Oxidizing OCaml with Modal Memory Management

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

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.

1 INTRODUCTION

Functional programming languages such as OCaml typically avoid troubling the user over memory management by leveraging automatic garbage collection. The effectiveness of automatic garbage collection is a key factor driving software engineering toward languages that put immutability first, thus aiding reasoning about our code and making it easier to write correct software. However, this comes at a cost:

- **Latency:** Any allocation can trigger a garbage collection cycle, which can lead to unexpected latencies. If programmers wish to write code that is guaranteed to be latency-free, they must avoid heap allocation—usually by tricks such as pre-allocating an array used as an unsafe arena allocator.
- **Cache-Locality:** Allocating a value on the minor heap advances the bump pointer, which may lead to a new cache line being fetched. However, in some cases, we can reuse the space of the value and avoid this cache miss.

To avoid these pitfalls, we wish to mix the safety and convenience of high-level functional programming with the ability to avoid garbage-collected heap allocations.

This paper describes how introducing *modes* to a type system can safely allow the allocation-reducing optimizations of *stack allocation* and *memory reuse*. 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. Indeed, we have implemented the stack allocation idea in a branch of the OCaml compiler, to general acclaim among the programmer colleagues using the feature. We say more about this implementation in Section 7.

Improvements in this direction are necessary in our business for certain performance-critical applications; if we were unable to improve the memory behavior of OCaml programs in this way, we would have to seriously consider switching these applications to Rust or some other lower-level language, thus losing out on the advanced type system and fluid expressiveness inherent in typed functional programming.

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

Authors' addresses: Anton Lorenzen, anton.lorenzen@ed.ac.uk, The University of Edinburgh, UK; Leo White, lwhite@janestreet.com, Jane Street, UK; Stephen Dolan, sdolan@janestreet.com, Jane Street, UK; Richard A. Eisenberg, reisenberg@janestreet.com, Jane Street, USA; Sam Lindley, sam.lindley@ed.ac.uk, The University of Edinburgh, UK.

```
let get_indented_lines filename prefix =
  let lines = read_lines_from_file filename in
  List.map (fun line -> prefix ^ line) lines
```

There are two separate sources of unnecessary allocation here:

(1) A naive implementation will allocate the closure for the function passed to `List.map` on the heap, as it contains a reference to the `prefix` argument to `get_indented_lines`. Yet we know that `List.map` does not save this closure anywhere, and we should be able to de-allocate this closure as soon as `List.map` is complete, without the need for a garbage-collection pass.
(2) The list `lines` is fresh: it was just produced by reading a file. It is not returned from `get_indented_lines` nor saved anywhere. We can thus safely reuse the memory created for it, making `List.map` unobservably destructive.

The techniques in this paper solve both problems, by allowing us to state that the function passed to `List.map` is `local` and that the list returned from a hypothetical `read_lines_from_file` is `unique`. Further examples comprise the bulk of Section 2.

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, where 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.

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 (potentially aliased, or `shared`) 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 c pair -> { 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 have to 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 `shared` value, giving us `unique < shared`. We can thus write code like this:

```
let process_parts_shared (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 `shared`. We still must be careful: any `shared` use of a variable makes *all* uses of that variable `shared`:

```
let process_parts_partial_sharing (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] It is easy to get confused between the uniqueness and affinity modes; it may be helpful to note that both uniqueness modes have 6 letters, while the affinity ones have 4. 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 shared); it is safe to forget this detail. In contrast, affinity is a statement about the future (a value cannot be shared); forgetting it could potentially make memory reuse

[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.

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 an affine value, it suffices to ensure that it is only used once—and thus once itself.

In the examples above, we have annotated values only if they are `unique` or `once`. This is because OCaml values have always been `shared` (without alias tracking, the safe assumption) and `many` (meaning that they are unrestricted). Mode inference (see Section 3.7) propagates modes to un-annotated variables. Where this is impossible, the legacy mode of `shared` or `many` is assumed.

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

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

The `@@ shared` annotation here ensures that even if the list itself is unique, its elements may be shared. 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_shared_elts` is `shared`, so may not. On the other hand, we cannot insert shared elements into a unique `list`, but we can insert shared elements into a unique `list_with_shared_elts`.

Not all modes give rise to a corresponding modality. For instance, it would be unsound to have a shared 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 `shared` and `many`:[2]

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

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

```
(* val graph_nodes : graph -> node shared list @ unique *)
(* val map_to_shared : ('a @ unique -> 'b) -> 'a list @ unique -> 'b list *)
```

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

```
let rev_graph_nodes graph =
  let nodes = graph_nodes graph in
  let rev_nodes = reverse nodes in
  map_to_shared (fun shared_node -> shared_node.s) rev_nodes
```

This function first gets a `unique` list of shared nodes: the list structure itself is unique (the `graph_nodes` function just produced it), but its contents are shared (because of cycles in the graph). We can then 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_shared` to remove the `shared` 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.

```
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]

[3]The `map_to_shared` 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.

[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.

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 shared references from a unique value, and if the references do not leave their scope, we can safely assume uniqueness of the value afterwards again. For instance, we can define a borrow combinator:

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

This erroneous code launders a unique value xs through a borrow, returning it uniquely *and* shared. 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 the unsound assumption that the global modality does not affect uniqueness. In order to ensure borrowed unique values remain local, the global modality incorporates the shared modality. Accordingly, the type definition for 'a global actually wraps a *shared* global value, which is why the unique annotation on the binding site for ys results in a mode error.

3 MODES AS TYPE QUALIFIERS

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:

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

axis	**minimum**	**maximum**	**legacy**	**modality**
affinity (a)	`many`	`once`	`many`	`many`
uniqueness (u)	`unique`	`shared`	`shared`	`shared`
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 @ shared
```

