Modes as Skew-Semirings

ANTON LORENZEN, University of Edinburgh, United Kingdom
ISHIKA GUPTA, University of Edinburgh, United Kingdom
SAM LINDLEY, University of Edinburgh, United Kingdom

The OxCaml compiler implements a bespoke mode system that enables features such as stack allocation, in-place reuse and safe concurrency. We show that OxCaml modes have the algebraic structure of skew-semirings and capture this structure in a graded modal calculus in call-by-push-value style. We prove the calculus sound for all skew-semirings, with reduction preserving both types and modes. We also introduce a bidirectional mode inference algorithm that generalises the core of OxCaml's inference algorithm to this setting. We then study the equational theory of OxCaml's mode system in our calculus and show that a suitable extension enables several natural refactorings not supported by the original theory. Our results clarify the theoretical basis of OxCaml's mode system and lay a foundation for future extensions.

1 Introduction

The OxCaml compiler [Jane Street 2026] extends the OCaml programming language for increased performance and additional safety in a concurrent setting. One of its main extensions is the mode system, which can perform safe stack allocation [Dolan and White 2022], track uniqueness and borrowing [Lorenzen et al. 2024], and guarantee the absence of data races [Georges et al. 2025]. Modes are now integrated deeply into the compiler and have motivated additional improvements to the language [Peters et al. 2026].

But what *is* a mode? And what kind of property could one expect to be enforceable using modes? This is surprisingly hard to answer in general without restricting oneself to the concrete instantiations given in previous work. Lorenzen et al. [2024] attempt to answer this question by relating the modes of OxCaml to the line of work on graded modal type systems [Abel and Bernardy 2020; Atkey 2018; Choudhury et al. 2021; McBride 2016], a general theory of resources that can enforce any property expressible as a semiring. There are obvious parallels between OxCaml's modes and graded modal types, but the devil is in the detail: OxCaml is a call-by-value language, while most work on graded modal types uses call-by-name semantics; and OxCaml's modes do not naturally satisfy the laws of a semiring.

Two recent developments have made it possible to overcome these two challenges. A growing list of researchers are adopting call-by-value semantics for (graded) modal types [Ahman 2023; Bianchini et al. 2023, 2026; Jang et al. 2024; Lorenzen et al. 2026; Tang and Lindley 2026; Tang et al. 2025], often by restricting boxing to values [Curien et al. 2016]. Moreover, Wood and Atkey [2020] observe that full semiring laws are not necessary to ensure soundness of graded modal type systems. Instead, *skew*-semirings suffice, a restriction of a semiring in which the associativity and distributivity laws hold only as inequalities.

In this paper, we present the existing mode system of OxCaml as an instance of a general *skew*-graded modal type theory. We present a call-by-value calculus parameterised by skew-semirings and show that the previously-studied modes of OxCaml give rise to a skew-semiring structure. From this perspective, a mode is thus any property of values expressible as a skew-semiring.

We use this skew-graded modal calculus to generalise various known results about OxCaml's mode system to arbitrary skew-semirings. First, we study the equational theory of our calculus. Here we notice a limitation: In the mode system of OxCaml it is sometimes impossible to abstract a

Authors' Contact Information: Anton Lorenzen, University of Edinburgh, United Kingdom, anton.lorenzen@ed.ac.uk; Ishika Gupta, University of Edinburgh, United Kingdom, s2773546@ed.ac.uk; Sam Lindley, University of Edinburgh, United Kingdom, Sam.Lindley@ed.ac.uk.

value out of a term, because the resulting term is no longer well-typed. We show that we would need the full laws of a semiring to recover the equational theory. In fact, there is a direct correspondence between the laws of a (skew-)semiring and the beta-reduction rules of the language: for any law that holds as a strict inequality, we can find a corresponding term where an inner value cannot be abstracted into a let-binding. Second, we give a new mode inference algorithm over arbitrary skew-semirings. The existing inference algorithm of OxCaml can infer all mode annotations in terms and types, but as it derives much of its power from the specific structure of the previously-studied modes, it does not directly generalise to arbitrary skew-semirings. Instead, we define a new bidirectional inference algorithm that can be used to infer all mode annotations in terms, and in contravariant positions of types.

We follow the formal mode theory introduced by Lorenzen et al. [2024] and extended by Georges et al. [2025]. However, at the time of writing, the OxCaml compiler often behaves slightly differently, as mode crossing [Peters et al. 2026], a technique for implicitly coercing between modes, significantly extends the power of the mode theory. In the last part of this paper, taking inspiration from mode crossing, we extend the skew-semiring of modes in order to enable more laws to hold as equalities. We compare our design against the approach implemented in the OxCaml compiler.

In summary, we make the following contributions:

- We show how to model the mode system of OxCaml in terms of semiring operations. We notice that the usual semiring laws only hold as inequalities, and show that this is enough for soundness but disallows some refactorings (Section 2).
- We define a *skew*-graded modal type system in a call-by-push-value (CBPV) [Levy 1999] calculus, parameterised by skew-semirings and modalities (Section 3).
- We prove that the usual CBPV semantics preserves modes for all skew-semirings (Section 4).
- We define the equational theory of our calculus, and show that the commuting conversions and most eta-laws hold for all skew-semirings, while the beta-laws rely on the full semiring laws (Section 5).
- We give a bidirectional mode inference algorithm that works generically for all skew-semirings with invertible modalities. The algorithm can infer all mode annotations in contravariant positions syntactically, without requiring a special solver (Section 6).
- We compare our approach with the implementation in the OxCaml compiler (Section 7).

For simplicity, the main examples in the body of the paper are based on the uniqueness and affinity modes of OxCaml [Lorenzen et al. 2024]. Appendix A incorporates portability and contention modes [Georges et al. 2025]. Appendix B provides definitions omitted from the body of the paper.

AI Use. The main statements in this document were first proven manually by the authors, and we now provide a Lean formalisation as an artifact. The artifact was generated from the manual proofs, and then extended further, using AI. The authors have checked its correctness and take responsibility for any errors that remain.

2 Overview

In this overview, we give a short introduction to the mode system of OxCaml and its semantics. Concurrently, we show how to model the mode system in terms of addition and multiplication operations and define the laws of a skew-semiring. We connect these aspects by the observation that each skew-semiring law corresponds to a beta-reduction in the calculus, and when the inequality is strict, the corresponding beta-expansion is not possible.

Our examples are written in a variant of OxCaml, which implements exactly the mode theory of Lorenzen et al. [2024] and Georges et al. [2025]. This version existed in the past, but is not actively

maintained). The actual OxCaml compiler, currently at version 5.2.0+ox, behaves a little differently due to mode crossing [Peters et al. 2026]; we will revisit this distinction in Section 7.

