi.org/10.1145/3622846

[4] Yuyan Bao, Guannan Wei, Oliver Bracevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–32. https://doi.org/10.1145/3485516

[5] Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondrej Lhoták, and Jonathan Immanuel Brachthäuser. 2023. Capturing Types. *ACM Transactions on Programming Languages and Systems* 45, 4 (2023), 21:1–21:52. https://doi.org/10.1145/3618003

[6] John Boyland, James Noble, and William Retert. 2001. Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In *European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science, Vol. 2072)*. Springer, 2–27. https://doi.org/10.1007/3-540-45337-7_2

[7] Luke Cheeseman, Matthew J. Parkinson, Sylvan Clebsch, Marios Kogias, Sophia Drossopoulou, David Chisnall, Tobias Wrigstad, and Paul Liétar. 2023. When Concurrency Matters: Behaviour-Oriented Concurrency. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 1531–1560. https://doi.org/10.1145/3622852

[8] Sylvan Clebsch, Juliana Franco, Sophia Drossopoulou, Albert Mingkun Yang, Tobias Wrigstad, and Jan Vitek. 2017. Orca: GC and type system co-design for actor languages. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017). https://doi.org/10.1145/3133896

[9] Derek Dreyer, Georg Neis, and Lars Birkedal. 2012. The impact of higher-order state and control effects on local relational reasoning. *J. Funct. Program.* 22, 4-5 (2012), 477–528. https://doi.org/10.1017/S095679681200024X

[10] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, static detection of race conditions and deadlocks. In *Symposium on Operating Systems Principles (SOSP)*. 237–252. https://doi.org/10.1145/1165389.945468

[11] Kasra Ferdowsi. 2023. The Usability of Advanced Type Systems: Rust as a Case Study. *CoRR* abs/2301.02308 (2023). https://doi.org/10.48550/arXiv.2301.02308

[12] Cormac Flanagan and Martín Abadi. 1999. Types for Safe Locking. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 1576)*. Springer, 91–108. http://users.soe.ucsc.edu/~cormac/papers/esop99.pdf

[13] Aïna Linn Georges, Benjamin Peters, Laila Elbeheiry, Leo White, Stephen Dolan, Richard A. Eisenberg, Chris Casinghino, François Pottier, and Derek Dreyer. 2024. *Appendix: Data Race Freedom à la Mode.* Technical Report.

[14] Paola Giannini, Marco Servetto, and Elena Zucca. 2016. Types for Immutability and Aliasing Control. In *Proceedings of the 17th Italian Conference on Theoretical Computer Science, Lecce, Italy, September 7-9, 2016 (CEUR Workshop Proceedings, Vol. 1720)*, Vittorio Bilò and Antonio Caruso (Eds.). CEUR-WS.org, 62–74. https://ceur-ws.org/Vol-1720/full5.pdf

[15] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and reference immutability for safe parallelism. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 21–40. https://www.cs.drexel.edu/~csg63/papers/oopsla12.pdf

[16] Philipp Haller and Alex Loiko. 2016. LaCasa: lightweight affinity and object capabilities in Scala. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016*. 272–291. https://doi.org/10.1145/2983990.2984042

[17] Ralf Jung. 2020. *Understanding and evolving the Rust programming language.* Ph. D. Dissertation. Saarland University, Saarbrücken, Germany. https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647

[18] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the Foundations of the Rust Programming Language. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 66:1–66:34. https://people.mpi-sws.org/~dreyer/papers/rustbelt/paper.pdf

[19] 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. https://people.mpi-sws.org/~dreyer/papers/iris-ground-up/paper.pdf

[20] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *European Conference on Object-Oriented Programming (ECOOP)*. 17:1–17:29. https://people.mpi-sws.org/~dreyer/papers/iris-weak/paper.pdf

[21] Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. *Proc. ACM Program. Lang.* ICFP. https://antonlorenzen.de/mode-inference.pdf

[22] Mae Milano, Joshua Turcotti, and Andrew C. Myers. 2022. A flexible type system for fearless concurrency. In *Programming Language Design and Implementation (PLDI)*. 458–473. https://doi.org/10.1145/3519939.3523443

[23] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Programming Language Design and Implementation (PLDI)*. 308–319. https://doi.org/10.1145/1133981.1134018

[24] Marco Servetto, David J Pearce, Lindsay Groves, and Alex Potanin. 2013. Balloon types for safe parallelisation over arbitrary object graphs. In *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, Vol. 107.

[25] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting Parallelism onto OCaml. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 113:1–113:30. https://doi.org/10.1145/3408995

[26] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A logical approach to type soundness. *JACM* (2024). https://iris-project.org/pdfs/2024-jacm-logical-type-soundness.pdf

[27] Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and Computation* 132, 2 (1997), 109–176. http://www.irisa.fr/prive/talpin/papers/ic97.pdf

[28] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: static race detection on millions of lines of code. In *Foundations of Software Engineering (FSE)*. 205–214. https://doi.org/10.1145/1287624.1287654

[29] Guannan Wei, Oliver Bracevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2024. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 393–424. https://doi.org/10.1145/3632856

[30] Yichen Xu, Aleksander Boruch-Gruszecki, and Martin Odersky. 2024. Degrees of Separation: A Flexible Type System for Safe Concurrency. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 1181–1207. https://doi.org/10.1145/3649853

[31] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. 2021. GhostCell: Separating permissions from data in Rust. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–30. https://plv.mpi-sws.org/rustbelt/ghostcell/paper.pdf

