Oxidizing OCaml with Modal Memory Management

ANTON LORENZEN, University of Edinburgh, United Kingdom
LEO WHITE, Jane Street, United Kingdom
STEPHEN DOLAN, Jane Street, United Kingdom
RICHARD A. EISENBERG, Jane Street, USA
SAM LINDLEY, University of Edinburgh, United Kingdom

Programmers can often improve the performance of their programs by reducing heap allocations: either by allocating on the stack or reusing existing memory in-place. However, without safety guarantees, these optimizations can easily lead to use-after-free errors and even type unsoundness. In this paper, we present a design based on *modes* which allows programmers to safely reduce allocations by using stack allocation and in-place updates of immutable structures. We focus on three mode axes: affinity, uniqueness and locality. Modes are fully backwards compatible with existing OCaml code and can be completely inferred. Our work makes manual memory management in OCaml safe and convenient and charts a path towards bringing the benefits of Rust to OCaml.

CCS Concepts: • **Theory of computation** → **Modal and temporal logics**; *Linear logic*; • **Software and its engineering** → *Garbage collection.*

Additional Key Words and Phrases: Modal Types, Type Qualifiers, Stack Allocation, Uniqueness Types

ACM Reference Format:
Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. *Proc. ACM Program. Lang.* 8, ICFP, Article 253 (August 2024), 30 pages. https://doi.org/10.1145/3674642

1 Introduction

Functional programming languages such as OCaml typically avoid troubling the user over memory management by providing automatic garbage collection. Automatic GC makes it practical to program with immutable data structures, simplifying reasoning and making it easier to write correct software. Yet GC is not magic. In return for its convenience, we pay a price on two fronts:

- **Latency:** In OCaml, *any* allocation can trigger a garbage collection cycle, which can lead to unexpected pauses. To write code free of such pauses, programmers must avoid heap allocation—usually by tricks such as pre-allocating an array used as an unsafe arena allocator.
- **Cache-Locality:** Garbage collectors cannot reuse memory immediately, increasing the number of cache misses in the program. For instance, a bump pointer allocator will not reuse an address until every other address in the young generation has been used.

Authors' Contact Information: Anton Lorenzen, University of Edinburgh, Edinburgh, United Kingdom, anton.lorenzen@ed.ac.uk; Leo White, Jane Street, London, United Kingdom, lwhite@janestreet.com; Stephen Dolan, Jane Street, London, United Kingdom, sdolan@janestreet.com; Richard A. Eisenberg, Jane Street, NYC, USA, reisenberg@janestreet.com; Sam Lindley, University of Edinburgh, Edinburgh, United Kingdom, Sam.Lindley@ed.ac.uk.

ACM 2475-1421/2024/8-ART253
https://doi.org/10.1145/3674642

To give you a flavor of the problem we are trying to solve, consider the following example:

```
let convert_currency exchange_rate prices =
  List.map (fun {num; denom} -> Frac.simplify
    { num = num * exchange_rate.num;
      denom = denom * exchange_rate.denom}) prices
```

We convert a list of prices (stored as fractions of ints) to a new currency. There are many sources of allocations in this code: we allocate a new record per price, the `Frac.simplify` function will allocate one more, and the `List.map` function will allocate a new list to store the results of the mapping. To avoid pauses, a performance-conscious OCaml programmer may choose to avoid functional data structures altogether and use mutable records and arrays instead. However, even with this change, the function will still perform heap allocations! The closure passed to the `map` function is itself allocated on the heap and could thus lead to a GC pause. To avoid such a pause, the programmer must only use closures that have no free variables—and will have to provide a `map` function that passes the `exchange_rate` as an argument to the closure, turning the above into:

```
let convert_currency exchange_rate prices =
  Array.map_inplace1 (fun exchange_rate price ->
      let price_num = price.num * exchange_rate.num in
      let price_denom = price.denom * exchange_rate.denom in
      let gcd = Frac.gcd price_num price_denom in
      price.num <- Int.div price_num gcd;
      price.denom <- Int.div price_denom gcd)
    exchange_rate prices
(* val Array.map_inplace1 : ('b -> 'a -> 'a) -> 'b -> 'a array -> unit *)
```

Finally, this function is free from GC-induced pauses. But at what cost?

In their quest for pause-free code, the programmer has had to give up not just immutable data structures, but also much of the convenience of functional programming. While this example is simplified, it is not just theoretical: functions such as `Array.map_inplace1` used to exist in our production code base!

This paper shows how to mix the safety and convenience of high-level functional programming with the ability to avoid garbage-collected heap allocations. Instead of strenuously avoiding heap allocation through imperative features, we can make the initial, purely functional code pause-free:

(1) If the input list `prices` is unique (and not used after the call to `convert_currency`), we can safely reuse its memory, making `List.map` unobservably destructive.
(2) Since we know that `List.map` does not save the closure passed to it anywhere, we can allocate the closure on the stack and de-allocate it as soon as `List.map` is complete, without the need for a garbage-collection pass.

We introduce *modes* to the OCaml type system to safely allow *stack allocation* and *memory reuse*. These techniques make the purely functional code above pause-free, by allowing us to state that the function passed to `List.map` is `local` and that the input list `prices` is `unique`. The designs of the two features mesh together well and form a cohesive whole; we are thus presenting them together, though either could be implemented separately. For both features, mode annotations can be fully inferred and all existing OCaml programs continue to be accepted in our implementation.

We have implemented the stack allocation idea in a branch of the OCaml compiler, to general acclaim among our programmer colleagues using the feature. We say more about this implementation in Section 7 and it is available as an artifact.

We offer the following contributions:

- We present the design of an OCaml extension enabling stack allocation, memory reuse and borrowing, based on three *mode* axes: affinity, uniqueness and locality (Sections 2 and 6).
- We describe a *mode calculus* modelling these features using type qualifiers, and show that we can *infer* all modes, including those of closures (Section 3).
- We prove soundness of our calculus against a *usage-aware store semantics* (Section 4)
- We relate our mode calculus to a *graded modal calculus* (suitably adapted to account for call-by-value side-effects) instantiated with a carefully chosen semiring (Section 5).
- We have implemented our design in a branch of the OCaml compiler (Section 7), and enabled stack allocation support by default. This branch has been successfully employed by hundreds of developers to write production software; adopters have described that these improvements have allowed them to "stay functional" even while writing heap-allocation-free code.

Section 8 discusses related work and Section 9 discusses future work. The appendix can be found in the supplementary material.

2 Programming with Modes

In this section we illustrate our design through examples written using our extension to OCaml. A summary of our definitions and terminology is given in Figure 1 on page 8.

2.1 Uniqueness

Immutable data structures are central to many functional programming languages. They allow programmers to reason about their code locally, without worrying about mutation introduced by other parts of the program. However, immutable data structures can be expensive to use, since any small update requires allocation and subsequent reclamation.

However, there is an escape hatch: if a value is guaranteed by the type system to have only one reference—what we call *unique*—it can unobservably be mutated in-place. For example, if the list given to the `reverse` function is unique, we can write an in-place reversal function as follows:

```
type 'a list = Nil | Cons of { hd : 'a; tl : 'a list }

let rec rev_append xs acc =
  match xs with
  | Nil -> acc
  | Cons x_xs -> rev_append x_xs.tl (Cons { overwrite x_xs with tl = acc })

let reverse xs = rev_append xs Nil
```

The `overwrite` prefix in the functional update (`Cons { overwrite x_xs with tl = acc }`) instructs the compiler to *reuse* the memory associated with the existing constructor, instead of allocating a fresh constructor.

As a first modal axis, we will thus consider *uniqueness*, giving us the ability to ensure that in-place reuse is safe. We distinguish `unique` values from ordinary (`aliased`) values by annotating them with the `unique` mode:

```
type ('a, 'b) pair = { fst : 'a; snd : 'b }
let update_snd : ((('a, 'b) pair) @ unique) -> 'c -> ('a, 'c) pair =
  fun pair c -> { overwrite pair with snd = c }
```

Given a unique `pair`, we can use projections to extract unique sub-parts of `pair` (assuming that we do not use `pair` after these assignments):

```
let process_parts (pair @ unique) =
  let x @ unique = pair.fst in
  let y @ unique = pair.snd in ...
```

For this example to be accepted (as it is in our prototype), we must carefully track the fact that `pair.fst` is an occurrence only of the `fst` field; the use of `pair.snd` on the following line is considered separate.

If, on the other hand, we tried to use `pair` again or tried to use `fst` twice as unique, we would get a mode error:

```
let process_parts_wrong (pair @ unique) =
  let x @ unique = pair.fst in
  let y @ unique = pair.fst in ...
(*                 ^^^^^^^^                                              *)
(*  Error: This value is used here, but it has already been used as unique:  *)
(*  Line 2, characters 19-26:                                             *)
(*  2 |   let x @ unique = pair.fst in                                     *)
(*                        ^^^^^^^^                                        *)
```

Uniqueness is considered a *deep* property by default; that is, we expect components of a unique value also to be unique. Individual types can override this default; see Section 2.3. Because of this assumption of depth, it is disallowed to ascribe uniqueness to values that are only partly unique:

```
let f (ys @ unique : int list) =
  let zs1 @ unique : int list = Cons {hd = 0; tl = ys} in
  let zs2 @ unique : int list = Cons {hd = 1; tl = ys} in ...
(*                                                ^^                    *)
(*  Error: This value is used here, but it has already been used as unique   *)
```

The assumption of depth is useful in practice. Returning to our `reverse` example, the uniqueness of `xs` implies the same for its tail `x_xs.tl` extracted in the match. Soundness requires that our in-place update work only on unique values, and this propagates to the argument:

```
val reverse : 'a list @ unique -> 'a list @ unique
```

However, while uniqueness is quite useful, its restrictions make some programming patterns (such as persistence or memoization) impossible. For this reason, it can be desirable to allocate unique values, use several in-place updating functions and then forget their uniqueness (making further in-place updates impossible). We thus have a *sub-moding* relation: any `unique` value is also a `aliased` value, giving us `unique < aliased`. We can thus write code like this:

```
let process_parts_aliased (pair @ unique) =
  let x = pair.fst in
  let y = pair.fst in  (* a repeated occurrence *)
  ...
```

No error arises here, because we just treat the `unique` `pair` as `aliased`. We still must be careful: any `aliased` use of a variable makes *all* uses of that variable `aliased`:

```
let process_parts_partial_aliasing (pair @ unique) =
  let x = pair.fst in
  let y = pair.snd in
  process_parts pair;
  ... x ...
(*    ^                                                                 *)
(*  Error: This value is used here, but it has already been used as unique   *)
```

This example demonstrates that our occurrence analysis tracks `x` as a component of `pair`; after a unique usage of `pair`, a use of `x` causes an error. (Because of this alias tracking, we do *not* error on the unique use of `pair` itself; neither `x` nor `y` has yet been used.)

2.2 Affinity

A uniqueness mode alone is not enough to create a safe uniqueness type system. To see why, consider the following example, where we capture the unique `xs` in the closure `f`:

```
let rejected =
  let xs @ unique : int list = [1;2;3] in
  let f = fun zs -> rev_append xs zs in
  let ys = f [4] in
  let zs = f [5] (* Oh no! zs and ys refer to the same memory! *)
  in ...
```

We have not put any restriction on the closure `f` and can thus invoke it twice. But since the reverse function uses in-place update on unique value `xs`, we end up with `ys` and `zs` both referring to the same memory. Worse, `ys` and `zs` refer have a tail of `[1;2;3]` since the memory underlying `xs` was reversed twice, cancelling out the effect.

Though other languages with support for in-place update solve this problem in different ways—see Section 8—our approach is to create a new mode for *affinity*. An affine value can be used at most `once`, in contrast to `many` times.[1] Only affine closures may close over unique values:

```
... let f @ once = fun zs -> rev_append xs zs in ...
```

Unlike with uniqueness, affinity cannot be forgotten. Uniqueness is a statement about the past (a value has not been aliased); it is safe to forget this detail. In contrast, affinity is a statement about the future (a value cannot be aliased); forgetting it could potentially make memory reuse observable [Marshall et al. 2022]. To prevent this sharing, it is not allowed to use an affine value `many` times, ensuring the affine closure is called only once:

```
... let fs = (f, f) in ...
(*               ^                                      *)
(* Error: This value has mode *once* and has already been used *)
```

Because affinity imposes a restriction, we can always safely assume a value to be affine; doing so restricts how we use the value later, but doing so is always safe. This dovetails well with our sub-moding feature, where `many` < `once`.

One may wonder at this point if we need to introduce a third mode axis for closures that capture affine values. Thankfully, we do not: to ensure that a closure can safely capture a `once` value, it suffices to ensure that it is only used once—and is thus `once` itself.

In the examples above, we have annotated values if they are `unique` or `once`. Such annotations are not necessary: mode inference (see Section 3.7) infers modes for un-annotated variables.

2.3 Modalities

While modes are naturally deep, there are cases where this behaviour is undesirable. For example, when we consider unique lists, we require only the cons cells to be unique to perform in-place update in a function like `reverse`. But our previous definition forces us also to ensure that the list *elements* are unique—a strong and unnecessary restriction. To override this restriction, we can

[1] An affine value is not required to be used. We could support linear values in our language, by adding a modal axis for relevant values which are required to be used at least once.

annotate fields of constructors with a *modality* which adjusts the underlying mode. Modalities are marked with `@@`s.

```
type 'a list_with_aliased_elts =
  Nil | Cons of { hd : 'a @@ aliased; tl : 'a list_with_aliased_elts }
```

The `@@ aliased` annotation here ensures that even if the list itself is unique, its elements may be aliased. Note that such a type must actually be distinct from the `list` type we defined above. An element extracted from a unique `list` is itself `unique` so may be updated in-place, whereas an element extracted from a unique `list_with_aliased_elts` is `aliased`, so may not. On the other hand, we cannot insert aliased elements into a unique `list`, but we can insert aliased elements into a unique `list_with_aliased_elts`.