2.1 Uniqueness and Aliasing

OxCaml uses modes to track runtime properties of values statically. In this work, we focus on the uniqueness mode, which tracks whether a value has exactly one reference pointing to it [Lorenzen et al. 2024]. For example, we can design the following API for mutable arrays of floats:

```
val alloc : length:int -> init:float -> Floatarray.t @ unique
val write : idx:int -> float -> Floatarray.t @ unique -> Floatarray.t @ unique
```

Modes are recorded on types using the `@` symbol (`@` binds more tightly than `->`). When we allocate a new array, we obtain a unique reference to it. We can then specify that writes to the array require a unique reference, which guarantees that no other part of the program can observe the write. However, modes do not force us to keep values unique forever. At any time, we can turn our unique reference into an *aliased* reference, which permits multiple references to the same value. At that point, it is no longer possible to write to the array, but we may still read from it:

```
val read : idx:int -> Floatarray.t @ aliased -> float
```

This submoding from unique to aliased references is useful in practice. A mutable data structure is often used in two phases: in the first phase, it is kept unique and mutated in-place, while in the second phase, it is aliased and only read from. Unlike a linear type system [Wadler 1990], which disallows ever aliasing linear values, modes let us express this two-phased design rather directly.

2.2 Affinity

Uniqueness is challenging to maintain in a higher-order language, where unique values may be captured in closures. Consider a function that captures or returns a unique array, and is then invoked multiple times:

```
let arr = alloc ~length:10 ~init:0.0 in
let f = (fun () -> arr) in
use_unique (f(), f())
```

This program should be rejected by the compiler, since it returns two references to the same array, and they cannot both be unique. To detect this, OxCaml uses an *affinity* mode, which tracks whether a value may be used more than once. Most values may be used *many* times, but closures that capture unique values may only be used *once*. As with uniqueness, we can turn a value that is usable many times into a value that is usable at most once, but not the other way around.

2.3 A Semi-Lattice of Modes

Modes are subject to a semi-lattice structure, as illustrated in the Hasse diagram in Figure 1. Lines between modes indicate the submoding relationship, whereby the lower mode may be coerced to the upper mode. Intuitively, this can be read as a notion of "demand" or "resources": if a function demands an argument at a higher mode, then it requires fewer resources than if it had demanded the argument at a lower mode. We write $a \leq b$ whenever a is below b in the diagram and they are connected by a path of lines. [1]

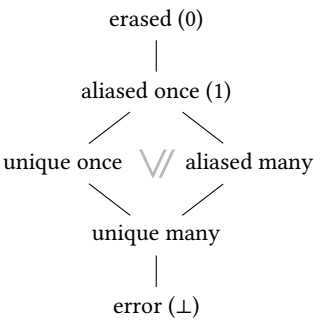


Fig. 1. A lattice of the uniqueness and affinity modes of OxCaml.

[1] Perhaps counter-intuitively, $a \leq b$ says that a needs "more resources" than b. It would be equally valid to use the opposite ordering, but then $\bot$ would be at the top of the lattice.

The lattice includes the modes of OxCaml in four pairs, where the first mode of each pair is either unique or aliased, and the second mode is either once or many. The `unique once` and `aliased many` modes are incomparable, as we can coerce from unique to aliased, and from many to once, but not the other way around.

Additionally, we include two modes not present in OxCaml. The `erased` (0) mode indicates that a variable is unused in a program. Graded modal type theories typically include this mode, as it is essential for an algebraic description of modes. The `error` ($\bot$) mode indicates that a variable is used in an ill-moded manner. This mode enables us to give an algebraic description of operations on modes, where a partial operation may return $\bot$, and unless stated otherwise, all operations below map $\bot$ to $\bot$. In practice, OxCaml rejects programs that would contain the error mode.

2.4 Graded Modal Types Over Semirings

At first glance, OxCaml's modes seem similar to the coeffects of Petricek et al. [2014], the resources of McBride [2016], the quantities and grades of Atkey [2018] and Choudhury et al. [2021], and the modalities of Abel and Bernardy [2020] and Abel et al. [2023]. These works also define annotations that track properties not directly expressible using types, and include a submoding relation.

The key idea of these graded modal type systems is to track resources in a program, by combining the use of variables using the operations of a semiring $(0, 1, +, \cdot)$. If a variable does not occur in a program, it is at mode 0; otherwise, we determine its mode from the context, and write `use_q x` for a use of `x` at mode q. If a variable is used at modes q and r in succession, as `use_q x; use_r x`, then its mode is $q + r$. Similarly, in `use_q (x, x)` the variable `x` is at mode $q + q$.

When a variable is used in a function, like `use_q (fun _ -> use_r x)`, its mode is $q \cdot r$, where we multiply the mode of the outer function with the mode of the variable in the function. Intuitively, the definition says that we need r-many resources in each of the q-times that the function can be called. This definition diverges somewhat from how graded modal type theories typically define multiplication, but makes sense in our setting: whereas most graded modal type theories are call-by-name, OxCaml is call-by-value.

2.5 Semiring Operations for OxCaml

We can specify the semantics of our mode lattice using the semiring operations. When a variable is used in sequence, it must be usable `many`-times, and neither use can be `unique`:

$$q + r = \begin{cases} r & \text{if } q = 0 \\ q & \text{if } r = 0 \\ \text{aliased many} & \text{if } q \text{ and } r \text{ are aliased} \\ \bot & \text{if } q \text{ or } r \text{ is unique} \end{cases}$$

A closure may contain a `unique` use if the closure itself can only be called `once`:

$$q \cdot r = \begin{cases} 0 & \text{if } q = 0 \text{ or } r = 0 \\ r & \text{if } q \text{ is once} \\ \text{aliased many} & \text{if } q \text{ is many and } r \text{ is aliased} \\ \bot & \text{if } q \text{ is many and } r \text{ is unique} \end{cases}$$

This is the key duality condition of OxCaml's mode system: a variable may only be used uniquely in a closure if the closure itself is used at most once [Lorenzen et al. 2024, Section 3.3]. Perhaps counter-intuitively, we also define $0 \cdot \bot = \bot \cdot 0 = 0$; we'll come back to this choice in Section 7.3.

2.6 From Semirings to Beta-Reduction

The definitions above suggest that OxCaml's modes would fit into the framework of graded modal type theory over a semiring. Unfortunately, though, the operations above do not satisfy the usual semiring laws. Consider the associativity law of multiplication, which states that $q \cdot (r \cdot s) = (q \cdot r) \cdot s$. In OxCaml, we have:

$$\begin{aligned}&\text{aliased many} \cdot (\text{unique once} \cdot \text{aliased once}) = \text{aliased many} \cdot \text{aliased once} = \text{aliased many}\\ &> \bot = \bot \cdot \text{aliased once} = (\text{aliased many} \cdot \text{unique once}) \cdot \text{aliased once}\end{aligned}$$

Thus multiplication is not associative.