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

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

(`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 shared = { s : 'a @@ shared }
```

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

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:

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

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 $P ::= T \times T \mid T \rightarrow T \ldots$ 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 and T_2 have qualifiers $\leq \mu$. The qualifiers inside computation types have no such constraints: $T_1 \rightarrow T_2$ can have any qualifier. Our system is similar, except we elide modes (qualifiers) from value types: $\tau ::= \tau \times \tau \mid \ldots$; we assume the subparts of a value are always at the same mode, unless specified otherwise.

Instead, we use *modalities* when a subpart of a value is at a different mode from that value. We support three modalities S, M and G, giving us three associated box types $\square^{\mathrm{S}}$, $\square^{\mathrm{M}}$ and $\square^{\mathrm{G}}$. Modalities act on modes, describing the mode of the box's contents given the mode of the box itself:

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

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

For instance, the type of a unique pair of a boxed shared value of type τ_1 and a unique value of type τ_2 is $(\Box^{\mathrm{S}}\tau_1) \times \tau_2$.

3.2 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 store `global` variables, and a `many` closure can only store `many` variables. Thus it must be the case that $a_1 \le a_2$ and $l_1 \le l_2$. But we must also ensure that `unique` variables are not captured by closures that can be invoked multiple times. We do so using the following dagger operation to relate each affinity with its corresponding uniqueness:

$$\textsc{once}^\dagger \coloneqq \textsc{unique} \qquad\qquad \textsc{many}^\dagger \coloneqq \textsc{shared}$$

Then we require that $u_1 \le a_2^\dagger$. 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* 🔒_μ to the context, which we write as $\Gamma, \text{🔒}_\mu$. Unlike the context containment operation of Walker [2005], our $\Gamma, \text{🔒}_\mu$ is not merely a predicate on the bindings of Γ, but may actually change modes to reflect that bindings have become shared:

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

In our inference rules (outlined in Section 3.7 and detailed in Appendix B), we evaluate the context operation *lazily*, by making $\Gamma, \text{🔒}_\mu$ 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.

3.3 Syntax

We define contexts as an ordered list of variables, which occur either with a type and mode $(x : \tau @ \mu)$ or as unusable $(x : -)$:

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

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 $\clubsuit$:

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

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 let $(x, y, z) = e \text{ in } e$ 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 until 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_2 inside a fresh region.

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 $x : \tau\ @\ \mu$, 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:

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

Contexts are ordered as follows:

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

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).

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 us to use a term at a more restrictive mode or assume that variables are given 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).

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

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

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

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

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

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

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

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

Fig. 2. Mode Calculus

The REUSE rule allows us to reuse a space credit to allocate a new pair. We demand that the space credit is UNIQUE and that all components of the pair are `global`. This 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 all 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 SHARED 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* $\Gamma_1, x : \tau_1 @ \mu_1, \Gamma' \vdash e : \tau_2 @ \mu_2$ *and* $\Gamma_2 \vdash v : \tau_1 @ \mu_1$ *and* $\Gamma_1 + \Gamma_2$ *is defined, then* $(\Gamma_1 + \Gamma_2), \Gamma' \vdash e[v/x] : \tau_2 @ \mu_2$.

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:

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

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

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

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

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]. A store semantics differs from a more traditional operational semantics in that substitutions $x \mapsto v$ are not applied immediately, but rather keep the value in the store $b \mapsto v$ under a fresh name b and substitute $x \mapsto b$ instead. Each binding in the store is annotated by a mode μ (or consumed) and its location, either in the global heap or a particular (local) stack frame. We denote addresses by b, c, d (to distinguish them from variables) and a generation n:

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

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:

$$\begin{gathered} - + \mu = \mu \\ (\textsc{many}, \textsc{shared}, l) + (a, \textsc{shared}, l) = (\textsc{many}, u, l) \end{gathered}$$

As usual for usage-aware semantics, $\hat{\mu}_1 + \mu_2$ is non-deterministic, where we can choose to either use a unique value as shared and keep it usable, use it as unique and mark it as consumed $(-)$, or even use it as shared and mark it as consumed $(-)$. We will assume that the + operator always chooses the most permissive mode for its first argument, which excludes the last option.

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 SHARED parts by the + operator). To show the soundness of stack allocation, we keep a counter $n \in \mathbb{N}$ 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 counter. At the end of a region, we delete all bindings that were allocated as local in the current stack frame:

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

We can use this to show the soundness of borrowing as well. To show that no further references remain when a borrowed use ends, we model borrowing by copying the borrowed value into the new stack frame. When the region ends the stack frame is deleted, which allows us to show that no further references to the borrowed value remain as long as we can show that the store has no dangling references.

Our store semantics is modelled using an abstract machine where each state is denoted by $S \mathbin{[\!]} E \rhd_n^{\mu} e$ with a store S, an evaluation context E (stored as a zipper), current mode μ, current stack frame n and expression e. 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, see Figure 9, 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 SHARED. In the reuse step, we overwrite the space credit by the new pair and move it to the end of the store to ensure that it is define 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 SHARED 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 call a store well-typed, if the modes only get more permissive when following pointers in the store, except for boxes and closures, which may also store more restrictive modes. Furthermore a variable should be allocated in the heap if and only if it is global and stack allocated variables can only refer to values in the same or an earlier stack frame. 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

$$
\begin{array}{rrcll}
\text{(unit)} & S \mathrel{[\!]} E \rhd^{\mu}_{n} () & \rightsquigarrow & S, d \mapsto^{\mu}_{n} () \mathrel{[\!]} E \rhd^{\mu}_{n} d & (d \text{ fresh}) \\
\text{(inl)} & S \mathrel{[\!]} E \rhd^{\mu}_{n} \mathrm{inl}\, b & \rightsquigarrow & S, d \mapsto^{\mu}_{n} \mathrm{inl}\, b \mathrel{[\!]} E \rhd^{\mu}_{n} d & (d \text{ fresh}) \\
\text{(inr)} & S \mathrel{[\!]} E \rhd^{\mu}_{n} \mathrm{inr}\, b & \rightsquigarrow & S, d \mapsto^{\mu}_{n} \mathrm{inr}\, b \mathrel{[\!]} E \rhd^{\mu}_{n} d & (d \text{ fresh}) \\
\text{(pair)} & S \mathrel{[\!]} E \rhd^{\mu}_{n} (b, c) & \rightsquigarrow & S, d \mapsto^{\mu}_{n} (b, c) \mathrel{[\!]} E \rhd^{\mu}_{n} d & (d \text{ fresh}) \\
\text{(box)} & S \mathrel{[\!]} E \rhd^{\mu}_{n} \mathrm{box}_{\nu}\, b & \rightsquigarrow & S, d \mapsto^{\mu}_{n} \mathrm{box}_{\nu}\, b \mathrel{[\!]} E \rhd^{\mu}_{n} d & (d \text{ fresh}) \\
\text{(lam)} & S \mathrel{[\!]} E \rhd^{\mu}_{n} \lambda x.e & \rightsquigarrow & S, d \mapsto^{\mu}_{n} \lambda x.e \mathrel{[\!]} E \rhd^{\mu}_{n} d & (d \text{ fresh}) \\
\text{(let)} & S \mathrel{[\!]} E \rhd^{\mu}_{n} \mathrm{let}\, x = b \,\mathrm{in}\, e & \rightsquigarrow & S \mathrel{[\!]} E \rhd^{\mu}_{n} e[x := b] & \\
\text{(app)} & S, b \mapsto^{\hat{\mu}_1 + \mu_2}_{m} \lambda x.e, S' \mathrel{[\!]} E \rhd^{\mu}_{n} b\, c & \rightsquigarrow & S, b \mapsto^{\hat{\mu}_1}_{m} \lambda x.e, S' \mathrel{[\!]} E \rhd^{\mu}_{n} e[x := c] &
\end{array}
$$

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

$$
\begin{array}{l}
\text{(split\_unique)} \quad S, b \mapsto^{\mu_1 + \mu_2}_{m} (c, d), S' \mathrel{[\!]} E \rhd^{\mu}_{n} \mathrm{let}_{\mu_2} (x, y, z) = b \,\mathrm{in}\, e \qquad (\mu_2 = (\_, \textsc{unique}, \_)) \\
\quad \rightsquigarrow S, b \mapsto^{\mu_2}_{m} \clubsuit, S' \mathrel{[\!]} E \rhd^{\mu}_{n} e[x := b, y := c, z := d] \\
\text{(split\_shared)} \quad S, b \mapsto^{\hat{\mu}_1 + \mu_2}_{m} (c, d), S' \mathrel{[\!]} E \rhd^{\mu}_{n} \mathrm{let}_{\mu_2} (x, y, z) = b \,\mathrm{in}\, e \qquad (\mu_2 = (\_, \textsc{shared}, \_)) \\
\quad \rightsquigarrow S, b \mapsto^{\hat{\mu}_1}_{m} (c, d), S' \mathrel{[\!]} E \rhd^{\mu}_{n} e[x := \mathtt{null}, y := c, z := d]
\end{array}
$$

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

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

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

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

with respect to Σ using n stack frames:

$$\frac{\mu = (\text{MANY}, \text{SHARED}, \text{GLOBAL})}{\varnothing :_n (\texttt{null} : \clubsuit \mathbin{@} \mu)}\ \text{WF-BASE} \qquad \frac{S :_m \Sigma_1 \qquad m \le n \qquad m \le k \le n \text{ or } \mu = (\_, \_, \text{GLOBAL})}{(S, b \mapsto^{\mu}_{k} \clubsuit) :_n (\Sigma_1, b : \clubsuit \mathbin{@} \mu)}\ \text{WF-SPACE}$$

$$\frac{S :_m \Sigma_1 \qquad m \le n}{(S, b \mapsto^{-}_{k} v) :_n (\Sigma_1, b : -)}\ \text{WF-UNUSED} \qquad \frac{S :_m (\Sigma_1 + \Sigma_2) \qquad \Sigma_2 \vdash v : \tau \mathbin{@} \mu \qquad m \le n \qquad m \le k \le n \text{ or } \mu = (\_, \_, \text{GLOBAL})}{(S, b \mapsto^{\mu}_{k} v) :_n (\Sigma_1, b : \tau \mathbin{@} \mu)}\ \text{WF-EXT}$$

Notice that this definition allows a variable to be marked unique in the store, even if there are several references to it. However, in the context, all such references may only use the value as shared. 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 $m \le n - k$, where k is the number of borrowing frames in E that need to be traversed to reach the variable.

We now formulate the progress and preservation lemmas (see Appendix A.3 for the proofs). We write $E[e : \tau \mathbin{@} \mu]$ to denote that expression e has type τ and mode μ in the evaluation context E.

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: $S \mathbin{[\!]} E \rhd^{\mu_1}_{n} e \rightsquigarrow S' \mathbin{[\!]} E' \rhd^{\mu_1'}_{n'} e'$.*

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 $S \mathbin{[\!]} E \rhd^{\mu_1}_{n} e \rightsquigarrow S' \mathbin{[\!]} E' \rhd^{\mu_1'}_{n'} e'$, 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 are a popular approach to creating type systems that track how values are used, but it is not immediately obvious how to use them to track uniqueness and affinity in the way that our mode calculus does. In this section we give a translation (by choosing an appropriate partially-ordered semiring) from our mode calculus into a graded modal calculus.

5.1 Modal Axiom K is Incompatible with Call-by-Value

Axiom K is a core axiom of modal logic, stating that if $\square(A \to B)$ and $\square A$, then also $\square B$. In our effectful call-by-value setting, this axiom is unsound. 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:

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

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:

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

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

Our graded calculus uses a fine-grain call-by-value [Levy et al. 2003] formulation in which terms are divided into value terms and computation terms. The full syntax and typing rules are given in Appendix C.

In the types, we include grades on boxes and for the arguments of functions, as usual for graded type theories. Following Abel and Bernardy [2020], but unlike other graded type theories, we order our grades such that if $q \leq r$ 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 $(\Gamma_1 \leq \Gamma_2)$ 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.

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

(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:

$$\{\textsc{unused}\ (\mathbf{0}),\ \textsc{shared once}\ (\mathrm{S}),\ \textsc{shared many}\ (\mathrm{S}\mathbf{M}),$$
$$\textsc{unique once}\ (\mathbf{1}),\ \textsc{unique many}\ (\mathbf{M}),\ \textsc{unique error}\ (\bot)\}$$

The ordering on the semiring corresponds to the ordering on contexts in our mode calculus. We choose SHARED 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 S should behave like the S modality and multiplying by $\mathbf{M}$ like the M modality. The rest of multiplication is forced by the semiring laws. Interesting equations include:

$$\begin{array}{lllll} \mathrm{S}+\mathrm{S}=\mathrm{S}\mathbf{M} & \mathbf{1}+\mathrm{S}=\bot & \mathrm{S}\cdot\mathrm{S}=\mathrm{S} & \mathrm{S}\cdot\mathbf{M}=\mathrm{S}\mathbf{M} & \mathrm{S}\cdot\bot=\mathrm{S}\mathbf{M} \\ \mathbf{1}+\mathbf{1}=\bot & \mathbf{M}+\mathbf{M}=\bot & \mathbf{M}\cdot\mathbf{M}=\mathbf{M} & \mathbf{M}\cdot\mathrm{S}=\mathrm{S}\mathbf{M} & \mathbf{M}\cdot\bot=\bot \end{array}$$

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 Sharable Closures

The naive semiring gives us a sound system, but one that prevents functions from being applied once they have been shared. 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 produce a function at grade 1. In particular, if V is a variable x then Γ_1 will be $\{x :^{\mathbf{1}}\ {}^{q}A \to B\}$. 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 $\mathrm{box}_{\mathrm{S}}$ construct for computations. That means that, given a binding $x :^{\mathrm{S}}\ {}^{q}A \to B$ 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 $\mathrm{box}_{\mathrm{S}}$ expression so that it no longer requires x uniquely:

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

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