Not all modes give rise to a corresponding modality. For instance, it would be unsound to have an aliased list which contains a unique value, since we cannot guarantee uniqueness of the inner value anymore. Similarly, we cannot capture affine values in a list at mode `many`. We can create a type that does nothing but wrap a modality. Here are the type declarations for `aliased` and `many`:[2]

```
type 'a aliased = { a : 'a @@ aliased } [@@unboxed]
type 'a many = { m : 'a @@ many } [@@unboxed]
```

The `[@@unboxed]` [Doligez 2016] annotations tell the compiler to omit any runtime allocation and indirection; these types exist only at compile time. Now, instead of writing `list_with_aliased_elts`, we can instead talk about `'a aliased list`: the `aliased` allows us to store aliased elements without affecting the uniqueness (and in-place updateability) of the overall list.

```
(* val graph_nodes : graph -> node aliased list @ unique *)
(* val map_to_aliased : ('a @ unique -> 'b) -> 'a list @ unique -> 'b list *)
let rev_graph_nodes graph =
  let nodes = graph_nodes graph in
  let rev_nodes = reverse nodes in
  map_to_aliased (fun aliased_node -> aliased_node.a) rev_nodes
```

This function first takes a `unique` list of aliased nodes: the list structure itself is unique (the `graph_nodes` function just produced it), but its contents are aliased (because of cycles in the graph). We can reverse the list in place, with `reverse`, which takes a `unique` list (the contents of which are immaterial). Finally, we assume the client of `rev_graph_nodes` just wants a `node list`, so we use `map_to_aliased` to remove the `aliased` type. [3]

2.4 Locality

The third (and last) modal axis we consider in this paper is *locality*. It describes values which cannot leave a region. Our regions are lexical, created around certain expressions in our grammar. For example, the body of a function is a region, as is the definition of a module-level variable. (More details in Section 6.2.) Accordingly, the following code results in an error.

```
let bad () = let xs @ local : int list = [1;2;3] in xs
(*                                        ^^                                  *)
(* Error: This local value escapes its region                                 *)
```

On the other hand, the following code is permitted.

[2] In our concrete syntax, modes can always be distinguished from other grammars; the namespace for modes is thus distinct from all other namespaces.

[3] The `map_to_aliased` function is one of many `map` functions we might like in a system with modes. Avoiding duplication in such functions would be the domain of *mode polymorphism* [Bernardy et al. 2017], which we do not attempt in this paper but expect to eventually incorporate in our implementation; see Section 6.6.

```
let good () = let xs @ local : int list = [1;2;3] in List.length xs
```

Annotating a definition with `local` means that the memory it points to is guaranteed not to be accessible outside its defining region. Not only can we not return `xs` from such a definition, we also cannot store it in a mutable cell.

Like affinity, locality describes a restriction on an operation (escape) to take place in the future. Accordingly, we can arbitrarily assume locality, but never forget it once it is assumed. We thus have `global < local`. Also echoing affinity, we can store global values in local ones via a modality, but not the other way around. Lastly, a closure can capture local values only if it is itself local.

2.5 Stack Allocation

We can use the locality mode to enable sound stack allocation. In the `good` example above, the list `xs` can be allocated on the stack. Stack allocation is particularly useful for function closures, which are usually not captured. For example, consider the `List.iter` function:

```
let rec iter f = function
  | [] -> ()
  | a :: l -> f a; iter f l
```

This function does not capture `f`, which can thus be made local. As such, call sites can allocate their closures on the stack.[4]

2.6 Borrowing

Connecting this locality mode to our uniqueness and affinity modes, we can use locality to implement safe *immutable borrows*, using the Rust terminology. We create aliased references from a unique value, and if the references do not leave their scope, we can safely assume uniqueness of the value again afterwards. For instance, we can define a `borrow` combinator:

```
val borrow : 'a @ unique -> ('a @ local -> 'b) -> ('a * 'b aliased) @ unique
```

The intended semantics of `borrow v f` is to pass the unique value `v` to `f` as a borrowed value and then return `v`, still as a unique value, paired up with the result of `f`. The reason this is sound is that the argument type of `f` is local, so there can be no other reference to `v` once `f` has returned.

We define `borrow` using the `&` syntax from Rust:

```
let borrow x f = let result = f &x in x, { a = result }
```

Though `x` is used twice, the first use is a borrow (meaning that the mode of the argument to `f` is `local`), so the second use can safely be treated as `unique`.

However, we have to be careful when using borrowing with the global modality. As with the `aliased` and `many` wrapper types, we can define a `global` wrapper type:

```
type 'a global = { g : 'a @@ global } [@@unboxed]
```

Now, consider the following sneaky trick.

```
let sneaky : int list @ unique -> (int list * int list aliased) @ unique =
  fun xs ->
    let global_xs @ unique : int list global = { g = xs } in
    let { g = (ys @ unique) }, { a = ys' } =
      borrow global_xs (fun { g = xs' } -> xs') in
    ys, { a = ys' }
(*  Error: ys is aliased; it cannot be treated as unique *)
```

[4] There is a delicate interaction between region placement and the tail-call optimization. After all, a naive design would place the end of a region after an apparent tail call, making the tail-call optimization invalid. We return to this point and explain the design in Section 6.3, though these details are not included in our formalization.

axis	minimum	maximum	legacy	modality
affinity (a)	`many`	`once`	`many`	`many`
uniqueness (u)	`unique`	`aliased`	`aliased`	`aliased`
locality (l)	`global`	`local`	`global`	`global`

A *mode* refers to members of any of the axes above, and also to the triple (a, u, l) of all three axes. We allow mode ascriptions (with `@`) in the following places:

- On types on either side of an arrow:
  ```
  graph @ local -> string @ unique
  ```
  When a modal axis is missing in a function type, we assume the legacy mode for that axis.
- On variable patterns:
  ```
  let f (x @ unique) = ... in ...
  ```
  Omitted modal axes are inferred.

Note that it is meaningless to give a type a mode without an arrow nearby; thus this is an error:

```
type t = string @ aliased
```

A *modality* is a function from a mode triple to a mode triple. We define three:

$$\texttt{aliased}\,(a, u, l) = (a, \texttt{aliased}, l)$$
$$\texttt{many}\,(a, u, l) = (\texttt{many}, u, l)$$
$$\texttt{global}\,(a, u, l) = (a, \texttt{aliased}, \texttt{global})$$

(`global` affects uniqueness as well as locality to soundly support borrowing; see Section 2.6).
These appear in modality ascriptions (with `@@`) on types for record fields:

```
type 'a aliased = { a : 'a @@ aliased }
```

If a record `r` has mode `m` and field `f` has modality `n`, then `r.f` has mode `n(m)`.

Fig. 1. Modes and Modalities

This erroneous code launders a `unique` value `xs` through a borrow, returning it uniquely *and* aliased. This would be disastrous, because later code could `reverse` the `unique` output, observably mutating the contents of the apparently-immutable second return value. The root cause of the problem here is that `borrow v f` is safe only because the argument `v` to `f` can not escape. But since the content of the `global` box can escape, there can be another reference to `v` after `f` returns. In order to ensure borrowed `unique` values remain `local`, the `global` modality incorporates the `aliased` modality. Accordingly, the type definition for `'a global` actually wraps an *aliased* global value, which is why the `unique` annotation on the binding site for `ys` results in a mode error.

3 Mode Calculus

In this section, we introduce a formal calculus of modes sufficient to capture the essence of the examples from Section 2. Our calculus is call-by-value and related to calculi for type qualifiers [Foster et al. 1999], and similar to the linear system proposed by Walker [2005].

The syntax of modes is given by the following grammar:

$$\text{(modes)}\quad \mu ::= (a, u, l)$$

$$\begin{array}{ll} \text{(affinities)} & a ::= \textsc{many} \mid \textsc{once} \\ \text{(uniquenesses)} & u ::= \textsc{unique} \mid \textsc{aliased} \\ \text{(localities)} & l ::= \textsc{global} \mid \textsc{local} \end{array}$$

A mode μ comprises a triple (a, u, l) of an affinity a, a uniqueness u, and a locality l. The following sub-moding relationships hold on the constituents of a mode:

$$\textsc{many} < \textsc{once} \qquad \textsc{unique} < \textsc{aliased} \qquad \textsc{global} < \textsc{local}$$

Modes are ordered pointwise, and when $\mu \leq \mu'$ a term at mode μ can be used at mode μ'. Modes form a lattice, where $\wedge$ gives the greatest lower bound and $\vee$ gives the least upper bound.

3.1 Boxes and Modalities

Systems based on type qualifiers, like Walker [2005], annotate all types with qualifiers. The syntax of types is defined by pretypes $P ::= T \times T \mid T \to T \ldots$ and qualified types $T ::= P \mathbin{@} \mu$, where μ is a type qualifier (in our setting qualifiers are *modes*). The qualifiers inside value types must preserve an ordering relationship: a pair $T_1 \times T_2$ can be qualified by μ only if both $T_1 = P_1 \mathbin{@} \mu_1$ and $T_2 = P_2 \mathbin{@} \mu_2$ have qualifiers $\mu_1, \mu_2 \leq \mu$. The qualifiers inside computation types have no such constraints: $T_1 \to T_2$ can have any qualifier.

In contrast, our system uses qualifiers on computation types, but omits them from value types. Instead, we introduce a box type $\Box^\nu \tau$ to represent the *modality* ν. Modalities act on modes, describing the mode of the box's contents given the mode of the box itself. For instance, the type of a unique pair of an aliased value of type τ_1 and a unique value of type τ_2 is $(\Box^{\mathrm{A}} \tau_1) \times \tau_2$.

We support three modalities A, M and G, giving us three associated box types $\Box^{\mathrm{A}}$, $\Box^{\mathrm{M}}$ and $\Box^{\mathrm{G}}$:

$$\mathrm{A}(a, u, l) = (a, \textsc{aliased}, l) \qquad \mathrm{M}(a, u, l) = (\textsc{many}, u, l) \qquad \mathrm{G}(a, u, l) = (a, \textsc{aliased}, \textsc{global})$$

Note that G affects both uniqueness and locality, which is necessary for borrowing (see Section 2.6).

Omitting qualifiers on value types avoids the need for side conditions on qualifiers in our type formation or introduction rules. That allows us to have a simple story for polymorphism (Section 3.8), unlike most type qualifier systems, which require either bounded polymorphism or qualifier polymorphism.

3.2 Syntax

Contexts are ordered lists of variables, marked with type and mode ($x : \tau \mathbin{@} \mu$) or as unusable ($x : -$):

$$\Gamma ::= \varnothing \mid \Gamma, x : - \mid \Gamma, x : \tau \mathbin{@} \mu$$

Our types consist of the unit type $\mathbb{1}$, sum types $\tau + \tau$ and product types $\tau \times \tau$. Furthermore, we have a function type $\tau \mathbin{@} \mu \to \tau \mathbin{@} \mu$ which stores the modes for the argument and result of the function and a box type $\Box^\nu \tau$ to represent the modality ν. To model in-place update, we also include a type for space credits ♣:

$$\tau ::= \mathbb{1} \mid \tau + \tau \mid \tau \times \tau \mid \Box^\nu \tau \mid \tau \mathbin{@} \mu \to \tau \mathbin{@} \mu \mid \clubsuit$$

Our syntax is largely standard. We introduce modalities with $\mathrm{box}_\nu\, e$ and eliminate them with $\mathrm{unbox}_\nu\, e$. Our elimination form for pairs $\mathrm{let}\,(x, y, z) = e\,\mathrm{in}\,e$ gives us access to the elements y, z of the pair and additionally the space credit x of the allocation itself, which can be reused (if unique) to allocate a new pair with reuse e in (e, e). Finally, we model borrowing by allowing x to be borrowed whilst y is computed. The final computation then has access to both x and y:

$$\begin{aligned} e ::= {} & x \mid () \mid \mathrm{inl}\, e \mid \mathrm{inr}\, e \mid (e, e) \mid \mathrm{box}_\nu\, e \mid \mathrm{unbox}_\nu\, e \mid \lambda x.\, e \mid e\, e \\ & \mid \mathrm{let}\, x = e\,\mathrm{in}\,e \mid \mathrm{let}\,(x, y, z) = e\,\mathrm{in}\,e \mid \mathrm{case}\, e\, \{\, \mathrm{inl}\, x \to e; \mathrm{inr}\, y \to e \,\} \\ & \mid \mathrm{reuse}\, e\,\mathrm{in}\,(e, e) \mid \mathrm{borrow}\, x = e\,\mathrm{for}\, y = e\,\mathrm{in}\,e \end{aligned}$$

We do not include an explicit construct for stack allocation regions, representing them instead as: borrow _ = () for $y = e_1$ in e_2, which already has the effect of running e_1 inside a fresh region.

3.3 Locks for Closures

Closures capture free variables from their environments, but when should a variable of mode (a_1, u_1, l_1) be available in a closure of mode (a_2, u_2, l_2)? We already saw that a `global` closure can only contain `global` variables, and a `many` closure can only contain `many` variables. Thus it must be the case that $a_1 \leq a_2$ and $l_1 \leq l_2$. But we must also ensure that variables are not captured as `unique` by closures that can be invoked multiple times. We do so using the following dagger operation to relate each affinity with its corresponding uniqueness:

$$\textsc{once}^\dagger := \textsc{unique} \qquad\qquad \textsc{many}^\dagger := \textsc{aliased}$$

Then we require that $u_1 \leq a_2^\dagger$. We need not use the uniqueness of the closure u_2, as it has no effect on the variables the closure may capture.