Can we define a graded modal type theory without the semiring laws? It turns out that this is not easily possible, as the semiring laws are directly connected to the reduction rules of the language. However, Wood and Atkey [2020] noticed that to show the soundness of beta-reduction, the semiring laws need only hold as *in*equalities. For example, consider:

```
let y = (fun _ -> use_s x) in use_q (fun _ -> use_r y)
```

Here y is used at mode $q \cdot r$, and so x is used at mode $(q \cdot r) \cdot s$. However, if we beta-reduce this term, then we obtain

```
use_q (fun _ -> use_r (fun _ -> use_s x))
```

where x is now used at mode $q \cdot (r \cdot s)$.

To prove soundness of a modal type theory, we must show that modes are preserved under beta-reduction. However, we can always weaken the mode of a variable, so if a reduction step changes the mode of a variable from q to r, we must ensure that $q \leq r$. Thus, we require that $(q \cdot r) \cdot s \leq q \cdot (r \cdot s)$; and it turns out that this inequality does indeed hold for OxCaml modes.

2.7 Beta-Expansion

If only one direction of the semiring laws is needed for soundness, why has most previous work not simply stated the laws as inequalities? We believe that part of the reason lies in the equational theory of the calculus. Whenever a law holds as a strict inequality, it is possible to find a term where the beta-reduction cannot be reversed, because the resulting term is ill-typed. We illustrate this point by adapting the example above:

```
use_many (fun _ -> use_unique (fun _ -> use_aliased x))
```

This creates a closure that can be run many times. Each time it runs, it creates a fresh unique closure containing an aliased x. We may wish to write:

```
let y = (fun _ -> use_aliased x) in use_many (fun _ -> use_unique y)
```

However, this term is not well-typed: because the resulting closure is invoked many times, it must not contain the unique variable y. We call this a failure of *beta-expansion*, and it shows that OxCaml's mode system does not enjoy the full equational theory of functional languages.

2.8 Right Distributivity

A similar issue arises with right distributivity of multiplication over addition. Consider the term:

```
let y = (fun _ -> use_r x) in use_q (y, y)
```

Here y is used at mode $q + q$ and x is used at mode $(q + q) \cdot r$. However, if we beta-reduce this term, then we obtain

```
use_q ((fun _ -> use_r x), (fun _ -> use_r x))
```

where x is now used at mode $q \cdot r + q \cdot r$. Again, we require only that $(q + q) \cdot r \leq q \cdot r + q \cdot r$ for soundness, and we need the reverse direction for beta-expansion. However, in OxCaml, this law holds only as an inequality. Consider the term:

```
use_unique ((fun _ -> use_aliased x), (fun _ -> use_aliased x))
```

It builds a tuple of two unique closures. We may wish to abstract this closure out and write:

```
let y = (fun _ -> use_aliased x) in use_unique (y, y)
```

However, this program is not well-typed: the resulting tuple is unique and thus expects its arguments to be unique, but y is aliased. In contrast, left distributivity does hold as an equality in OxCaml. Dually to the example above, this implies that pairs can always be substituted in and out of functions.

2.9 Calling Functions at Unit

In order to complete the semiring structure, we must still define a unit for multiplication. In our calculus, the unit mode is used to constrain the mode at which a function is called. In our running example, we choose the `aliased once` mode. Erased functions must not be called: since OxCaml is an impure language, calling an erased function could cause an otherwise observable side-effect. All other modes can be subtyped to the aliased mode.

Beta Reduction. Consider `(fun () -> use_q x)()`, where x is used at mode q in a closure that is immediately called. This term contains x at mode $1 \cdot q$, and after beta-reduction, we obtain `use_q x`, which is at mode q. Thus we require that $1 \cdot q \leq q$ for beta-reduction, and the converse $1 \cdot q \geq q$ for beta-expansion. Both hold in OxCaml.

Eta Expansion. Given the use of a function `use_q f`, we can perform eta-expansion to obtain `use_q (fun x -> f x)`, where f is called at mode 1 and thus required at mode $q \cdot 1$. We need $q \leq q \cdot 1$ to perform eta-expansion, and the converse $q \geq q \cdot 1$ to perform eta-reduction.

Only the first holds in OxCaml. Eta-reduction is not always possible, because eta-expansion may upgrade the mode of a closure. For `f @ aliased`, the program `use_unique (fun z -> f z)` type-checks even though f itself is not unique — and so it is not possible to undo the eta-expansion.

2.10 Why OxCaml's Modalities are not Modes

In OxCaml, a *modality* is used to change the mode of a record field. For example, we might write

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

in order to indicate that the `aliased` field will always be aliased, even if the record itself is unique. Given the mode of the record q, we write $q * \mu$ for the mode of the field under the modality μ.

The design of modalities is a bit strange when taking the perspective of graded modal type theory, but they make sense when considering the semiring laws as inequalities. First, graded modal type theory typically defines modalities through multiplication from the left, just like our design for functions. However, in OxCaml, it makes more sense to view modalities as a multiplication from the *right*. The aliased modality is defined as:

$$q * \text{aliased} = q \cdot \texttt{aliased once}$$

The distinction between left- and right-multiplication does not matter in semirings, where multiplication is associative. In our skew-semiring, however, we cannot define the aliased modality as left-multiplication. Moreover, in semiring-based graded modal calculi, modalities are typically (a subset of) modes. This can be motivated algebraically: in a semiring, we have

$$q * \mu = (q \cdot 1) * \mu = q \cdot (1 * \mu)$$

and thus a modality can always be identified with the mode that it sends 1 to. But in a skew-semiring, this calculation holds only as an inequality, so $q * \mu \leq q \cdot (1 * \mu)$, and it is possible to implement modalities that are not modes. OxCaml also includes the `many` modality, defined as:

$$q * \text{many} = \begin{cases} 0 & \text{if } q = 0 \\ \text{aliased many} & \text{if } q \text{ is aliased} \\ \text{unique many} & \text{if } q \text{ is unique} \\ \bot & \text{if } q = \bot \end{cases}$$

This modality cannot be expressed as multiplication by a mode.