Both these problems would be addressed if we had a type constructor R such that $\Box_{\mathrm{S}} \circ R$ was a comonad. This would allow us to use $R\,(A \to B)$ as the type of functions that can still be applied after they have been shared. Similarly, placing such a function in a $\mathrm{box}_{\mathrm{S}}$ expression would still require the values in its context at $\Box_{\mathrm{S}} \circ R$, which is at least as strong as the original requirements.

If we had an element $\mathrm{S} \to \mathbf{1}$ in our semiring that acted as a right residual to S:

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

then $\Box_{\mathrm{S}\to\mathbf{1}}$ would be a right adjoint to $\Box_{\mathrm{S}}$, and $\Box_{\mathrm{S}} \circ \Box_{\mathrm{S}\to\mathbf{1}}$ would be the desired comonad.

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

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 additional elements:

$$\{\textsc{unshared once}\ (\mathrm{S} \to \mathbf{1}),\ \textsc{unshared many}\ (\mathrm{S} \to \mathbf{M}),\ \textsc{unshared error}\ (\mathrm{S} \to \bot)\}$$

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

$$\begin{array}{lll} \mathrm{S} \to \mathbf{1} + \mathrm{S} \to \mathbf{1} = \mathrm{S} \to \bot & \mathrm{S} \cdot (\mathrm{S} \to \mathbf{1}) = \mathrm{S} \to \mathbf{1} & \mathbf{M} \cdot (\mathrm{S} \to \mathbf{1}) = \mathrm{S} \to \bot \\ \mathrm{S} \to \mathbf{1} + \mathbf{1} = \mathrm{S} \to \bot & (\mathrm{S} \to \mathbf{1}) \cdot \mathrm{S} = \mathrm{S} & (\mathrm{S} \to \mathbf{1}) \cdot \mathbf{M} = \mathrm{S} \to \mathbf{M} \end{array}$$

$$\begin{tikzcd}
 & & & \textsc{unused}(0) & & \\
 & & & \textsc{shared once}(\mathrm{S}) \arrow[u] & & \\
 & & \textsc{unique once}(\mathbf{1}) \arrow[ur] & & \textsc{shared many}(\mathbf{SM}) \arrow[ul] & \\
 & \textsc{unshared once}(\mathrm{S} \to \mathbf{1}) \arrow[ur] & & \textsc{unique many}(\mathbf{M}) \arrow[ul] \arrow[ur] & & \\
 & & \textsc{unshared many}(\mathrm{S} \to \mathbf{M}) \arrow[ul] \arrow[ur] & & \textsc{unique error}(\bot) \arrow[ul] & \\
 & & & \textsc{unshared error}(\mathrm{S} \to \bot) \arrow[ul] \arrow[ur] & &
\end{tikzcd}$$

Fig. 4. Ordering of the Extended Semiring

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:

$$[\![\tau_1 \mathbin{@} \mu_1 \longrightarrow \tau_2 \mathbin{@} \mu_2]\!] = \Box^{(\mathrm{S}\to\mathbf{1})}\,({}^{(\mathrm{S}\to\mathbf{1})\cdot[\![\mu_1]\!]}[\![\tau_1]\!] \longrightarrow \Box^{(\mathrm{S}\to\mathbf{1})\cdot[\![\mu_2]\!]}[\![\tau_2]\!]))$$

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

Similarly, contexts are translated as:

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

with the grades on variable bindings multiplied by $\mathrm{S} \to \mathbf{1}$.

We translate judgements $\Gamma \vdash e : t@\mu$ into judgements of the form $[\![\Gamma]\!] \vdash^{c} M : \Box^{(\mathrm{S}\to\mathbf{1})\cdot[\![\mu]\!]}[\![t]\!]$. The rules are quite mechanical, so for brevity we show only a few of instances of the translation:

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

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

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:

- In the right-hand side of a **let**: In `let x = &y in expr`, both x and y can be used as `shared` within `expr`. However, y remains in scope after the **let** is done evaluating, and it can be used `uniquely`.
- In a function argument: We treat `f ... &y ...` just like `let x = &y in f ... x ....` (If the same variable is borrowed multiple times in the same function call, only one borrow is inferred.)

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 $⨟$ 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:

$$(\Gamma_1, x : \tau \mathbin{@}^{\mathbb{B}} (\_, \mathtt{shared}, l)) ⨟ (\Gamma_2, x : \tau \mathbin{@}^{b} (\_, u, l)) = (\Gamma_1 ⨟ \Gamma_2), x : \tau \mathbin{@}^{b} (\mathtt{many}, u, l)$$

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.

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

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

$$\lceil \varnothing \rceil = \varnothing \qquad\qquad \lfloor \varnothing \rfloor = \varnothing$$

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

Fig. 5. Rules for Borrowing

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 available as shared 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 shared. 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. It can be written only 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
    let n = n - 1 in exclave loop n (f n :: 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.

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 a region.

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. In cases where 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 experimenting with the idea of introducing a non-curried function arrow to the language. This is solidly future work, but we see it 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

This paper presents a postfix syntax for mode information, introduced with `@` for modes and `@@` for modalities. Writing modes postfix 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.

(OCaml uses `@` and `@@` in expressions as list-append and function application, respectively. Our syntax, though, appears only in types and patterns, where these symbols are unused.)

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, shared 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. Accordingly, despite the fact that we have used `@` and `@@` to discriminate between modes and modalities in this paper, the final design actually distinguishes between inline (or zipped) mode annotations and a mode/modality that applies to an entire type.

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 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`.

It is the same with our `get_x` example: the next usage of `get_x` will determine 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 expect and return `local` types, which means it would no longer be suitable to pass as the function argument to the standard `List.map`, say, which expects a function whose argument and return are at the legacy mode.

A future expansion of our ideas to encompass full mode polymorphism will remove the need for this extra little complication.

7 IMPLEMENTATION

We have included our prototype implementation as part of our submission. It is a work in progress; we detail what is implemented and what is not in this section. This is current as of the end of February 2024.

Locality. The features around locality 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. We believe these features are unused outside of our development tests. The reason we have held back encouraging our colleagues to adopt these features is that we are still in the process of implementing memory reuse; without that key feature, the motivation for uniqueness and affinity is less than we would like, and so we have not pushed the features.

Syntax. The syntax we have implemented is different from what is described in this paper. It is, in fact, our experience getting feedback from programmers that informed the ideas in Section 6.5. The implemented syntax puts mode annotations in prefix, using new keywords. A noteworthy advantage of working in a branch of the compiler—instead of eagerly trying to include our innovations in the main OCaml compiler—is that we can test out syntax, gather feedback, and then iterate.

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 share a value that it is given, and uniqueness, since the values it is given cannot have been shared. There have been many languages and calculi which use this insight to model memory and resource management, by integrating linearity/uniqueness with standard functional programming.

The simplest approach is to entirely separate the linear/unique world (containing values that cannot be and have not been shared) 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 chosen to build linear (or affine, if usage is not mandatory) rather than unique type systems. It is always safe to turn a nonlinear value into a linear one by promising to use it 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. Moreover, that work also relies on adding substructural aspects to a type system to prevent duplication of the value to be 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 (though support for linearity in the type system is a necessary component).

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 more precise linear types than many other systems 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 a shared value by forgetting that it is unique, 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 does 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. Different systems deal with this in different ways, often by introducing multiple function types (for instance, `Fn` and `FnOnce` in Rust). Instead, we follow [Marshall et al. 2022] in supporting both linearity and uniqueness, and use locks to ensure that closing over unique values yields linear closures.

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 major 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 to provide static guarantees that memory will be reused; even in a higher-order setting.

8.2 Regions and Locality

Stack allocation of memory is very efficient, but requires that all references to the memory be gone when the stack is popped. A major line of work in type systems to enforce this 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 much more expressive than our `local` and `global` modes, introducing an arbitrary number of regions named by *region variables* with which every type is annotated. This resulted in complicated types, although this was less of a concern than usual as these region-annotated types were produced during compilation and rarely became user-visible. However, these complicated types made separate compilation more difficult. [Tofte et al. 2004].

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]. Such an analysis examines the control-flow at compile time to determine which values are guaranteed to be 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 can be brittle depending on the exact heuristics used.

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 improve this aspect of the user interface; these tokens allow the user to elide the threading.

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, which tracks the precise region in which a borrowed value may be used using *lifetime variables*. This is much more expressive than our system (which tracks only `local` and `global`), but requires much heavier annotation to thread lifetime variables through types. 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]. These systems are parameterised by some form of ordered semiring and can support a wide array of uses. 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 SHARED 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 have been a number of attempts to define generic multimodal type systems that are parameterised by some collection of modes and modalities, including Multimodal Dependent Type Theory [Gratzer et al. 2020] and Multimodal Adjoint Type Theory [Shulman 2023]. Such theories do not yet support substructural type systems or side-effects, but could potentially model a system such as ours.

Our system combines both uniqueness and side-effects, which is a key part of what takes our design away from much of the existing graded modal types work. Other attempts to understand the interaction between these features 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 Call-By-Push-Value.

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.

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.

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.

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.

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.

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.

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

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.

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.

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.

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.

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.

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.

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.

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.

Jeffrey S Foster, Manuel Fähndrich, and Alexander Aiken. 1999. *ACM SIGPLAN Notices* 34, 5 (1999), 192–203.

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.

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

Neelakantan R. Krishnaswami, Cécilia Pradic, and Nick Benton. 2015. Integrating Linear and Dependent Types. In *Principles of Programming Languages (POPL)*. 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.

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.

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.

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.

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.

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

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.

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.

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.

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.

Michael Shulman. 2023. Semantics of multimodal adjoint type theory. *arXiv preprint arXiv:2303.02572* (2023).

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.

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.

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).

Jesse A. Tov and Riccardo Pucella. 2011. Practical affine types. In *POPL*. ACM, 447–458.

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.
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.
Andrew K. Wright. 1995. Simple Imperative Polymorphism. *LISP and Symbolic Computation* 8 (1995), 343–355. https://doi.org/10.1007/BF01018828

A PROOFS

A.1 Modes as Type Qualifiers

We write that $\Gamma_1 + \Gamma_2$ is defined if for all variables x occurring in both Γ_1 and Γ_2 at modes (a_1, u_1, l_1) and (a_2, u_2, l_2) respectively, we have $u_1 = u_2 = \textsc{shared}$ and $l_1 = l_2$.

Lemma A.1. *If $(\Gamma_1 + \Gamma_2) + (\Gamma_3 + \Gamma_4)$ is defined, then $\Gamma_i + \Gamma_j$ is defined for $i < j$.*

Proof. Clear. □

Lemma A.2 (Context joining is monotone). *If $\Gamma_1 + \Gamma_2$ is defined, then $\Gamma_1 \geq \Gamma_1 + \Gamma_2$ and $\Gamma_2 \geq \Gamma_1 + \Gamma_2$.*

Proof. By induction on the contexts. If x occurs in both Γ_1 and Γ_2 at modes (a_1, u_1, l_1) and (a_2, u_2, l_2) respectively, then it occurs in $\Gamma_1 + \Gamma_2$ at mode (a_3, u_3, l_3) with $l_1 = l_2 = l_3$ and $a_3 = \textsc{many}$ and $u_1 = u_2 = \textsc{shared}$. Thus $(a_1, u_1, l_1) \geq (a_3, u_3, l_3)$ and $(a_2, u_2, l_2) \geq (a_3, u_3, l_3)$ □

Lemma A.3 (Weakening variables). *If $\Gamma, \Gamma' \vdash e : \tau_2 \mathbin{@} \mu_2$, then $\Gamma, x : \tau_1 \mathbin{@} \mu_1, \Gamma' \vdash e : \tau_2 \mathbin{@} \mu_2$.*

Proof. By straightforward induction on the typing derivation. □

Lemma A.4 (Monotonicity of locks). *If $\Gamma \leq \Gamma'$ and $\mu \geq \mu'$, then $\Gamma, \text{🔒}_{\mu} \leq \Gamma', \text{🔒}_{\mu'}$*

Proof. Induction on the length of Γ, Γ'. The nontrivial case is a variable x present in both Γ and Γ', where we have:

$$\begin{aligned} x &: (a_1, u_1, l_1) \in \Gamma & \mu &= (a_2, u_2, l_2) \\ x &: (a_1', u_1', l_1') \in \Gamma' & \mu' &= (a_2', u_2', l_2') \\ (a_1, u_1, l_1) &\leq (a_1', u_1', l_1') & (a_2', u_2', l_2') &\leq (a_2, u_2, l_2) \end{aligned}$$