A OPERATIONAL SEMANTICS

$$\frac{\pi' \text{ fresh in } s}{(h, s, fs, \mathsf{fork}(e)) \rightsquigarrow_\pi (h, s[\pi' := []], (), [(\pi', e)])}$$

$$\frac{n = |s[\pi]| \quad \iota \text{ fresh in } fs}{(h, s, fs, \lambda^{\textbf{local}} f\ x, e) \rightsquigarrow_\pi (h, s[\pi][n := \iota], fs \uplus \{\iota\}, (\lambda^{((\pi,n),\iota)} f\ x, e), [])}$$

$$\frac{\ell \text{ fresh in } h \quad \iota \text{ fresh in } fs}{(h, s, fs, \lambda^{\textbf{global}} f\ x, e) \rightsquigarrow_\pi (h[\ell := \iota], s, fs \uplus \{\iota\}, (\lambda^{(\ell,\iota)} f\ x, e), [])}$$

$$\frac{n = |s[\pi]| \quad s' = s[\pi][n := (\textsc{r}_0, v)]}{(h, s, fs, \mathsf{alloc}^{\textbf{local}}(v)) \rightsquigarrow_\pi (h, s', fs, (\pi, n), [])} \qquad \frac{\ell \text{ fresh in } h \quad h' = h[\ell := (\textsc{r}_0, v)]}{(h, s, fs, \mathsf{alloc}^{\textbf{global}}(v)) \rightsquigarrow_\pi (h', s, fs, \ell, [])}$$

$$\frac{n = |s[\pi]|}{(h, s, fs, \mathsf{region}(e)) \rightsquigarrow_\pi (h, s, fs, \mathsf{end}^n(e), [])} \qquad \frac{s' = s[\pi := \lfloor s[\pi] \rfloor_{<n}]}{(h, s, fs, \mathsf{end}^n(v)) \rightsquigarrow_\pi (h, s', v, [])}$$

$$\frac{h[\ell] = (\textsc{r}_n, v) \quad h' = h[\ell := (\textsc{r}_{n+1}, v)]}{(h, s, fs, !^{\text{NA}_1} \ell) \rightsquigarrow_\pi (h', s, fs, !^{\text{NA}_2} \ell, [])} \qquad \frac{h[\ell] = (\textsc{r}_{n+1}, v) \quad h' = h[\ell := (\textsc{r}_n, v)]}{(h, s, fs, !^{\text{NA}_2} \ell) \rightsquigarrow_\pi (h', s, fs, v, [])}$$

$$\frac{h[\ell] = (\textsc{r}_0, w) \quad h' = h[\ell := (\textsc{wr}, w)]}{(h, s, fs, \ell \xleftarrow{\text{NA}_1} v) \rightsquigarrow_\pi (h', s, \ell \xleftarrow{\text{NA}_1} v, [])} \qquad \frac{h[\ell] = (\textsc{wr}, w) \quad h' = h[\ell := (\textsc{r}_0, v)]}{(h, s, fs, \ell \xleftarrow{\text{NA}_2} v) \rightsquigarrow_\pi (h', s, (), [])}$$

$$\frac{h[\ell] = (\textsc{r}_n, v)}{(h, s, fs, !^{\text{AT}} \ell) \rightsquigarrow_\pi (h, s, fs, v, [])} \qquad \frac{h[\ell] = (\textsc{r}_0, w) \quad h' = h[\ell := (\textsc{r}_0, v)]}{(h, s, fs, \ell \xleftarrow{\text{AT}} v) \rightsquigarrow_\pi (h', s, (), [])}$$

Fig. 9. Selected rules of the operational semantics.

A selection of the small-step reduction rules appears in Fig. 9.

How lock states are used. The first step of a non-atomic load ($!^{\text{NA}_1}$) requires the lock state to be a read state $\textsc{r}_m$ and increases the number of readers by one by changing the lock state to $\textsc{r}_{m+1}$. The second step of a non-atomic load ($!^{\text{NA}_2}$) decreases the number of readers back to $\textsc{r}_m$. The first step of a non-atomic store ($\xleftarrow{\text{NA}_1}$) requires the lock state to be $\textsc{r}_0$ — indicating that no other thread is trying to read or write this address — and sets the lock state to $\textsc{wr}$. The second step of a non-atomic store ($\xleftarrow{\text{NA}_2}$) releases this address by reverting the lock state to $\textsc{r}_0$.

B PROGRAM LOGIC

In this appendix, we present a program logic for DRFCamlLang. The program logic depends on the following three resource predicates, given here with their intuitive meanings:

$\pi \hookrightarrow n$	stack of thread π has size n
$n \Mapsto_\pi w$	stack of thread π stores w at offset n
$\ell \mapsto w$	heap location ℓ stores w

Each of these predicates describes exclusive ownership over fragments of the global state. A step that does not alter the global state does not require exclusive ownership; it requires just shared knowledge about some fragment. To that end, we define $n \Mapsto_\pi^q w$ and $\ell \mapsto^q w$, where q is a fraction, to describe fractional ownership over state fragments. The lock state is entirely abstracted away: it is not explicit at the level of the program logic. Instead, the distinction between atomic and non-atomic accesses is expressed in the logic through the rules for *invariants*, which we will return to in §5.

We define the program logic in terms of Iris's weakest preconditions [19], adjusted to work on languages where the thread-id's are visible at the level of the operational semantics (similar adjustments have been made in *e.g.,* [20], where thread-id's were paired with expressions; we pair them with steps in the operational semantics instead). Weakest precondition propositions are denoted by $\mathsf{wp}\ e\ \{\Phi\}_\pi$, and intuitively express that expression e may execute in thread π and does not get stuck, and if it reduces to a value v then $\Phi(v)$ holds. Hoare triples have a similar interpretation, and are derived from weakest preconditions. Finally, some of the rules use the so-called later modality, denoted $\triangleright$, to indicate that a step has been taken. Intuitively, $\triangleright P$ means that P holds one step later.

In the remainder of this appendix, we present a selection of program logic rules for DRFCamlLang. First, we present the rules that allocate new state fragments, namely fork, stack allocation, and heap allocation.

$$\frac{\triangleright(\forall \pi.\ \pi \hookrightarrow 0 \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\top\}_\pi) \qquad \triangleright \Phi(())}{\mathsf{wp}\ \mathsf{fork}(e)\ \{\Phi\}_{\pi'}}$$

$$\{\pi \hookrightarrow n\}\ \mathsf{alloc}^{\mathsf{local}}(v)\ \left\{w.\ w = (\pi, n) * n \Mapsto_\pi v * \pi \hookrightarrow n+1\right\}_\pi$$

$$\{\top\}\ \mathsf{alloc}^{\mathsf{global}}(v)\ \{w.\ \exists \ell, w = \ell * \ell \mapsto v\}_\pi$$

Fork spawns a new thread of some thread-id π, and allocates an empty stack. The proof obligation of the spawned thread is a new weakest precondition — now parameterized by π — which may depend on the newly allocated stack size predicate $\pi \hookrightarrow 0$. The stack size predicate is then used for subsequent stack allocations. Stack allocation uses $\pi \hookrightarrow n$ to allocate a new stack fragment predicate $n \Mapsto_\pi v$, increasing the stack size to $\pi \hookrightarrow n+1$. Finally, heap allocation does not depend on any resources, and returns a freshly allocated $\ell \mapsto v$.