2.11 Laws for Modalities

Again, we can derive laws for how modalities interact with our other operations by considering beta-reduction:

```
let y = { aliased = x } in use_q (fun _ -> use_r y)
```

Here y is used at mode $q \cdot r$ and x at mode $(q \cdot r) * \text{aliased}$. However, if we beta-reduce this term, we obtain

```
use_q (fun _ -> use_r { aliased = x })
```

where x is now used at mode $q \cdot (r * \text{aliased})$. For beta-reduction to be sound, we thus require that $(q \cdot r) * \mu \leq q \cdot (r * \mu)$. This inequality is strict in OxCaml, and we cannot always let-bind a record out of a closure:

```
use_many (fun _ -> use_unique { aliased = x })
```

Similarly, for the addition operation, we have:

```
let y = { aliased = x } in use_r (y , y)
```

Here y is used at mode $r + r$, and x is used at mode $(r + r) * \text{aliased}$. However, if we beta-reduce this term, we obtain:

```
use_r ({ aliased = x } , { aliased = x })
```

where x is now used at mode $r * \text{aliased} + r * \text{aliased}$. For beta-reduction to be sound, we thus require that $(r + r) * \mu \leq r * \mu + r * \mu$. This inequality is also strict in OxCaml, and we cannot always let-bind a record out of a tuple:

```
use_unique ({ aliased = x } , { aliased = x })
```

2.12 Skew-Semirings

In Figure 2 we collect together all of the laws that we consider in this paper, inspired by the skew-semiring laws of Wood and Atkey [2020]. We divide our figure into three parts: the structural laws at the top, soundness laws on the left and equational laws on the right.

The structural laws enforce important structural properties of our calculus. We use a meet operation $\sqcap$ to induce a partial order $\leq$ on modes by setting $q \leq r$ iff $q \sqcap r = q$. We require that all operations distribute over meets, which ensures that they are monotone in this order. We also include the law $q \cdot 0 = 0$, which is useful for weakening, and the law $1 \cdot q = q$, which is useful for the equational theory and mode inference (Wood and Atkey [2020] require only $q \cdot 0 \geq 0$ and $1 \cdot q \leq q$.)

The soundness laws are required for beta-reduction to preserve modes, and for eta-expansion to be sound. While eta-expansion is not needed for soundness, per se, we still include it in this column due to its widespread use in normalisation algorithms. All laws in this column hold for our instantiation of OxCaml's uniqueness and affinity modes.

Structural Laws.

$$q \sqcap (r \sqcap s) = (q \sqcap r) \sqcap s \qquad q \sqcap r = r \sqcap q \qquad q \sqcap q = q$$
$$q + (r + s) = (q + r) + s \qquad q + r = r + q \qquad q + 0 = 0 + q = q$$
$$(q \sqcap r) + s = q + s \sqcap r + s \qquad q + (r \sqcap s) = q + r \sqcap q + s$$
$$(q \sqcap r) \cdot s = q \cdot s \sqcap r \cdot s \qquad q \cdot (r \sqcap s) = q \cdot r \sqcap q \cdot s$$
$$(q \sqcap r) * \mu = q * \mu \sqcap r * \mu \qquad q \cdot 0 = 0 \qquad 1 \cdot q = q$$

Beta Reduction Laws.

$$0 \cdot q \leq 0$$
$$0 * \mu \leq 0$$
$$(q \cdot r) \cdot s \leq q \cdot (r \cdot s)$$
$$q \cdot r + q \cdot s \leq q \cdot (r + s)$$
$$(q + r) \cdot s \leq q \cdot s + r \cdot s$$
$$(q \cdot r) * \mu \leq q \cdot (r * \mu)$$
$$(q + r) * \mu \leq q * \mu + r * \mu$$

Eta Expansion.

$$q \leq q \cdot 1$$

Beta Expansion Laws.

$$0 \cdot q \geq 0$$
$$0 * \mu \geq 0$$
$$(q \cdot r) \cdot s \geq q \cdot (r \cdot s)$$
$$q \cdot r + q \cdot s \geq q \cdot (r + s)$$
$$(q + r) \cdot s \geq q \cdot s + r \cdot s$$
$$(q \cdot r) * \mu \geq q \cdot (r * \mu)$$
$$(q + r) * \mu \geq q * \mu + r * \mu$$

Eta Reduction.

$$q \geq q \cdot 1$$

Abstracting out of Cases.

$$q + r = q \sqcap r$$

Fig. 2. The collected laws for skew-semirings.

The equational laws are largely the opposite of the soundness laws, and can be used to undo the respective transformations. These laws are not needed for soundness, but can be useful for refactoring programs. We also include a law for "abstracting out of cases", enabling hoisting a use out of one branch of a case. This rule can be convenient for refactoring, but is generally not available in uniqueness type systems, where the variable might still be used differently in another branch. In fact, this refactoring is possible only if the sequencing operator is trivial, as it is for the the portability and contention modes [Georges et al. 2025].

3 Graded Call-By-Push-Value

In this section, we present our calculus for skew-graded modal type theory. We take inspiration from Torczon et al. [2024] and present our calculus in call-by-push-value style [Levy 1999]. While our approach requires a separate judgement for values, we prefer CBPV over other approaches (like Levy et al. [2003]) mainly due to its elegant equational theory. For that purpose, we need an intrinsically-typed syntax, but we will omit the gray type annotations from the rest of the paper.

Syntax.

$$
\begin{aligned}
A &::= U\,\underline{B} \mid \Box^{\mu} A \mid A_1 \times A_2 \mid A_1 + A_2 \mid \mathbb{1} \\
\underline{B} &::= F^{q} A \mid {}^{q}A \to \underline{B} \\
v &::= x \mid \mathsf{thunk}\ e \mid \mathsf{box}_{\mu}\ v \mid (v_1, v_2) \mid () \mid \mathsf{inl}_A\ v \mid \mathsf{inr}_A\ v \\
e &::= \mathsf{return}_q\ v \mid \mathsf{force}\ v \mid \lambda x :^{q} A.\, e \mid e\ (v :^{q} A) \mid \mathsf{let}\, x :^{q} A \leftarrow e_1\, \mathsf{in}\, e_2 \mid v ;_q e \\
&\quad \mid \mathsf{let}_q\, \mathsf{box}_{\mu}\ (x \colon A) = v\, \mathsf{in}\, e \mid \mathsf{let}_q (x \colon A_1, y \colon A_2) = v\, \mathsf{in}\, e \mid \mathsf{case}_q\ v\ \{x \colon A_1.\, e_1;\ y \colon A_2.\, e_2\}
\end{aligned}
$$

We use the judgement $\gamma\Gamma \vdash v :^{q} A$ for values and the judgement $\gamma\Gamma \vdash e : \underline{B}$ for expressions. Contexts Γ are (ordered) lists of variables with types, and usage contexts γ are lists of modes of the same length. We occasionally write $x :^{q} A$ for $(q)(x : A)$. The skew-semiring operations $\sqcap, +, \cdot$ act pointwise on usage contexts and we write $\underline{0}$ for the all-zero usage context.