If the variable is present in $\Gamma', \text{🔒}_{\mu'}$ (because $a_1' \leq a_2'$ and $l_1' \leq l_2'$) we must show it is also in $\Gamma, \text{🔒}_{\mu}$ with a submode, that is we need to prove:

$$(a_1, u_1 \vee a_2^{\dagger}, l_1) \leq (a_1', u_1' \vee a_2'^{\dagger}, l_1') \qquad a_1 \leq a_2 \qquad l_1 \leq l_2$$

which follow by transitivity from the assumptions above and the antimonotonicity of $(-)^{\dagger}$. □

Since $\Gamma = \Gamma, \text{🔒}_{\textsc{once,shared,local}}$ by definition of locking and this is the largest mode, we have as a corollary of the above that $\Gamma \leq \Gamma, \text{🔒}_{\mu}$.

As another corollary, if $\Gamma, \text{🔒}_{\mu_1} \vdash e : \tau \mathbin{@} \mu_2$ and $\mu_1 \leq \mu_1'$, then $\Gamma, \text{🔒}_{\mu_1'} \vdash e : \tau \mathbin{@} \mu_2$ by application of the sub rule.

Lemma A.5 (Duplicating values). *If $\Gamma \vdash v : \tau \mathbin{@} (\textsc{many}, u, l)$ then $\Gamma = \Gamma_1 + \Gamma_2$ and $\Gamma_i \vdash v : \tau \mathbin{@} (\textsc{many}, \textsc{shared}, l)$.*

Proof. We obtain Γ_1, Γ_2 by using the same affinity and locality as in Γ for all free variables of v and setting their uniqueness to shared. The first claim then follows by sequencing join. We show the second claim by induction on the typing derivation of v. For variables, this follows since we use the same affinity and locality and use the variable as shared. For closures, a $\text{🔒}_{(\textsc{many}, u, l)}$ is applied to their body which turns all unique bindings into shared, so any variables used by the closure must be present with the same modes in Γ_i. For other value constructors, it follows directly from the inductive hypothesis. □

Lemma A.6 (Substitution). *If $\Gamma_1, x : \tau_1 \mathbin{@} \mu_1, \Gamma_2 \vdash e : \tau_2 \mathbin{@} \mu_2$ and $\Gamma' \vdash v : \tau_1 \mathbin{@} \mu_1$ and $\Gamma_1 + \Gamma'$ is defined, then $(\Gamma_1 + \Gamma'), x : -, \Gamma_2 \vdash e[v/x] : \tau_2 \mathbin{@} \mu_2$.*

Proof. By induction on the typing derivation of e.

- Case VAR, matching variables: We have $\Gamma' \vdash v : \tau_2 @ \mu_2$ and we we show that $(\Gamma_1 + \Gamma'), \Gamma_2 \vdash v : \tau_1 @ (a_2, u_2, l_2)$ by first using monotonicity of context joining and then weakening variables repeatedly.
- Case VAR, mismatching variables: Clear.
- Case SUB: If x is weakened $\mu_1 \leq \mu_3$, then use the SUB-rule to obtain $\Gamma' \vdash v : \tau_1 @ \mu_3$ and apply the inductive hypothesis. If x becomes unused, the claim follows directly.
- Case PAIR: If x is only used in one component, use the inductive hypothesis on that component. If x is used in both components and is split, use the duplicating values and invoke the inductive hypothesis.
- Case LAM: The inductive hypothesis yields:

$$((\Gamma_1, 🔒_{\mu_2}) + \Gamma'), x : -, (\Gamma_2, 🔒_{\mu_2}) \vdash (\lambda y.e)[v/x] : \tau_2 @ \mu_2$$

  from which the result follows by the SUB rule since $\Gamma_i \leq \Gamma_i, 🔒_{\mu_2}$.
- All other cases are clear, since they do not modify the context beyond adding new variables, or joining contexts (which can be done analogous to the PAIR rule).

□

A.2 Borrowing

A.2.1 Approach. We first recast our **let**$y = \&x$ **in** e construct into a more general borrow! $y = (x_1, \ldots, x_n)$ for e construct, allowing simultaneous borrowing. (This transformation is not necessary for this proof, but instead relates the simpler surface construct to a more general one that we have already worked with.)

We need to be able to access the continuation of the `borrow!` construct. We can do this by transforming the program into A-normal form (ANF), which floats all compound computations out of let-bindings and thus makes the continuation explicit. Returning to our introductory example, we can desugar the program as follows:

$$\begin{array}{l} \text{case}\, b \\ \quad \text{inl}\, y \rightarrow \text{borrow! } z = z \text{ for } x = (f\, z) \text{ in } h\,(x, z) \\ \quad \text{inr}\, y \rightarrow \text{borrow! } z = z \text{ for } x = (g\, z) \text{ in } h\,(x, z) \end{array}$$

...where the final computation $h\,(x, z)$ got duplicated into both branches. To achieve this, we perform a standard ANF transformation, which can easily be shown to be type-preserving in our extended calculus:

LEMMA A.7 (NORMALIZATION TO ANF IS TYPE-PRESERVING). *If* $\Gamma \vdash e_1 : \tau @ \mu$ *and* $e_1 \rightsquigarrow_{ANF} e_2$, *then* $\Gamma \vdash e_2 : \tau @ \mu$.

Unfortunately, though, a translation to ANF is not quite enough. This is because a standard ANF transformation can not normalize the `borrow!` construct. In particular, if `borrow!`'s are nested, the inner `borrow!` can not be moved to obtain full access to its continuation:

$$\text{let}\, x = (\text{borrow! } y = \overline{y} \text{ for } \text{ let}\, z = e_1 \text{ in borrow! } z' = \overline{z'} \text{ for } h\,(y, z, z')) \text{ in } e_2$$

Here, the borrow of $z' = \overline{z'}$ should allow us to use $\overline{z'}$ again in e_2. However, to achieve this with our original borrowing rule, we would need to ensure that the borrow of $z' = v_2$ and e_2 are in the same scope. Unfortunately, there is no easy transformation that would allow us to do this (e.g., e_2 can not be moved into the scope of the borrow since it may not be global). Instead, we will explicitly pass the borrowed variables out of their scope to obtain a term in our mode calculus:

$$\begin{aligned}&\text{borrow! } y = \overline{y} \text{ for } x = (\text{let}\, z = e_1 \,\text{in}\, \text{borrow! } z' = \overline{z'} \text{ for } x' = h\,(y, z, z') \text{ in } (x', \overline{z'}))\\ &\qquad\qquad \text{in let}\, \overline{y} = y \,\text{in}\, \text{let}\, (\_, x, \overline{z'}) = x \,\text{in}\, e_2\end{aligned}$$

This is possible since our BORROW rule ensures that borrowed variables are global and so are allowed to escape the region. We can show that the transformed program can then be checked in the mode calculus:

LEMMA A.8 (PASSING BORROWED VARIABLES IS TYPE-PRESERVING). *If* $\Gamma \vdash e_1 : \tau @ \mu$ *and* $e_1 \rightsquigarrow_{pass} e_2$, *then* $\Gamma \vdash e_2 : \tau @ \mu$.

A.2.2 Proofs.

LEMMA A.9 (CONTEXT JOINING IS ASSOCIATIVE). $\Gamma_1 \fatsemi (\Gamma_2 \fatsemi \Gamma_3) = (\Gamma_1 \fatsemi \Gamma_2) \fatsemi \Gamma_3$

PROOF. By induction over the contexts. We only consider the cases where a variable occurs in all three contexts with a mode and borrowing annotation. If a variable is omitted in at least one of the three contexts, associativity follows easily. For the purpose of this proof, we will consider a mode μ as a collection of three functions $a(u), l(u), b$ that map uniqueness to affinity, locality and a borrowing tag $\in \{\mathbb{B}, \mathbb{O}\}$—corresponding to the flow of information.

Our sequencing join operation is then written as follows:

$$\begin{aligned} seq(a_1, a_2)(u) &= \text{MANY}\\ seq(l_1, l_2)(u) &= l_1(\text{SHARED}) \wedge l_2(\text{SHARED})\\ seq(b_1, b_2) &= \mathbb{O}\end{aligned}$$

And the borrowing join operation is written as follows:

$$\begin{aligned} bor(a_1, a_2)(u) &= \text{MANY}\\ bor(l_1, l_2)(u) &= l_1(\text{SHARED}) \wedge l_2(u)\\ bor(\mathbb{B}, b_2) &= b_2\end{aligned}$$

Given as input affinities (a_1, a_2, a_3), uniqueness u and localities (l_1, l_2, l_3) and borrowing tags (b_1, b_2, b_3), we have to consider four cases, depending on whether (b_1, b_2) are borrowing:

$$\begin{aligned} seq(seq(a_1, a_2), a_3)(u) &= \text{MANY}\\ &= seq(a_1, seq(a_2, a_3))(u)\\ seq(seq(l_1, l_2), l_3)(u) &= seq(l_1, l_2)(\text{SHARED}) \wedge l_3(\text{SHARED})\\ &= (l_1(\text{SHARED}) \wedge l_2(\text{SHARED})) \wedge l_3(\text{SHARED})\\ &= l_1(\text{SHARED}) \wedge (l_2(\text{SHARED}) \wedge l_3(\text{SHARED}))\\ &= l_1(\text{SHARED}) \wedge seq(l_2, l_3)(\text{SHARED})\\ &= seq(l_1, seq(l_2, l_3))(u)\\ seq(seq(\mathbb{O}, \mathbb{O}), b_3) &= \mathbb{O}\\ &= seq(\mathbb{O}, seq(\mathbb{O}, b_3))\end{aligned}$$

$$\begin{aligned}
seq(seq(a_1, a_2), a_3)(u) &= \textsc{many} \\
&= seq(a_1, bor(a_2, a_3))(u) \\
seq(seq(l_1, l_2), l_3)(u) &= seq(l_1, l_2)(\textsc{shared}) \land l_3(\textsc{shared}) \\
&= (l_1(\textsc{shared}) \land l_2(\textsc{shared})) \land l_3(\textsc{shared}) \\
&= l_1(\textsc{shared}) \land (l_2(\textsc{shared}) \land l_3(\textsc{shared})) \\
&= l_1(\textsc{shared}) \land bor(l_2, l_3)(\textsc{shared}) \\
&= seq(l_1, bor(l_2, l_3))(u) \\
seq(seq(\mathbb{O}, \mathbb{B}), b_3) &= \mathbb{O} \\
&= seq(\mathbb{O}, bor(\mathbb{B}, b_3))
\end{aligned}$$

$$\begin{aligned}
seq(bor(a_1, a_2), a_3)(u) &= \textsc{many} \\
&= bor(a_1, seq(a_2, a_3))(u) \\
seq(bor(l_1, l_2), l_3)(u) &= bor(l_1, l_2)(\textsc{shared}) \land l_3(\textsc{shared}) \\
&= (l_1(\textsc{shared}) \land l_2(\textsc{shared})) \land l_3(\textsc{shared}) \\
&= l_1(\textsc{shared}) \land (l_2(\textsc{shared}) \land l_3(\textsc{shared})) \\
&= l_1(\textsc{shared}) \land seq(l_2, l_3)(u) \\
&= bor(l_1, seq(l_2, l_3))(u) \\
seq(bor(\mathbb{B}, \mathbb{O}), b_3) &= \mathbb{O} \\
&= bor(\mathbb{B}, seq(\mathbb{O}, b_3))
\end{aligned}$$

$$\begin{aligned}
bor(bor(a_1, a_2), a_3)(u) &= \textsc{many} \\
&= bor(a_1, bor(a_2, a_3))(u) \\
bor(bor(l_1, l_2), l_3)(u) &= bor(l_1, l_2)(\textsc{shared}) \land l_3(u) \\
&= (l_1(\textsc{shared}) \land l_2(\textsc{shared})) \land l_3(u) \\
&= l_1(\textsc{shared}) \land (l_2(\textsc{shared}) \land l_3(u)) \\
&= l_1(\textsc{shared}) \land bor(l_2, l_3)(u) \\
&= bor(l_1, bor(l_2, l_3))(u) \\
bor(bor(\mathbb{B}, \mathbb{B}), b_3) &= b_3 \\
&= bor(\mathbb{B}, seq(\mathbb{B}, b_3))
\end{aligned}$$

□

We write that $\Gamma_1 \fatsemi \Gamma_2$ is defined if for all variables x occurring in both Γ_1 and Γ_2 at modes $(a_1, u_1, l_1)^{b_1}$ and $(a_2, u_2, l_2)^{b_2}$ respectively, we have $u_1 = \textsc{shared}$ and either $u_2 = \textsc{shared}$ or $b_1 = \mathbb{B}$.

Lemma A.10. *If $(\Gamma_1 \fatsemi \Gamma_2) \fatsemi (\Gamma_3 \fatsemi \Gamma_4)$ is defined, then $\Gamma_i \fatsemi \Gamma_j$ is defined for $i < j$.*

Proof. By induction on the contexts. By the assumption, we have that a variable is unique in Γ_i only if it is unused in all Γ_j $(i < j)$. Similarly, a variable is only unique in Γ_j if it is borrowed in all Γ_i $(i < j)$. □

Lemma A.11 (Context joining is monotone). *If* $\Gamma_1 ⨟ \Gamma_2$ *is defined, then* $\Gamma_1 \geq \Gamma_1 ⨟ \Gamma_2$ *and* $\Gamma_2 \geq \Gamma_1 ⨟ \Gamma_2$.

Proof. By induction on the contexts. Obvious for the empty context and if (at least) one of the variables is unused. Otherwise, we have $a \geq \textsc{many}$ for any affinity a and $l_i \geq l_1 \wedge l_2$ for any locality l_i (i=1,2). □

Lemma A.12 (Borrowing values). *If* $\Gamma \vdash v : \tau \mathbin{@} (\textsc{many}, u, l)$ *then* $\Gamma = \lceil \Gamma_1 \rceil ⨟ \Gamma_2$ *and* $\lceil \Gamma_1 \rceil \vdash v : \tau \mathbin{@} (\textsc{many}, \textsc{shared}, l)$ *and* $\Gamma_2 \vdash v : \tau \mathbin{@} (\textsc{many}, u, l)$.

Proof. We obtain Γ_1 from Γ by making all variables shared. We let Γ_2 be Γ. Then we can see the first claim by induction on the typing derivation of v. For variables, this follows directly. For closures, this follows from the duplicating closures lemma. For all other value constructors it follows directly from the inductive hypothesis. □

Lemma A.13 (Substitution). *If* $\Gamma_1, x : \tau_1 \mathbin{@} \mu_1, \Gamma_2 \vdash e : \tau_2 \mathbin{@} \mu_2$ *and* $\Gamma' \vdash v : \tau_1 \mathbin{@} \mu_1$ *and* $\Gamma_1 ⨟ \Gamma'$ *is defined, then* $(\Gamma_1 ⨟ \Gamma'), \Gamma_2 \vdash e[v/x] : \tau_2 \mathbin{@} \mu_2$.

Proof. By induction on the typing derivation of e, using as a second inductive hypothesis that: If $\Gamma_1, x : \tau_1 \mathbin{@^{\mathbb{B}}} \mu_1, \Gamma_2 \vdash e : \tau_2 \mathbin{@} \mu_2$ and $\lceil \Gamma' \rceil \vdash v : \tau_1 \mathbin{@} \mu_1$ and $\Gamma_1 ⨟ \lceil \Gamma' \rceil$ is defined, then $(\Gamma_1 ⨟ \lceil \Gamma' \rceil), \Gamma_2 \vdash e[v/x] : \tau_2 \mathbin{@} \mu_2$.

- Case var, matching variables: Since x is not borrowed in the var-rule, the first premise applies. Then use the weakening variables lemma repeatedly to obtain exactly Γ'.
- Case var, mismatching variables: Clear.
- Case sub: If x is weakened $\mu_1 \leq \mu_3$, then use the sub-rule to obtain $\Gamma' \vdash v : \tau_1 \mathbin{@} \mu_3$. If x becomes unused, the claim follows directly.
- Case pair: If x is only used in one component, use the inductive hypothesis on that component. If x is used in both components and is split using a borrowing join, then use the borrowing values lemma.
- Case lam: x can not be marked borrowed in the context, since the context of the lam rule was applied to the $\lfloor \Gamma \rfloor$ operation. If x occurs as owned, then we can invoke either inductive hypothesis, depending on x was marked as borrowed before this annotation was removed. The inductive hypothesis yields:

  $$((\Gamma_1, \text{🔒}_{\mu_2}) + \Gamma'), x : -, (\Gamma_2, \text{🔒}_{\mu_2}) \vdash (\lambda y.e)[v/x] : \tau_2 \mathbin{@} \mu_2$$

  from which the result follows by the sub rule since $\Gamma_i \leq \Gamma_i, \text{🔒}_{\mu_2}$.
- Case region: If x is borrowed in this region, then to show the second hypothesis, use the first inductive hypothesis.
- All other cases are clear, since they do not modify the context beyond adding new locks or variables, or joining contexts (which can be done analogous to the pair rule).

□

We show the transformation into ANF in two steps, by first normalizing to fine-grain call-by-value. In our translation, we will accumulate the tail context of an expression, that contains all computations in the order in which they occur in the program:

$$E ::= ? \mid \text{let}\, x = e \,\text{in}\, E$$

Lemma A.14 (Splitting of tail contexts). *If* $\Gamma_1 \vdash E[e] : \tau \mathbin{@} \mu$, *then* $\Gamma_1 = \Gamma_2 ⨟ \Gamma_3$ *and* $\Gamma_2 \vdash E[()] : \mathbb{1} \mathbin{@} \mu$ *and* $\Gamma_3, \Gamma_4 \vdash e : \tau \mathbin{@} \mu$, *where* Γ_4 *contains all variables bound in E.*

Proof. By induction on E.

- If $E = ?$, then let Γ_2 contain all variables from Γ_1 as unused. Let Γ_3 be Γ_1 and Γ_4 be the empty context. Then the claim follows.
- Let $E = \text{let}\, x = e' \,\text{in}\, E'$. Then $\Gamma_1 = \Gamma_1' \fatsemi \Gamma_1''$, and $\Gamma_1' \vdash e'$ and $\Gamma_1'', x \vdash E'[e] : \tau @ \mu$. By the induction hypothesis, we obtain $(\Gamma_1'', x) = (\Gamma_2, x) \fatsemi (\Gamma_3, x)$ and $\Gamma_2, x \vdash E'[()] : \mathbb{1} @ \mu$ and $(\Gamma_3, x), \Gamma_4 \vdash e : \tau @ \mu$. Then $\Gamma_1 = \Gamma_1' \fatsemi \Gamma_2 \fatsemi \Gamma_3$ and $\Gamma_1' \fatsemi \Gamma_2 \vdash E[()] : \mathbb{1} @ \mu$ and $\Gamma_3, (x, \Gamma_4) \vdash e : \tau @ \mu$.

□

Lemma A.15 (Joining of tail contexts). *If* $\Gamma_1 = \Gamma_2 \fatsemi \Gamma_3$ *is defined and* $\Gamma_2 \vdash E[()] : \mathbb{1} @ \mu$ *and* $\Gamma_3, \Gamma_4 \vdash e : \tau @ \mu$ *where* Γ_4 *contains all variables bound in E, then* $\Gamma_1 \vdash E[e] : \tau @ \mu$.

Proof. By induction on E.

- If $E = ?$, then the claim follows directly using the sub-rule and monotonicity of context join.
- Let $E = \text{let}\, x = e' \,\text{in}\, E'$. Then $\Gamma_2 = \Gamma_2' \fatsemi \Gamma_2''$, and $\Gamma_2' \vdash e'$ and $\Gamma_2'', x \vdash E'[()] : \mathbb{1} @ \mu$. By the induction hypothesis, we obtain $(\Gamma_2'', x) \fatsemi (\Gamma_3, x) \vdash E'[e] : \tau @ \mu$. Then $\Gamma_2' \fatsemi \Gamma_2'' \fatsemi \Gamma_3 \vdash \text{let}\, x = e' \,\text{in}\, E'[e] : \tau @ \mu$.