We enforce the latter constraint by applying a *lock* $🔒_\mu$ to the context, which we write as $\Gamma, 🔒_\mu$. Unlike the context containment operation of Walker [2005], our $\Gamma, 🔒_\mu$ is not merely a predicate on the bindings of Γ, but may actually change modes to reflect that bindings have become aliased:

$$\begin{aligned}
\varnothing, 🔒_{(a_2,u_2,l_2)} &= \varnothing \\
\Gamma, x : -, 🔒_{(a_2,u_2,l_2)} &= \Gamma, 🔒_{(a_2,u_2,l_2)}, x : - \\
\Gamma, x : \tau \mathbin{@} (a_1, u_1, l_1), 🔒_{(a_2,u_2,l_2)} &= \begin{cases} \Gamma, 🔒_{(a_2,u_2,l_2)}, x : \tau \mathbin{@} (a_1, u_1 \vee a_2^\dagger, l_1) & \text{if } a_1 \leq a_2 \text{ and } l_1 \leq l_2 \\ \Gamma, 🔒_{(a_2,u_2,l_2)}, x : - & \text{otherwise} \end{cases}
\end{aligned}$$

In our inference rules (outlined in Section 3.7 and detailed in Appendix B), we evaluate the context operation *lazily*, by making $\Gamma, 🔒_\mu$ part of the syntax of contexts (rather than an operation on contexts as it is here) and applying the locks in the variable rule. This adjustment enables us to infer the modes of locks, and consequently infer the modes of closures as well as other values.

3.4 Joining Usages

We use the typing context to track the usage of variables and join contexts to combine usages. Our context consists of a list of variables $x : \tau \mathbin{@} \mu$, containing a mode μ at which x is used, or $x : -$ if it is unused. To combine the usages of variables in two contexts, we use a (partial) join operation $+$, which assumes that the two contexts contain the same variables, in the same order:

$$\begin{aligned}
\varnothing + \varnothing &= \varnothing \\
(\Gamma_1, x : -) + (\Gamma_2, x : -) &= (\Gamma_1 + \Gamma_2), x : - \\
(\Gamma_1, x : \tau \mathbin{@} \mu) + (\Gamma_2, x : -) &= (\Gamma_1 + \Gamma_2), x : \tau \mathbin{@} \mu \\
(\Gamma_1, x : -) + (\Gamma_2, x : \tau \mathbin{@} \mu) &= (\Gamma_1 + \Gamma_2), x : \tau \mathbin{@} \mu \\
(\Gamma_1, x : \tau \mathbin{@} (_, \texttt{aliased}, l)) + (\Gamma_2, x : \tau \mathbin{@} (_, \texttt{aliased}, l)) &= (\Gamma_1 + \Gamma_2), x : \tau \mathbin{@} (\texttt{many}, _, l)
\end{aligned}$$

Contexts are ordered as follows:

$$\begin{aligned}
\Gamma, x : -, \Gamma' &\geq \Gamma, x : \tau \mathbin{@} \mu, \Gamma' \\
\Gamma, x : \tau \mathbin{@} \mu_1, \Gamma' &\geq \Gamma, x : \tau \mathbin{@} \mu_2, \Gamma' \qquad \text{if } \mu_1 \geq \mu_2
\end{aligned}$$

3.5 Typing Rules

We present the typing rules for our calculus in Figure 2. We write _ for a meta-variable that only appears once in a rule (hence its actual name is immaterial). Whenever we have more than one premise, we join the contexts of the premises using the + operation (the only exception is the CASE rule, where both branches use the same context).

$$\frac{}{\Gamma, x : \tau \mathbin{@} \mu, \Gamma' \vdash x : \tau \mathbin{@} \mu}\ \textsc{var} \qquad \frac{\Gamma_1 \geq \Gamma_2 \qquad \mu_1 \leq \mu_2 \qquad \Gamma_1 \vdash e : \tau \mathbin{@} \mu_1}{\Gamma_2 \vdash e : \tau \mathbin{@} \mu_2}\ \textsc{sub}$$

$$\frac{\Gamma, \text{🔒}_{\mu_3}, x : \tau_1 \mathbin{@} \mu_1 \vdash e : \tau_2 \mathbin{@} \mu_2}{\Gamma \vdash \lambda x.\, e : (\tau_1 \mathbin{@} \mu_1 \to \tau_2 \mathbin{@} \mu_2) \mathbin{@} \mu_3}\ \textsc{lam} \qquad \frac{\Gamma_1 \vdash e_1 : (\tau_1 \mathbin{@} \mu_1 \to \tau_2 \mathbin{@} \mu_2) \mathbin{@} \_ \qquad \Gamma_2 \vdash e_2 : \tau_1 \mathbin{@} \mu_1}{\Gamma_1 + \Gamma_2 \vdash e_1\, e_2 : \tau_2 \mathbin{@} \mu_2}\ \textsc{app}$$

$$\frac{}{\Gamma \vdash () : \mathbb{1} \mathbin{@} \mu}\ \textsc{unit} \qquad \frac{\Gamma \vdash e : \tau_1 \mathbin{@} \mu}{\Gamma \vdash \mathrm{inl}\, e : \tau_1 + \tau_2 \mathbin{@} \mu}\ \textsc{inl} \qquad \frac{\Gamma \vdash e : \tau_2 \mathbin{@} \mu}{\Gamma \vdash \mathrm{inr}\, e : \tau_1 + \tau_2 \mathbin{@} \mu}\ \textsc{inr}$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \mathbin{@} \mu \qquad \Gamma_2 \vdash e_2 : \tau_2 \mathbin{@} \mu}{\Gamma_1 + \Gamma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2 \mathbin{@} \mu}\ \textsc{pair} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \times \tau_2 \mathbin{@} \mu_1 \qquad \Gamma_2, x : \clubsuit \mathbin{@} \mu_1, y : \tau_1 \mathbin{@} \mu_1, z : \tau_2 \mathbin{@} \mu_1 \vdash e_2 : \tau_3 \mathbin{@} \mu_2}{\Gamma_1 + \Gamma_2 \vdash \mathrm{let}\,(x, y, z) = e_1\,\mathrm{in}\, e_2 : \tau_3 \mathbin{@} \mu_2}\ \textsc{split}$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \mathbin{@} \mu_1 \qquad \Gamma_2, x : \tau_1 \mathbin{@} \mu_1 \vdash e_2 : \tau_2 \mathbin{@} \mu_2}{\Gamma_1 + \Gamma_2 \vdash \mathrm{let}\, x = e_1\,\mathrm{in}\, e_2 : \tau_2 \mathbin{@} \mu_2}\ \textsc{let} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 + \tau_2 \mathbin{@} \mu_1 \qquad \Gamma_2, x_1 : \tau_1 \mathbin{@} \mu_1 \vdash e_2 : \tau_3 \mathbin{@} \mu_2 \qquad \Gamma_2, x_2 : \tau_2 \mathbin{@} \mu_1 \vdash e_3 : \tau_3 \mathbin{@} \mu_2}{\Gamma_1 + \Gamma_2 \vdash \mathrm{case}\ e_1\, \{\, \mathrm{inl}\, x_1 \to e_2; \mathrm{inr}\, x_2 \to e_3 \,\} : \tau_3 \mathbin{@} \mu_2}\ \textsc{case}$$

$$\frac{\Gamma \vdash e : \tau \mathbin{@} \nu(\mu)}{\Gamma \vdash \mathrm{box}_\nu\, e : \Box^\nu \tau \mathbin{@} \mu}\ \textsc{box} \qquad \frac{\Gamma \vdash e : \Box^\nu \tau \mathbin{@} \mu}{\Gamma \vdash \mathrm{unbox}_\nu\, e : \tau \mathbin{@} \nu(\mu)}\ \textsc{unbox}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash e_1 : \clubsuit \mathbin{@} (\_, \textsc{unique}, \textsc{global}) \\ \Gamma_2 \vdash e_2 : \tau_1 \mathbin{@} (a, u, \textsc{global}) \\ \Gamma_3 \vdash e_3 : \tau_2 \mathbin{@} (a, u, \textsc{global})\end{array}}{\Gamma_1 + \Gamma_2 + \Gamma_3 \vdash \mathrm{reuse}\, e_1\ \mathrm{in}\ (e_2, e_3) : \tau_1 \times \tau_2 \mathbin{@} (a, u, \textsc{global})}\ \textsc{reuse}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash e_1 : \tau_1 \mathbin{@} (\textsc{many}, u_1, l_1) \\ \Gamma_2, x : \tau_1 \mathbin{@} (\textsc{many}, \textsc{aliased}, \textsc{local}) \vdash e_2 : \tau_2 \mathbin{@} (a_2, u_2, \textsc{global}) \\ \Gamma_3, x : \tau_1 \mathbin{@} (\textsc{many}, u_1, l_1), y : \tau_2 \mathbin{@} (a_2, u_2, \textsc{global}) \vdash e_3 : \tau_3 \mathbin{@} \mu\end{array}}{\Gamma_1 + \Gamma_2 + \Gamma_3 \vdash \mathrm{borrow}\, x = e_1\ \mathrm{for}\ y = e_2\ \mathrm{in}\ e_3 : \tau_3 \mathbin{@} \mu}\ \textsc{borrow}$$

Fig. 2. Mode Calculus

The VAR rule is standard, which may be surprising to the reader familiar with modal calculi. Since our locks are defined as operations on the context, their effect has already been applied in Γ, so no side-conditions are necessary. However, the algorithmic rules that we use for mode inference (see Appendix B) treat locks lazily, and consequently have a more complicated variable rule.

The SUB rule allows a term to be used at a more restrictive mode or to use a variable at a more permissive mode. We have submoding but not subtyping: the SUB rule does not change types. The LAM rule differs from the usual one in that it introduces a lock, ensuring that GLOBAL functions do not have LOCAL free variables, and that MANY functions do not have UNIQUE free variables. (Per the definition of locking, the uniqueness of a function has no effect). As we already restrict usage of

variables inside a closure, the mode of the function is immaterial in the APP rule. Our rules for the unit, sum, and product types are standard; note that modes commute with both sums and products. The SPLIT rule gives us access not only to the elements of the pair, but also its space credit. This may seem unsafe: what if the pair is not UNIQUE? However, we also store a mode for the space credit and can only reuse it if the space credit is UNIQUE. The LET rule and the CASE rule are standard. The BOX and UNBOX rules allow us to box and unbox a term at a modality (ν ranges over the modalities of Figure 1). The REUSE rule allows us to reuse a space credit to allocate a new pair. We demand that the space credit is UNIQUE and that the new components of the pair are `global`. The latter restriction is imposed by our garbage collector, which assumes that there are never pointers from the heap to the stack, making it unsafe to reuse a heap-allocated pair with stack-allocated elements. We avoid this problem by requiring that the new elements of the reused pair are `global` and thus not stack-allocated. Finally, the BORROW rule allows us to borrow the result of evaluating e_1 for the duration of e_2. We demand that e_1 yields a `many` value since we make x available to both e_2 and e_3. In e_2 it is ALIASED which ensures that no in-place update can take place. Furthermore, it is LOCAL while the return mode of e_2 is GLOBAL, which ensures that x may not escape. Once e_2 is evaluated, we can safely make x available to e_3 (as well as the result y of e_2).

3.6 Substitution

We give a substitution lemma for our calculus here, though we defer discussion of the semantics until Section 4:

LEMMA 3.1 (SUBSTITUTION). *If* $\Gamma_1, x : \tau_1 \mathbin{@} \mu_1, \Gamma' \vdash e : \tau_2 \mathbin{@} \mu_2$ *and* $\Gamma_2 \vdash v : \tau_1 \mathbin{@} \mu_1$ *and* $\Gamma_1 + \Gamma_2$ *is defined, then* $(\Gamma_1 + \Gamma_2), \Gamma' \vdash e[v/x] : \tau_2 \mathbin{@} \mu_2$.

Recall that $\Gamma_1 + \Gamma_2$ is only defined (Section 3.4) if any variable used by both Γ_1 and Γ_2 is not used uniquely by either, ensuring that the unique variables in e and v are disjoint.

3.7 Inference

Our calculus admits a type inference algorithm that can infer all types and modes in a program. The mode inference works by generating inequality constraints that can be solved by a simple solver. We cannot generate constraints based on the rules in Figure 2 directly because the LAM rule uses locks, which act on the modes in Γ, which are still being inferred. We instead switch to a different presentation of contexts, where locks are part of the syntax of contexts:

$$\Gamma ::= \varnothing \mid \Gamma, x : - \mid \Gamma, x : \tau \mathbin{@} \mu \mid \Gamma, \text{🔒}_\mu$$

and their effect is applied lazily in the VAR rule, rather than eagerly in the LAM rule:

$$\text{VAR}\ \frac{a_1 \le a_2 \wedge \bigwedge_{\text{🔒}_{(a_i,\_,\_)} \in \Gamma'} a_i \qquad u_1 \vee \bigvee_{\text{🔒}_{(a_i,\_,\_)} \in \Gamma'} \dagger(a_i) \le u_2 \qquad l_1 \le l_2 \wedge \bigwedge_{\text{🔒}_{(\_,\_,l_i)} \in \Gamma'} l_i}{\Gamma, x : \tau \mathbin{@} (a_1, u_1, l_1), \Gamma' \vdash x : \tau \mathbin{@} (a_2, u_2, l_2)}$$

The inequalities in the premises of this rule are the constraints that our solver satisfies. We offer the details in Appendix B.

3.8 Polymorphism

To focus on the essence of our design, we omit polymorphism from our discussion. However, it would be straightforward to add type polymorphism to our calculus:

$$\frac{\Gamma \vdash e : \tau \mathbin{@} \mu \qquad \alpha \notin \mathrm{FTV}(\Gamma)}{\Gamma \vdash e : (\forall \alpha.\, \tau) \mathbin{@} \mu}\ \text{GEN} \qquad\qquad \frac{\Gamma \vdash e : (\forall \alpha.\, \tau_1) \mathbin{@} \mu}{\Gamma \vdash e : \tau_1[\tau_2/\alpha] \mathbin{@} \mu}\ \text{INST}$$

For example, we can give a type to the identity function as $\mathrm{id} : \forall \alpha.\alpha \mathbin{@} \mu \to \alpha \mathbin{@} \mu$ for any fixed mode μ. In particular, there is no need for additional constraints between the type variable α and the fixed mode μ. We can choose μ to be the legacy mode to obtain exactly the type of the identity function in OCaml. This makes our system fully backwards compatible with existing OCaml code and all existing OCaml programs continue to check in our implementation.