Value typing rules. $\boxed{\gamma\Gamma \vdash v :^{q} A}$

$$\frac{}{\underline{0}\Gamma, x :^{q} A, \underline{0}\Gamma' \vdash x :^{q} A}\ \textsc{var} \qquad \frac{\gamma_1 \geq \gamma_2 \qquad \gamma_1\Gamma \vdash v :^{q} A}{\gamma_2\Gamma \vdash v :^{q} A}\ \textsc{vsub}$$

$$\frac{\gamma\Gamma \vdash e : \underline{B}}{(q \cdot \gamma)\Gamma \vdash \mathsf{thunk}\ e :^{q} U\,\underline{B}}\ \textsc{thunk} \qquad \frac{}{\underline{0}\Gamma \vdash () :^{q} \mathbb{1}}\ \textsc{unit} \qquad \frac{\gamma\Gamma \vdash v :^{q*\mu} A}{\gamma\Gamma \vdash \mathsf{box}_{\mu}\ v :^{q} \Box^{\mu} A}\ \textsc{box}$$

$$\frac{\gamma_1\Gamma \vdash v_1 :^{q} A \qquad \gamma_2\Gamma \vdash v_2 :^{q} B}{(\gamma_1 + \gamma_2)\Gamma \vdash (v_1, v_2) :^{q} A \times B}\ \textsc{pair} \qquad \frac{\gamma\Gamma \vdash v :^{q} A}{\gamma\Gamma \vdash \mathsf{inl}_B\ v :^{q} A + B}\ \textsc{inl} \qquad \frac{\gamma\Gamma \vdash v :^{q} B}{\gamma\Gamma \vdash \mathsf{inr}_A\ v :^{q} A + B}\ \textsc{inr}$$

Expression typing rules. $\boxed{\gamma\Gamma \vdash e : \underline{B}}$

$$\frac{\gamma\Gamma \vdash v :^{q} A}{\gamma\Gamma \vdash \mathsf{return}_q\ v : F^{q} A}\ \textsc{return} \qquad \frac{\gamma\Gamma \vdash v :^{1} U\,\underline{B}}{\gamma\Gamma \vdash \mathsf{force}\ v : \underline{B}}\ \textsc{force} \qquad \frac{\gamma_1 \geq \gamma_2 \qquad \gamma_1\Gamma \vdash e : \underline{B}}{\gamma_2\Gamma \vdash e : \underline{B}}\ \textsc{esub}$$

$$\frac{\gamma\Gamma, x :^{q} A \vdash e : \underline{B}}{\gamma\Gamma \vdash \lambda x :^{q} A.\, e :\ {}^{q}A \to \underline{B}}\ \textsc{lam} \qquad \frac{\gamma_1\Gamma \vdash e :\ {}^{q}A \to \underline{B} \qquad \gamma_2\Gamma \vdash v :^{q} A}{(\gamma_1 + \gamma_2)\Gamma \vdash e\, v : \underline{B}}\ \textsc{app}$$

$$\frac{\gamma_1\Gamma \vdash e_1 : F^{q} A \qquad \gamma_2\Gamma, x :^{q} A \vdash e_2 : \underline{B}}{(\gamma_1 + \gamma_2)\Gamma \vdash \mathsf{let}\, x \leftarrow e_1\, \mathsf{in}\, e_2 : \underline{B}}\ \textsc{bind} \qquad \frac{\gamma_1\Gamma \vdash v :^{q} \Box^{\mu} A \qquad \gamma_2\Gamma, x :^{q*\mu} A \vdash e : \underline{B}}{(\gamma_1 + \gamma_2)\Gamma \vdash \mathsf{let}_q\, \mathsf{box}_{\mu}\, x = v\, \mathsf{in}\, e : \underline{B}}\ \textsc{unbox}$$

$$\frac{\gamma_1\Gamma \vdash v :^{q} \mathbb{1} \qquad \gamma_2\Gamma \vdash e : \underline{B}}{(\gamma_1 + \gamma_2)\Gamma \vdash v ;_q e : \underline{B}}\ \textsc{seq} \qquad \frac{\gamma_1\Gamma \vdash v :^{q} A_1 \times A_2 \qquad \gamma_2\Gamma, x :^{q} A_1, y :^{q} A_2 \vdash e : \underline{B}}{(\gamma_1 + \gamma_2)\Gamma \vdash \mathsf{let}_q (x, y) = v\, \mathsf{in}\, e : \underline{B}}\ \textsc{split}$$

$$\frac{\gamma_1\Gamma \vdash v :^{q} A_1 + A_2 \qquad \gamma_2\Gamma, x :^{q} A_1 \vdash e_1 : \underline{B} \qquad \gamma_3\Gamma, y :^{q} A_2 \vdash e_2 : \underline{B} \qquad 1 \geq q}{(\gamma_1 + (\gamma_2 \sqcap \gamma_3))\Gamma \vdash \mathsf{case}_q\ v\ \{x.\, e_1;\ y.\, e_2\} : \underline{B}}\ \textsc{case}$$

We define "$\mathsf{let}_q\, x = v\, \mathsf{in}\, e$" as "$\mathsf{let}\, x \leftarrow \mathsf{return}_q\ v\, \mathsf{in}\, e$", with the derivable typing rule:

$$\frac{\gamma_1\Gamma \vdash v :^{q} A \qquad \gamma_2\Gamma, x :^{q} A \vdash e : \underline{B}}{(\gamma_1 + \gamma_2)\Gamma \vdash \mathsf{let}_q\, x = v\, \mathsf{in}\, e : \underline{B}}\ \textsc{let}$$

3.1 Design Choices

Our calculus is derived from the usual presentation of graded modal types, with several adjustments.

Output Modes. Following McBride [2016], we use a value judgement $\gamma\Gamma \vdash v :^q A$, where the γ context describes how often each variable in Γ is used in v, assuming that we are producing q copies of v. Just like McBride [2016], we can define the [var] rule without the multiplicative unit: instead, we require the variable as often as the number of copies requested.

The output mode q may be viewed as a delayed multiplication of the context γ, and in other graded modal calculi, our judgement might equivalently be written $(q\gamma)\Gamma \vdash v : A$. We make the q annotation explicit in order to be able to implement modalities via right-multiplication. In the [box] rule, the modality μ multiplies the output mode q from the right, corresponding to $((q * \mu)\gamma)\Gamma \vdash v : A$, instead of the usual $(q(\mu \cdot \gamma))\Gamma \vdash v : A$. The latter term also associates the multiplication the wrong way, which is a problem in the [unbox] rule, which multiplies q by μ to yield a $(q * \mu)$ annotation: without output modes, beta-reduction would require multiplication to be associative. Wood and Atkey [2020] avoid this problem by restricting their box-elimination rule to only extract the modality without permitting multiplication by a grade.