Once allocated, resource fragments are used to reason about load and store operations. Below we show rules for non-atomic load and store over heap locations. Note that since the load operation does not alter state (insofar as it does not alter the value pointed to by the location), it suffices to assume fractional ownership over the location ℓ.

$$\{\ell \mapsto^q v\}\ !^{\mathrm{NA}_1}\ell\ \{w.\ w = v * \ell \mapsto^q v\}_\pi \qquad\qquad \{\ell \mapsto v\}\ \ell \leftarrow^{\mathrm{NA}_1} w\ \{w'.\ w' = () * \ell \mapsto w\}_\pi$$

Finally, we describe the rules for $\mathsf{region}(e)$ and $\mathsf{end}^n(v)$. Starting a region only requires knowledge of the current stack size, as expressed by $\pi \hookrightarrow n$ (note that the thread-id of the stack size predicate matches that of the weakest precondition). Ending a region, on the other hand, requires more resources. Since $\mathsf{end}^n(v)$ deallocates *all* stack locations at and above the cutoff n, the proof rule requires every stack fragment predicate from n to the top of the stack, namely $m-1$. Each of these are consumed by the proof rule, and the stack size predicate is returned with the new size n.

$$\frac{\pi \hookrightarrow n \qquad \triangleright(\pi \hookrightarrow n \mathrel{-\!\!*} \mathsf{wp}\ \mathsf{end}^n(e)\ \{\Phi\}_\pi)}{\mathsf{wp}\ \mathsf{region}(e)\ \{\Phi\}_\pi}$$

$$\frac{n \leq m}{\left\{\pi \hookrightarrow m * \mathop{\Huge *}_{k\in[n,m)} k \Mapsto_\pi -\right\}\ \mathsf{end}^n(v)\ \{w.\ w = v * \pi \hookrightarrow n\}_\pi}$$

Each proof rule is derived from the definition of weakest preconditions, which itself is proved sound by the following adequacy theorem.

Theorem B.1 (Adequacy of the Weakest Precondition). *Let Φ be a first-order predicate. If $\vdash \mathsf{wp}\ e\ \{\Phi\}_\pi$ and $(\sigma, e) \rightsquigarrow^*_\pi (\sigma', e', [(\pi_1, e_1), \cdots, (\pi_n, e_n)])$, then:*

(1) $\forall i \in [1, n].\ e_i$ is a value $\vee\ (\sigma', e_i) \rightsquigarrow_{\pi_i} -$
(2) if e' is a value, then $\Phi(e')$ holds

Proof. Follows the proof of adequacy of Iris's weakest preconditions, now with thread-ids. □

The adequacy statement gives rise to the following corollary, stating that if one can prove a weakest precondition statement for some expression e, then executing that expression does not cause a data race.

Corollary B.1. *If $\vdash \mathsf{wp}\ e\ \{\Phi\}_\pi$ then executing the closed program e (with an initially empty heap and stack, and with thread identifier π) cannot cause a data race.*

Proof. Apply Theorem 3.1 followed by Theorem B.1. □

C TYPING RULES

Figures 10 and 11 contain all typing rules of DRFCaml.

D FRACTIONAL INVARIANTS

We use *fractional invariants* to model shared access to non-atomic references. Fractional invariants are a variant of fractured borrows from RustBelt [18], but without the lifetime logic. As with fractured borrows, fractional invariants grant concurrent and non-atomic access to some resource. Crucially, if access to the invariant is shared, access to its contents might only be partial.

Fractional invariants use fractional resource tokens to get partial access to the resources in P. Let $\lambda v\ q.\ P(v, q)$ be a predicate over some v of parameterized typed W — which we will refer to as a view — and some fraction q, and let $[\mathsf{FrTok} : \gamma : v]_q$ denote the access token of name γ, at fraction q and view v. We write $\mathsf{FrInv}^{\mathcal{N},\gamma}(P)$ to denote a fractional invariant, under the namespace $\mathcal{N}$ and with name γ. The following lemma let's us open the fractional invariant:

$$\mathcal{N}^{\uparrow} \subseteq \mathcal{E} \rightarrow \mathsf{FrInv}^{\mathcal{N},\gamma}(P) \mathrel{-\!\!*} [\mathsf{FrTok} : \gamma : v]_q \ {}_{\mathcal{E}}{\Rrightarrow\!\!*}_{\mathcal{E}} \triangleright P(v, q) * (\triangleright P(v, q) \ {}_{\mathcal{E}}{\Rrightarrow\!\!*}_{\mathcal{E}} \ [\mathsf{FrTok} : \gamma : v]_q)$$

Here, ${}_{\mathcal{E}_1}{\Rrightarrow\!\!*}_{\mathcal{E}_2}$ denotes the so-called fancy update modality, which allows us to open invariants included in the mask $\mathcal{E}_1$, and restricting further accesses to $\mathcal{E}_2$. Note however, that in the above lemma, the mask does not change! Instead, the access token of fraction q is lost, and can only be regained by relinquishing $P(v, q)$, thus preventing $P(v, q)$ from being extracted twice. In fact, it is precisely because the mask does not change that the resources can be accessed non-atomically.