So far, our implementation does not allow polymorphism over modes, even though we see no fundamental obstacle to adding it. Perhaps surprisingly, we have not found this to be a significant problem in practice as we show in Section 7.

4 Usage-Aware Store Semantics

In order to formulate soundness for our mode calculus, we give a usage-aware store semantics [Abel and Bernardy 2020; Choudhury et al. 2021]. We use a call-by-value abstract machine, where each step of the evaluation is denoted by a machine state $S \,[\!]\, E \rhd_n^\mu e$, with a store S, an evaluation context E (stored as a zipper), current mode μ, current stack frame n and expression e. In this semantics, bindings $x \mapsto v$ are not substituted immediately, but rather we keep the value in the store $S, b \mapsto v$ under a fresh name b and substitute $x \mapsto b$ instead. Each binding in the store is annotated by a mode μ (or *consumed*, written $-$) and its stack-frame $n \in \mathbb{N}$. We denote store addresses by b, c, d (to distinguish them from variables) and define:

$$
\begin{aligned}
v &::= b \mid () \mid \mathrm{inl}\, b \mid \mathrm{inr}\, b \mid (b, b) \mid \mathrm{box}_\nu\, b \mid \lambda x.\, e \\
\hat{\mu} &::= \mu \mid - \\
S &::= \varnothing \mid S, b \mapsto_n^{\hat{\mu}} v
\end{aligned}
$$

We show the soundness of in-place reuse, stack allocation, and borrowing using this semantics.

First, we show the soundness of in-place reuse via the mode annotations in the store. In our semantics, we only change the mode $\hat{\mu}$ of a binding when accessing an address using mode μ_2: In that case we split $\hat{\mu}$ into $\hat{\mu}_1 + \mu_2$ and keep $\hat{\mu}_1$ in the store. Since we do not allow modes to be unused ($x : -$) in our terms, the join operator $\hat{\mu}_1 + \mu_2$ is defined with a case for $-$ only on the left:

$$
\begin{aligned}
(\textsc{many}, \textsc{aliased}, l) + (a, \textsc{aliased}, l) &= (\textsc{many}, u, l) \\
- + \mu &= \mu
\end{aligned}
$$

When the evaluation requests a ALIASED value, it needs to be MANY in the store and we keep it in the store as ALIASED. When the evaluation uses a value as UNIQUE or it is ONCE in the store, it is marked as consumed afterwards. Through the usage-aware semantics, we can easily derive the soundness of in-place updates: A value is unique in the store only if it was created as unique and was not used before (in which case it would have been split up into aliased parts by the + operator). If an aliased value in the store is accessed as unique, our semantics is stuck.

Second, to show the soundness of stack allocation, we keep a stack-frame counter n that is incremented whenever we enter a region and decremented whenever we leave a region. We then mark every value in the store with the current stack-frame. At the end of a region, we delete all bindings that were allocated as local in the current stack frame:

$$
\varnothing - n = \varnothing \qquad (S, b \mapsto_m^{\hat{\mu}} v) - n = \begin{cases} S - n & \text{if } \hat{\mu} = (a, u, \textsc{local}) \text{ and } m = n \\ S - n, b \mapsto_m^{\hat{\mu}} v & \text{otherwise} \end{cases}
$$

Our invariant ensures that the pointers in our store follow a predictable pattern: Variables allocated in the heap can only refer only to other heap-allocated values, and stack allocated variables can only refer to values in the same or an earlier stack frame.

Third, we show the soundness of borrowing. Our main concern is that no further references remain when a borrowed use ends. Thus, we model borrowing by copying the borrowed value into the new stack frame (see bottom of Figure 3). This copy is deep, but we can stop copying when the data is guaranteed to be aliased: on an aliased box, a global box (which implies aliased), and a lambda (which is guaranteed to be many by the borrow rule and thus only contains aliased data). When the region ends the stack frame is deleted, which allows us to conclude that no further references to the borrowed value remain from the soundness of stack allocation.

Putting these arguments together, we define a typing relation for stores. We denote typing contexts that contain store addresses instead of variables by Σ. We write $S :_n \Sigma$ to denote that the store S is well-typed with respect to Σ using n stack frames:

$$\frac{\mu = (\textsc{many}, \textsc{aliased}, \textsc{global})}{\varnothing :_n (\texttt{null} : \clubsuit \mathbin{@} \mu)}\ \textsc{wf-base} \qquad \frac{S :_m \Sigma_1 \qquad m \leq n \qquad m \leq k \leq n \text{ or } \mu = (\_, \_, \textsc{global})}{(S, b \mapsto^{\mu}_{k} \clubsuit) :_n (\Sigma_1, b : \clubsuit \mathbin{@} \mu)}\ \textsc{wf-space}$$

$$\frac{S :_m \Sigma_1 \qquad m \leq n}{(S, b \mapsto^{-}_{k} v) :_n (\Sigma_1, b : -)}\ \textsc{wf-unused} \qquad \frac{S :_m (\Sigma_1 + \Sigma_2) \qquad \Sigma_2 \vdash v : \tau \mathbin{@} \mu \qquad m \leq n \qquad m \leq k \leq n \text{ or } \mu = (\_, \_, \textsc{global})}{(S, b \mapsto^{\mu}_{k} v) :_n (\Sigma_1, b : \tau \mathbin{@} \mu)}\ \textsc{wf-ext}$$

We construct an empty store using the WF-BASE rule, where we allow a logical binding to `null`, which is the space credit corresponding to unsuccessful reuses. We allow consumed bindings in the store through the WF-UNUSED rule. The most important rule is the WF-EXT rule, which allows us to extend the store by a new binding to a value. For this, we require that the value could have been produced using the elements defined in the store before it: we split the typing context of the existing store into Σ_1 and Σ_2 and use Σ_2 to check the value. This mirrors our substitution lemma 3.1 and ensures that we could reconstruct the value at each usage site of the binding. Notice that this definition allows a variable to be marked UNIQUE in the store even if there are several references to it, as long as all such references only use the value as ALIASED. The WF-SPACE rule is similar, but give it separately since we do not have a constructor for space credits in the source language.

We show the reduction steps in Figure 3. We omit the steps that merely load and unload expressions from the evaluation context from the main text, since these steps do not modify the store and merely adjust the current mode (they can be found in the appendix in Figure 9 and 10). However, we include these steps for the `borrow` construct, since these are the steps were the stack frame counter is incremented and decremented.

The steps for the value constructors are straightforward: We allocate the value at a fresh address d using the current mode and stack frame counter. In the elimination steps, we split the mode of the accessed binding using the + operator and keep $\hat{\mu}_1$ in the store. The complement μ_2 is given by the expression, except in the (app) step, where any μ_2 is allowed (mirroring the APP rule which ignores the mode of the closure).

For the split steps, we split on whether this step requires a UNIQUE value or not. If the value is UNIQUE, then b is not used later to access the pair. Thus, we can safely overwrite it to point to a space credit. Otherwise, we keep the binding of b to the pair in the store and use a special space credit `null` which is ALIASED. In the reuse step, we overwrite the space credit by the new pair and

$$
\begin{array}{lrcll}
\text{(unit)} & S \,[\!]\, E \rhd^{\mu}_{n} () & \rightsquigarrow & S, d \mapsto^{\mu}_{n} () \,[\!]\, E \rhd^{\mu}_{n} d & (d \text{ fresh}) \\
\text{(inl)} & S \,[\!]\, E \rhd^{\mu}_{n} \mathrm{inl}\, b & \rightsquigarrow & S, d \mapsto^{\mu}_{n} \mathrm{inl}\, b \,[\!]\, E \rhd^{\mu}_{n} d & (d \text{ fresh}) \\
\text{(inr)} & S \,[\!]\, E \rhd^{\mu}_{n} \mathrm{inr}\, b & \rightsquigarrow & S, d \mapsto^{\mu}_{n} \mathrm{inr}\, b \,[\!]\, E \rhd^{\mu}_{n} d & (d \text{ fresh}) \\
\text{(pair)} & S \,[\!]\, E \rhd^{\mu}_{n} (b, c) & \rightsquigarrow & S, d \mapsto^{\mu}_{n} (b, c) \,[\!]\, E \rhd^{\mu}_{n} d & (d \text{ fresh}) \\
\text{(box)} & S \,[\!]\, E \rhd^{\mu}_{n} \mathrm{box}_{\nu}\, b & \rightsquigarrow & S, d \mapsto^{\mu}_{n} \mathrm{box}_{\nu}\, b \,[\!]\, E \rhd^{\mu}_{n} d & (d \text{ fresh}) \\
\text{(lam)} & S \,[\!]\, E \rhd^{\mu}_{n} \lambda x.e & \rightsquigarrow & S, d \mapsto^{\mu}_{n} \lambda x.e \,[\!]\, E \rhd^{\mu}_{n} d & (d \text{ fresh}) \\
\text{(let)} & S \,[\!]\, E \rhd^{\mu}_{n} \mathrm{let}\, x = b \,\mathrm{in}\, e & \rightsquigarrow & S \,[\!]\, E \rhd^{\mu}_{n} e[x := b] &
\end{array}
$$

$$
\text{(app)} \quad S, b \mapsto^{\hat{\mu}_1+\mu_2}_{m} \lambda x.e, S' \,[\!]\, E \rhd^{\mu}_{n} b\, c \rightsquigarrow S, b \mapsto^{\hat{\mu}_1}_{m} \lambda x.e, S' \,[\!]\, E \rhd^{\mu}_{n} e[x := c]
$$

$$
\begin{aligned}
\text{(unbox)} \quad & S, b \mapsto^{\hat{\mu}_1+\mu_2}_{m} \mathrm{box}_{\nu}\, c, S' \,[\!]\, E \rhd^{\mu_2}_{n} \mathrm{unbox}_{\nu}\, b \\
& \rightsquigarrow S, b \mapsto^{\hat{\mu}_1}_{m} \mathrm{box}_{\nu}\, c, S' \,[\!]\, E \rhd^{\mu_2}_{n} c
\end{aligned}
$$

$$
\begin{aligned}
\text{(case}_{\mathrm{l}}\text{)} \quad & S, b \mapsto^{\hat{\mu}_1+\mu_2}_{m} \mathrm{inl}\, c, S' \,[\!]\, E \rhd^{\mu}_{n} \mathrm{case}_{\mu_2}\, b\, \{\, \mathrm{inl}\, x \to e_1; \mathrm{inr}\, y \to e_2 \,\} \\
& \rightsquigarrow S, b \mapsto^{\hat{\mu}_1}_{m} \mathrm{inl}\, c, S' \,[\!]\, E \rhd^{\mu}_{n} e_1[x := c]
\end{aligned}
$$

$$
\begin{aligned}
\text{(case}_{\mathrm{r}}\text{)} \quad & S, b \mapsto^{\hat{\mu}_1+\mu_2}_{m} \mathrm{inr}\, c, S' \,[\!]\, E \rhd^{\mu}_{n} \mathrm{case}_{\mu_2}\, b\, \{\, \mathrm{inl}\, x \to e_1; \mathrm{inr}\, y \to e_2 \,\} \\
& \rightsquigarrow S, b \mapsto^{\hat{\mu}_1}_{m} \mathrm{inr}\, c, S' \,[\!]\, E \rhd^{\mu}_{n} e_2[y := c]
\end{aligned}
$$

$$
\begin{aligned}
\text{(split\_unique)} \quad & S, b \mapsto^{\mu_1+\mu_2}_{m} (c, d), S' \,[\!]\, E \rhd^{\mu}_{n} \mathrm{let}_{\mu_2}(x, y, z) = b \,\mathrm{in}\, e \qquad (\mu_2 = (\_, \textsc{unique}, \_)) \\
& \rightsquigarrow S, b \mapsto^{\mu_2}_{m} \clubsuit, S' \,[\!]\, E \rhd^{\mu}_{n} e[x := b, y := c, z := d]
\end{aligned}
$$

$$
\begin{aligned}
\text{(split\_aliased)} \quad & S, b \mapsto^{\hat{\mu}_1+\mu_2}_{m} (c, d), S' \,[\!]\, E \rhd^{\mu}_{n} \mathrm{let}_{\mu_2}(x, y, z) = b \,\mathrm{in}\, e \qquad (\mu_2 = (\_, \textsc{aliased}, \_)) \\
& \rightsquigarrow S, b \mapsto^{\hat{\mu}_1}_{m} (c, d), S' \,[\!]\, E \rhd^{\mu}_{n} e[x := \mathtt{null}, y := c, z := d]
\end{aligned}
$$

$$
\begin{aligned}
\text{(reuse)} \quad & S, b \mapsto^{\mu_1}_{m} \clubsuit, S' \,[\!]\, E \rhd^{\mu}_{n} \mathrm{reuse}\, b \,\mathrm{with}\, (c, d) \\
& \rightsquigarrow S, S', b \mapsto^{\mu}_{n} (c, d) \,[\!]\, E \rhd^{\mu}_{n} b
\end{aligned}
$$

$$
\begin{aligned}
\text{(enter\_region)} \quad & S \,[\!]\, E \rhd^{\mu}_{n} \mathrm{borrow}\, x = b \,\mathrm{for}\, y = e_2 \,\mathrm{in}\, e_3 \\
& \rightsquigarrow S, \mathrm{copy}(n+1, S, b, b') \,[\!]\, \mathrm{borrow}\, x = b \,\mathrm{for}\, y = E \,\mathrm{in}\, e_3 \rhd^{\mu}_{n+1} e_2[x := b'] \\
\text{(leave\_region)} \quad & S \,[\!]\, \mathrm{borrow}\, x = b \,\mathrm{for}\, y = E \,\mathrm{in}\, e_3 \rhd^{\mu}_{n+1} c \\
& \rightsquigarrow S - (n+1) \,[\!]\, E \rhd^{\mu}_{n} e_3[x := b, y := c]
\end{aligned}
$$