Value Trick. Unlike McBride [2016], we have no output mode on expressions. In McBride's (call-by-name) calculus, one can create ρ copies of a function's return value by running the function ρ times. This makes it necessary to "split" the resources of the function, which causes the substitution lemma to fail in the APP rule [Atkey 2018]. We avoid this pitfall thanks to our call-by-(push-)value semantics, where modes track properties of values, instead of properties of computations. We follow the recipe of Lorenzen et al. [2026], who advocate multiplying the usage context only when it emerges from a rule that checks a value. This happens naturally in a call-by-push-value syntax where output modes are restricted to values. Our design is remarkably similar to that of Torczon et al. [2024], with the exception of their [coeff-letin] rule, which multiplies the γ_1 context by a grade, even though it is used to check an expression. In our calculus, the [bind] rule takes an expression and does not multiply the γ_1 context, while the [unbox] rule restricts its scrutinee to be a value.

Multiplicative Unit. In the [force] rule, we require that the thunk is available at mode 1. This matches other graded modal calculi, where the grade of the function is usually not multiplied, and so would be introduced as 1 by the [var] rule. In the [case] rule, we introduce a constraint that $1 \geq q$, following Choudhury et al. [2021], although this is not necessary for soundness, and it might be possible to parameterise the calculus by another mode for this purpose.

Types Do Not Depend on Modes. Thanks to the type annotations in the syntax, we can compute from each term its type. In particular, its type is not affected by usage contexts or output modes:

Theorem 3.1 (Unique Typing). *If* $\gamma_1\Gamma \vdash v :^q A_1$ *and* $\gamma_2\Gamma \vdash v :^r A_2$, *then* $A_1 = A_2$.
If $\gamma_1\Gamma \vdash e : \underline{B_1}$ *and* $\gamma_2\Gamma \vdash e : \underline{B_2}$, *then* $\underline{B_1} = \underline{B_2}$.

We write $\Gamma \vdash v : A$ whenever $\gamma\Gamma \vdash v :^q A$, and $\Gamma \vdash e : \underline{B}$ whenever $\gamma\Gamma \vdash e : \underline{B}$, for some γ and q.

4 Semantics

In this section, we restate the usual operational semantics for call-by-push-value [Levy 1999] to prove the soundness of our typing judgement. In particular, we show that reduction preserves the usage contexts of a term, so that we never need more resources than we had before the reduction. All results in this section rely only on the structural laws and the beta-reduction laws of skew-semirings.

Reduction Rules. $\boxed{e \rightsquigarrow e'}$

$$\begin{aligned}
\text{force}\,(\text{thunk}\ e) &\rightsquigarrow e \\
(\lambda x :^q A.\, e)\, v &\rightsquigarrow e[v/x] \\
\text{let}\, x \leftarrow \text{return}_q\ v \,\text{in}\, e &\rightsquigarrow e[v/x] \\
\text{let}_q\, \text{box}_\mu\, x = \text{box}_\mu\ v \,\text{in}\, e &\rightsquigarrow e[v/x] \\
\text{let}_q (x, y) = (v_1, v_2) \,\text{in}\, e &\rightsquigarrow e[v_1/x, v_2/y] \\
\text{case}_q\ (\text{inl}_{A_2}\ v)\ \{x.\, e_1;\ y.\, e_2\} &\rightsquigarrow e_1[v/x] \\
\text{case}_q\ (\text{inr}_{A_1}\ v)\ \{x.\, e_1;\ y.\, e_2\} &\rightsquigarrow e_2[v/y] \\
();_q\ e &\rightsquigarrow e
\end{aligned}$$

Terminal computations.

$$t ::= \text{return}_q\ v \mid \lambda x :^q A.\, e$$

Evaluation contexts.

$$E ::= \cdot \mid E\, v \mid \text{let}\, x \leftarrow E \,\text{in}\, e$$

$$\frac{e \rightsquigarrow e'}{E[e] \rightsquigarrow E[e']}\ \text{CTX}$$

Every closed computation is either terminal or decomposes uniquely as $E[e]$ with e a redex. The semantics is mode-erasing as no rule inspects the mode annotations on syntax such as let_q, case_q and return_q. The progress property thus depends only on the correctness of types, rather than modes, and we do not prove it again. Instead, our focus is on preservation of modes:

Theorem 4.1 (Preservation). *If $\gamma\Gamma \vdash e : \underline{B}$ and $e \rightsquigarrow e'$, then $\gamma\Gamma \vdash e' : \underline{B}$.*

To prove this claim, we rely on the usual substitution lemma for our calculus:

Lemma 4.2 (Substitution).

$$\frac{\gamma_1\Gamma \vdash v :^q A \qquad \gamma_2\Gamma, x :^q A, \gamma_2'\Gamma' \vdash v' :^{q'} B}{(\gamma_1 + \gamma_2)\Gamma, \gamma_2'\Gamma' \vdash v'[v/x] :^{q'} B} \qquad \frac{\gamma_1\Gamma \vdash v :^q A \qquad \gamma_2\Gamma, x :^q A, \gamma_2'\Gamma' \vdash e : \underline{B}}{(\gamma_1 + \gamma_2)\Gamma, \gamma_2'\Gamma' \vdash e[v/x] : \underline{B}}$$

To perform substitutions, we must split the value judgement in the premise according to how x is used in the term. This requires us to prove various helper lemmas, which are stated below. These helper lemmas rely directly on the beta-reduction laws of skew-semirings:

Lemma 4.3 (Splitting of Output Modes). *If $\gamma\Gamma \vdash v :^{q+r} A$, then there exist γ_1 and γ_2 with $\gamma_1 + \gamma_2 \geq \gamma$ such that $\gamma_1\Gamma \vdash v :^q A$ and $\gamma_2\Gamma \vdash v :^r A$.*

Lemma 4.4 (Splitting of Multiplied Output Modes). *If $\gamma\Gamma \vdash v :^{q \cdot q'} A$, then there exists a γ' such that $q \cdot \gamma' \geq \gamma$ and $\gamma'\Gamma \vdash v :^{q'} A$.*

Lemma 4.5 (Usage of Values at Mode Zero). *If $\gamma\Gamma \vdash v :^0 A$, then $\underline{0} \geq \gamma$.*

5 Equational Theory

In this section, we define an equational theory for our graded CBPV calculus. Typically, the judgement $\Gamma \vdash e \equiv e' : \underline{B}$, states that e and e' are equivalent computations of type $\underline{B}$, in the sense that they either both diverge or both terminate with the same value in any context Γ. However, our semantics is entirely standard, and so this interpretation of equivalence would not be surprising. Instead, we are interested in showing that equivalence preserves typing.