Note that if one owns the full fraction $[\mathsf{FrTok} : \gamma : v]_1$, the invariant behaves like a non-atomic, cancellable invariant. Furthermore, full ownership enables the change of the view v as follows:

$$\mathcal{N}^{\uparrow} \subseteq \mathcal{E} \rightarrow \mathsf{FrInv}^{\mathcal{N},\gamma}(P) \mathrel{-\!\!*} [\mathsf{FrTok} : \gamma : v]_1 \ {}_{\mathcal{E}}{\Rrightarrow\!\!*}_{\mathcal{E}} \triangleright P(v, 1) * (\forall v'.\ \triangleright P(v', 1) \ {}_{\mathcal{E}}{\Rrightarrow\!\!*}_{\mathcal{E}} \ [\mathsf{FrTok} : \gamma : v']_1)$$

$$\frac{}{\Gamma \vdash () : \mathbb{1} \mathbin{@} m}\ \textsc{Unit} \qquad \frac{}{\Gamma \vdash b : \mathbb{B} \mathbin{@} m}\ \textsc{Bool} \qquad \frac{}{\Gamma \vdash z : \mathbb{Z} \mathbin{@} m}\ \textsc{Int} \qquad \frac{}{\Gamma, x : \tau \mathbin{@} m, \Gamma' \vdash x : \tau \mathbin{@} m}\ \textsc{Var}$$

$$\frac{\Gamma, \text{🔒}_{(l,o,p)}, x : \tau_1 \mathbin{@} m_1 \vdash e : \tau_2 \mathbin{@} m_2}{\Gamma \vdash \lambda^l x, e : (\tau_1 \mathbin{@} m_1 \to \tau_2 \mathbin{@} m_2) \mathbin{@} (l, o, u, p, c)}\ \textsc{NonRecLam}$$

$$\frac{\Gamma, \text{🔒}_{(l,\mathbf{many},p)}, x : \tau_1 \mathbin{@} m_1, f : (\tau_1 \mathbin{@} m_1 \to \tau_2 \mathbin{@} m_2) \mathbin{@} (l, \mathbf{many}, u, p, c) \vdash e : \tau_2 \mathbin{@} m_2}{\Gamma \vdash \lambda^l f\ x, e : (\tau_1 \mathbin{@} m_1 \to \tau_2 \mathbin{@} m_2) \mathbin{@} (l, \mathbf{many}, u, p, c)}\ \textsc{RecLam}$$

$$\frac{\Gamma_1 \vdash e_1 : (\tau_1 \mathbin{@} m_1 \to \tau_2 \mathbin{@} m_2) \mathbin{@} m_3 \qquad \Gamma_2 \vdash e_2 : \tau_1 \mathbin{@} m_1}{\Gamma_1 + \Gamma_2 \vdash e_1(e_2) : \tau_2 \mathbin{@} m_2}\ \textsc{App} \qquad \frac{\Gamma_1 \vdash e_1 : \mathbb{1} \mathbin{@} m_1 \qquad \Gamma_2 \vdash e_2 : \tau \mathbin{@} m_2}{\Gamma_1 + \Gamma_2 \vdash (e_1; e_2) : \tau \mathbin{@} m_2}\ \textsc{Seq}$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \mathbin{@} m_1 \qquad \Gamma_2, x : \tau_1 \mathbin{@} m_1 \vdash e_2 : \tau_2 \mathbin{@} m_2}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2 : \tau_2 \mathbin{@} m_2}\ \textsc{Let}$$

$$\frac{\begin{array}{c} \Gamma_1 \vdash e_1 : \tau_1 \mathbin{@} (l, \mathbf{many}, u', p', c') \\ \Gamma_2, x : \tau_1 \mathbin{@} (\mathbf{local}, \mathbf{many}, \mathbf{aliased}, p', c') \vdash e_2 : \tau_2 \mathbin{@} (\mathbf{global}, o, u, p, c) \\ \Gamma_3, x : \tau_1 \mathbin{@} (l, \mathbf{many}, u', p', c'), y : \tau_2 \mathbin{@} (\mathbf{global}, o, u, p, c) \vdash e_3 : \tau_3 \mathbin{@} m \end{array}}{\Gamma_1 + \Gamma_2 + \Gamma_3 \vdash \mathsf{borrow}\ x := e_1\ \mathsf{for}\ y := e_2\ \mathsf{in}\ e_3 : \tau_3 \mathbin{@} m}\ \textsc{Borrow}$$

$$\frac{\Gamma, \text{🔒}_{(\mathbf{global},o,\mathbf{portable})} \vdash e : \tau_1 \mathbin{@} m_1}{\Gamma \vdash \mathsf{fork}(e) : \mathbb{1} \mathbin{@} m_2}\ \textsc{Fork} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \mathbin{@} m \qquad \Gamma_2 \vdash e_2 : \tau_2 \mathbin{@} m}{\Gamma_1 + \Gamma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2 \mathbin{@} m}\ \textsc{Pair}$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \times \tau_2 \mathbin{@} m_1 \qquad \Gamma_2, y : \tau_2 \mathbin{@} m_1, x : \tau_1 \mathbin{@} m_1 \vdash e_2 : \tau_3 \mathbin{@} m_2}{\Gamma_1 + \Gamma_2 \vdash \mathsf{unpair}\ e_1\ \mathsf{as}\ (x, y)\ \mathsf{in}\ e_2 : \tau_3 \mathbin{@} m_2}\ \textsc{Unpair}$$

$$\frac{\Gamma \vdash e : \tau_1 \mathbin{@} m}{\Gamma \vdash \mathsf{inl}(e) : \tau_1 + \tau_2 \mathbin{@} m}\ \textsc{Inl} \qquad \frac{\Gamma \vdash e : \tau_2 \mathbin{@} m}{\Gamma \vdash \mathsf{inr}(e) : \tau_1 + \tau_2 \mathbin{@} m}\ \textsc{Inr}$$

$$\frac{\begin{array}{c} \Gamma_1 \vdash e : \tau_1 + \tau_2 \mathbin{@} m_1 \\ \Gamma_2, x : \tau_1 \mathbin{@} m_1 \vdash e_1 : \tau_3 \mathbin{@} m_2 \\ \Gamma_2, x : \tau_2 \mathbin{@} m_1 \vdash e_2 : \tau_3 \mathbin{@} m_2 \end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathsf{case}\ e\ \{\mathsf{inl}\ x \to e_1; \mathsf{inr}\ x \to e_2\} : \tau_3 \mathbin{@} m_2}\ \textsc{Case} \qquad \frac{\begin{array}{c} \Gamma_1 \vdash e : \mathbb{B} \mathbin{@} m_1 \\ \Gamma_2 \vdash e_1 : \tau \mathbin{@} m_2 \\ \Gamma_2 \vdash e_2 : \tau \mathbin{@} m_2 \end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau \mathbin{@} m_2}\ \textsc{If}$$