□

For the definition of the translation to fine-grain call-by-value, see Figure 6.

Lemma A.16 (Normalization to fine-grain is type-preserving). *If* $\Gamma \vdash e : \tau @ \mu$ *and* $e \rightsquigarrow E \mid v$, *then* $\Gamma \vdash E[v] : \tau @ \mu$.

Proof. By induction on the translation.

- Case var, case unit: Clear.
- Case inl, case inr, case box: Apply the inductive hypothesis to e. Split the typing derivation of $E[v]$. Then join the typing derivation to $E[\text{inl}\, v]$.
- Case pair: Apply the inductive hypothesis to e_1. Apply the inductive hypothesis to e_2. Then derive $\text{let}\, x_1 = E_1[v_1] \,\text{in}\, \text{let}\, x_2 = E_2[v_2] \,\text{in}(x_1, x_2)$ using variable weakening by x_1.
- Case lam: Apply the inductive hypothesis to e. Then derive $\lambda x.\, E[v]$.
- Case app: Apply the inductive hypothesis to e_1. Apply the inductive hypothesis to e_2. Then derive $\text{let}\, x_1 = E_1[v_1] \,\text{in}\, \text{let}\, x_2 = E_2[v_2] \,\text{in}\, \text{let}\, x = x_1\, x_2 \,\text{in}\, x$ using variable weakening by x_1.
- Case unbox: Apply the inductive hypothesis to e. Split the typing derivation of $E[v]$. Then join the typing derivation to derive $\text{let}\, x = E[\text{unbox}_\mu\, v] \,\text{in}\, x$.
- Case let: Apply the inductive hypothesis to e_1. Apply the inductive hypothesis to e_2. Then derive $\text{let}\, x = E_1[v_1] \,\text{in}\, E_2[v_2]$ using variable weakening by x.
- Case split: Apply the inductive hypothesis to e_1. Apply the inductive hypothesis to e_2. Then derive $\text{let}\, x' = E_1[v_1] \,\text{in}\, \text{let}\, (x, y, z) = x' \,\text{in}\, E_2[v_2]$ using variable weakening by x'.
- Case case: Apply the inductive hypothesis to e_1. Apply the inductive hypothesis to e_2. Apply the inductive hypothesis to e_3. Then derive $\text{let}\, x' = E_1[v_1] \,\text{in}\, \text{let}\, x'' = \text{case}\, x' \,\{\, \text{inl}\, x \longrightarrow E_2[v_2]; \text{inr}\, y \longrightarrow E_3[v_3] \,\}\, \text{in}\, x''$ using variable weakening by x'.
- Case reuse: Apply the inductive hypothesis to e_1. Apply the inductive hypothesis to e_2. Apply the inductive hypothesis to e_3. Then derive $\text{let}\, x_1 = E_1[v_1] \,\text{in}\, \text{let}\, x_2 = E_2[v_2] \,\text{in}\, \text{let}\, x_3 = E_3[v_3] \,\text{in}\, \text{let}\, x = \text{reuse}\, x_1 \,\text{in}\, (x_2, x_3) \,\text{in}\, x$ using variable weakening by x_1.
- Case borrow: Apply the inductive hypothesis to e. Then derive $\text{let}\, x' = \text{borrow!}\, x = (x_1, \ldots, x_n) \,\text{for}\, E[v] \,\text{in}\, x'$.

□

$$\frac{}{x \rightsquigarrow ? \mid x}\text{VAR} \qquad \frac{}{() \rightsquigarrow ? \mid ()}\text{UNIT} \qquad \frac{e \rightsquigarrow E \mid v}{\mathsf{inl}\, e \rightsquigarrow E \mid \mathsf{inl}\, v}\text{INL} \qquad \frac{e \rightsquigarrow E \mid v}{\mathsf{inr}\, e \rightsquigarrow E \mid \mathsf{inr}\, v}\text{INR}$$

$$\frac{e \rightsquigarrow E \mid v}{\mathsf{box}_\mu\, e \rightsquigarrow E \mid \mathsf{box}_\mu\, v}\text{BOX} \qquad \frac{e \rightsquigarrow E \mid v}{\lambda x.\, e \rightsquigarrow ? \mid \lambda x.\, E[v]}\text{LAM}$$

$$\frac{e_1 \rightsquigarrow E_1 \mid v_1 \qquad e_2 \rightsquigarrow E_2 \mid v_2 \qquad x_1, x_2 \text{ fresh}}{(e_1, e_2) \rightsquigarrow \mathsf{let}\, x_1 = E_1[v_1]\, \mathsf{in}\, \mathsf{let}\, x_2 = E_2[v_2]\, \mathsf{in}\, ? \mid (x_1, x_2)}\text{PAIR}$$

$$\frac{e_1 \rightsquigarrow E_1 \mid v_1 \qquad e_2 \rightsquigarrow E_2 \mid v_2 \qquad x, x_1, x_2 \text{ fresh}}{e_1\, e_2 \rightsquigarrow \mathsf{let}\, x_1 = E_1[v_1]\, \mathsf{in}\, \mathsf{let}\, x_2 = E_2[v_2]\, \mathsf{in}\, \mathsf{let}\, x = x_1\, x_2\, \mathsf{in}\, ? \mid x}\text{APP}$$

$$\frac{e \rightsquigarrow E \mid v \qquad x \text{ fresh}}{\mathsf{unbox}_\mu\, e \rightsquigarrow \mathsf{let}\, x = E[\mathsf{unbox}_\mu\, v]\, \mathsf{in}\, ? \mid x}\text{UNBOX} \qquad \frac{e_1 \rightsquigarrow E_1 \mid v_1 \qquad e_2 \rightsquigarrow E_2 \mid v_2}{\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \rightsquigarrow \mathsf{let}\, x = E_1[v_1]\, \mathsf{in}\, E_2 \mid v_2}\text{LET}$$

$$\frac{e_1 \rightsquigarrow E_1 \mid v_1 \qquad e_2 \rightsquigarrow E_2 \mid v_2 \qquad x' \text{ fresh}}{\mathsf{let}\, (x, y, z) = e_1\, \mathsf{in}\, e_2 \rightsquigarrow \mathsf{let}\, x' = E_1[v_1]\, \mathsf{in}\, \mathsf{let}\, (x, y, z) = x'\, \mathsf{in}\, E_2 \mid v_2}\text{SPLIT}$$

$$\frac{e_1 \rightsquigarrow E_1 \mid v_1 \qquad e_2 \rightsquigarrow E_2 \mid v_2 \qquad e_3 \rightsquigarrow E_3 \mid v_3 \qquad x', x'' \text{ fresh}}{\begin{array}{c}\mathsf{case}\, e_1 \,\{\, \mathsf{inl}\, x \longrightarrow e_2; \mathsf{inr}\, y \longrightarrow e_3 \,\} \rightsquigarrow \\ \mathsf{let}\, x' = E_1[v_1]\, \mathsf{in}\, \mathsf{let}\, x'' = \mathsf{case}\, x' \,\{\, \mathsf{inl}\, x \longrightarrow E_2[v_2]; \mathsf{inr}\, y \longrightarrow E_3[v_3] \,\}\, \mathsf{in}\, ? \mid x''\end{array}}\text{CASE}$$

$$\frac{e_1 \rightsquigarrow E_1 \mid v_1 \qquad e_2 \rightsquigarrow E_2 \mid v_2 \qquad e_3 \rightsquigarrow E_3 \mid v_3 \qquad x, x_1, x_2, x_3 \text{ fresh}}{\begin{array}{c}\mathsf{reuse}\, e_1\, \mathsf{in}\, (e_2, e_3) \rightsquigarrow \\ \mathsf{let}\, x_1 = E_1[v_1]\, \mathsf{in}\, \mathsf{let}\, x_2 = E_2[v_2]\, \mathsf{in}\, \mathsf{let}\, x_3 = E_3[v_3]\, \mathsf{in}\, \mathsf{let}\, x = \mathsf{reuse}\, x_1\, \mathsf{in}\, (x_2, x_3)\, \mathsf{in}\, ? \mid x\end{array}}\text{REUSE}$$

$$\frac{e \rightsquigarrow E \mid v \qquad x' \text{ fresh}}{\mathsf{borrow!}\, x = (x_1, \ldots, x_n)\, \mathsf{for}\, e \rightsquigarrow \mathsf{let}\, x' = \mathsf{borrow!}\, x = (x_1, \ldots, x_n)\, \mathsf{for}\, E[v]\, \mathsf{in}\, ? \mid x'}\text{BORROW}$$

Fig. 6. Normalizing Coarse-Grain to Fine-Grain Call-by-Value.

A.2.3 Translation to ANF. Our translation to ANF uses the following tail context:

$$\begin{aligned} E ::= {} & ? \mid \mathsf{let}\, x = v\, \mathsf{in}\, E \mid \mathsf{let}\, x = v_1\, v_2\, \mathsf{in}\, E \mid \mathsf{let}\, x = \mathsf{unbox}_\mu\, v\, \mathsf{in}\, E \mid \mathsf{let}\, (x, y, z) = v\, \mathsf{in}\, E \\ & \mid \mathsf{case}\, v \,\{\, \mathsf{inl}\, x \longrightarrow E; \mathsf{inr}\, y \longrightarrow E \,\} \mid \mathsf{let}\, x = \mathsf{reuse}\, v\, \mathsf{in}\, (v_1, v_2)\, \mathsf{in}\, E \\ & \mid \mathsf{let}\, x = \mathsf{borrow!}\, y = (x_1, \ldots, x_n)\, \mathsf{for}\, e\, \mathsf{in}\, E \end{aligned}$$

Again, we obtain a splitting and a joining lemma for tail contexts:

Lemma A.17 (Splitting of tail contexts). *If $\Gamma_1 \vdash E[e] : \tau \,@\, \mu$, then $\Gamma_1 = \Gamma_2 \,⨾\, \Gamma_3$ and $\Gamma_2 \vdash E[()] : \mathbb{1} \,@\, \mu$ and $\Gamma_3, \Gamma_4 \vdash e : \tau \,@\, \mu$, where Γ_4 contains all variables bound in all branches of E.*

Proof. By induction on E.

- If $E = ?$, then let Γ_2 contain all variables from Γ_1 as unused. Let Γ_3 be Γ_1 and Γ_4 be the empty context. Then the claim follows.

- Let $E = \text{let}\, x = e' \,\text{in}\, E'$. Then $\Gamma_1 = \Gamma_1' \fatsemi \Gamma_1''$, and $\Gamma_1' \vdash e'$ and $\Gamma_1'', x \vdash E'[e] : \tau \mathbin{@} \mu$. By the induction hypothesis, we obtain $(\Gamma_1'', x) = (\Gamma_2, x) \fatsemi (\Gamma_3, x)$ and $\Gamma_2, x \vdash E'[()] : \mathbb{1} \mathbin{@} \mu$ and $(\Gamma_3, x), \Gamma_4 \vdash e : \tau @ \mu$. Then $\Gamma_1 = \Gamma_1' \fatsemi \Gamma_2 \fatsemi \Gamma_3$ and $\Gamma_1' \fatsemi \Gamma_2 \vdash E[()] : \mathbb{1} @ \mu$ and $\Gamma_3, (x, \Gamma_4) \vdash e : \tau @ \mu$.
- Let $E = \text{let}\,(x, y, z) = v \,\text{in}\, E'$. Similar to the previous case.
- Let $E = \text{case}\, v \,\{\, \text{inl}\, x \rightarrow E'; \text{inr}\, y \rightarrow E'' \,\}$. Then $\Gamma_1 = \Gamma_1' \fatsemi \Gamma_1''$, and $\Gamma_1' \vdash v$ and $\Gamma_1'', x \vdash E'[e] : \tau \mathbin{@} \mu$ and $\Gamma_1'', y \vdash E''[e] : \tau \mathbin{@} \mu$. By the induction hypothesis, we obtain $(\Gamma_1'', x) = (\Gamma_2, x) \fatsemi (\Gamma_3, x)$ and $(\Gamma_1'', y) = (\Gamma_2', y) \fatsemi (\Gamma_3', y)$ and $\Gamma_2, x \vdash E'[()] : \mathbb{1} \mathbin{@} \mu$ and $\Gamma_2', x \vdash E''[()] : \mathbb{1} \mathbin{@} \mu$ and $(\Gamma_3, x), \Gamma_4 \vdash e : \tau \mathbin{@} \mu$ and $(\Gamma_3', y), \Gamma_4' \vdash e : \tau \mathbin{@} \mu$. Then $\Gamma_1 = \Gamma_1' \fatsemi \Gamma_2 \fatsemi \Gamma_3$ and $\Gamma_1 = \Gamma_1' \fatsemi \Gamma_2' \fatsemi \Gamma_3'$. Let Γ_3'' be the smallest context with $\Gamma_3'' \geq \Gamma_3$ and $\Gamma_3'' \geq \Gamma_3'$. Let Γ_4'' be the smallest context with $\Gamma_4'' \geq \Gamma_4$ and $\Gamma_4'' \geq \Gamma_4'$. Then $(\Gamma_3'', y), \Gamma_4'' \vdash e : \tau \mathbin{@} \mu$. Let Γ_2'' be the biggest context such that $\Gamma_1 = \Gamma_2'' \fatsemi \Gamma_3''$. Then $\Gamma_2'' \leq \Gamma_2$ and $\Gamma_2'' \leq \Gamma_2'$. Thus $\Gamma_2'' \vdash E'[()] : \mathbb{1} \mathbin{@} \mu$ and and $\Gamma_2'' \vdash E''[()] : \mathbb{1} \mathbin{@} \mu$.

□

where we used that:

Lemma A.18 (Intersection of contexts). *If $\Gamma \vdash e : \tau \mathbin{@} \mu$ and $\Gamma' \vdash e : \tau \mathbin{@} \mu$. Let Γ'' be the smallest context with $\Gamma'' \geq \Gamma$ and $\Gamma'' \geq \Gamma'$. Then $\Gamma'' \vdash e : \tau \mathbin{@} \mu$.*

Proof. By induction over the typing derivation of e.

- Case var: Assume that x is at mode (a_1, u_1, l_1) in Γ and at mode (a_1', u_1', l_1') in Γ'. Then it is at mode $(a_1 \vee a_1', u_1 \vee u_1', l_1 \vee l_1')$ in Γ''. Then each inequality $a_1 \vee a_1' \leq a_2$ is fulfilled since it holds iff $a_1 \leq a_2$ and $a_1' \leq a_2$.
- Case sub: Let Γ''' be the smallest context with $\Gamma''' \geq \Gamma_1$ and $\Gamma''' \geq \Gamma_1'$, where $\Gamma_1 \geq \Gamma$ and $\Gamma_1' \geq \Gamma'$. Then apply the induction hypothesis to Γ'''. Since $\Gamma''' \geq \Gamma''$, use the sub-rule to obtain the claim.
- All other cases are clear.

□

Lemma A.19 (Joining of tail contexts). *If $\Gamma_1 = \Gamma_2 \fatsemi \Gamma_3$ is defined and $\Gamma_2 \vdash E[()] : \mathbb{1} \mathbin{@} \mu$ and $\Gamma_3, \Gamma_4 \vdash e : \tau \mathbin{@} \mu$ where Γ_4 contains all variables bound in all branches of E, then $\Gamma_1 \vdash E[e] : \tau \mathbin{@} \mu$.*

Proof. By induction on E.

- If $E = ?$, then the claim follows directly using the sub-rule and monotonicity of context join.
- Let $E = \text{let}\, x = e' \,\text{in}\, E'$. Then $\Gamma_2 = \Gamma_2' \fatsemi \Gamma_2''$, and $\Gamma_2' \vdash e'$ and $\Gamma_2'', x \vdash E'[()] : \mathbb{1} \mathbin{@} \mu$. By the induction hypothesis, we obtain $(\Gamma_2'', x) \fatsemi (\Gamma_3, x : -) \vdash E'[e] : \tau \mathbin{@} \mu$. Then $\Gamma_2' \fatsemi \Gamma_2'' \fatsemi \Gamma_3 \vdash \text{let}\, x = e' \,\text{in}\, E'[e] : \tau \mathbin{@} \mu$.
- Let $E = \text{let}\,(x, y, z) = v \,\text{in}\, E'$. Similar to the previous case.
- Let $E = \text{case}\, v \,\{\, \text{inl}\, x \rightarrow E'; \text{inr}\, y \rightarrow E'' \,\}$. Then $\Gamma_2 = \Gamma_2' \fatsemi \Gamma_2''$, and $\Gamma_2' \vdash v$ and $\Gamma_2'', x \vdash E'[()] : \mathbb{1} \mathbin{@} \mu$ and $\Gamma_2'', y \vdash E''[()] : \mathbb{1} \mathbin{@} \mu$. By the induction hypothesis, we obtain $(\Gamma_2'', x) \fatsemi (\Gamma_3, x : -) \vdash E'[e] : \tau \mathbin{@} \mu$ and $(\Gamma_2'', y) \fatsemi (\Gamma_3, y : -) \vdash E''[e] : \tau \mathbin{@} \mu$. Then $\Gamma_2' \fatsemi \Gamma_2'' \fatsemi \Gamma_3 \vdash \text{case}\, v \,\{\, \text{inl}\, x \rightarrow E'[e]; \text{inr}\, y \rightarrow E''[e] \,\} : \tau \mathbin{@} \mu$.

□

For the definition of the translation to ANF, see Figure 7.

Lemma A.20 (Normalization to fine-grain is type-preserving). *If $\Gamma \vdash v : \tau \mathbin{@} \mu$ and $v \rightsquigarrow v'$, then $\Gamma \vdash v' : \tau \mathbin{@} \mu$. If $\Gamma \vdash e : \tau \mathbin{@} \mu$ and $e \rightsquigarrow E \mid v$, then $\Gamma \vdash E[v] : \tau \mathbin{@} \mu$.*

Proof. By induction on the translation. For values this follows directly from the inductive hypothesis.

$$\frac{}{x \rightsquigarrow x}\ \textsc{var} \qquad \frac{}{() \rightsquigarrow ()}\ \textsc{unit} \qquad \frac{v_1 \rightsquigarrow v_2}{\mathrm{inl}\, v_1 \rightsquigarrow \mathrm{inl}\, v_2}\ \textsc{inl} \qquad \frac{v_1 \rightsquigarrow v_2}{\mathrm{inr}\, v_1 \rightsquigarrow \mathrm{inr}\, v_2}\ \textsc{inr}$$

$$\frac{v_1 \rightsquigarrow w_1 \qquad v_2 \rightsquigarrow w_2}{(v_1, v_2) \rightsquigarrow (w_1, w_2)}\ \textsc{pair} \qquad \frac{v_1 \rightsquigarrow v_2}{\mathrm{box}_\mu\, v_1 \rightsquigarrow \mathrm{box}_\mu\, v_2}\ \textsc{box} \qquad \frac{e_1 \rightsquigarrow E \mid e_2}{\lambda x.\, e_1 \rightsquigarrow \lambda x.\, E[e_2]}\ \textsc{lam}$$

$$\frac{v_1 \rightsquigarrow v_2}{v_1 \rightsquigarrow ? \mid v_2}\ \textsc{value} \qquad \frac{v_1 \rightsquigarrow v_1' \qquad v_2 \rightsquigarrow v_2' \qquad x \text{ fresh}}{v_1\, v_2 \rightsquigarrow \mathrm{let}\, x = v_1'\, v_2' \,\mathrm{in}\, ? \mid x}\ \textsc{app}$$

$$\frac{v_1 \rightsquigarrow v_1' \qquad x \text{ fresh}}{\mathrm{unbox}_\mu\, v_1 \rightsquigarrow \mathrm{let}\, x = \mathrm{unbox}_\mu\, v_1' \,\mathrm{in}\, ? \mid x}\ \textsc{unbox} \qquad \frac{e_1 \rightsquigarrow E_1 \mid v_1 \qquad e_2 \rightsquigarrow E_2 \mid v_2}{\mathrm{let}\, x = e_1 \,\mathrm{in}\, e_2 \rightsquigarrow E_1[\mathrm{let}\, x = v_1 \,\mathrm{in}\, E_2] \mid v_2}\ \textsc{let}$$

$$\frac{v_1 \rightsquigarrow v_1' \qquad e \rightsquigarrow E \mid v_2}{\mathrm{let}\,(x, y, z) = v_1 \,\mathrm{in}\, e \rightsquigarrow \mathrm{let}\,(x, y, z) = v_1' \,\mathrm{in}\, E \mid v_2}\ \textsc{split}$$

$$\frac{v_1 \rightsquigarrow v_1' \qquad e_1 \rightsquigarrow E_1 \mid v_2 \qquad e_2 \rightsquigarrow E_2 \mid v_3 \qquad x' \text{ fresh}}{\begin{array}{c}\mathrm{case}\, v_1 \,\{\, \mathrm{inl}\, x \longrightarrow e_1; \mathrm{inr}\, y \longrightarrow e_2 \,\} \rightsquigarrow \\ \mathrm{case}\, v_1' \,\{\, \mathrm{inl}\, x \longrightarrow E_1[\mathrm{let}\, x' = v_2 \,\mathrm{in}\, ?]; \mathrm{inr}\, y \longrightarrow E_2[\mathrm{let}\, x' = v_3 \,\mathrm{in}\, ?] \,\} \mid x'\end{array}}\ \textsc{case}$$

$$\frac{v_1 \rightsquigarrow v_1' \qquad v_2 \rightsquigarrow v_2' \qquad v_3 \rightsquigarrow v_3' \qquad x \text{ fresh}}{\mathrm{reuse}\ v_1\ \mathrm{in}(v_2, v_3) \rightsquigarrow \mathrm{let}\, x = (\mathrm{reuse}\ v_1'\ \mathrm{in}\ (v_2', v_3'))\ \mathrm{in}\, ? \mid x}\ \textsc{reuse}$$

$$\frac{e \rightsquigarrow E \mid v \qquad x' \text{ fresh}}{\mathrm{borrow!}\ x = (x_1, \dots, x_n)\ \mathrm{for}\ e_1 \rightsquigarrow \mathrm{let}\, x' = \mathrm{borrow!}\ x = (x_1, \dots, x_n)\ \mathrm{for}\ E[v]\ \mathrm{in}\, ? \mid x'}\ \textsc{borrow}$$

Fig. 7. Normalizing Fine-Grain Call-by-Value to ANF

- Case APP: Apply the inductive hypothesis to v_1. Apply the inductive hypothesis to v_2. Then derive $\mathrm{let}\, x = v_1\, v_2 \,\mathrm{in}\, x$ using the APP-rule.
- Case UNBOX: Apply the inductive hypothesis to v_1. Then derive $\mathrm{let}\, x = \mathrm{unbox}_\mu\, v_1' \,\mathrm{in}\, x$ using the UNBOX-rule.
- Case LET: Apply the inductive hypothesis to v_1. Apply the inductive hypothesis to e_2. Split the typing derivation of $E_1[v_1]$ into E_1 and v_1. Then use the LET rule to derive $\mathrm{let}\, x = v_1 \,\mathrm{in}\, E_2[v_2]$. Join the tail contexts to obtain $E_1[\mathrm{let}\, x = v_1 \,\mathrm{in}\, E_2[v_2]]$
- Case SPLIT: Apply the inductive hypothesis to v_1. Apply the inductive hypothesis to e. Then derive $\mathrm{let}\,(x, y, z) = v_1' \,\mathrm{in}\, E[v_2]$ using the SPLIT-rule.