$$
\mathrm{copy}(n, S, b, b') = \begin{cases}
b' \mapsto^{(\textsc{many},\textsc{aliased},\textsc{local})}_{n} () & \text{if } b \mapsto^{\mu}_{m} () \in S \\
\mathrm{copy}(n, S, c, c'), b' \mapsto^{(\textsc{many},\textsc{aliased},\textsc{local})}_{n} \mathrm{inl}\, c' & \text{if } b \mapsto^{\mu}_{m} \mathrm{inl}\, c \in S \\
\mathrm{copy}(n, S, c, c'), b' \mapsto^{(\textsc{many},\textsc{aliased},\textsc{local})}_{n} \mathrm{inr}\, c' & \text{if } b \mapsto^{\mu}_{m} \mathrm{inr}\, c \in S \\
\mathrm{copy}(n, S, c, c'), \mathrm{copy}(n, S, d, d'), b' \mapsto^{(\textsc{many},\textsc{aliased},\textsc{local})}_{n} (c', d') & \\
\qquad\qquad\qquad\qquad\qquad\qquad & \text{if } b \mapsto^{\mu}_{m} (c, d) \in S \\
\mathrm{copy}(n, S, c, c'), b' \mapsto^{(\textsc{many},\textsc{aliased},\textsc{local})}_{n} \mathrm{box}_{\mathrm{M}}\, c' & \text{if } b \mapsto^{\mu}_{m} \mathrm{box}_{\mathrm{M}}\, c \in S \\
b' \mapsto^{(\textsc{many},\textsc{aliased},\textsc{local})}_{n} \mathrm{box}_{\mathrm{A}}\, c & \text{if } b \mapsto^{\mu}_{m} \mathrm{box}_{\mathrm{A}}\, c \in S \\
b' \mapsto^{(\textsc{many},\textsc{aliased},\textsc{local})}_{n} \mathrm{box}_{\mathrm{G}}\, c & \text{if } b \mapsto^{\mu}_{m} \mathrm{box}_{\mathrm{G}}\, c \in S \\
b' \mapsto^{(\textsc{many},\textsc{aliased},\textsc{local})}_{n} \lambda x.\, e & \text{if } b \mapsto^{\mu}_{m} \lambda x.\, e \in S)
\end{cases}
$$

Fig. 3. Usage-Aware Store Semantics: Reduction Steps

move it to the end of the store to ensure that it is defined after the elements of the pair. Since this rule demands a UNIQUE space credit, it can be safely overwritten (and is not `null`).

When entering a region, we increase the stack frame counter and copy the borrowed value into the new stack frame. This copying operation is deep and applies to all children of the value up to closures and global boxes. We do not have to copy those, since they can only hold ALIASED values (remember that b is MANY by the REGION rule). When leaving a region, we decrement the stack frame counter and delete all local bindings in that stack frame.

We now formulate the progress and preservation lemmas (see Appendix A.3 for the proofs). We write $?$ for the empty evaluation context and $E[e : \tau \mathbin{@} \mu]$ to denote that expression e has type τ and mode μ in the evaluation context E. Further, we call a pair (E, e) well-formed for (S, n) if all free variables of $E[e]$ are either global or are stack-allocated in a frame $m \leq n - k$, where k is the number of borrowing frames in E that need to be traversed to reach the variable.

LEMMA 4.1 (PROGRESS). *If $\Sigma \vdash E[e : \tau_1 \mathbin{@} \mu_1] : \tau_2 \mathbin{@} \mu_2$ and $S :_n \Sigma$ and (E, e) is well-formed for (S, n), then either execution concludes with $E = ?$ and $e = b$ for some store address b or a step is possible: $S \,[\!]\, E \rhd_n^{\mu_1} e \rightsquigarrow S' \,[\!]\, E' \rhd_{n'}^{\mu_1'} e'$.*

LEMMA 4.2 (PRESERVATION). *If $\Sigma \vdash E[e : \tau_1 \mathbin{@} \mu_1] : \tau_2 \mathbin{@} \mu_2$ and (E, e) is well-formed for (S, n), and $S :_n \Sigma$ and $S \,[\!]\, E \rhd_n^{\mu_1} e \rightsquigarrow S' \,[\!]\, E' \rhd_{n'}^{\mu_1'} e'$, then $\Sigma' \vdash E'[e' : \tau_1' \mathbin{@} \mu_1'] : \tau_2 \mathbin{@} \mu_2$ and (E', e') is well-formed for (S', n'), and $S' :_{n'} \Sigma'$.*

5 Translation to a Graded Modal Calculus

While our mode calculus is modelled using type qualifiers, there is a close relationship to graded modal calculi [Abel and Bernardy 2020; Choudhury et al. 2021; Orchard et al. 2019; Petricek et al. 2014], which annotate variables in the context with grades from a partially-ordered semiring. Such calculi underpin a popular approach to creating type systems that track how values are used, forming the basis of both linear types in Haskell [Bernardy et al. 2017] and linear resources in Idris [Brady 2021], so it is important to understand their relationship to our own approach.

In this section we give a translation from our mode calculus into a graded modal calculus, by choosing an appropriate partially-ordered semiring. Whilst we do not do so here, this translation can also form the basis of a categorical semantics for our mode calculus, built on the existing categorical semantics [Brunel et al. 2014; Gaboardi et al. 2016; Katsumata 2018] for graded modal calculi, which assume grades form a partially-ordered semiring.

5.1 Incompatibility between Axiom K and Call-by-Value

Axiom K is a core axiom of modal logic, stating that if $\Box(A \to B)$ and $\Box A$, then also $\Box B$. In our effectful call-by-value setting, this axiom can not model the modes in this paper. To see why, and how to fix the problem, let us consider a simple example in the style of Choudhury et al. [2021]. Variables in the context are annotated by a grade q, which in our example will be a natural number that denotes the number of times the variable may be used. The typing rules for variables, applications and boxes are:

$$\frac{}{0 \cdot \Gamma, x :^1 A \vdash x : A}\ \text{VAR} \qquad \frac{\Gamma_1 \vdash e_1 : {}^q A \to B \qquad \Gamma_2 \vdash e_2 : A}{\Gamma_1 + q \cdot \Gamma_2 \vdash e_1\, e_2 : B}\ \text{APP} \qquad \frac{\Gamma \vdash e : A}{q \cdot \Gamma \vdash \mathrm{box}_q\, e : \Box^q A}\ \text{BOX}$$

The VAR rule allows us to use a variable one time, the APP rule allows us to pass an argument e_2 to a function that will use the argument q times as long as we multiply the grades of the variables in the argument by q. Similarly, the BOX rule allows us to create a box that can be used q times as long as we multiply the grades by q.

With these three rules we can derive a version of the modal Axiom K:

$$\dfrac{\dfrac{\dfrac{}{f :^{1} ({}^{1}A \to B), x :^{0} A \vdash f : A \to B}\,\textsc{var} \quad \dfrac{}{f :^{0} ({}^{1}A \to B), x :^{1} A \vdash x : A}\,\textsc{var}}{f :^{1} ({}^{1}A \to B), x :^{1} A \vdash f\,x : B}\,\textsc{app}}{f :^{q\cdot 1} ({}^{1}A \to B), x :^{q\cdot 1} A \vdash \mathrm{box}_q(f\,x) : \Box_q B}\,\textsc{box}$$

This rule allows us to use the result of $(f\,x)$ q times if we can use both f and x q times. In a call-by-name setting, $(f\,x)$ can be treated as a thunk which is evaluated anew every time it is used—this only requires that f and x are available q times, as guaranteed by the rules. In a call-by-value setting, however, $(f\,x)$ evaluates to a value which may not be usable q times. Concretely, a function such as `open_file` : $\mathbb{1} \to$ `File` can be used many times and so can a unit value, but the resulting `File` should only be used affinely.

Thankfully, it is not hard to rule out Axiom K by restricting the BOX rule to values:

$$\dfrac{\Gamma \vdash V : A}{q \cdot \Gamma \vdash \mathrm{box}_q\, V : \Box^q A}\,\textsc{box}$$

This prevents the derivation above since it is no longer possible to box an application.

5.2 Graded Call-by-Value Calculus

We use a fine-grain call-by-value [Levy et al. 2003] formulation in which terms are divided into value terms and computation terms. (See Appendix C for full syntax and typing rules.) In the types, we include grades on boxes and for the arguments of functions, as is usual for graded type theories. Following Abel and Bernardy [2020], but unlike other graded type theories, we order our grades such that if $q \le r$ then a value at grade q can be used at grade r, maintaining consistency with the ordering of our modes. For contexts, multiplication $(q \cdot \Gamma)$, addition $(\Gamma_1 + \Gamma_2)$ and ordering $(\Gamma_1 \preceq \Gamma_2)$ are defined point-wise on the grades of their bindings.

In addition to the semiring, our rules are parameterised by a choice of grade σ that is required of values of sum types when they are eliminated.

$$\dfrac{\Gamma_1 \vdash^{\mathrm{v}} V : A + B \qquad q \preceq \sigma \qquad \Gamma_2, x :^{q} A \vdash^{\mathrm{c}} M : C \qquad \Gamma_2, y :^{q} B \vdash^{\mathrm{c}} N : C}{q \cdot \Gamma_1 + \Gamma_2 \vdash^{\mathrm{c}} \mathrm{case}_q\, V\, \{\, \mathrm{inl}\, x \to M; \mathrm{inr}\, y \to N \,\} : C}\,\textsc{case}$$

(The superscripts on turnstyles distinguish the value typing judgement, $\vdash^{\mathrm{v}}$, from the computation typing judgement, $\vdash^{\mathrm{c}}$.) In existing graded calculi σ is usually chosen to be 1 but, as noted by Choudhury et al. [2021], that choice is not necessary for general type soundness.

5.3 The Naive Semiring

Now that we have a graded calculus, we need to construct an appropriate semiring to model our modes. For brevity we restrict attention to uniqueness and affinity, but the same techniques also apply to locality and borrowing.

The *plus* operation of the semiring should correspond to the *join* operation on contexts from our mode calculus. That means we need a **0** grade to represent the $x : -$ case in contexts, grades for each of the modes, and a $\bot$ grade to represent the cases where the join operation is undefined. That gives us the following elements for our semiring:

$$\{\textsc{unused}\ (\mathbf{0}),\ \textsc{aliased once}\ (\mathbf{A}),\ \textsc{aliased many}\ (\mathbf{AM}),$$
$$\textsc{unique once}\ (\mathbf{1}),\ \textsc{unique many}\ (\mathbf{M}),\ \textsc{unique error}\ (\bot)\}$$

The ordering on the semiring corresponds to the ordering on contexts in our mode calculus. We choose ALIASED ONCE for σ, essentially allowing any sum value to be eliminated.

The *multiplication* operation of the semiring should correspond to the *modalities* of our mode calculus: multiplying by $\mathbf{A}$ should behave like the A modality and multiplying by $\mathbf{M}$ like the M modality. The rest of multiplication is forced by the semiring laws. Interesting equations include:

$$\mathbf{A}+\mathbf{A}=\mathbf{AM} \qquad \mathbf{1}+\mathbf{A}=\bot \qquad \mathbf{A}\cdot\mathbf{A}=\mathbf{A} \qquad \mathbf{A}\cdot\mathbf{M}=\mathbf{AM} \qquad \mathbf{A}\cdot\bot=\mathbf{AM}$$
$$\mathbf{1}+\mathbf{1}=\bot \qquad \mathbf{M}+\mathbf{M}=\bot \qquad \mathbf{M}\cdot\mathbf{M}=\mathbf{M} \qquad \mathbf{M}\cdot\mathbf{A}=\mathbf{AM} \qquad \mathbf{M}\cdot\bot=\bot$$

Using this semiring as grades, we can safely enable in-place updates. When splitting a pair, we now also get access to its space credit and we add a `reuse` construct which allows us to reuse the space credit for a new pair. The extended syntax and typing rules are shown in Appendix C.3.

5.4 Extending the Semiring for Aliasable Closures

The naive semiring gives us a sound system, but one that prevents functions from being applied once they have been aliased. Consider the APP rule of our graded calculus:

$$\frac{\Gamma_1 \vdash^{\mathrm{v}} V : {}^{q}A \to B \qquad \Gamma_2 \vdash^{\mathrm{v}} W : A}{\Gamma_1 + q\cdot\Gamma_2 \vdash^{\mathrm{c}} V\,W : B}\ \textsc{app}$$

Γ_1 are the inputs required for V to produce a function at grade 1. In particular, if V is a variable x then Γ_1 will be $\{x :^{1} {}^{q}A \to B\}$. Γ_1 isn't multiplied by a grade in the conclusion, and it can't be multiplied by a grade in a surrounding rule because there is no $\mathrm{box}_{\mathbf{A}}$ construct for computations. That means that, given a binding $x :^{\mathbf{A}} {}^{q}A \to B$ in the context there is no way to apply it.[5]

Relatedly, a function that closes over a use of a value at grade UNIQUE MANY:

$$x :^{\mathbf{M}} A \vdash^{\mathrm{c}} \lambda y.\, N : B \to C$$

can be placed in a $\mathrm{box}_{\mathbf{A}}$ expression so that it no longer requires x uniquely:

$$x :^{\mathbf{A}\cdot\mathbf{M}} A \vdash^{\mathrm{c}} \mathrm{box}_{\mathbf{A}}(\lambda y.\, N) : \Box^{\mathbf{A}}(B \to C)$$

This is sound only because aliased functions can never be applied, and is why we can't simply adjust the APP rule to allow applying aliased functions.