$$\frac{\Gamma \vdash e : \tau \mathbin{@} (\mathbf{global}, o, u, p, c)}{\Gamma \vdash e : \mathsf{region}(e) \mathbin{@} (\mathbf{global}, o, u, p, c)}\ \textsc{Region} \qquad \frac{\begin{array}{c} \Gamma_1 \vdash e_1 : \tau_1 \mathbin{@} m \qquad \Gamma_2 \vdash e_2 : \tau_2 \mathbin{@} m \\ \mathit{binopTyped}(\oplus, \tau_1, \tau_2, \tau_3) \end{array}}{\Gamma_1 + \Gamma_2 \vdash e_1 \oplus e_2 : \tau_3 \mathbin{@} m}\ \textsc{BinOp}$$

$$\begin{array}{c} \textsc{UnOp} \\ \dfrac{\begin{array}{c} \Gamma \vdash e : \tau_1 \mathbin{@} m \\ \mathit{unopTyped}(\oplus, \tau_1, \tau_2) \end{array}}{\Gamma \vdash \oplus(e) : \tau_2 \mathbin{@} m} \end{array} \qquad \frac{\begin{array}{c} \Gamma_1 \geq \Gamma_2 \\ m_1 \leq m_2 \\ \Gamma_1 \vdash e : \tau \mathbin{@} m_1 \end{array}}{\Gamma_2 \vdash e : \tau \mathbin{@} m_2}\ \textsc{Sub} \qquad \begin{array}{c} \textsc{Box} \\ \dfrac{\Gamma \vdash e : \tau \mathbin{@} \eta(m)}{\Gamma \vdash \mathsf{box}(e) : \Box^{\eta} \tau \mathbin{@} m} \end{array} \qquad \begin{array}{c} \textsc{Unbox} \\ \dfrac{\Gamma \vdash e : \Box^{\eta} \tau \mathbin{@} m}{\Gamma \vdash \mathsf{unbox}(e) : \tau \mathbin{@} \eta(m)} \end{array}$$

Fig. 10. Typing Rules

$$\textsc{NaAlloc}\quad \frac{\Gamma \vdash e : \tau \mathrel{@} (l, \textbf{many}, u, p, \textbf{uncontended})}{\Gamma \vdash \mathsf{alloc}^{l}(e) : \mathsf{ref}_{p}(\tau) \mathrel{@} (l, o, u', p, c)}$$

$$\textsc{NaLoad}\quad \frac{\Gamma \vdash e : \mathsf{ref}_{p}(\tau) \mathrel{@} (l, o', u', p', c) \qquad c \neq \textbf{contended}}{\Gamma \vdash {!}^{\mathrm{NA}} e : \tau \mathrel{@} (l, o, \textbf{aliased}, p, c)}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash e_1 : \mathsf{ref}_{p}(\tau) \mathrel{@} (l', o', u', p', \textbf{uncontended}) \\ \Gamma_2 \vdash e_2 : \tau \mathrel{@} (\textbf{global}, \textbf{many}, u, p, \textbf{uncontended})\end{array}}{\Gamma_1 + \Gamma_2 \vdash e_1 \leftarrow^{\mathrm{NA}} e_2 : \mathbb{1} \mathrel{@} m_2}\ \textsc{NaStore}$$

$$\textsc{AtAlloc}\quad \frac{\Gamma \vdash e : \tau \mathrel{@} (\textbf{global}, \textbf{many}, u, \textbf{portable}, c)}{\Gamma \vdash \mathsf{alloc}^{\textbf{global}}(e) : \mathsf{atomic}(\tau) \mathrel{@} (l, o, u', p, c')}$$

$$\textsc{AtLoad}\quad \frac{\Gamma \vdash e : \mathsf{atomic}(\tau) \mathrel{@} m}{\Gamma \vdash {!}^{\mathrm{AT}} e : \tau \mathrel{@} (l, o, \textbf{aliased}, p, \textbf{contended})}$$

$$\frac{\Gamma_1 \vdash e_1 : \mathsf{atomic}(\tau) \mathrel{@} m_1 \qquad \Gamma_2 \vdash e_2 : \tau \mathrel{@} (\textbf{global}, \textbf{many}, u, \textbf{portable}, c)}{\Gamma_1 + \Gamma_2 \vdash e_1 \leftarrow^{\mathrm{AT}} e_2 : \mathbb{1} \mathrel{@} m_2}\ \textsc{AtStore}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash e_1 : \mathsf{atomic}(\tau) \mathrel{@} m_1 \\ \Gamma_2 \vdash e_2 : \tau \mathrel{@} m_2 \qquad \Gamma_3 \vdash e_3 : \tau \mathrel{@} (\textbf{global}, \textbf{many}, u, \textbf{portable}, c) \qquad \mathit{typeCmpSafe}(\tau)\end{array}}{\Gamma_1 + \Gamma_2 + \Gamma_3 \vdash \mathsf{cmpXchg}(e_1, e_2, e_3) : \tau \times \mathbb{B} \mathrel{@} (l, o, \textbf{aliased}, p, \textbf{contended})}\ \textsc{CmpXchg}$$

$$\frac{\Gamma_1 \vdash e_1 : \mathsf{atomic}(\tau) \mathrel{@} m \qquad \Gamma_2 \vdash e_2 : \tau \mathrel{@} (\textbf{global}, \textbf{many}, u, \textbf{portable}, c)}{\Gamma_1 + \Gamma_2 \vdash \mathsf{xchg}(e_1, e_2) : \tau \mathrel{@} (l, o, \textbf{aliased}, p, \textbf{uncontended})}\ \textsc{Xchg}$$

$$\frac{\Gamma_1 \vdash e_1 : \mathsf{atomic}(\mathbb{Z}) \mathrel{@} m_1 \qquad \Gamma_2 \vdash e_2 : \mathbb{Z} \mathrel{@} m_2}{\Gamma_1 + \Gamma_2 \vdash \mathsf{faa}(e_1, e_2) : \mathbb{Z} \mathrel{@} m_3}\ \textsc{Faa}$$

Fig. 11. Typing Rules for the Higher Order Store

With fractional invariants in mind, we can now more precisely specify the type of $\varepsilon_{\mathsf{mut}}$ and $\varepsilon_{\mathsf{ro}}$: rather than tracking references directly, we track the set of fractional token names that are currently available, together with the portability mode of the associated reference. As such, $\varepsilon_{\mathsf{mut}}$ and $\varepsilon_{\mathsf{ro}}$ are sets of portability mode and token name pairs (p, γ). Similarly, Δ will in part contain the fractional token names of temporarily owned references, together with an abstract notion of non-reference locals.