- Case CASE: Apply the inductive hypothesis to v_1. Apply the inductive hypothesis to e_1. Apply the inductive hypothesis to e_2. Split the typing derivation of the tail context $E_1[v_2]$ into E_1 and v_2. Split the typing derivation of the tail context $E_2[v_3]$ into E_2 and v_3. Then derive $E_1[\mathrm{let}\, x' = v_2 \,\mathrm{in}\, x']$ and $E_2[\mathrm{let}\, x' = v_3 \,\mathrm{in}\, x']$ using the LET-rule and joining the tail contexts. Then derive $\mathrm{case}\, v_1' \,\{\, \mathrm{inl}\, x \longrightarrow E_1[\mathrm{let}\, x' = v_2 \,\mathrm{in}\, x']; \mathrm{inr}\, y \longrightarrow E_2[\mathrm{let}\, x' = v_3 \,\mathrm{in}\, x'] \,\}$.
- Case REUSE: Apply the inductive hypothesis to v_1. Apply the inductive hypothesis to v_2. Apply the inductive hypothesis to v_3. Then derive $\mathrm{let}\, x = \mathrm{reuse}\ v_1'\ \mathrm{in}(v_2', v_3') \,\mathrm{in}\, x$ using the REUSE-rule.

- Case BORROW: Apply the inductive hypothesis to e. Then derive let $x' =$ borrow! $x = (x_1, \dots, x_n)$ for $E[v]$ in x' using the BORROW-rule.

□

A.2.4 Translation to Passing Borrowed Variables. This translation passes borrowed variables explicitly from a scope, thus allowing us to check the program in the mode calculus. For expressions, we compute a set B of variables that are borrowed but not used in the expression:

For any mode $\mu = (a, u, l)$, we write $\text{box}_\mu\, \tau$ to denote:

- If $(a, u) =$ (MANY, SHARED): Then $\Box_\mu \tau := \Box_M \Box_S \tau$.
- If $(a, u) =$ (MANY, UNIQUE): Then $\Box_\mu \tau := \Box_M \tau$.
- If $(a, u) =$ (ONCE, SHARED): Then $\Box_\mu \tau := \Box_S \tau$.
- If $(a, u) =$ (ONCE, UNIQUE): Then $\Box_\mu \tau := \tau$.

For a set of borrowed variables B, we write $\tau(B)$ for the type of an n-tuple of borrowed variables in B:

- If $B = \varnothing$: Then $\tau(B) := \mathbb{1}$.
- If $B = (b_1 : \tau_1 @ \mu_1), B'$: Then $\tau(B) := \Box_{\mu_1} \tau_1 \times \tau(B')$.

We write $\text{cfv}(e)$ for the *consumed* free variables of an expression e, which includes all free variables of e except those occurring only in the n-tuples $(x_1, \dots, x_n)$ of BORROW! expressions.

The translation to the mode calculus is given in Figure 8.

LEMMA A.21 (PASSING BORROWED VARIABLES IS TYPE-PRESERVING). *If* $\Gamma \vdash e_1 : \tau @ (a, u, l)$ *and* $e_1 \rightsquigarrow B \mid e_2$*, then*

- *if* $x \in fv(e_1)$ *and* x *is marked as borrowed in* Γ*, then* $x \in B$
- *if* $l =$ *LOCAL, then* $B = \varnothing$
- $\Gamma \vdash e_2 : (\Box_{(a,u,l)} \tau) \times \tau(B) @ (\textit{ONCE}, \textit{UNIQUE}, l)$.

PROOF. By induction on the translation.

- Case VAR, UNIT: The variable x can not be borrowed. and the set of borrowed variables is clearly empty.
- Case INL, INR, PAIR and BOX: The claim follows directly from the inductive hypothesis.
- Case LAM: The claim follows from the inductive hypothesis. Notice that in our new LAM-rule, we delete all borrowing annotations from the context.
- Case VALUE: Follows directly from the inductive hypothesis.
- Case APP, UNBOX, REUSE, SPLIT, LET: The claim follows directly from the inductive hypothesis, since the type and mode of the expression is the type and mode of e'.
- Case CASE: Apply the inductive hypothesis to both e_1 and e_2. In the CASE-rule, we assume that both expressions are checked with the same context. Thus, if a variable is marked borrowed in the context used to check the case statement, then it is also marked borrowed in both branches. If the mode of the case statement is LOCAL, then this is also the mode of the branches and the intersection of two empty sets is empty. Thus the claim follows.
- Case BORROW: Apply the inductive hypothesis to e. Since all variables $x_1, \dots, x_n$ are marked as borrowed in the context, we need to add them to the set of borrowed variables. We can do so by splitting the x re-introduced by the BORROW-rule (of the mode calculus) into $x_1, \dots, x_n$ and adding it to the tuple.

□

We can finish the transformation by wrapping any transformed expression e into $\text{let}(\_, x, \_) = e$ in $\text{unbox}_\mu\, x$. Then we obtain the original type and mode:

$$\frac{}{x \rightsquigarrow x}\ \textsc{var} \qquad \frac{}{() \rightsquigarrow ()}\ \textsc{unit} \qquad \frac{v_1 \rightsquigarrow v_2}{\mathrm{inl}\, v_1 \rightsquigarrow \mathrm{inl}\, v_2}\ \textsc{inl} \qquad \frac{v_1 \rightsquigarrow v_2}{\mathrm{inr}\, v_1 \rightsquigarrow \mathrm{inr}\, v_2}\ \textsc{inr}$$

$$\frac{v_1 \rightsquigarrow w_1 \qquad v_2 \rightsquigarrow w_2}{(v_1, v_2) \rightsquigarrow (w_1, w_2)}\ \textsc{pair} \qquad \frac{v_1 \rightsquigarrow v_2}{\mathrm{box}_\mu\, v_1 \rightsquigarrow \mathrm{box}_\mu\, v_2}\ \textsc{box} \qquad \frac{e \rightsquigarrow B \mid e'}{\lambda x.\, e \rightsquigarrow \lambda x.\, e'}\ \textsc{lam}$$

$$\frac{v_1 \rightsquigarrow v_2}{v_1 \rightsquigarrow \varnothing \mid (\mathrm{box}_\mu\, v_2, ())}\ \textsc{value} \qquad \frac{v_1 \rightsquigarrow v_1' \qquad v_2 \rightsquigarrow v_2' \qquad e \rightsquigarrow B \mid e'}{\mathrm{let}\, x = v_1\, v_2 \,\mathrm{in}\, e \rightsquigarrow B \mid \mathrm{let}\, x = v_1'\, v_2' \,\mathrm{in}\, e'}\ \textsc{app}$$

$$\frac{v \rightsquigarrow v' \qquad e \rightsquigarrow B \mid e'}{\mathrm{let}\, x = \mathrm{unbox}_\mu\, v \,\mathrm{in}\, e \rightsquigarrow B \mid \mathrm{let}\, x = \mathrm{unbox}_\mu\, v' \,\mathrm{in}\, e'}\ \textsc{unbox}$$

$$\frac{v_1 \rightsquigarrow v_1' \qquad v_2 \rightsquigarrow v_2' \qquad v_3 \rightsquigarrow v_3' \qquad e \rightsquigarrow B \mid e'}{\mathrm{let}\, x = \mathrm{reuse}\, v_1 \,\mathrm{in}\, (v_2, v_3) \,\mathrm{in}\, e \rightsquigarrow B \mid \mathrm{let}\, x = \mathrm{reuse}\, v_1' \,\mathrm{in}\, (v_2', v_3') \,\mathrm{in}\, e'}\ \textsc{reuse}$$

$$\frac{v \rightsquigarrow v' \qquad e \rightsquigarrow B \mid e'}{\mathrm{let}\, (x, y, z) = v \,\mathrm{in}\, e \rightsquigarrow B \mid \mathrm{let}\, (x, y, z) = v' \,\mathrm{in}\, e'}\ \textsc{split} \qquad \frac{v \rightsquigarrow v' \qquad e \rightsquigarrow B \mid e'}{\mathrm{let}\, x = v \,\mathrm{in}\, e \rightsquigarrow B \mid \mathrm{let}\, x = v' \,\mathrm{in}\, e'}\ \textsc{let}$$

$$\frac{v_1 \rightsquigarrow v_1' \qquad e_1 \rightsquigarrow B_1 \mid e_1' \qquad e_2 \rightsquigarrow B_2 \mid e_2' \qquad x', y' \text{ fresh}}{\begin{array}{c}\mathrm{case}\, v_1 \,\{\, \mathrm{inl}\, x \rightarrow e_1; \mathrm{inr}\, y \rightarrow e_2 \,\} \rightsquigarrow B_1 \cap B_2 \mid \mathrm{case}\, v_1' \,\{ \\ \mathrm{inl}\, x \rightarrow \mathrm{let}(\_, x', B_1) = e_1' \,\mathrm{in}\, (x', B_1 \cap B_2); \\ \mathrm{inr}\, y \rightarrow \mathrm{let}(\_, y', B_2) = e_2' \,\mathrm{in}\, (y', B_1 \cap B_2) \,\}\end{array}}\ \textsc{case}$$

$$\frac{e \rightsquigarrow B \mid e' \qquad y \text{ fresh}}{\begin{array}{c}\mathrm{borrow!}\, x = (x_1, \ldots, x_n) \,\mathrm{for}\, e \rightsquigarrow B, x_1, \ldots, x_n \mid \mathrm{borrow}\, x = (x_1, \ldots, x_n) \\ \mathrm{for}\, y = e' \,\mathrm{in}\,\mathrm{let}\, (x_1, \ldots, x_n) = x \,\mathrm{in}\,\mathrm{let}\, (\_, y, B) = y \,\mathrm{in}\, (y, (B, x_1, \ldots, x_n))\end{array}}\ \textsc{borrow}$$

Fig. 8. Translation to Passing Borrowed Variables

Lemma A.22 (Passing borrowed variables is type-preserving). *If* $\Gamma \vdash e_1 : \tau \mathbin{@} \mu$ *and* $e_1 \rightsquigarrow_{pass} e_2$, *then* $\Gamma \vdash e_2 : \tau \mathbin{@} \mu$.

A.3 Semantics

In Figure 9 and 10, we extend the semantics of the mode calculus with rules to load and unload expressions from our evaluation context. These rules are quite straightforward and do not modify the store or current stack frame number. The only reason why we list them separately rather than wrapping them up in a single step-rule is that we need to adjust the mode of the state depending on the expression we are loading or unloading.

In the rules we further assume that the syntax of our language is annotated with the modes of sub-expressions. For example, we write application as $f^{\,\mu_1} x$ where μ_1 is the mode of the argument x. We also include a sub_μ term former for the sub-rule, which is annotated with the mode of the inner expression.

We store the evaluation context as a zipper in the semantics, and will write $E[e]$ to mean that e is in the hole of the zipper E. Similarly, we write $E[E']$ to mean that E' is in the hole of the zipper E.

$$
\begin{aligned}
E \coloneqq\ & ?\mid \mathrm{inl}\,E \mid \mathrm{inr}\,E \mid (E,e) \mid (a,E) \mid \mathrm{box}_\mu\ E_\mu \mid (E\,{}^{\mu}e)_\mu \mid (b\,E)_\mu \mid \mathrm{let}_\mu\,x = E\,\mathrm{in}_\mu\,e \\
& \mid \mathrm{sub}_\mu\,E \ \mid \mathrm{unbox}_\mu\ E_\mu \mid \mathrm{let}_\mu\,(x,y,z) = E\,\mathrm{in}_\mu\,e \mid \mathrm{case}_\mu\ E_\mu\,\{\,\mathrm{inl}\,x \to e; \mathrm{inr}\,y \to e\,\} \\
& \mid \mathrm{reuse}\,E\ \mathrm{with}_\mu\,(e,e) \mid \mathrm{reuse}\,b\ \mathrm{with}_\mu\,(E,e) \mid \mathrm{reuse}\,b\ \mathrm{with}_\mu\,(c,E) \\
& \mid \mathrm{borrow}\ x = E\ \mathrm{for}\ y = e\ \mathrm{in}_\mu\,e \mid \mathrm{borrow}\ x = b\ \mathrm{for}\ y = E\ \mathrm{in}_\mu\,e
\end{aligned}
$$

$$
\begin{array}{rll}
(\mathit{inl}_1) & S \,[]\, E \triangleright^{\mu}_{n} \mathrm{inl}\,e & \rightsquigarrow S \,[]\, \mathrm{inl}\,E \triangleright^{\mu}_{n} e \\
(\mathit{inl}_2) & S \,[]\, \mathrm{inl}\,E \triangleright^{\mu}_{n} b & \rightsquigarrow S \,[]\, E \triangleright^{\mu}_{n} \mathrm{inl}\,b \\
(\mathit{inr}_1) & S \,[]\, E \triangleright^{\mu}_{n} \mathrm{inr}\,e & \rightsquigarrow S \,[]\, \mathrm{inr}\,E \triangleright^{\mu}_{n} e \\
(\mathit{inr}_2) & S \,[]\, \mathrm{inr}\,E \triangleright^{\mu}_{n} b & \rightsquigarrow S \,[]\, E \triangleright^{\mu}_{n} \mathrm{inr}\,b \\
(\mathit{pair}_1) & S \,[]\, E \triangleright^{\mu}_{n} (e_1,e_2) & \rightsquigarrow S \,[]\, (E,e_2) \triangleright^{\mu}_{n} e_1 \\
(\mathit{pair}_2) & S \,[]\, (E,e_2) \triangleright^{\mu}_{n} b & \rightsquigarrow S \,[]\, (b,E) \triangleright^{\mu}_{n} e_2 \\
(\mathit{pair}_3) & S \,[]\, (b,E) \triangleright^{\mu}_{n} c & \rightsquigarrow S \,[]\, E \triangleright^{\mu}_{n} (b,c) \\
(\mathit{box}_M) & S \,[]\, E \triangleright^{(a,u,l)}_{n} \mathrm{box}_M\,e & \rightsquigarrow S \,[]\, \mathrm{box}_M\,E_{(a,u,l)} \triangleright^{(\textsc{many},u,l)}_{n} e \\
(\mathit{box}_S) & S \,[]\, E \triangleright^{(a,u,l)}_{n} \mathrm{box}_S\,e & \rightsquigarrow S \,[]\, \mathrm{box}_S\,E_{(a,u,l)} \triangleright^{(a,\textsc{shared},l)}_{n} e \\
(\mathit{box}_G) & S \,[]\, E \triangleright^{(a,u,l)}_{n} \mathrm{box}_G\,e & \rightsquigarrow S \,[]\, \mathrm{box}_G\,E_{(a,u,l)} \triangleright^{(a,\textsc{shared},\textsc{global})}_{n} e \\
(\mathit{box}_2) & S \,[]\, \mathrm{box}_\nu\,E_\mu \triangleright^{-}_{n} b & \rightsquigarrow S \,[]\, E \triangleright^{\mu}_{n} \mathrm{box}_\nu\,b \\
(\mathit{app}_1) & S \,[]\, E \triangleright^{\mu_1}_{n} e_1\,{}^{\mu_2}e_2 & \rightsquigarrow S \,[]\, (E\,{}^{\mu_2}e_2)_{\mu_1} \triangleright^{(\textsc{once},\textsc{shared},\textsc{local})}_{n} e_1 \\
(\mathit{app}_2) & S \,[]\, (E\,{}^{\mu_2}e_2)_{\mu_1} \triangleright^{-}_{n} b & \rightsquigarrow S \,[]\, (b\,{}^{\mu_2}E)_{\mu_1} \triangleright^{\mu_2}_{n} e_2 \\
(\mathit{app}_3) & S \,[]\, (b\,{}^{\mu_2}E)_{\mu_1} \triangleright^{-}_{n} c & \rightsquigarrow S \,[]\, E \triangleright^{\mu_1}_{n} b\,{}^{\mu_2}c \\
(\mathit{let}_1) & S \,[]\, E \triangleright^{\mu_1}_{n} \mathrm{let}_{\mu_2}\,x = e_1\,\mathrm{in}\,e_2 & \rightsquigarrow S \,[]\, \mathrm{let}\,x = E\,\mathrm{in}_{\mu_1}\,e_2 \triangleright^{\mu_2}_{n} e_1 \\
(\mathit{let}_2) & S \,[]\, \mathrm{let}\,x = E\,\mathrm{in}_{\mu_1}\,e_2 \triangleright^{-}_{n} b & \rightsquigarrow S \,[]\, E \triangleright^{\mu_1}_{n} \mathrm{let}_{\mu_2}\,x = b\,\mathrm{in}\,e_2 \\
(\mathit{sub}_1) & S \,[]\, E \triangleright^{\mu_1}_{n} \mathrm{sub}_{\mu_2}\,e & \rightsquigarrow S \,[]\, \mathrm{sub}_{\mu_1} E \triangleright^{\mu_2}_{n} e \\
(\mathit{sub}_2) & S \,[]\, \mathrm{sub}_\mu\,E \triangleright^{-}_{n} b & \rightsquigarrow S \,[]\, E \triangleright^{\mu}_{n} b \\
(\mathit{unbox}_M) & S \,[]\, E \triangleright^{(a,u,l)}_{n} \mathrm{unbox}_M\,e & \rightsquigarrow S \,[]\, \mathrm{unbox}_M\ E_{(a,u,l)} \triangleright^{(\textsc{once},u,l)}_{n} e \\
(\mathit{unbox}_S) & S \,[]\, E \triangleright^{(a,u,l)}_{n} \mathrm{unbox}_S\,e & \rightsquigarrow S \,[]\, \mathrm{unbox}_S\ E_{(a,u,l)} \triangleright^{(a,\textsc{shared},l)}_{n} e \\
(\mathit{unbox}_G) & S \,[]\, E \triangleright^{(a,u,l)}_{n} \mathrm{unbox}_G\,e & \rightsquigarrow S \,[]\, \mathrm{unbox}_G\ E_{(a,u,l)} \triangleright^{(a,\textsc{shared},\textsc{local})}_{n} e \\
(\mathit{unbox}_2) & S \,[]\, \mathrm{unbox}_{\mu_1}\,E_{\mu_2} \triangleright^{-}_{n} b & \rightsquigarrow S \,[]\, E \triangleright^{\mu_2}_{n} \mathrm{unbox}_{\mu_1}\,b \\
(\mathit{split}_1) & S \,[]\, E \triangleright^{\mu_1}_{n} \mathrm{let}_{\mu_2}\,(x,y,z) = e_1\,\mathrm{in}\,e_2 & \rightsquigarrow S \,[]\, \mathrm{let}_{\mu_2}(x,y,z) = E\,\mathrm{in}_{\mu_1}\,e_2 \triangleright^{\mu_2}_{n} e_1 \\
(\mathit{split}_2) & S \,[]\, \mathrm{let}_{\mu_2}(x,y,z) = E\,\mathrm{in}_\mu\,e \triangleright^{-}_{n} b & \rightsquigarrow S \,[]\, E \triangleright^{\mu}_{n} \mathrm{let}_{\mu_2}(x,y,z) = b\,\mathrm{in}_\mu\,e \\
(\mathit{reuse}_1) & S \,[]\, E \triangleright^{\mu}_{n} \mathrm{reuse}\ e_1\ \mathrm{in}\ (e_2,e_3) & \rightsquigarrow S \,[]\, \mathrm{reuse}\ E\ \mathrm{in}_\mu\,(e_2,e_3) \triangleright^{(\textsc{once},\textsc{unique},\textsc{global})}_{n} e_1 \\
(\mathit{reuse}_2) & S \,[]\, \mathrm{reuse}\ E\ \mathrm{in}_\mu\,(e_2,e_3) \triangleright^{-}_{n} a & \rightsquigarrow S \,[]\, \mathrm{reuse}\ a\ \mathrm{in}_\mu\,(E,e_3) \triangleright^{\mu}_{n} e_2 \\
(\mathit{reuse}_3) & S \,[]\, \mathrm{reuse}\ b\ \mathrm{in}_\mu\,(E,e_3) \triangleright^{-}_{n} c & \rightsquigarrow S \,[]\, \mathrm{reuse}\ b\ \mathrm{in}_\mu\,(c,E) \triangleright^{\mu}_{n} e_3 \\
(\mathit{reuse}_4) & S \,[]\, \mathrm{reuse}\ b\ \mathrm{with}_\mu\,(c,E) \triangleright^{-}_{n} d & \rightsquigarrow S \,[]\, E \triangleright^{\mu}_{n} \mathrm{reuse}\ b\ \mathrm{with}_\mu\,(c,d)
\end{array}
$$

Fig. 9. Usage-Aware Store Semantics: Building up the Evaluation Context (1)

$$
\begin{aligned}
(\mathit{case}_1)\ & S \,[]\, E \rhd_n^{\mu_1} \mathrm{case}_{\mu_2}\ e_1 \{\, \mathrm{inl}\, x \longrightarrow e_2; \mathrm{inr}\, y \longrightarrow e_3 \,\} \\
& \rightsquigarrow S \,[]\, \mathrm{case}_{\mu_2}\ E_{\mu_1} \{\, \mathrm{inl}\, x \longrightarrow e_2; \mathrm{inr}\, y \longrightarrow e_3 \,\} \rhd_n^{\mu_2} e_1 \\
(\mathit{case}_2)\ & S \,[]\, \mathrm{case}_{\mu_2}\ E_{\mu} \{\, \mathrm{inl}\, x \longrightarrow e_1; \mathrm{inr}\, y \longrightarrow e_2 \,\} \rhd_{\overline{n}}^{-} b \\
& \rightsquigarrow S \,[]\, E \rhd_n^{\mu} \mathrm{case}_{\mu_2}\ b \{\, \mathrm{inl}\, x \longrightarrow e_1; \mathrm{inr}\, y \longrightarrow e_2 \,\} \\
(\mathit{borrow}_1)\ & S \,[]\, E \rhd_n^{\mu_1} \mathrm{borrow}_{\mu_2}\ x = e_1\ \mathrm{for}_{\mu_3}\ y = e_2\ \mathrm{in}\ e_3 \\
& \rightsquigarrow S \,[]\, \mathrm{borrow}_{\mu_2} x = E\ \mathrm{for}_{\mu_3}\ y = e_2\ \mathrm{in}_{\mu_1}\ e_3 \rhd_n^{\mu_2} e_1 \\
(\mathit{borrow}_2)\ & S \,[]\, \mathrm{borrow}_{\mu_2}\ x = E\ \mathrm{for}_{\mu_3}\ y = e_2\ \mathrm{in}_{\mu_1}\ e_3 \rhd_{\overline{n}}^{-} b \\
& \rightsquigarrow S \,[]\, E \rhd_n^{\mu_3} \mathrm{borrow}_{\mu_2}\ x = b\ \mathrm{for}_{\mu_3}\ y = e_2\ \mathrm{in}_{\mu_1}\ e_3 \\
(\mathit{borrow}_3)\ & S \,[]\, E \rhd_n^{\mu_1} \mathrm{borrow}_{\mu_2} x = b\ \mathrm{for}_{\mu_3}\ y = c\ \mathrm{in}_{\mu_1} e_3 \\
& \rightsquigarrow S \,[]\, E \rhd_n^{\mu_1} e_3[x := b, y := c]
\end{aligned}