We solve this problem by insisting that closing over a UNIQUE use gives a ONCE function, which permits us to safely allow applying ALIASED functions. This solution is not obviously modal though: it requires prohibiting some values with only UNIQUE MANY free variables from being UNIQUE MANY, which runs counter to the box rule of modal calculi. However, by extending the underlying semiring, we can in fact model this solution in our graded modal calculus.

What we require is an element $\mathbf{A} \to \mathbf{1}$ in our semiring that acts as a right residual to $\mathbf{A}$:

$$\mathbf{A}\cdot q \le r \iff q \le (\mathbf{A}\to\mathbf{1})\cdot r$$

then $\Box_{\mathbf{A}\to\mathbf{1}}$ would be a right adjoint to $\Box_{\mathbf{A}}$, and $\Box_{\mathbf{A}} \circ \Box_{\mathbf{A}\to\mathbf{1}}$ would be a comonad. This would allow us to use $\Box_{\mathbf{A}\to\mathbf{1}}\,(A \to B)$ as the type of functions that can be applied after they have been aliased (we refer to these as *coaliased* functions). Note that placing such a function in a $\mathrm{box}_{\mathbf{A}}$ expression would require the values in its context at $\mathbf{A}\cdot\mathbf{A} \to \mathbf{1}$, which is at least as strong as the original context of the function.

Our naive semiring does not contain such an element, so we would like to embed it into a larger semiring that does. There is an initial such embedding which extends the semiring with three

[5] Technically, we could put the application inside a lambda and then put that in a $\mathrm{box}_{\mathbf{A}}$ expression, but that just gets us another aliased lambda we can't apply.

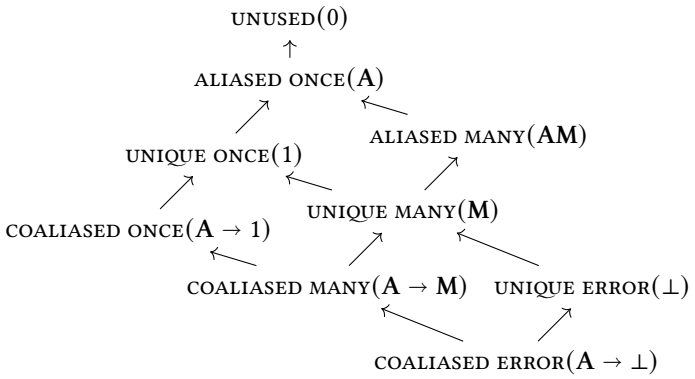

Fig. 4. Ordering of the Extended Semiring

additional elements:

$$\{\text{COALIASED ONCE } (\mathbf{A} \to \mathbf{1}),\ \text{COALIASED MANY } (\mathbf{A} \to \mathbf{M}),\ \text{COALIASED ERROR } (\mathbf{A} \to \bot)\}$$

The ordering for this extended semiring is shown in Figure 4, and the full set of equations for addition and multiplication are in Appendix D. The most interesting of those equations are:

$$\mathbf{A} \to \mathbf{1} + \mathbf{A} \to \mathbf{1} = \mathbf{A} \to \bot \qquad \mathbf{A} \cdot (\mathbf{A} \to \mathbf{1}) = \mathbf{A} \to \mathbf{1} \qquad \mathbf{M} \cdot (\mathbf{A} \to \mathbf{1}) = \mathbf{A} \to \bot$$
$$\mathbf{A} \to \mathbf{1} + \mathbf{1} = \mathbf{A} \to \bot \qquad (\mathbf{A} \to \mathbf{1}) \cdot \mathbf{A} = \mathbf{A} \qquad (\mathbf{A} \to \mathbf{1}) \cdot \mathbf{M} = \mathbf{A} \to \mathbf{M}$$

By using the extended semiring and wrapping functions in $\square^{(\mathbf{A}\to\mathbf{1})}$ we avoid the issues with the naive semiring. A function that closes over a use of a value at grade UNIQUE MANY will now require that value at grade COALIASED MANY, and it will continue to do so even if placed in a $\text{box}_{\mathbf{A}}$ expression. Any attempt to wrap such a function in $\square^{\mathbf{M}}$ will result in an error grade, essentially forcing the function to be treated as ONCE. This mirrors how closing over a (MANY, UNIQUE, l) value produces a (ONCE, u, l) function in our mode calculus.

5.5 Translation from the Mode Calculus

We can now translate our mode calculus to the graded calculus. Our modes and modalities are translated directly into grades in the obvious way. The translation for types is mostly straightforward, with the exception of arrow types, which are translated as:

$$[\![\tau_1 \mathbin{@} \mu_1 \longrightarrow \tau_2 \mathbin{@} \mu_2]\!] = \square^{(\mathbf{A}\to\mathbf{1})} ({}^{(\mathbf{A}\to\mathbf{1})\cdot[\![\mu_1]\!]}[\![\tau_1]\!] \longrightarrow \square^{(\mathbf{A}\to\mathbf{1})\cdot[\![\mu_2]\!]}[\![\tau_2]\!]))$$

which has been wrapped in an $\mathbf{A} \to \mathbf{1}$ box. Note that the grades on the parameter and return have also been multiplied by $\mathbf{A} \to \mathbf{1}$. This ensures that parameters and results always stand for values that can in turn be used inside a closure. Similarly, contexts are translated as:

$$[\![\varnothing]\!] = \varnothing \qquad [\![\Gamma, x : -]\!] = [\![\Gamma]\!] \qquad [\![\Gamma, x : \tau \mathbin{@} \mu]\!] = [\![\Gamma]\!], x :^{(\mathbf{A}\to\mathbf{1})\cdot[\![\mu]\!]} [\![\tau]\!]$$

with the grades on variable bindings multiplied by $\mathbf{A} \to \mathbf{1}$. We translate judgements $\Gamma \vdash e : \tau \mathbin{@} \mu$ into judgements of the form $[\![\Gamma]\!] \vdash^{\mathrm{c}} M : \square^{(\mathbf{A}\to\mathbf{1})\cdot[\![\mu]\!]}[\![t]\!]$. The rules are quite mechanical, so for brevity we show only a few of instances of the translation:

$$\begin{aligned}&[\![\Gamma \vdash \lambda x.\, e : (\tau_1 \mathbin{@} \mu_1 \longrightarrow \tau_2 \mathbin{@} \mu_2) \mathbin{@} \mu_3]\!] \\ &\quad = [\![\Gamma]\!] \vdash^{\mathrm{c}} \text{return } (\text{box}_{(\mathbf{A}\to\mathbf{1})\cdot[\![\mu_3]\!]} \ (\text{box}_{(\mathbf{A}\to\mathbf{1})} \ (\lambda x.\, [\![e]\!]))) \\ &\qquad\quad : \square^{(\mathbf{A}\to\mathbf{1})\cdot[\![\mu_3]\!]} \square^{\mathbf{A}\to\mathbf{1}} ({}^{(\mathbf{A}\to\mathbf{1})\cdot[\![\mu_1]\!]}[\![\tau_1]\!] \longrightarrow \square^{(\mathbf{A}\to\mathbf{1})\cdot[\![\mu_2]\!]}[\![\tau_2]\!])\end{aligned}$$

$$\frac{\Gamma_1 \vdash x : \tau \mathbin{@} (\textsc{many}, \_, \textsc{global}) \qquad \Gamma_2, y : \tau \mathbin{@} (\textsc{many}, \textsc{aliased}, \textsc{local}) \vdash e : \tau_2 \mathbin{@} (a, u, \textsc{global})}{\Gamma_2 \fatsemi \lceil \Gamma_1 \rceil \vdash \mathbf{let}\ y = \&x\ \mathbf{in}\ e : \tau_2 \mathbin{@} (a, u, \textsc{global})}\ \textsc{borrow}$$

$$\frac{\Gamma, \blacksquare_{\mu_3}, x : \tau_1 \mathbin{@} \mu_1 \vdash e : \tau_2 \mathbin{@} \mu_2}{\lfloor \Gamma \rfloor \vdash \lambda x.\, e : (\tau_1 \mathbin{@} \mu_1 \to \tau_2 \mathbin{@} \mu_2) \mathbin{@} \mu_3}\ \textsc{lam}$$

$$\lceil \varnothing \rceil = \varnothing \qquad \lfloor \varnothing \rfloor = \varnothing$$
$$\lceil \Gamma, x : \tau \mathbin{@}^b \mu \rceil = \lceil \Gamma \rceil, x : \tau \mathbin{@}^{\mathbb{B}} \mu \qquad \lfloor \Gamma, x : \tau \mathbin{@}^b \mu \rfloor = \lfloor \Gamma \rfloor, x : \tau \mathbin{@} \mu$$

Fig. 5. Rules for Borrowing

$$\begin{aligned}
&[\![\Gamma \vdash e_1\ e_2 : \tau \mathbin{@} \mu]\!] \\
&\quad = [\![\Gamma]\!] \vdash^{\mathrm{c}} [\![e_1]\!] \text{ to } x_1.\ [\![e_2]\!] \text{ to } x_2. \\
&\qquad\qquad \mathrm{let\,box}_{(\mathbf{A}\to\mathbf{1})\cdot[\![\mu_1]\!]}\ x_3 = x_1 \text{ in let box}_{(\mathbf{A}\to\mathbf{1})\cdot[\![\mu_2]\!]}\ x_4 = x_2 \text{ in} \\
&\qquad\qquad \mathrm{let}_{(\mathbf{A}\to\mathbf{1})\cdot[\![\mu_1]\!]}\ \mathrm{box}_{\mathbf{A}\to\mathbf{1}}\ x_5 = x_3 \text{ in} \\
&\qquad\qquad x_5\ x_4 \\
&\qquad : \Box^{(\mathbf{A}\to\mathbf{1})\cdot[\![\mu]\!]}([\![\tau_1]\!] \times [\![\tau_2]\!])
\end{aligned}$$

6 Surface-Language Design Decisions

Not only do we need a sound theory supporting stack allocation and memory reuse, but we also need to incorporate these ideas into a surface language that is approachable and expressive. This section describes our decisions around the design of these language features, as used by the programmer.

6.1 Borrowing

While the borrowing rule in our mode calculus captures the essence of borrowing in our system, it is much less flexible in practice than the borrowing construct available in our surface language. Programmers can implicitly place borrows around functions that take local arguments and have a global return value using the `&` syntax from Rust:

```
let apply_and_reverse b f g x =
  let result = if b then f &x else g &x in result, reverse x
```

Here, we use `x` twice, but since the first use (in either branch of the `if`) is a borrow, we can still destructively reverse `x` and return it as part of a `unique` pair.

We allow borrowing with `&` in two different contexts:

- right-hand side of a `let`: In `let x = &y in expr`, `x` can be used as `aliased` within `expr`; however, `y` remains in scope after the `let` is done evaluating, and it can be used `uniquely`.
- function argument: We treat `f ... &y ...` just like `let x = &y in f ... x ...`.

The typing rules that explain how borrowing works are in Figure 5, which we now explain.

Borrowing Annotations. The typing context here comprises a list of bindings $x : \tau \mathbin{@}^b \mu$, newly carrying a borrowing annotation b. This annotation is $\mathbb{B}$ if the variable has been returned after a borrow, or omitted otherwise. We extend the $+$ operation of Section 3.4 to an associative, non-commutative operator $\fatsemi$ by adding an annotation variable b to every equation of $+$ (requiring that the b be the same when combining two used variable bindings) and adding the following equation:

$$(\Gamma_1, x : \tau \mathbin{@}^{\mathbb{B}} (\_, \texttt{aliased}, l)) \fatsemi (\Gamma_2, x : \tau \mathbin{@}^{b} (\_, u, l)) = (\Gamma_1 \fatsemi \Gamma_2), x : \tau \mathbin{@}^{b} (\texttt{many}, u, l)$$

This new equation says that if the variable has been borrowed in Γ_1, we can join it with any use in Γ_2. The borrow annotation from Γ_2 is retained in the joined annotation, allowing us to borrow a variable multiple times.

When comparing contexts, we allow for borrowing annotations to be forgotten:

$$\Gamma, x : -, \Gamma' \geq \Gamma, x : \tau \mathbin{@}^{\mathbb{B}} \mu, \Gamma'$$
$$\Gamma, x : \tau \mathbin{@}^{\mathbb{B}} \mu, \Gamma' \geq \Gamma, x : \tau \mathbin{@} \mu, \Gamma'$$

Adding and Removing Borrowing Annotations. We see in the BORROW rule that y is both aliased and local in e, where e itself is required to be global. (This requirement means that e cannot be a conduit for letting y escape.) Γ_1 will have a usage for x and no usage for all other variables. In the result of the rule, the binding for x is marked with $\mathbb{B}$ in $\lceil \Gamma_1 \rceil$. This is then combined with Γ_2, leading to x being marked borrowed in the conclusion of the rule. This setup means that any unique use of x *before* the borrow will be rejected: when a $\mathbb{B}$ is on the right of $\fatsemi$, the left-hand usage must be aliased. However, a unique use of x *after* the borrow is fine, using our new equation above for $\fatsemi$.

Furthermore, we have to adjust the LAM rule to ensure that borrowing annotations do not escape a closure. This is necessary, since a closure may be invoked at a later point in the control-flow, where the previously borrowed variable may have been used. As such, we have to delete all borrowing annotations using the $\lfloor \Gamma \rfloor$ operator.

Elaboration. To show the soundness of this surface-level borrowing construct, we desugar the syntax into our mode calculus. The key step is a translation to ANF, which allows us to reason cleanly about what gets evaluated after the borrow. The details are in Appendix A.2.