E LOGICAL RELATION

Figures 12 and 13 (almost) contain the full definition of the logical relation. The full list of Core Conditions of the Logical Relation, *i.e.,* the conditions on semantic types, is as follows:

Definition E.1 (Core Conditions of the Logical Relation)**.**

(1) $\Delta' \supseteq \Delta \implies \llbracket \tau \rrbracket_m^{\varepsilon_{\mathsf{mut}},\varepsilon_{\mathsf{ro}},\Delta}(v) \mathrel{-\!\!*} \llbracket \tau \rrbracket_m^{\varepsilon_{\mathsf{mut}},\varepsilon_{\mathsf{ro}},\Delta'}(v)$

(2) if $m.l = \textbf{global}$ then $\llbracket \tau \rrbracket_m^{\varepsilon_{\mathsf{mut}},\varepsilon_{\mathsf{ro}},\Delta}(v) \mathrel{-\!\!*} \llbracket \tau \rrbracket_m^{\varepsilon_{\mathsf{mut}},\varepsilon_{\mathsf{ro}},\Delta'}(v)$

$$\boxed{\llbracket \tau \rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\boldsymbol{m}} : \mathrm{Value} \longrightarrow \mathit{iProp}}$$

$$\begin{array}{lll}
\llbracket \mathbb{1} \rrbracket^{-,-,-}_{-}(v) & \triangleq & v = () \\
\llbracket \mathbb{B} \rrbracket^{-,-,-}_{-}(v) & \triangleq & \exists b.\ v = b \\
\llbracket \mathbb{Z} \rrbracket^{-,-,-}_{-}(v) & \triangleq & \exists z.\ v = z \\
\llbracket \tau_1 + \tau_2 \rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\boldsymbol{m}}(v) & \triangleq & (\exists v_1.\ v = \mathsf{inl}(v_1) * \llbracket \tau_1 \rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\boldsymbol{m}}(v_1)) \lor \\
 & & (\exists v_2.\ v = \mathsf{inr}(v_2) * \llbracket \tau_2 \rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\boldsymbol{m}}(v_2)) \\
\llbracket \tau_1 \times \tau_2 \rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\boldsymbol{m}}(v) & \triangleq & \exists v_1\ v_2.\ v = (v_1, v_2) * \llbracket \tau_1 \rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\boldsymbol{m}}(v_1) * \llbracket \tau_2 \rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\boldsymbol{m}}(v_2) \\
\llbracket \Box^{\eta} \tau \rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\boldsymbol{m}}(v) & \triangleq & \llbracket \tau \rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\eta(\boldsymbol{m})}(v)
\end{array}$$

$$\begin{array}{l}
\llbracket \tau_1 \mathbin{@} m_1 \longrightarrow \tau_2 \mathbin{@} m_2 \rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\boldsymbol{m}}(v) \quad \triangleq \\
\quad v = \lambda \ldots\ * \forall \pi\ \varepsilon'_{\mathrm{mut}}\ \varepsilon'_{\mathrm{ro}}\ \Delta'\ q.\ (\varepsilon'_{\mathrm{mut}}, \varepsilon'_{\mathrm{ro}}) \sqsupseteq^{\boldsymbol{m}.p} (\varepsilon_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}}) \longrightarrow \Delta' \sqsupseteq^{\boldsymbol{m}.l,\boldsymbol{m}.p} \Delta \longrightarrow \Box^{\boldsymbol{m}.o} \forall n\ v_1. \\
\quad (\llbracket \tau_1 \rrbracket^{\varepsilon'_{\mathrm{mut}},\varepsilon'_{\mathrm{ro}},\Delta'}_{m_1}(v_1) * \mathcal{L}(\pi, n, \varepsilon'_{\mathrm{mut}}, \varepsilon'_{\mathrm{ro}}, \Delta') * \mathcal{M}\textsc{em}(\varepsilon'_{\mathrm{mut}}, \varepsilon'_{\mathrm{ro}}, q)) \mathrel{-\!\!*} \mathcal{E}\llbracket \tau_2 \rrbracket^{\varepsilon'_{\mathrm{mut}},\varepsilon'_{\mathrm{ro}},\Delta'}_{\pi,n,q,m_2}(v(v_1))
\end{array}$$

$$\llbracket \mathsf{ref}_p(\tau) \rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\boldsymbol{m}}(v) \quad \triangleq \quad \exists a.\ v = a * p \leq \boldsymbol{m}.p\ *$$

$$\begin{cases}
\top & \boldsymbol{m}.c = \mathbf{contended} \\
\phi_{\mathbf{H}}(p, \ell)(\mathsf{filter}(p, \varepsilon_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}}), 1) & \boldsymbol{m}.c = \mathbf{uncontended} \land a = \ell \land \\
 & \boldsymbol{m}.u = \mathbf{unique} \\
\exists \gamma.\ (p, \gamma) \in \varepsilon_{\mathrm{mut}} * \mathsf{FrInv}^{\mathcal{N}_{\mathrm{log}}.\ell,\gamma}\ (\phi_{\mathbf{H}}(p, \ell)) & \boldsymbol{m}.c = \mathbf{uncontended} \land a = \ell \land \\
 & \boldsymbol{m}.u = \mathbf{aliased} \land \boldsymbol{m}.l = \mathbf{global} \\
\exists \gamma.\ (p, \gamma) \in \varepsilon_{\mathrm{mut}} \cup \Delta\ * & \boldsymbol{m}.c = \mathbf{uncontended} \land a = \ell \land \\
\mathsf{FrInv}^{\mathcal{N}_{\mathrm{log}}.\ell,\gamma}\ (\phi_{\mathbf{H}}(p, \ell)) & \boldsymbol{m}.u = \mathbf{aliased} \land \boldsymbol{m}.l = \mathbf{local} \\
\exists \gamma.\ (\mathbf{portable}, \gamma) \in \varepsilon_{\mathrm{mut}} \cup \Delta \cup \varepsilon_{\mathrm{ro}}\ * & \boldsymbol{m}.c = \mathbf{shared} \land a = \ell \land \\
\mathsf{FrInv}^{\mathcal{N}_{\mathrm{log}}.\ell,\gamma}\ (\phi_{\mathbf{H}}(\mathbf{portable}, \ell))\ * & \boldsymbol{m}.u = \mathbf{aliased} \\
\exists \Delta'\ \gamma.\ \Delta' \subseteq \Delta * (p, \gamma) \in \Delta\ * & \boldsymbol{m}.c \neq \mathbf{contended} \land a = (\pi, n) \\
\mathsf{FrInv}^{\mathcal{N}_{\mathrm{log}}.(\pi,n),\gamma}\ (\phi_{\mathbf{S}}(\Delta', p, \pi, n)) & \boldsymbol{m}.l = \mathbf{local}
\end{cases}$$