$$

Fig. 10. Usage-Aware Store Semantics: Building up the Evaluation Context (2)

We will by a small abuse of notation consider the hole ? to be a term of our mode calculus of any type and mode. Similarly, we will in the following occasionally omit the type and mode annotations of terms, where they are clear to save space and improve readability. We can split and join the typing derivations of evaluation contexts:

Lemma A.23 (Splitting of evaluation contexts). *If* $\Sigma_1 \vdash E[e : \tau_1 @ \mu_1] : \tau_2 @ \mu_2$, *then* $\Sigma_1 = \Sigma_2 + \Sigma_4$ *and* $\Sigma_2, \Sigma_3 \vdash e : \tau_1 @ \mu_1$, *and* $\Sigma_4 \vdash E[\,?\,] : \tau_2 @ \mu_2$, *where* Σ_3 *contains all borrowed variables in E.*

Proof. By induction on E.

- Case $E = ?$: Then $\Sigma_3 = \varnothing$ and the claim follows.
- Case $E = \mathrm{sub} E'$, case $E = \mathrm{inl}\, E'$, case $E = \mathrm{inr}\, E'$, case $E = \mathrm{box}_\nu\, E'$, case $E = \mathit{unbox}_\nu E'$: Follows directly from the inductive hypothesis.
- Case $E = (E', e)$, case $E = (b, E')$, case $E = \mathrm{let}\, x = E'\, \mathrm{in}\, e$, case $E = \mathrm{case}\, E' \{\, \mathrm{inl}\, x \longrightarrow e_1; \mathrm{inr}\, y \longrightarrow e_2 \,\}$, case $E = \mathrm{reuse}\, E'$ with (e_1, e_2), case $E = \mathrm{reuse}\, b$ with (E', e), case $E = \mathrm{reuse}\, b$ with (c, E'), case $E = \mathrm{borrow}\, x = E'$ for $y = e_1$ in e_2: In each case, we have $\Sigma_1 = \Sigma'_1 + \cdots + \Sigma'_n$ corresponding to the sub-expressions. Invoke the inductive hypothesis on E' and its corresponding context Σ'_i. Then we obtain $\Sigma'_i = \Sigma''_i + \Sigma'''_i$. Then, let $\Sigma_2 = \Sigma''_i$ and $\Sigma_3 = \Sigma'''_i + \sum_{j \neq i} \Sigma'_j$. We have $\Sigma_1 = \Sigma_2 + \Sigma_3$ since the + operation is associative and commutative. Then the claim follows.
- Case $E = \mathrm{borrow}\, x = b$ for $y = E'$ in e: We have $\Sigma_1 = \Sigma'_1 + \Sigma'_2 + \Sigma'_3$. Apply the inductive hypothesis to E' and Σ'_2, x. Then we obtain $\Sigma'_2, x = (\Sigma''_2, x) + (\Sigma'''_2, x)$. Let $\Sigma_2 = \Sigma''_2$ and $\Sigma_3 = \Sigma'_1 + \Sigma'''_2 + \Sigma'_3$. Then $\Sigma_1 = \Sigma_2 + \Sigma_3$ and the claim follows, where x is part of the borrowed variables in E.

□

Lemma A.24 (Joining of evaluation contexts). *If* $\Sigma_1 = \Sigma_2 + \Sigma_4$ *is defined and* $\Sigma_2, \Sigma_3 \vdash e : \tau_1 @ \mu_1$ *and* $\Sigma_4 \vdash E[\,? : \tau_1 @ \mu_1] : \tau_2 @ \mu_2$, *where* Σ_3 *contains all borrowed variables in E, then* $\Sigma_1 \vdash E[e] : \tau_2 @ \mu_2$.

Proof. By induction on E.

- Case $E = ?$: Then $\Sigma'_2 = \Sigma_3 = \varnothing$ and the claim follows.

- Case $E = \mathrm{sub} E'$, case $E = \mathrm{inl}\, E'$, case $E = \mathrm{inr}\, E'$, case $E = \mathrm{box}_\nu E'$, case $E = \mathit{unbox}_\nu E'$: Follows directly from the inductive hypothesis.
- Case $E = (E', e)$, case $E = (b, E')$, case $E = \mathrm{let}\, x = E' \,\mathrm{in}\, e$, case $E = \mathrm{case}\, E' \{ \mathrm{inl}\, x \to e_1; \mathrm{inr}\, y \to e_2 \}$, case $E = $ reuse E' with (e_1, e_2), case $E = $ reuse b with (E', e), case $E = $ reuse b with (c, E'), case $E = $ borrow $x = E'$ for $y = e_1$ in e_2: In each case, we have $\Sigma_4 = \Sigma'_1 + \cdots + \Sigma'_n$ corresponding to the sub-expressions. Invoke the inductive hypothesis on E' and its corresponding context Σ'_i (which is defined thanks to monotonicity of the + operation). Then we obtain $\Sigma_2 + \Sigma'_i \vdash E'[e] : \tau_2 \mathbin{@} \mu_2$. Then $\Sigma_2 + \Sigma_4 \vdash E[e] : \tau_2 \mathbin{@} \mu_2$ due to associativity and commutativity of the + operation.
- Case $E = $ borrow $x = b$ for $y = E'$ in e: We have $\Sigma_4 = \Sigma'_1 + \Sigma'_2 + \Sigma'_3$. Apply the inductive hypothesis to E' and Σ'_2, x. Then we obtain $(\Sigma_2 + \Sigma'_2), x \vdash E'[e] : \tau_2 \mathbin{@} \mu_2$. Then $\Sigma_2 + \Sigma_4 \vdash E[e] : \tau_2 \mathbin{@} \mu_2$ due to associativity and commutativity of the + operation.

□

Lemma A.25 (Substitution from store). *If $(S, b \mapsto^{\mu_1}_m v, S') :_n (\Sigma_1, b : \tau @ \mu_2, \Sigma_2)$ and $\mu_2 = \mu_3 + \mu_4$, then there is a context Σ' with $\Sigma' \vdash v : \tau @ \mu_4$ and $\Sigma_1 + \Sigma'$ is defined and $(S, b \mapsto^{\mu'_1}_m v, S') :_n ((\Sigma_1 + \Sigma'), b : \tau \mathbin{@} \mu_3, \Sigma_2)$.*

Proof. By induction on S'. If S' is empty, then by the wf-ext rule, we have a context Σ' with $\Sigma' \vdash v : \tau \mathbin{@} \mu_2$ and $S :'_n (\Sigma_1 + \Sigma')$ (and thus $\Sigma_1 + \Sigma'$ is defined). If μ_3 is $-$, then $(S, b \mapsto^{-}_m v) :_{n'} ((\Sigma_1 + \Sigma'), b : -)$ by the wf-unused rule. If μ_3 is not $-$, then μ_2 is many and μ_3, μ_4 are shared. By the duplicating values lemma, we can split $\Sigma' = \Sigma'' + \Sigma'''$ with $\Sigma'' \vdash v : \tau \mathbin{@} \mu_3$ and $\Sigma''' \vdash v : \tau \mathbin{@} \mu_4$. Then we have $(S, b \mapsto^{\mu_3}_m v) :_{n'} ((\Sigma_1 + \Sigma'''), b : \tau \mathbin{@} \mu_3)$ by the wf-ext rule and $\Sigma_1 + \Sigma'''$ is defined by monotonicity of the + operation.

If S' is extended by an unused variable, the claim follows directly from the inductive hypothesis. If S' is extended by $c \mapsto^{\mu}_k w$ such that $\Sigma'_1, c : \tau \mathbin{@} \mu'_3, \Sigma'_2 \vdash w : \tau' \mathbin{@} \mu$ and $(S, b \mapsto^{\mu'_1}_m v, S') :_{m'} ((\Sigma_1 + \Sigma'_1), b : \tau \mathbin{@} \mu_3 + \mu'_3, \Sigma_2 + \Sigma'_2)$, we can use the inductive hypothesis to obtain the context Σ' with $(S, b \mapsto^{\mu'_1}_m v, S') :_{m'} ((\Sigma_1 + \Sigma'_1 + \Sigma'), b : \tau \mathbin{@} \mu_3 + \mu'_3, \Sigma_2 + \Sigma'_2)$. Then by the wf-ext rule, $(S, b \mapsto^{\mu'_1}_m v, S', c \mapsto^{\mu}_k w) :_{n'} ((\Sigma_1 + \Sigma'), b : \tau \mathbin{@} \mu_3, \Sigma_2, c : \tau' \mathbin{@} \mu)$. □

Lemma A.26 (Removing unused variables). *If $(S, b : -, S') :_n (\Sigma, b : -, \Sigma')$, then $S, S' :_n \Sigma, \Sigma'$.*

Proof. By induction on S'. If S' is empty, then by the wf-unused rule, we have $S :_n \Sigma$. If S' is extended by an unused address, the claim follows directly from the inductive hypothesis. If S' is extended otherwise, we notice that b is unused inside Σ_2. If we denote by Σ'_2 the version of Σ_2 without b, then we still have $\Sigma'_2 \vdash v : \tau \mathbin{@} \mu$. The claim then follows from the inductive hypothesis. □

Lemma A.27 (Copying to stack). *If $(S, b \mapsto^{\mu_1}_m v, S') :_n (\Sigma_1, b : \tau \mathbin{@} \mu_2, \Sigma_2)$ where μ_2 is many, then $(S, b \mapsto^{\mu_1}_m v, S', copy(n{+}k, b)) :_{n+k} (\Sigma_1, b : \tau @ \mu_2, \Sigma_2, \Sigma_3)$, where $b' : \tau @ (\text{MANY}, \text{SHARED}, \text{LOCAL}) \in \Sigma_3$ and $k \geq 0$.*

Proof. By induction on the cases of the copy(n, b) operation. This is well-founded since the copy operation considers increasingly earlier addresses in the store. If $b \mapsto^{\mu}_m ()$, then the claim follows directly from the wf-ext rule. If $b \mapsto^{\mu}_m v$, where $v = \mathrm{inl}\, c, \mathrm{inr}\, c, (\mathrm{box}_{\mathrm{M}}\, c)$, then the claim follows from the inductive hypothesis and the wf-ext rule. If $b \mapsto^{\mu}_m (c, d)$, we have to invoke the inductive hypothesis twice. First we use $k = 1$ and then we invoke it on the result with $k = 0$. Then the claim follows from the wf-ext rule. If $b \mapsto^{\mu}_m \lambda x.e$, then we can obtain a substitution from the store for $\Sigma' \vdash \lambda x.e : \tau \mathbin{@} (\text{MANY}, \text{SHARED}, \_)$. Using the sub-rule, we can assume that the locality is local. Then the claim follows from the wf-ext rule. If $b \mapsto^{\mu}_m (\mathrm{box}_{\mathrm{G}}\, c)$ or $b \mapsto^{\mu}_m (\mathrm{box}_{\mathrm{S}}\, c)$, then we can obtain a substitution from the store for $\Sigma' \vdash c : \tau$ @ (MANY, SHARED, GLOBAL). Using the SUB-rule, we can assume that the locality is LOCAL. Then the claim follows from the WF-EXT rule. □

We call (E, e) *well-formed* for (S, n) if:

- for all $b \in \mathrm{fv}(e)$, $b \mapsto^{\mu}_{m} v \in S$ and $m \leq n$ or μ is GLOBAL.
- $E = \text{borrow } x = b \text{ for } y = E' \text{ in } e'$, implies that $n = n' + 1$ and $(E', (b, e'))$ is well-formed for (S, n').
- E is any other constructor of our evaluation context $E = t(E')$ implies that (E', t) is well-formed for (S, n).

In particular, both lemmas above will have $\Sigma_3 = \varnothing$ for well-formed evaluation contexts, since all free variables are store addresses rather than borrowed variables. Notice that unloading from the evaluation context preserves this property, but loading a borrow $x = b$ for E in e_3 frame onto the evaluation context does not preserve this property in general. This is because the inner term e might have x as a free variable, which is not in the store. To preserve well-formedness, it is necessary to substitute all occurrences of x as in the (enter_region) rule.

LEMMA A.28 (PROGRESS). *If $\Sigma \vdash E[e : \tau_1 \text{ @ } \mu_1] : \tau_2 \text{ @ } \mu_2$ 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: $S \mathbin{[]} E \rhd^{\mu_1}_{n} e \rightsquigarrow S' \mathbin{[]} E' \rhd^{\mu_1'}_{n'} e'$.*

PROOF. If $e = b$, but $E \neq ?$, then we can unload a frame from the evaluation context and the claim follows. If we can load a frame onto the evaluation context, then the claim follows. Otherwise, we split the typing derivation $\Sigma \vdash E[e : \tau_1 \text{ @ } \mu_1] : \tau_2 \text{ @ } \mu_2$ into $\Sigma = \Sigma_1 + \Sigma_2$ and $\Sigma_1 \vdash E[\,?] : \tau_2 \text{ @ } \mu_2$ and $\Sigma_2 \vdash e : \tau_1 \text{ @ } \mu_1$. Then $S :_n \Sigma_1 + \Sigma_2$. Perform case-analysis on e:

- Case VAR: Then $e = b$ for some store address b. By assumption $E = ?$. Thus the execution concludes.
- Case UNIT, case INL, case INR, case PAIR, case BOX, case LAM, case LET: Obvious.
- Case APP: We have $\Sigma_2 \vdash b\,c : \tau_1 \text{ @ } \mu_1$ and thus $\Sigma_2 \vdash b : \tau_3 \text{ @ } \mu_3 \to \tau_1 \text{ @ } \mu_1$. By the store typing, we thus have $b \mapsto \lambda x. \in S$ and we can take an (app) step.
- Case UNBOX: We have $\Sigma_2 \vdash \text{unbox}\, b : \tau_1 \text{ @ } \mu_1$ and thus $\Sigma_2 \vdash b : \text{box}_\nu\, \tau_1 \text{ @ } \mu_1'$. By the store typing, we thus have $b \mapsto \text{box}_\nu\, c$ and we can take an (unbox) step.
- Case CASE: We have $\Sigma_2 \vdash \text{case}\, b\{\, \text{inl}\, x \to e_1; \text{inr}\, y \to e_2\,\} : \tau_1 \text{ @ } \mu_1$ and thus $\Sigma_2 \vdash b : \tau_3 + \tau_4 \text{ @ } \mu'$. By the store typing, we thus have $b \mapsto \text{inl}\, c$ or $b \mapsto \text{inr}\, c$ and we can take a (case_l) or (case_r) step.
- Case SPLIT: We have $\Sigma_2 \vdash \text{let}(x, y, z) = b \,\text{in}\, e_1 : \tau_1 \text{ @ } \mu_1$ and thus $\Sigma_2 \vdash b : \tau_3 \times \tau_4 \text{ @ } \mu'$. By the store typing, we thus have $b \mapsto (c, d)$ and we can take a (split_unique) step or a (split_shared) step depending on the uniqueness of μ'.
- Case REUSE: We have $\Sigma_2 \vdash \text{reuse } b \text{ with } (c, d) : \tau_1 \text{@} \mu_1$ and thus $\Sigma_2 \vdash b : \clubsuit\text{@}(\_, \text{UNIQUE}, \text{GLOBAL})$. By the store typing, we thus have $b \mapsto \clubsuit \in S$ (and in particular, $b \neq$ `null`) and we can take a (reuse) step.
- Case BORROW: We can take either a (enter_region), (leave_region) or (borrow_3) step, which imposes no restrictions on the store or evaluation context. For the (leave_region) step, we can assume that $n = n' + 1$ due to the well-formedness of the evaluation context.

□

LEMMA A.29 (PRESERVATION). *If $\Sigma \vdash E[e : \tau_1 \text{ @ } \mu_1] : \tau_2 \text{ @ } \mu_2$ and (E, e) is well-formed for (S, n), and $S :_n \Sigma$ and $S \mathbin{[]} E \rhd^{\mu_1}_{n} e \rightsquigarrow S' \mathbin{[]} E' \rhd^{\mu_1'}_{n'} e'$, then $\Sigma' \vdash E'[e' : \tau_1' \text{ @ } \mu_1'] : \tau_2 \text{ @ } \mu_2$ and (E', e') is well-formed for (S', n'), and $S' :_{n'} \Sigma'$.*

Proof. By case-analysis on the reduction relation. This is obvious for all rules that merely load and unload expressions from the evaluation context since neither the store nor the expression $E[e]$ changes. Otherwise, we split the typing derivation $\Sigma \vdash E[e : \tau_1 @ \mu_1] : \tau_2 @ \mu_2$ into $\Sigma = \Sigma_1 + \Sigma_2$ and $\Sigma_1 \vdash E[\,?\,] : \tau_2 @ \mu_2$ and $\Sigma_2 \vdash e : \tau_1 @ \mu_1$. Then $S :_n \Sigma_1 + \Sigma_2$.

- Case (unit), case inl, case inr, case (pair), case box, case lam: Since $S :_n \Sigma_1 + \Sigma_2$, we can apply the WF-EXT rule to obtain $(S, d \mapsto^{\mu}_{n} e) :_n (\Sigma_1, d : \tau_1 @ \mu)$.
- Case (let): If $\Sigma \vdash \text{let}\, x = b \,\text{in}\, e$, then $\Sigma = \Sigma_1 + \Sigma_2$ and $\Sigma_1 \vdash b$ and $\Sigma_2, x \vdash e$. Thus we can apply the substitution lemma to obtain $\Sigma \vdash e[x := b]$.
- Case (app): Obtain a substitution for b from the store and apply it to $b\,c$, thus obtaining a well-typed expression $(\lambda x.e')\,c$. Then $\Sigma \vdash (\lambda x.e')\,c$, which implies $\Sigma = \Sigma_1 + \Sigma_2$ and $\Sigma_1 \vdash \lambda x.e'$ and $\Sigma_2 \vdash c$. Thus we can apply the substitution lemma again to obtain $e'[x := c]$.
- Case (unbox): Obtain a substitution for b from the store and apply it to $\text{unbox}\, b$, thus obtaining a well-typed expression $\text{unbox}(\text{box}_v\, c)$. Then c has the same type and mode.
- Case (case$_\text{l}$): Obtain a substitution for b from the store and apply it to $\text{case}\, b\{\, \text{inl}\, x \longrightarrow e_1; \text{inr}\, y \longrightarrow e_2 \,\}$, thus obtaining a well-typed expression $\text{case}\,(\text{inl}\, c)\{\, \text{inl}\, x \longrightarrow e_1; \text{inr}\, y \longrightarrow e_2 \,\}$. Then $\Sigma \vdash \text{case}\,(\text{inl}\, c)\{\, \text{inl}\, x \longrightarrow e_1; \text{inr}\, y \longrightarrow e_2 \,\}$ implies $\Sigma = \Sigma_1 + \Sigma_2$ and $\Sigma_1 \vdash \text{inl}\, c$ and $\Sigma_2, x \vdash e_1$. Then apply the substitution lemma to obtain $\Sigma \vdash e_1[x := c]$.
- Case (case$_\text{r}$): Obtain a substitution for b from the store and apply it to $\text{case}\, b\{\, \text{inr}\, x \longrightarrow e_1; \text{inr}\, y \longrightarrow e_2 \,\}$, thus obtaining a well-typed expression $\text{case}\,(\text{inr}\, c)\{\, \text{inl}\, x \longrightarrow e_1; \text{inr}\, y \longrightarrow e_2 \,\}$. Then $\Sigma \vdash \text{case}\,(\text{inl}\, c)\{\, \text{inl}\, x \longrightarrow e_1; \text{inr}\, y \longrightarrow e_2 \,\}$ implies $\Sigma = \Sigma_1 + \Sigma_2$ and $\Sigma_1 \vdash \text{inr}\, c$ and $\Sigma_2, y \vdash e_2$. Then apply the substitution lemma to obtain $\Sigma \vdash e_2[y := c]$.
- Case (split_unique): Obtain a substitution for b from the store and apply it to $\text{let}(x, y, z) = b \,\text{in}\, e$, thus obtaining a well-typed expression $\text{let}(x, y, z) = (c, d) \,\text{in}\, e$. Then $\Sigma \vdash \text{let}(x, y, z) = (c, d) \,\text{in}\, e$ implies $\Sigma = \Sigma_1 + \Sigma_2$ and $\Sigma_1 \vdash (c, d)$ and $\Sigma_2, x, y, z \vdash e$. Since $\mu_2 = (\_, \textsc{unique}, \_)$, we have $b : -$ in the store. Thus we can remove the unused address b. In its place, we then create a space credit $b \mapsto^{\mu_2}_{m} \clubsuit$ using the wf-space rule. Then apply the substitution lemma to obtain $\Sigma \vdash e[x := b, y := c, z := d]$.