6.2 Region Placement

We describe local values as unable to escape their region. But what is a region? Though our formalism supports regions through the explicit `borrow` construct, we have no such syntax in the surface language. Instead, we assume regions surround function and loop bodies. Thus, defining a function also defines a region around its contents, and writing a loop (**`for`** or **`while`** in OCaml) surrounds its body in a region. However, sometimes a user does not want a region. For example, we might want to write an implementation for

```
val init_local : len:int -> (int -> 'a @ local) -> 'a list @ local
```

that creates a stack-allocated list. Yet the body of this function must somehow return its `local` result. Our approach is to introduce a new keyword, **`exclave`**, that ends a region prematurely. The construct **`exclave`** e ends the current region and then executes e in the outer region. We cannot re-enter a region once it has ended, so **`exclave`** is only supported in tail position of an existing region (e.g. function). Values allocated in an **`exclave`** are placed in the memory from an outer region. This is exactly the behavior we want for `init_local`: its allocated cons cells should be in the region of the caller, not in the region of `init_local`. Concretely, here is the implementation of `init_local`:

```
let init_local ~len f =
  let rec loop n acc =
    if n = 0 then acc else exclave loop (n-1) (f (n-1) :: acc)
  in exclave loop len []
```

We need to write **`exclave`** in both the outer function and its inner helper; this allows the local list to be returned without crossing a region boundary.

6.3 Tail Calls

Leaving a region is a run-time concern: the implementation must move a stack pointer to release the memory in the region. Yet this bit of cleanup interferes with the tail-call optimization, where the calling function's stack frame is lost before jumping to a function called in tail position [Clinger 1998]. To preserve tail calls, we end the function's region before performing any tail calls; any local values allocated in the function's region are unavailable for passing to tail calls. (This is distinct from `exclave`, which allows stack allocation in tail positions; if you leave off the `exclave`, tail-position allocations must be on the heap.)

This would prevent most tail-recursive functions, including our `iter` example from Section 2.5, from having local parameters. In our implementation, but not formalized in this paper, we include a regional mode between local and global, representing values that may escape only the current region. This is sufficient to allow `iter` to accept its argument `f` with local mode, since `f` can be given regional mode inside the body of `iter`, and thus safely passed to the tail call.

OCaml currently optimizes all calls in tail position into tail calls. If the programmer wishes to stack-allocate values in the current region and pass them to a call in tail position, we require the programmer to annotate the call to instruct the compiler to not perform a tail-call optimization.

6.4 Currying and Partial Application

Consider a function `f : t1 @ local -> t2 -> t3` and a partial application `f x`. The partial application will, at run-time, allocate a closure that captures both `f` and `x`. Thus, because a closure capturing a local must itself be local, we require `f x` to be at the local mode; it cannot escape from a region. Yet the type of `f` suggests that `f x` is global: there is no local annotation on the tail `t2 -> t3`. (That is, we do not see `f : t1 @ local -> (t2 -> t3) @ local`.)

Because any function that takes a local argument has this problem, we interpret the original type for `f`—with only one `@local`—as meaning the second. That is, all partial applications of a function that takes a local parameter must themselves be local. This happens invisibly to programmers; we can understand this as a slightly-unexpected interpretation of the concrete syntax of function types. It applies to local parameters, as we see here, but also to `once` or `unique` ones, where both of those induce partial applications to be `once`.

Interestingly, the complications here go away if we imagine a change to the language forbidding currying. That is, if a function of type `t1 -> t2 -> t3` were unambiguously a two-argument function, we would not have to propagate mode information down the spines of functions. If we had `f : t1 -> t2 -> t3` and wanted to apply it only to one argument `x`, we could easily write `fun y -> f x y`. This new closure would infer its own result mode, possibly allocating the closure on the stack or possibly on the heap.

Given this simplification in the absence of currying—and the fact that currying does not preserve semantics in the presence of effects—we are planning to experiment with introducing a non-curried function arrow to the language. We see this as a promising direction, removing awkward mode propagation, clarifying the semantics of function applications in the presence of side effects, making closure allocation more apparent, and improving type errors arising from over- or under-application.

6.5 Syntax

We present a postfix syntax for mode information, introduced with `@` for modes and `@@` for modalities. This syntax was motivated by a desire to reduce the number of new keywords in the language: if a mode or modality is always introduced with a special operator, then we have syntactic freedom in the specification of the mode or modality. We can even imagine user-written mode or modality abbreviations, living in a namespace distinct from the many namespaces OCaml already has.

Beyond just the syntax included in this paper, we have designed what we call an *unzipped syntax*, which we believe will make mode-heavy code easier to read. We conjecture that most users, most of the time, will not care deeply about the modes on a function. Instead, when reading a module interface, the programmer will want to see types, not modes. We thus want a syntax where mode information can be separated from type information. For example, consider a `fold` function on local, aliased lists, building a unique result:

```
val fold : ('a @ unique -> 'b @ local -> 'a @ unique) @ local
         -> 'a @ unique -> 'b list @ local -> 'a @ unique
```

A programmer reading this type signature has a hard time finding the types among all the modes. While we will continue to support the syntax as written here, we also plan to support a syntax where all the mode information is placed after all the type information, thus:

```
val fold : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
       @@ (unique -> local -> unique) local -> unique -> local -> unique
```

Now a programmer can easily spot the type independent of the mode. Note that we use a `@@` marker for an unzipped mode signature; these are available only on *type schemes*, not on types within a larger type expression.

6.6 Generalization

Suppose we have a function:

```
let get_x r = r.x
```

What modes should be in the inferred type of `get_x`? Assuming there is no modality on the field `x`, then any modes will do. But, lacking mode polymorphism, we must choose *some* modes. Our answer is to echo OCaml's existing treatment of the value restriction [Wright 1995], and infer a type for `get_x` that is *weakly polymorphic* in its modes.

Weak polymorphism in OCaml describes the situation where inferring a generalized type for a definition cannot commit to a specific type, but instead leaves a type variable to be solved later. Once this type variable is solved for, the resulting type is used at all occurrences of the definition. The canonical example of weak polymorphism is in inferring the type of `let cell = ref []`. The value restriction says that the type of `cell` cannot be `'a list ref`; inferring that polymorphic type would be unsound. Instead, we infer a weakly polymorphic type: the next use of `cell` will determine the type of the contents of the `ref`.

Our `get_x` example is similar: the next use of `get_x` determines the modes in its inferred type. For example, if `get_x` is used to extract the field of a `local` argument, then it will both take and return `local` types, which means it would no longer be suitable to pass as the function argument to `List.map`, say, which takes a function whose argument and return are at the legacy mode.

The fact that a function works only with one set of modes—instead of supporting true mode polymorphism—means that we must sometimes duplicate functions. For example, we might have both a global and a local `get_x` function. As of April 2024, we count 85 functions duplicated across different locality modes in our corporate codebase. This duplication is unfortunate, but a cost well worth paying to support the wide use of locality, as described in more detail in Section 7. A future extension to encompass full mode polymorphism will remove the need for this duplication.

6.7 Modules

A challenge in adding any feature to OCaml is to figure out how it integrates with OCaml's advanced and expressive module system. We can largely sidestep this design challenge as we think about how modes interact with modules: we can see modules as much like records, but with (possibly)

Table 1. Performance of baseline and modified `ocamldep`, showing allocation count per iteration, GC cycles per thousand iterations, and average runtime.

	allocs/iter	GCs/1k iters	time
`baseline`	247 kB	30	485 μs
`locals`	18 kB	2	490 μs
`baseline+extra`	942 kB	131	1131 μs
`locals+extra`	712 kB	98	1038 μs

existential types. In our case, because types are largely independent from modes (except for the modes we write on function types), the existential types inherent in modules present no challenge.

Although we have not yet implemented these extensions, the design we envisage includes the ability for module variables to bind modules at a given mode (just like normal variables) and to assign modalities to module elements (just like record fields).

7 Implementation

Our prototype is work in progress; we detail what is implemented and what is not in this section.

Locality. The locality features described in this paper are fully implemented, including using stack allocation for local values. This feature has met widespread adoption among our colleagues, where the ability to avoid heap allocation has enabled several projects to simplify implementations over what they wrote previously. In some of these applications, requirements are such that garbage collection is simply not acceptable; the code previously had to pre-allocate blocks of memory to be carefully managed during the execution of a low-latency program. Now, these programmers can program in a more functional style, which they describe as increasing their productivity.

As of February 2024, our codebase has 187,765 `.mli` files. Of these, 2,648 have a use of `local` or `global`, with a total of 27,382 occurrences. The feature has been available to programmers for about a year, and we are pleased with this uptake. We looked only in interfaces, not implementations, as the implementations can generally infer locality, whereas interfaces must write it down explicitly.

Uniqueness and Affinity. We have implemented the uniqueness and affinity modes, including alias-tracking uniqueness analysis, but these have not been widely advertised to our colleagues and they are unused outside of our development tests. We have held back encouraging our colleagues to adopt these features because we have not yet implemented memory reuse and our implementation of borrowing was only recently completed.

7.1 Case Study: ocamldep

As a demonstration of the performance benefits of stack allocation enabled by locals, we modified `ocamldep`, a program that ships with the OCaml compiler which computes the dependencies of an OCaml source file for use by a build system. This is a small but nontrivial program (about 600 lines, not counting lexing, parsing and command-line argument processing code that is shared with the OCaml compiler), and is written in an idiomatic mostly-functional style.

We made a number of changes to enable stack allocation, detailed in Appendix E. Primarily, we changed the signature (but not the implementation) of several standard library functions to mark arguments as local, and we marked various calls occurring at the end of a function as not requiring tail-call optimization, as this would otherwise inhibit stack allocation (see Section 6.3). Additionally, in one function we used the global modality to allow part of a structure to be stack-allocated.

These changes reduced heap allocation during dependency computation by more than 90%, as measured by repeatedly running `ocamldep` on the syntax tree of a large OCaml source file. Note that while this decreases the allocation rate and the number of GC cycles significantly (as detailed in Table 1), it does *not* make this simple example run faster. This is to be expected: heap allocation and garbage collection are efficient in OCaml, particularly for the short-lived allocations that are easy to convert to stack allocation. The situation changes when we embed our simple example in a larger program. The latter two lines of Table 1 do some additional work between iterations of dependency computation, by randomly updating a few hundred keys of a long-lived map. This additional work contains no local allocations, and is identical between the baseline and locals versions.

However, when this additional work is present, the version with locals runs approx. 9% faster overall. This is because the reduction in memory allocation leaves more space available for the rest of the program to use between garbage collections. This is the main mechanism by which stack allocation improves performance: while code using stack allocation might not in isolation run any faster than the heap-allocating version, it runs with a smaller footprint, polluting the CPU cache and GC heap much less, leaving more resources for other code and improving system performance.

It's worth noting that some classic compiler optimizations can achieve most of the same gains: aggressive inlining and specialization of iteration functions will remove the need to allocate many closures, and unboxing transformations can remove some additional allocations. However, locals and stack allocations have several advantages: they apply reliably (are visible in types, rather than dependent on inlining heuristics), they do not cause code duplication (unlike inlining), and they do not require a global transformation to use some instances of a type locally (unlike unboxing).

8 Related Work

8.1 Linearity, Affinity and Uniqueness

Substructural type systems based on linear logic force variables to be used exactly once, providing both linearity, since the program cannot alias a value that it is given, and uniqueness, since the values it is given cannot have been aliased. Several languages have used this insight for memory and resource management, by integrating linearity/uniqueness with standard functional programming.

The simplest approach is to separate the linear/unique world (values that cannot be and have not been aliased) from the functional world, by having separate variables for each, sometimes even bound by separate typing contexts. Examples include LNL [Benton and Wadler 1996], the dependent LNL_D [Krishnaswami et al. 2015], and Walker's linear type system [Walker 2005].

Linearity. Desiring less strict separation between the linear/unique and the functional world, many authors have built linear (or affine) rather than unique type systems. Nonlinear values can be turned into a linear ones by promising to use them only once, but allowing this means that linear values may no longer be assumed unique. F° [Mazurak et al. 2010] expresses this with subkinding, an approach also implemented in the Links web programming language [Cooper et al. 2006] in order to support session typing [Lindley and Morris 2017; Tang et al. 2024]. Linear Haskell [Bernardy et al. 2017; Spiwack 2018] also opts for linearity, sharing our goal of allowing unobservable memory reuse, using a substructural type system to prevent duplication of values being updated in-place. Because it does not track uniqueness, the key property allowing safe update—the lack of any aliases—can be assured only when the type of the value is abstract and is produced by a carefully audited interface which ensures uniqueness. That is, the safety of in-place updates is a property of a module boundary and API, not of the type system (although it relies on linearity in the type system). The Alms [Tov and Pucella 2011] and Quill [Morris 2016] systems both use qualified types

(not to be confused with type qualifiers!) to track linearity. They offer precise linear types at the cost of more complex constraints.

Uniqueness. Other systems take the opposite approach, supporting unique rather than linear types. It is always safe to turn a unique value into an aliased value by forgetting its uniqueness, but this means that unique values may be used more than once. Clean [Barendsen and Smetsers 1995, 1996; De Vries et al. 2008] takes this approach, as do Pony [Clebsch et al. 2017] (using capabilities to track aliasing and mutability), and Rust [Matsakis and Klock 2014]. Mezzo [Pottier and Protzenko 2013] uses singleton types to control aliasing less restrictively, allowing multiple references to a unique value but only in statically-tracked ways. However, all uniqueness type systems must contend with the issue detailed in Section 2.2, in that closing over a *unique* value yields a *linear* closure. A common solution is to introduce multiple function types (e.g., `Fn` and `FnOnce` in Rust). We instead follow [Marshall et al. 2022] in supporting both linearity and uniqueness. Rather than statically tracking uniqueness, in-place updates can also be made safe by dynamically detecting uniqueness using reference counting [Didrich et al. 1994; Reinking et al. 2021; Ullrich and de Moura 2019]. A key advantage of that approach is that it can reuse all memory that happens to be unique at runtime, even if this property is hard to track in a type system. However, it provides few guarantees that memory is actually reused at runtime beyond a first-order check [Lorenzen et al. 2023]. Our system could be used to complement reference counting with static guarantees that memory will be reused, even in a higher-order setting.