$$\llbracket \mathsf{atomic}(\tau) \rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\boldsymbol{m}}(v) \quad \triangleq \quad \exists \ell.\ v = \ell\ *$$

$$\begin{cases}
\phi_{\mathbf{At}}(\ell) & \boldsymbol{m}.u = \mathbf{unique} \\
\boxed{\phi_{\mathbf{At}}(\ell)}^{\mathcal{N}_{\mathrm{at}}.\ell} & \boldsymbol{m}.u = \mathbf{aliased} \land \boldsymbol{m}.l = \mathbf{global} \\
\boxed{\phi_{\mathbf{At}}(\ell)}^{\mathcal{N}_{\mathrm{at}}.\ell} \lor \ \exists \gamma.\ (\mathbf{portable}, \gamma) \in \Delta\ * & \boldsymbol{m}.u = \mathbf{aliased} \land \boldsymbol{m}.l = \mathbf{local} \\
\qquad\qquad\quad \mathsf{CInv}^{\mathcal{N}_{\mathrm{at}}.\ell,\gamma}\ (\phi_{\mathbf{At}}(\ell)) &
\end{cases}$$

where

$$\begin{array}{lll}
\phi_{\mathbf{H}}(p, \ell) & \triangleq & \lambda(\varepsilon_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}})\ q.\ \exists v.\ \ell \mapsto^{q} v * \llbracket \tau \rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\emptyset}_{(\mathbf{global},\mathbf{many},\mathbf{aliased},p,\mathbf{uncontended})}(v) \\
\phi_{\mathbf{S}}(\Delta, p, \pi, n) & \triangleq & \lambda(\varepsilon_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}})\ q.\ \exists v.\ n \Mapsto^{q}_{\pi} v * \llbracket \tau \rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{(\mathbf{local},\mathbf{many},\mathbf{aliased},p,\mathbf{uncontended})}(v) \\
\phi_{\mathbf{At}}(\ell) & \triangleq & \exists v.\ \ell \mapsto v * \llbracket \tau \rrbracket^{\emptyset,\emptyset,\emptyset}_{(\mathbf{global},\mathbf{many},\mathbf{aliased},\mathbf{portable},\mathbf{contended})}(v) \\
\mathsf{filter}(p, \varepsilon_{\mathrm{mut}}) & \triangleq & \begin{cases} (\mathsf{filter}_{\mathsf{p}}(\varepsilon_{\mathrm{mut}}), \emptyset) & p = \mathbf{portable} \\ (\varepsilon_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}}) & p = \mathbf{nonportable} \end{cases}
\end{array}$$

Fig. 12. Value Relation

$$\boxed{\mathcal{E}\llbracket\tau\rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\pi,n,q,\boldsymbol{m}} : \text{Expression} \rightarrow \mathit{iProp}}$$

$$\begin{array}{lcl}
\mathcal{E}\llbracket\tau\rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{\pi,n,q,\boldsymbol{m}}(e) & \triangleq & \mathrm{wp}\ e \left\{ v.\ \begin{array}{l} \exists n'\ \Delta'\ \varepsilon'_{\mathrm{mut}}.\ n \le n' \wedge \Delta \subseteq \Delta' \wedge \varepsilon_{\mathrm{mut}} \subseteq \varepsilon'_{\mathrm{mut}} \\ * \llbracket\tau\rrbracket^{\varepsilon'_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta'}_{\boldsymbol{m}}(v) \\ * \mathcal{L}(\pi, n', \varepsilon'_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}}, \Delta') \\ * \mathcal{M}\textsc{em}(\varepsilon'_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}}, q) \\ * \mathrm{collectFrames}(n, n', \pi, \Delta, \Delta') \end{array} \right\}_{\pi} \\
\mathcal{L}(\pi, n, \varepsilon_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}}, \Delta) & \triangleq & \pi \hookrightarrow n * \mathop{\Large *}_{x\in\Delta} [\mathsf{FrTok} : x.\gamma : (\varepsilon_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}})]_1 * \textit{view resource based on } x \\
\mathcal{M}\textsc{em}(\varepsilon_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}}, q) & \triangleq & (\mathop{\Large *}_{(p,\gamma)\in\varepsilon_{\mathrm{mut}}} \exists \varepsilon'_{\mathrm{mut}}\ \varepsilon'_{\mathrm{ro}}.\ (\varepsilon_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}}) \sqsupseteq (\varepsilon'_{\mathrm{mut}}, \varepsilon'_{\mathrm{ro}}) * \\
& & \qquad [\mathsf{FrTok} : \gamma : (\varepsilon'_{\mathrm{mut}}, \varepsilon'_{\mathrm{ro}})]_1 * \cdots) * \\
& & \mathop{\Large *}_{(p,\gamma)\in\varepsilon_{\mathrm{ro}}} \exists \varepsilon'_{\mathrm{mut}}\ \varepsilon'_{\mathrm{ro}}.\ \varepsilon_{\mathrm{mut}} \cup \varepsilon_{\mathrm{ro}} \supseteq \varepsilon'_{\mathrm{mut}}\varepsilon'_{\mathrm{ro}} * \\
& & \qquad [\mathsf{FrTok} : \gamma : (\varepsilon'_{\mathrm{mut}}, \varepsilon'_{\mathrm{ro}})]_q * \cdots
\end{array}$$

where

$$\begin{array}{lcl}
\mathrm{collectFrames}(n, n', \pi, \Delta, \Delta') & \triangleq & \mathop{\Large *}_{m\in[n,n')} \exists x.\ (x \in \Delta' * x \notin \Delta \vee [\mathsf{FrTok} : x.\gamma : -]_1) * \\
& & \qquad ([\mathsf{FrTok} : x.\gamma : -]_1 \Rrightarrow m \Mapsto_{\pi} -)
\end{array}$$