- Case (split_shared): Obtain a substitution for b from the store and apply it to $\text{let}(x, y, z) = b \,\text{in}\, e$, thus obtaining a well-typed expression $\text{let}(x, y, z) = (c, d) \,\text{in}\, e$. Then $\Sigma \vdash \text{let}(x, y, z) = (c, d) \,\text{in}\, e$ implies $\Sigma = \Sigma_1 + \Sigma_2$ and $\Sigma_1 \vdash (c, d)$ and $\Sigma_2, x, y, z \vdash e$. Furthermore, $\Sigma_1 \vdash \texttt{null} : \clubsuit @ (\textsc{many}, \textsc{shared}, l)$. Then apply the substitution lemma to obtain $\Sigma \vdash e[x := \texttt{null}, y := c, z := d]$.
- Case (reuse): Since $b : \clubsuit @ (\_, \textsc{unique}, \textsc{global}) \in \Sigma$, we have that $b \neq \texttt{null}$. Obtain a substitution for b from the store and apply it to reuse b with (c, d), thus obtaining the expression reuse $\clubsuit$ with (c, d). Since b was unique, we can remove the (now unused) address b from the store. Then $\Sigma \vdash (c, d)$ and we can allocate the pair at the (now fresh) address b as in the (pair) step.
- Case (enter_region): By the copying to stack lemma, we have that $(S, \text{copy}(n + 1, b)) :_{n+1} \Sigma$. Then we can obtain a substitution for b' from the store and apply it to $e_2[x := b']$. Then (E', e') is well-formed for $(S', n + 1)$, since (E, e_2) was well-formed for (S, n) and b' is annotated by $n + 1$.
- Case (leave_region): We have that c is GLOBAL and (borrow $x = b$ for E' in e_3, c) is well-formed for $(S, n+1)$. By the definition of well-formedness, we thus have that (E, borrow $x = b$ for c in e_3) is well-formed for (S, n). Then there is a context $\Sigma' \vdash E$[borrow $x = b$ for c in e_3] such that $(S - (n + 1)) :_n \Sigma'$ (which can be obtained by deleting all variables from Σ that are deleted by the (S - (n+1)) operation — these are not free variables of the term). Then we can apply the substitution lemma to obtain $\Sigma' \vdash e_3[x := b, y := c]$.

□

B MODE INFERENCE

Our implementation features mode inference that can completely infer modes. We can describe that inference as a combination of a simple constraint solver and syntax-directed typing rules that generate constraints for that solver.

We extend our mode expressions to include inference variables (α) both for whole mode triples and for each individual mode axis:

$$\begin{aligned} \mu &::= (a, u, l) \mid \alpha \\ a &::= \textsc{many} \mid \textsc{once} \mid \alpha \\ u &::= \textsc{unique} \mid \textsc{shared} \mid \alpha \\ l &::= \textsc{global} \mid \textsc{local} \mid \alpha \end{aligned}$$

We also define *positive mode expressions* (a^+, u^+ and l^+) and *negative mode expressions* (a^-, u^- and l^-) for each mode axis:

$$\begin{aligned} \mu^+ &::= (a^+, u^+, l^+) & \qquad \mu^- &::= (a^-, u^-, l^-) \\ a^+ &::= a \mid a^+ \vee a^+ \mid \dagger^{-1}(u^-) & a^- &::= a \mid a^- \wedge a^- \\ u^+ &::= u \mid u^+ \vee u^+ \mid \dagger(a^-) & u^- &::= u \mid u^- \wedge u^- \\ l^+ &::= l \mid l^+ \vee l^+ & l^- &::= l \mid l^- \wedge l^- \end{aligned}$$

The positive expressions are all extended with join ($\vee$) and the negative expressions with meet ($\wedge$). The positive uniqueness expressions also include applications of the $\dagger$ function to negative affinity expressions, whilst positive affinity expressions include applications of the inverse of the $\dagger$ function to negative uniqueness expressions.

Our simple constraint solver is able to solve sets of constraints of the forms:

$$a^+ \leq a^- \qquad u^+ \leq u^- \qquad l^+ \leq l^- \qquad \mu^+ \leq \mu^- \qquad \mu = \mu$$

Solving such constraints is essentially just transitive closure.

The rules in Figure 2 have three issues that prevent them being used directly to generate constraints for our solver:

(1) The LAM rule uses locks, which act on the modes in Γ which are still being inferred.
(2) The SUB rule is not syntax-directed.
(3) The context joining operation (+) is non-deterministic when used to split a context

We resolve issue (1) by switching to a different presentation of contexts, as described in Section 3.7.

We resolve issue (2) by integrating submoding directly into carefully chosen other rules, making the system syntax-directed.

We resolve (3) by using a context splitting operation that is parameterised by the free variables of the subterms:

$$\curlyvee_{(X,Y)}(\Gamma) = \Gamma_X + \Gamma_Y$$

where Γ_X and Γ_Y have the same variables as Γ, but have empty bindings for any variables not in X or Y respectively.

The resulting syntax-directed system is formulated in Figure 11. We indicate generation of a new constraint to solve with side-conditions on the rules. The definition of $\curlyvee$ is:

$$\begin{aligned}
\curlyvee_{(X,Y)}(\varnothing) &= \varnothing + \varnothing \\
\curlyvee_{(X,Y)}(\Gamma, \text{🔒}_\mu) &= \Gamma_X, \text{🔒}_\mu + \Gamma_Y, \text{🔒}_\mu \\
&\text{where } \curlyvee_{(X,Y)}(\Gamma) = \Gamma_X + \Gamma_Y \\
\curlyvee_{(X,Y)}(\Gamma, x : -) &= \Gamma_X, x : - + \Gamma_Y, x : - \\
&\text{where } \curlyvee_{(X,Y)}(\Gamma) = \Gamma_X + \Gamma_Y \\
\curlyvee_{(X,Y)}(\Gamma, x : \tau @ \mu) &= \Gamma_X, x : \tau @ \mu + \Gamma_Y, x : - \\
&\text{where } x \in X,\ x \notin Y,\ \curlyvee_{(X,Y)}(\Gamma) = \Gamma_X + \Gamma_Y \\
\curlyvee_{(X,Y)}(\Gamma, x : \tau @ \mu) &= \Gamma_X, x : - + \Gamma_Y, x : \tau @ \mu \\
&\text{where } x \notin X,\ x \in Y,\ \curlyvee_{(X,Y)}(\Gamma) = \Gamma_X + \Gamma_Y \\
\curlyvee_{(X,Y)}(\Gamma, x : \tau @ (a, u, l)) &= \Gamma_X, x : \tau @ (a, \mathtt{shared}, l) + \Gamma_Y, x : \tau @ (a, \mathtt{shared}, l) \\
&\text{where } a \leq \mathtt{many},\ x \in X,\ x \in Y,\ \curlyvee_{(X,Y)}(\Gamma) = \Gamma_X + \Gamma_Y
\end{aligned}$$

Note that the last case includes a side condition on a that adds a constraint to be solved by our constraint solver. The operation is associative on the sets of variables, so we allow it to be used on n-tuples of sets rather than just pairs..

VAR

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

LAM

$$\frac{\Gamma, \text{🔒}_{\mu_1}, x : \tau_1 \mathbin{@} \mu_2 \vdash e : \tau_2 \mathbin{@} \mu_3}{\Gamma \vdash \lambda x.\, e : (\tau_1 \mathbin{@} \mu_2 \to \tau_2 \mathbin{@} \mu_3) \mathbin{@} \mu_1}$$

$$\frac{\Gamma \vdash e : \tau_1 \mathbin{@} \mu}{\Gamma \vdash \mathrm{inl}\, e : \tau_1 + \tau_2 \mathbin{@} \mu} \ \text{INL}$$

APP

$$\frac{\begin{array}{c}\curlyvee_{(\mathrm{FV}(e_1),\mathrm{FV}(e_2))}(\Gamma) = \Gamma_1 + \Gamma_2 \\ \Gamma_1 \vdash e_1 : (\tau_1 \mathbin{@} \mu_1 \to \tau_2 \mathbin{@} \mu_2) \mathbin{@} \mu_3 \\ \Gamma_2 \vdash e_2 : \tau_1 \mathbin{@} \mu_1 \qquad \mu_2 \leq \mu_4\end{array}}{\Gamma \vdash e_1\, e_2 : \tau_2 \mathbin{@} \mu_4}$$

$$\frac{\Gamma \vdash e : \tau_1 \mathbin{@} \mu}{\Gamma \vdash \mathrm{inr}\, e : \tau_1 + \tau_2 \mathbin{@} \mu} \ \text{INR}$$

SPLIT

$$\frac{\begin{array}{c}\curlyvee_{(\mathrm{FV}(e_1),\mathrm{FV}(e_2))}(\Gamma) = \Gamma_1 + \Gamma_2 \\ \Gamma_1 \vdash e_1 : \tau_1 \times \tau_2 \mathbin{@} \mu_1 \\ \Gamma_2, x : \clubsuit \mathbin{@} \mu_1, y : \tau_1 \mathbin{@} \mu_1, z : \tau_2 \mathbin{@} \mu_1 \vdash e_2 : \tau_3 \mathbin{@} \mu_2\end{array}}{\Gamma \vdash \mathrm{let}\,(x, y, z) = e_1\, \mathrm{in}\, e_2 : \tau_3 \mathbin{@} \mu_2}$$

$$\frac{}{\Gamma \vdash () : \mathbb{1} \mathbin{@} \mu} \ \text{UNIT}$$

LET

$$\frac{\begin{array}{c}\curlyvee_{(\mathrm{FV}(e_1),\mathrm{FV}(e_2))}(\Gamma) = \Gamma_1 + \Gamma_2 \\ \Gamma_1 \vdash e_1 : \tau_1 \mathbin{@} \mu_1 \qquad \Gamma_2, x : \tau_1 \mathbin{@} \mu_1 \vdash e_2 : \tau_2 \mathbin{@} \mu_2\end{array}}{\Gamma \vdash \mathrm{let}\ x = e_1\, \mathrm{in}\, e_2 : \tau_2 \mathbin{@} \mu_2}$$

$$\frac{\begin{array}{c}\curlyvee_{(\mathrm{FV}(e_1),\mathrm{FV}(e_2))}(\Gamma) = \Gamma_1 + \Gamma_2 \\ \Gamma_1 \vdash e_1 : \tau_1 \mathbin{@} \mu_1 \\ \Gamma_2 \vdash e_2 : \tau_2 \mathbin{@} \mu_2 \qquad \mu_1 \vee \mu_2 \leq \mu_3\end{array}}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \mathbin{@} \mu_3} \ \text{PAIR}$$

CASE

$$\frac{\begin{array}{c}\curlyvee_{(\mathrm{FV}(e_1),\mathrm{FV}(e_2)\cup\mathrm{FV}(e_3))}(\Gamma) = \Gamma_1 + \Gamma_2 \\ \Gamma_1 \vdash e_1 : \tau_1 + \tau_2 \mathbin{@} \mu_1 \\ \Gamma_2, x_1 : \tau_1 \mathbin{@} \mu_1 \vdash e_2 : \tau_3 \mathbin{@} \mu_2 \\ \Gamma_2, x_2 : \tau_2 \mathbin{@} \mu_1 \vdash e_3 : \tau_3 \mathbin{@} \mu_3 \qquad \mu_2 \vee \mu_3 \leq \mu_4\end{array}}{\Gamma \vdash \mathrm{case}\ e_1 \,\{\, \mathrm{inl}\ x_1 \to e_2; \mathrm{inr}\ x_2 \to e_3 \,\} : \tau_3 \mathbin{@} \mu_4}$$

BORROW

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

REUSE

$$\frac{\begin{array}{c}\curlyvee_{(\mathrm{FV}(e_1),\mathrm{FV}(e_2),\mathrm{FV}(e_3))}(\Gamma) = \Gamma_1 + \Gamma_2 + \Gamma_3 \\ \Gamma_1 \vdash e_1 : \clubsuit \mathbin{@} (a_1, \textsc{unique}, \textsc{global}) \\ \Gamma_2 \vdash e_2 : \tau_1 \mathbin{@} (a_2, u_1, \textsc{global}) \\ \Gamma_3 \vdash e_3 : \tau_2 \mathbin{@} (a_3, u_2, \textsc{global}) \qquad a_2 \vee a_3 \leq a_4 \qquad u_1 \vee u_2 \leq u_3\end{array}}{\Gamma \vdash \mathrm{reuse}\ e_1\ \mathrm{in}\ (e_2, e_3) : \tau_1 \times \tau_2 \mathbin{@} (a_4, u_3, \textsc{global})}$$

Fig. 11. Syntax-Directed Inference Rules

$$\frac{\Gamma \vdash e : \tau \mathbin{@} (\textsc{many}, u, l)}{\Gamma \vdash \mathrm{box}_{\mathrm{M}}\, e : \square^{\mathrm{M}}\tau \mathbin{@} (a, u, l)}\ \textsc{M-box} \qquad \frac{\Gamma \vdash e : \square^{\mathrm{M}}\tau \mathbin{@} (a_1, u, l)}{\Gamma \vdash \mathrm{unbox}_{\mathrm{M}}\, e : \tau \mathbin{@} (a_2, u, l)}\ \textsc{M-unbox}$$

$$\frac{\Gamma \vdash e : \tau \mathbin{@} (a, u_1, l)}{\Gamma \vdash \mathrm{box}_{\mathrm{S}}\, e : \square^{\mathrm{S}}\tau \mathbin{@} (a, u_2, l)}\ \textsc{S-box} \qquad \frac{\Gamma \vdash e : \square^{\mathrm{S}}\tau \mathbin{@} (a, u, l)}{\Gamma \vdash \mathrm{unbox}_{\mathrm{S}}\, e : \tau \mathbin{@} (a, \textsc{shared}, l)}\ \textsc{S-unbox}$$

$$\frac{\Gamma \vdash e : \tau \mathbin{@} (a, u_1, \textsc{global})}{\Gamma \vdash \mathrm{box}_{\mathrm{G}}\, e : \square^{\mathrm{G}}\tau \mathbin{@} (a, u_2, l)}\ \textsc{G-box} \qquad \frac{\Gamma \vdash e : \square^{\mathrm{G}}\tau \mathbin{@} (a, u, l_1)}{\Gamma \vdash \mathrm{unbox}_{\mathrm{G}}\, e : \tau \mathbin{@} (a, \textsc{shared}, l_2)}\ \textsc{G-unbox}$$

Fig. 12. Syntax-Directed Inference Rules for Boxes

C GRADED CALCULUS

C.1 Syntax

$$\begin{aligned}
\Gamma &::= \varnothing \mid \Gamma, x :^q A \\
A, B, C &::= \mathbb{1} \mid A + B \mid A \times B \mid \square^q A \mid {}^q A \to A \\
V, W &::= x \mid () \mid \mathrm{inl}\, V \mid \mathrm{inr}\, V \mid (V, W) \\
&\quad\mid \mathrm{box}_q\, V \mid \lambda x.\, M \\
M, N &::= \mathrm{return}_q\ V \mid V\, W \mid M \ \mathrm{to}\ x.\, N \\
&\quad\mid \mathrm{let}_q\ \mathrm{box}_r\, x = V \,\mathrm{in}\, M \mid \mathrm{let}_q\ (x, y) = V \,\mathrm{in}\, M \\
&\quad\mid \mathrm{case}_q\, V \,\{\, \mathrm{inl}\, x \to M ; \mathrm{inr}\, y \to N \,\}
\end{aligned}$$

C.2 Typing Rules

$$\frac{}{0 \cdot \Gamma_1, x :^1 A \vdash^{\mathrm{v}} x : A, 0 \cdot \Gamma_2}\ \textsc{var} \qquad \frac{}{0 \cdot \Gamma \vdash^{\mathrm{v}} () : \mathbb{1}}\ \textsc{unit} \qquad \frac{\Gamma \vdash^{\mathrm{v}} V : A}{\Gamma \vdash^{\mathrm{v}} \mathrm{inl}\, V : A + B}\ \textsc{inl}$$

$$\frac{\Gamma \vdash^{\mathrm{v}} V : B}{\Gamma \vdash^{\mathrm{v}} \mathrm{inr}\, V : A + B}\ \textsc{inr} \qquad \frac{\Gamma_1 \vdash^{\mathrm{v}} V : A \qquad \Gamma_2 \vdash^{\mathrm{v}} W : B}{\Gamma_1 + \Gamma_2 \vdash^{\mathrm{v}} (V, W) : A \times B}\ \textsc{pair} \qquad \frac{\Gamma \vdash^{\mathrm{v}} V : A}{q \cdot \Gamma \vdash^{\mathrm{v}} \mathrm{box}_q\, V : \Box^q A}\ \textsc{box}$$

$$\frac{\Gamma, x :^q A \vdash^{\mathrm{c}} M : B}{\Gamma \vdash^{\mathrm{v}} \lambda x. M : {}^q A \to B}\ \textsc{lam} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash^{\mathrm{v}} V : {}^q A \to B \\ \Gamma_2 \vdash^{\mathrm{v}} W : A\end{array}}{\Gamma_1 + q \cdot \Gamma_2 \vdash^{\mathrm{c}} V\, W : B}\ \textsc{app}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash^{\mathrm{v}} V : \Box^r A \\ \Gamma_2, x :^{q \cdot r} A \vdash^{\mathrm{c}} M : B\end{array}}{q \cdot \Gamma_1 + \Gamma_2 \vdash^{\mathrm{c}} \mathrm{let}_q\, \mathrm{box}_r\, x = V \,\mathrm{in}\, M : B}\ \textsc{unbox} \qquad \frac{\Gamma_1 \vdash^{\mathrm{v}} V : A}{q \cdot \Gamma_1 \vdash^{\mathrm{c}} \mathrm{return}\ V : A}\ \textsc{return}$$

$$\frac{\Gamma_1 \vdash^{\mathrm{c}} M : A \qquad \Gamma_2, x :^1 A \vdash^{\mathrm{c}} N : B}{\Gamma_1 + \Gamma_2 \vdash^{\mathrm{c}} M \,\mathrm{to}\, x. N : B}\ \textsc{let} \qquad \frac{\Gamma_1 \vdash^{\mathrm{c}} M : B \qquad \Gamma_2 \le \Gamma_1}{\Gamma_2 \vdash^{\mathrm{c}} M : B}\ \textsc{sub}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash^{\mathrm{v}} V : A \times B \\ \Gamma_2, x :^q A, y :^q B \vdash^{\mathrm{c}} M : C\end{array}}{q \cdot \Gamma_1 + \Gamma_2 \vdash^{\mathrm{c}} \mathrm{let}_q\, (x, y) = V \,\mathrm{in}\, M : C}\ \textsc{split}$$

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

C.3 Extension to In-Place Update

$$\begin{array}{rl} A, B ::= & \cdots \mid \clubsuit \\ M, N ::= & \cdots \mid \mathrm{let}_q\, (x, y, z) = V \,\mathrm{in}\, M \\ & \mid \mathrm{reuse}_q\, V \text{ in } (W, W') \end{array}$$

$$\frac{\Gamma_1 \vdash^{\mathrm{v}} V : \clubsuit \qquad \Gamma_2 \vdash^{\mathrm{v}} W : A \qquad \Gamma_3 \vdash^{\mathrm{v}} W' : A}{\Gamma_1 + q \cdot \Gamma_2 + q \cdot \Gamma_3 \vdash^{\mathrm{c}} \mathrm{reuse}_q\, V \text{ in } (W, W') : \Box^q (A \times B)}\ \textsc{reuse}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash^{\mathrm{v}} V : A \times B \\ \Gamma_2, x :^q \clubsuit, y :^q A, z :^q B \vdash^{\mathrm{c}} M : C\end{array}}{q \cdot \Gamma_1 + \Gamma_2 \vdash^{\mathrm{c}} \mathrm{let}_q (x, y, z) = V \,\mathrm{in}\, M : C}\ \textsc{destruct}$$

D EQUATIONS FOR THE EXTENDED SEMIRING

Addition:

$$\begin{aligned} q + \mathbf{0} = \mathbf{0} + q &= q \\ \{\mathbf{S}, \mathbf{SM}\} + \{\mathbf{S}, \mathbf{SM}\} &= \mathbf{SM} \\ \{\mathbf{1}, \mathbf{M}, \bot\} + \{\mathbf{S}, \mathbf{SM}\} = \{\mathbf{S}, \mathbf{SM}\} + \{\mathbf{1}, \mathbf{M}, \bot\} &= \bot \\ \{\mathbf{1}, \mathbf{M}, \bot\} + \{\mathbf{1}, \mathbf{M}, \bot\} &= \bot \\ (\mathbf{S} \to \{\mathbf{1}, \mathbf{M}, \bot\}) + \{\mathbf{S}, \mathbf{SM}, \mathbf{1}, \mathbf{M}, \bot\} = \{\mathbf{S}, \mathbf{SM}, \mathbf{1}, \mathbf{M}, \bot\} + (\mathbf{S} \to \{\mathbf{1}, \mathbf{M}, \bot\}) &= \mathbf{S} \to \bot \\ (\mathbf{S} \to \{\mathbf{1}, \mathbf{M}, \bot\}) + (\mathbf{S} \to \{\mathbf{1}, \mathbf{M}, \bot\}) &= \mathbf{S} \to \bot \end{aligned}$$

Multiplication:

$$\begin{aligned}
\mathbf{0} \cdot q = q \cdot \mathbf{0} &= \mathbf{0} & \{\mathsf{S}, \mathsf{S} \to \mathbf{1}\} \cdot (\mathsf{S} \to a) &= \mathsf{S} \to a \\
\mathbf{1} \cdot q = q \cdot \mathbf{1} &= q & \{\mathsf{SM}, \mathbf{M}, \bot, \mathsf{S} \to \mathbf{M}, \mathsf{S} \to \bot\} \cdot \mathsf{S} \to a &= \mathsf{S} \to \bot \\
\mathsf{S} \cdot \mathsf{S} &= \mathsf{S} & (\mathsf{S} \to \mathbf{1}) \cdot \mathsf{S} &= \mathsf{S} \\
\mathsf{S} \cdot \{\mathsf{SM}, \mathbf{M}, \bot\} &= \mathsf{SM} & (\mathsf{S} \to \{\mathbf{M}, \bot\}) \cdot \mathsf{S} &= \mathsf{SM} \\
\{\mathbf{M}, \bot\} \cdot \{\mathsf{S}, \mathsf{SM}\} &= \mathsf{SM} & (\mathsf{S} \to a) \cdot (\mathsf{SM}) &= \mathsf{SM} \\
\mathbf{M} \cdot \mathbf{M} &= \mathbf{M} & (\mathsf{S} \to \{\mathbf{1}, \mathbf{M}\}) \cdot \mathbf{M} &= \mathsf{S} \to \mathbf{M} \\
\mathbf{M} \cdot \bot &= \bot & (\mathsf{S} \to \bot) \cdot \mathbf{M} &= \mathsf{S} \to \bot \\
\bot \cdot \{\mathbf{1}, \mathbf{M}, \bot\} &= \bot & (\mathsf{S} \to a) \cdot \bot &= \mathsf{S} \to \bot
\end{aligned}$$