8.2 Regions and Locality

Stack allocation of memory is efficient, but requires that all references to the memory be gone when the stack is popped. A key line of work in type systems to enforce this invariant is the *region calculus* [Tofte and Talpin 1997] as implemented in MLKit, initially to replace garbage collection and later alongside it [Elsman and Hallenberg 2020; Tofte et al. 2002]. The region calculus is more expressive than our two locality modes, annotating every type with a *region variable*, tracking an arbitrary number of regions. This yields complicated types, though the system can remain usable because such region-annotated types are inferred and rarely user-visible. However, separate compilation is challenging in this setting [Tofte et al. 2004].

Rust [Matsakis and Klock 2014] also uses (user-visible) region variables called *lifetimes*. They can express patterns that our locality modes cannot: for instance, an arena allocator can allocate new values whose lifetime is a particular enclosing region, giving them a status between our `local` and `global`. However, this approach relies on heavy type annotations to specify thread lifetimes.

Instead of a type system, it is possible to implement stack allocation and memory reuse using an escape analysis [Bruynooghe 1986; Park and Goldberg 1992], by examining control-flow at compile time to determine which values are unique (or non-escaping). While such an analysis can be more powerful than a type system, it provides no guarantees to the user that memory will be reused, and often depends on brittle heuristics.

8.3 Borrowing

When a unique value is used multiple times in sequence, one way to track its uniqueness is to thread it through each operation, returning it each time so that the returned value gets used just once. This is the standard approach in Linear Haskell, whose designers proposed using implicit linearity tokens [Spiwack 2023; Spiwack et al. 2022] to make the style more ergonomic.

By contrast, borrowing allows a unique value to be directly used multiple times within a region, and to regain its uniqueness once that region has ended. Rust [Matsakis and Klock 2014] has a sophisticated borrow-checker based on lifetime variables, which is more expressive than our

system but requires heavier annotation. In particular, higher-order functions in Rust that pass newly borrowed values to their callbacks must have higher-rank types [Beingessner et al. 2017], while in our system such functions have rank-1 types that can be fully inferred.

8.4 Modal Type Systems

As discussed in Section 5, our system is closely related to graded modal type systems [Abel and Bernardy 2020; Atkey 2018; Orchard et al. 2019; Wood and Atkey 2022], which are parameterised by an ordered semiring. They are especially suited to comonadic modalities such as the bounded exponential modality from bounded linear logic. Their combination with side-effects, and especially how that interacts with monadic modalities such as our ALIASED modality, is less well studied.

Another modal approach to combining different forms of substructural typing is to use modalities to represent adjunctions between these different forms, as pioneered by Benton [1994] and since generalized to other modalities [Jang et al. 2024; Licata and Shulman 2016; Pruiksma et al. 2018].

Our mode system combines multiple modes and modalities. There are a number of generic multimodal type systems parameterised by some collection of modes and modalities [Gratzer et al. 2020; Shulman 2023], which could in principle model our system, although they do not yet support substructural type systems or side-effects.

The interaction of uniqueness and side-effects is a key aspect of our system absent from much prior work on graded modal types. Other attempts to understand this interaction include the work of Curien et al. [2016], which extends the categorical semantics of Call-By-Push-Value with linearity and the exponential modality, and the work of Torczon et al. [2023], which combines graded modal types with CBPV. Ahman [2023] also uses modal types in a call-by-value setting by restricting the box and unbox rules to values.

9 Future Work

We are exploring extending our system with several new modes which can be used to track other properties of values. In particular, we are interested in modes that track whether values are thread-shared or not, which could enable us to add safe concurrency primitives to OCaml 5. Moreover, we are exploring the use of modes to track user-defined effects. We are also interested in allowing polymorphism over modes.

Acknowledgments

We thank Wenhao Tang and Danel Ahman for helpful discussions and Nick Barnes, Francois Pottier, Guillaume Munch-Maccagnoni and the anonymous reviewers for feedback. We thank Zesen Qian at Jane Street for substantial work on the implementation of type checking for modes. We thank Guillaume Bury, Pierre Chambart, Nathanaëlle Courant, Vincent Laviron at OCamlPro, and Mark Shinwell at Jane Street for extending the stack allocation implementation to a new optimizer. We thank the rest of the OCaml Language team at Jane Street for improving, debugging and maintaining this extended implementation in production. This work was supported by UKRI Future Leaders Fellowship MR/T043830/1 (EHOP).

References

Andreas Abel and Jean-Philippe Bernardy. 2020. A unified view of modalities in type systems. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–28. https://doi.org/10.1145/3408972

Danel Ahman. 2023. When Programs Have to Watch Paint Dry.. In *FoSSaCS*. 1–23. https://doi.org/10.1007/978-3-031-30829-1_1

Robert Atkey. 2018. Syntax and semantics of quantitative type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. 56–65. https://doi.org/10.1145/3209108.3209189

Erik Barendsen and Sjaak Smetsers. 1995. Uniqueness Type Inference. In *Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs*. 189–206. https://doi.org/10.1007/BFb0026821

Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical structures in computer science* 6, 6 (1996), 579–612. https://doi.org/10.1017/S0960129500070109

Aria Beingessner, Steve Klabnik, and Yuki Okushi. 2017. Higher-Rank Trait Bounds (HRTBs) (The Rustonomicon, sec. 3.7). https://doc.rust-lang.org/nomicon/hrtb.html.

Nick Benton and Philip Wadler. 1996. Linear logic, monads and the lambda calculus. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 420–431. https://doi.org/10.5555/788018.788785

P Nick Benton. 1994. A mixed linear and non-linear logic: Proofs, terms and models. In *International Workshop on Computer Science Logic*. Springer, 121–135.

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.* 2, POPL, Article 5 (dec 2017), 29 pages. https://doi.org/10.1145/3158093

Edwin Brady. 2021. Idris 2: Quantitative type theory in practice. *arXiv preprint arXiv:2104.00480* (2021). https://doi.org/10.48550/arXiv.2104.00480

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A core quantitative coeffect calculus. In *European Symposium on Programming Languages and Systems*. Springer, 351–370. https://doi.org/10.1007/978-3-642-54833-8_19

Maurice Bruynooghe. 1986. *Compile time garbage collection.* Katholieke Universiteit Leuven. Departement Computerwetenschappen.

Pritam Choudhury, Harley Eades III, Richard A Eisenberg, and Stephanie Weirich. 2021. A graded dependent type system with a usage-aware semantics. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–32. https://doi.org/10.1145/3434331

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), 1–28. https://doi.org/10.1145/3133896

William D Clinger. 1998. Proper tail recursion and space efficiency. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. 174–185. https://doi.org/10.1145/277650.277719

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web Programming Without Tiers. In *FMCO (Lecture Notes in Computer Science, Vol. 4709)*. Springer, 266–296. https://doi.org/10.1007/978-3-540-74792-5_12

Pierre-Louis Curien, Marcelo Fiore, and Guillaume Munch-Maccagnoni. 2016. A theory of effects and resources: adjunction models and polarised calculi. *ACM SIGPLAN Notices* 51, 1 (2016), 44–56. https://doi.org/10.1145/2837614.2837652

Edsko De Vries, Rinus Plasmeijer, and David M Abrahamson. 2008. Uniqueness typing simplified. In *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers 19*. Springer, 201–218. https://doi.org/10.1007/978-3-540-85373-2_12

Klaus Didrich, Andreas Fett, Carola Gerke, Wolfgang Grieskamp, and Peter Pepper. 1994. OPAL: Design and implementation of an algebraic programming language. In *Programming Languages and System Architectures*. Springer, 228–244. https://doi.org/10.1007/3-540-57840-4_34

Damien Doligez. 2016. Unboxed types. Pull request against OCaml source. https://github.com/ocaml/ocaml/pull/606

Martin Elsman and Niels Hallenberg. 2020. On the effects of integrating region-based memory management and generational garbage collection in ML. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 95–112. https://doi.org/10.1007/978-3-030-39197-3_7

Jeffrey S Foster, Manuel Fähndrich, and Alexander Aiken. 1999. A theory of type qualifiers. *ACM SIGPLAN Notices* 34, 5 (1999), 192–203. https://doi.org/10.1145/301618.301665

Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining effects and coeffects via grading. *ACM SIGPLAN Notices* 51, 9 (2016), 476–489. https://doi.org/10.1145/2951913.2951939

Daniel Gratzer, GA Kavvos, Andreas Nuyts, and Lars Birkedal. 2020. Multimodal dependent type theory. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*. 492–506. https://doi.org/10.1145/3373718.3394736

Junyoung Jang, Sophia Roshal, Frank Pfenning, and Brigitte Pientka. 2024. Adjoint Natural Deduction (Extended Version). *CoRR* abs/2402.01428 (2024).

Shin-ya Katsumata. 2018. A double category theoretic analysis of graded linear exponential comonads. In *Foundations of Software Science and Computation Structures: 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14–20, 2018. Proceedings 21*. Springer, 110–127. https://doi.org/10.1007/978-3-319-89366-2_6

Neelakantan R. Krishnaswami, Cécilia Pradic, and Nick Benton. 2015. Integrating Linear and Dependent Types. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2775051.2676969 http://www.cs.bham.ac.uk/~krishnan/dlnl-paper.pdf.

Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Inf. Comput.* 185, 2 (2003), 182–210. https://doi.org/10.1016/S0890-5401(03)00088-9

Daniel R Licata and Michael Shulman. 2016. Adjoint logic with a 2-category of modes. In *Logical Foundations of Computer Science: International Symposium, LFCS 2016, Deerfield Beach, FL, USA, January 4-7, 2016. Proceedings*. Springer, 219–235. https://doi.org/10.1007/978-3-319-27683-0_16

Sam Lindley and J Garrett Morris. 2017. Lightweight Functional Session Types. In *Behavioural Types: from Theory to Tools*. River Publishers, 265–286.

Anton Lorenzen, Daan Leijen, and Wouter Swierstra. 2023. FP^2: Fully in-Place Functional Programming. *Proceedings of the ACM on Programming Languages* 7, ICFP (2023), 275–304. https://doi.org/10.1145/3607840

Daniel Marshall, Michael Vollmer, and Dominic Orchard. 2022. Linearity and uniqueness: An entente cordiale. In *European Symposium on Programming*. Springer International Publishing Cham, 346–375. https://doi.org/10.1007/978-3-030-99336-8_13

Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. *Ada Lett.* 34, 3 (oct 2014), 103–104. https://doi.org/10.1145/2692956.2663188

Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. 2010. Lightweight linear types in system fdegree. In *TLDI*. ACM, 77–88. https://doi.org/10.1145/1708016.1708027

J. Garrett Morris. 2016. The best of both worlds: linear functional programming without compromise. In *ICFP*. ACM, 448–461. https://doi.org/10.1145/3022670.2951925

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–30. https://doi.org/10.1145/3341714

Young Gil Park and Benjamin Goldberg. 1992. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*. 116–127. https://doi.org/10.1145/143103.143125

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A Calculus of Context-dependent Computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. ACM, 123–135. https://doi.org/10.1145/2628136.2628160

François Pottier and Jonathan Protzenko. 2013. Programming with Permissions in Mezzo. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. ACM, 173–184. https://doi.org/10.1145/2500365.2500598

Klaas Pruiksma, William Chargin, Frank Pfenning, and Jason Reed. 2018. Adjoint logic. *Unpublished manuscript, April* (2018).

Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: Garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 96–111. https://doi.org/10.1145/3453483.3454032

Michael Shulman. 2023. Semantics of multimodal adjoint type theory. *arXiv preprint arXiv:2303.02572* (2023). https://doi.org/10.48550/arXiv.2303.02572

Arnaud Spiwack. 2018. Linear types. A GHC Proposal. https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0111-linear-types.rst

Arnaud Spiwack. 2023. Linear constraints proposal. A GHC Proposal. https://github.com/tweag/ghc-proposals/blob/linear-constraints/proposals/0621-linear-constraints.rst

Arnaud Spiwack, Csongor Kiss, Jean-Philippe Bernardy, Nicolas Wu, and Richard A. Eisenberg. 2022. Linearly qualified types: generic inference for capabilities and uniqueness. *Proc. ACM Program. Lang.* 6, ICFP, Article 95 (aug 2022), 28 pages. https://doi.org/10.1145/3547626

Wenhao Tang, Daniel Hillerström, Sam Lindley, and J. Garrett Morris. 2024. Soundly Handling Linearity. *Proc. ACM Program. Lang.* 8, POPL (2024), 1600–1628. https://doi.org/10.1145/3632896

Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation* 17 (2004), 245–265. https://doi.org/10.1023/B:LISP.0000029446.78563.a4

Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. 2002. Programming with regions in the ML Kit (for version 4). IT University of Copenhagen.

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and computation* 132, 2 (1997), 109–176.

Cassia Torczon, Emmanuel Suárez Acevedo, Shubh Agrawal, Joey Velez-Ginorio, and Stephanie Weirich. 2023. Effects and Coeffects in Call-By-Push-Value (Extended Version). *arXiv preprint arXiv:2311.11795* (2023). https://doi.org/10.48550/arXiv.2311.11795

Jesse A. Tov and Riccardo Pucella. 2011. Practical affine types. In *POPL*. ACM, 447–458. https://doi.org/10.1145/1925844.1926436

Sebastian Ullrich and Leonardo de Moura. 2019. Counting immutable beans: Reference counting optimized for purely functional programming. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*.

1–12. https://doi.org/10.1145/3412932.3412935
David Walker. 2005. Substructural type systems. *Advanced topics in types and programming languages* (2005), 3–44.
James Wood and Robert Atkey. 2022. A Framework for Substructural Type Systems. In *ESOP (Lecture Notes in Computer Science, Vol. 13240)*. Springer, 376–402. https://doi.org/10.1007/978-3-030-99336-8_14
Andrew K. Wright. 1995. Simple Imperative Polymorphism. *LISP and Symbolic Computation* 8 (1995), 343–355. https://doi.org/10.1007/BF01018828

Received 2024-02-28; accepted 2024-06-18