Fig. 13. Expression Relation and Auxiliary Definitions

(3) if $m.p$ = **portable** and $m.c$ = **contended** then
$\Delta' \supseteq \mathrm{atomics}(\Delta) \implies \llbracket\tau\rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{m}(v) \mathrel{-\!\!*} \llbracket\tau\rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta'}_{m}(v)$
where $\mathrm{atomics}(\Delta')$ is an operation which extracts all those elements of Δ' associated to atomically accessible values.

(4) $(\varepsilon'_{\mathrm{mut}}, \varepsilon'_{\mathrm{ro}}) \sqsupseteq (\varepsilon_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}}) \implies \llbracket\tau\rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{m}(v) \mathrel{-\!\!*} \llbracket\tau\rrbracket^{\varepsilon'_{\mathrm{mut}},\varepsilon'_{\mathrm{ro}},\Delta}_{m}(v)$
where $(\varepsilon'_{\mathrm{mut}}, \varepsilon'_{\mathrm{ro}}) \sqsupseteq (\varepsilon_{\mathrm{mut}}, \varepsilon_{\mathrm{ro}}) \triangleq \varepsilon'_{\mathrm{mut}} \supseteq \varepsilon_{\mathrm{mut}} \wedge \varepsilon'_{\mathrm{mut}} \cup \varepsilon'_{\mathrm{ro}} \supseteq \varepsilon_{\mathrm{ro}}$

(5) if $m.p$ = **portable** and $m.c$ = **uncontended** then
$\llbracket\tau\rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{m}(v) \mathrel{-\!\!*} \llbracket\tau\rrbracket^{\mathrm{filter_p}(\varepsilon_{\mathrm{mut}}),\mathrm{filter_p}(\varepsilon'_{\mathrm{ro}}),\Delta}_{m}(v)$ where $\mathrm{filter_p}$ is an operation which extracts all those elements of $\varepsilon_{\mathrm{mut}}$ and $\varepsilon_{\mathrm{ro}}$ associated to **portable** references

(6) if $m.p$ = **portable** and $m.c$ = **contended** then
$\llbracket\tau\rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{m}(v) \mathrel{-\!\!*} \llbracket\tau\rrbracket^{\varepsilon'_{\mathrm{mut}},\varepsilon'_{\mathrm{ro}},\Delta}_{m}(v)$

(7) if $c \le$ **shared** then
$\llbracket\tau\rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{(\mathbf{global},\mathbf{many},\mathbf{aliased},\mathbf{portable},c)}(v) \mathrel{-\!\!*} \llbracket\tau\rrbracket^{\emptyset,\varepsilon_{\mathrm{mut}}\cup\varepsilon_{\mathrm{ro}},\Delta}_{(\mathbf{global},\mathbf{many},\mathbf{aliased},\mathbf{portable},\mathbf{shared})}(v)$

(8) if $m.o$ = **many** and $m.u$ = **aliased** then $\mathrm{Persistent}(\llbracket\tau\rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{m}(v))$

(9) *The borrow condition, which is here omitted, states that validity can temporarily be turned* **local** *and* **aliased** *by extending* Δ

(10) if $m \le m'$ then $\cdots * \llbracket\tau\rrbracket^{\varepsilon_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{m}(v) \mathrel{\Rrightarrow\!\!*}_{\top} \exists \varepsilon'_{\mathrm{mut}}.\ \varepsilon_{\mathrm{mut}} \subseteq \varepsilon'_{\mathrm{mut}} * \cdots * \llbracket\tau\rrbracket^{\varepsilon'_{\mathrm{mut}},\varepsilon_{\mathrm{ro}},\Delta}_{m'}(v)$

To help explain these conditions, we restate them in words:

(1) The set of locals can always grow.

(2) If the mode is **global**, validity does not depend on any locals.

(3) If the mode is **portable** and **contended**, validity does not depend on *non-atomic* locals. Here, $\mathrm{atomics}(\Delta)$ is an operation which extracts all those elements of Δ associated to atomically accessible values.

(4) Enlarging the sets of accessible mutable and immutable references, or allowing mutable access to previously immutable references, does not invalidate any existing values.

(5) If the mode is **portable** and **uncontended**, validity does not depend on **nonportable** references. Here, $\mathrm{filter}_\mathrm{p}$ is an operation which extracts all those elements of $\varepsilon_{\mathrm{mut}}$ associated to **portable** references.
(6) If the mode is **portable** and **contended**, validity does not depend on any references.
(7) An **uncontended** mode can be turned **shared** by moving all mutable accessible references to the immutable set of accessible references.
(8) If the mode is **many** and **aliased**, validity is persistent, which means it can be freely duplicated.
(9) A condition used for turning **unique** values **aliased**, and then back to **unique**.
(10) Validity is preserved across mode weakening. Here, we omit some of the auxiliary ghost resources allocated by the lemma.

F CAPSULE API

The Capsule API is implemented as follows:

```
module Key = struct
  type 'k t = unit
end

let create _ = ()

module Data = struct
  type ('a, 'k) t = 'a

  let create f = Obj.magic (f ())
  let map key f v = (key, Obj.magic (f (Obj.magic v)))
  let extract key f v = (key, f (Obj.magic v))
  let both v w = Obj.magic (v, w)

  let map_shared key f v = Obj.magic (f (Obj.magic v))
  let extract_shared key f v = f (Obj.magic v)
  let destroy key v = Obj.magic v
end
```