Definition 5.1 (Equivalence). We write $\Gamma \vdash e \equiv e' : \underline{B}$ if for all γ we have that $\gamma\Gamma \vdash e : \underline{B}$ implies $\gamma\Gamma \vdash e' : \underline{B}$ and, vice versa, $\gamma\Gamma \vdash e' : \underline{B}$ implies $\gamma\Gamma \vdash e : \underline{B}$.

Note that by Theorem 3.1, the type $\underline{B}$ is independent of the usage context γ.

Corollary 5.2 (Equivalence is an Equivalence Relation). *The relation $\equiv$ is reflexive, symmetric and transitive.*

$$\frac{}{\Gamma \vdash e \equiv e : \underline{B}}\ \text{REFL} \qquad \frac{\Gamma \vdash e \equiv e' : \underline{B}}{\Gamma \vdash e' \equiv e : \underline{B}}\ \text{SYMM} \qquad \frac{\Gamma \vdash e \equiv e' : \underline{B} \qquad \Gamma \vdash e' \equiv e'' : \underline{B}}{\Gamma \vdash e \equiv e'' : \underline{B}}\ \text{TRANS}$$

Theorem 5.3 (Congruence). *The equivalence relation $\equiv$ is a congruence with respect to the typing rules of the calculus.*

5.1 Eta Laws for Values

We decompose the equational theory of our language into several laws. Diverging slightly from CBPV, we aim to present the eta-laws for values without including the commuting conversions (which we tackle below). Levy [1999] presents the eta-laws for values v bound in an arbitrary expression $e[v/z]$. This turns out to require most of the semiring laws, and we return to this point in Section 5.6. Instead, we use a restricted form of the eta-laws, where e may only return the value:

Theorem 5.4 (Eta Laws for Values). *Assuming only the eta-expansion and eta-reduction laws:*

$$\begin{aligned}
&\Gamma \vdash \mathrm{return}_q\ v \equiv \mathrm{return}_q\ (\mathrm{thunk}\ (\mathrm{force}\ v)) && : F^q\ (U\ \underline{C})\\
&\Gamma \vdash \mathrm{return}_q\ v \equiv \mathrm{let}_q (x, y) = v\ \mathrm{in}\ \mathrm{return}_q\ (x, y) && : F^q\ (A_1 \times A_2)\\
&\Gamma \vdash \mathrm{return}_q\ v \equiv v;_q\ \mathrm{return}_q\ () && : F^q\ \mathbb{1}\\
&\Gamma \vdash \mathrm{return}_q\ v \equiv \mathrm{let}_q\ \mathrm{box}_\mu\ x = v\ \mathrm{in}\ \mathrm{return}_q\ (\mathrm{box}_\mu\ x) && : F^q\ (\Box^\mu A)\\
&\Gamma \vdash \mathrm{return}_q\ v \equiv \mathrm{case}_q\ v\ \{x.\ \mathrm{return}_q\ (\mathrm{inl}_{A_2}\ x);\\
&\qquad\qquad y.\ \mathrm{return}_q\ (\mathrm{inr}_{A_1}\ y)\} && : F^q\ (A_1 + A_2) \qquad \text{for } 1 \geq q
\end{aligned}$$

The first equation requires the eta-expansion and eta-reduction laws, while all other equations hold structurally.

5.2 The Missing Eta Laws for Computations

For computations, the eta-laws are:

$$\Gamma \vdash e \equiv \lambda x :^q A.\, e\, x : {}^qA \to \underline{B} \qquad\qquad \Gamma \vdash e \equiv \mathrm{let}\, x \leftarrow e\, \mathrm{in}\, \mathrm{return}_q\ x : F^q A$$

However, these laws do not hold in the calculus. While it is trivial to rewrite the left to the right expression, it is impossible to rewrite from the right to the left: eta-expansion always works, but eta-reduction can fail.

This happens because our types may contain mode annotations (e.g. in F^qA and ${}^qA \to B$), and we do not have a subtyping relation to perform submoding in types. An eta-expanded term may thus use the [esub] rule to change the mode recorded in the type, and then eta-reduction will fail to type-check. It should be possible to add a subtyping relation to our calculus, without assuming additional laws on skew-semirings.

5.3 Commuting Conversions

Following Maurer et al. [2017], we state the commuting conversion laws as permuting evaluation contexts with tail contexts. A tail context T is a term that ends in a hole $\cdot$ in tail position. The grammar for tail contexts is given on the right. Together with evaluation contexts, it covers all parts of our syntax, except for thunks and lambdas.

$$\begin{aligned}
T ::= {} & \cdot \mid \mathrm{let}\, x \leftarrow e\, \mathrm{in}\, T\\
& \mid v;_q\ T\\
& \mid \mathrm{let}_q\, \mathrm{box}_\mu\, x = v\, \mathrm{in}\, T\\
& \mid \mathrm{let}_q (x, y) = v\, \mathrm{in}\, T\\
& \mid \mathrm{case}_q\ v\ \{x.\, T_1;\ y.\, T_2\}
\end{aligned}$$

Unlike an evaluation context, a tail context may contain several holes and is filled with a vector of expressions $\overline{e}$:

Theorem 5.5 (Commuting Conversions). *Assuming only the structural laws:*

$$\frac{\mathrm{bv}(T) \cap \mathrm{fv}(E) = \emptyset}{\Gamma \vdash E[T[\overline{e}]] \equiv T[\overline{E[e]}] : \underline{B}}\ \textsc{commute}$$

5.4 Beta Laws

To state the beta-laws, we must track at which mode a value is used in a term. To do so, we internalise a substitution operator in our syntax:

$$\frac{\gamma_1\Gamma \vdash v :^q A \qquad \gamma_2\Gamma, x :^q A, \gamma_2'\Gamma' \vdash e : \underline{B}}{(\gamma_1 + \gamma_2)\Gamma, \gamma_2'\Gamma' \vdash e[v/_q x] : \underline{B}}\ \textsc{subst}$$

This judgement states that v could be substituted for x in e at mode q, but does not actually perform the substitution. With this, we can state the following beta-laws:

Theorem 5.6 (Beta Laws). *Assuming only the structural laws:*

$$\Gamma \vdash ();_q\ e \equiv e : \underline{B} \qquad \Gamma \vdash \mathrm{let}\, x \leftarrow \mathrm{return}_q\ v \,\mathrm{in}\, e \equiv e[v/_q x] : \underline{B}$$
$$\Gamma \vdash \mathrm{force}\,(\mathrm{thunk}\ e) \equiv e : \underline{B} \qquad \Gamma \vdash \mathrm{let}_q\, \mathrm{box}_\mu\, x = \mathrm{box}_\mu\ v \,\mathrm{in}\, e \equiv e[v/_{q*\mu} x] : \underline{B}$$
$$\Gamma \vdash (\lambda x :^q A.\, e)\, v \equiv e[v/_q x] : \underline{B} \qquad \Gamma \vdash \mathrm{let}_q(x, y) = (v_1, v_2) \,\mathrm{in}\, e \equiv e[v_1/_q x, v_2/_q y] : \underline{B}$$

and assuming $1 \geq q$ *and* $\gamma\Gamma, x :^q A_1 \vdash e_1 : \underline{B} \iff \gamma\Gamma, y :^q A_2 \vdash e_2 : \underline{B}$ *for all* γ*:*

$$\Gamma \vdash \mathrm{case}_q\,(\mathrm{inl}_{A_2}\ v)\,\{x : A_1.\, e_1;\ y : A_2.\, e_2\} \equiv e_1[v/_q x] : \underline{B}$$
$$\Gamma \vdash \mathrm{case}_q\,(\mathrm{inr}_{A_1}\ v)\,\{x : A_1.\, e_1;\ y : A_2.\, e_2\} \equiv e_2[v/_q y] : \underline{B}$$

Thanks to the internalised substitution, the beta-laws require neither the beta-reduction, nor the beta-expansion laws. The beta-reduction laws are necessary to reduce the internalised substitution $e[v/_q x]$ to the actual substitution $e[v/x]$, as shown in Section 4. Conversely, the beta-expansion laws are necessary to recover the internalised substitution from the actual substitution.

5.5 Substitution Inversion

To perform beta-expansion, we must recover the mode q at which a value is used in a term. This process is the inverse of a substitution:

Theorem 5.7 (Substitution Inversion). *Assuming the beta-expansion laws and* $q + r = q \sqcap r$*: If* $\Gamma \vdash v : A$ *and* $\delta\Gamma, \delta'\Gamma' \vdash e[v/x] : \underline{B}$*, then there is a* q *with* $\delta\Gamma, \delta'\Gamma' \vdash e[v/_q x] : \underline{B}$*.*

The requirement that $q + r = q \sqcap r$ is needed to be able to abstract values out of case expressions, and it can be dropped if x does not occur under a case-split in e. Similar to the Substitution Lemma 4.2, we rely on several structural lemmas. In this case, the lemmas *merge* the value judgements based on how x is used in the term using the beta-expansion laws.

Lemma 5.8 (Merging of Output Modes). *If* $\gamma_1\Gamma \vdash v :^q A$ *and* $\gamma_2\Gamma \vdash v :^r A$*, then* $(\gamma_1 + \gamma_2)\Gamma \vdash v :^{q+r} A$*.*

Lemma 5.9 (Multiplication of Output Modes). *If* $\gamma\Gamma \vdash v :^{q'} A$*, then* $(q \cdot \gamma)\Gamma \vdash v :^{q \cdot q'} A$*.*

Lemma 5.10 (Discarding of Values). *If* $\gamma\Gamma \vdash v :^q A$*, then* $\underline{0}\Gamma \vdash v :^0 A$*.*

5.6 Full Eta-Laws

We can obtain the usual eta-laws of call-by-push-value by combining our restricted eta-laws with the beta-laws and commuting conversions. For example, consider the full eta-law for products:

$$\begin{aligned}
&\Gamma \vdash e[v/z] \\
&\quad \equiv \mathrm{let}\, z \leftarrow \mathrm{return}_q\ v \,\mathrm{in}\, e && \text{(beta-expansion)} \\
&\quad \equiv \mathrm{let}\, z \leftarrow (\mathrm{let}_q(x, y) = v \,\mathrm{in}\, \mathrm{return}_q\,(x, y)) \,\mathrm{in}\, e && \text{(eta-law for products)} \\
&\quad \equiv \mathrm{let}_q(x, y) = v \,\mathrm{in}\, \mathrm{let}\, z \leftarrow \mathrm{return}_q\,(x, y) \,\mathrm{in}\, e && \text{(commuting conversion)} \\
&\quad \equiv \mathrm{let}_q(x, y) = v \,\mathrm{in}\, e[(x, y)/z] : \underline{B} && \text{(beta-reduction)}
\end{aligned}$$

This derivation invokes most of the laws of the semiring and thus does not hold in OxCaml in its full generality. However, the restricted eta-laws do hold in OxCaml, and our presentation shows that the failure of the full eta-laws is really due to the failure of beta-expansion.

6 Inference

In this section, we derive an inference algorithm for modes. Our approach is inspired by, but not identical to, the inference algorithm implemented in the OxCaml compiler. The mode solver in the OxCaml compiler makes strong assumptions on the operations of the skew-semiring, which even rule out our design for uniqueness and affinity augmented by the erased mode. Instead, we use a simpler inference algorithm that makes only one additional assumption on the skew-semiring, namely we require a modality inversion q/μ such that:

$$r \geq q * \mu \text{ iff } r/\mu \geq q \qquad\qquad q/\mu \sqcap r/\mu = (q \sqcap r)/\mu$$

First, we show how to infer all modes in the program assuming that the mode annotations on arrow types are given. Then, we show how to infer the mode annotations on contravariant positions of arrow types, assuming that the mode annotations on covariant positions are given. For example, consider the usual `map` function on lists, which has the type:

```
val map : ('a @ m1 -> 'b @ m2) @ m3 -> 'a list @ m4 -> 'b list @ m5
```

Here `m1` and `m5` are in covariant positions whereas `m2, m3, m4` are in contravariant positions. Our inference algorithm assumes that `m1` and `m5` are given (or defaults them to the legacy mode `aliased many`), and infers the other mode annotations. In particular, this allows us to infer unused arguments and local function arguments, which makes it possible to perform stack allocation and erasure.

6.1 Inference Calculus

Our inference calculus uses the same syntax as the mode system of Lorenzen et al. [2024]:

Syntax.

$$\begin{aligned} A, B &::= A \mathbin{@} q \to B \mathbin{@} r \mid \Box^{\mu} A \mid A \times B \mid A + B \mid \mathbb{1} \\ e &::= x \mid \lambda x.\, e \mid e_1\, e_2 \mid \mathrm{box}_{\mu}\ e \mid \mathrm{unbox}_{\mu}\ e \mid (e_1, e_2) \mid () \mid \mathrm{inl}\ e \mid \mathrm{inr}\ e \\ &\quad \mid \mathrm{let}\, x = e_1\, \mathrm{in}\, e_2 \mid e_1;\, e_2 \mid \mathrm{let}(x, y) = e_1\, \mathrm{in}\, e_2 \mid \mathrm{case}\ e\ \{x.e_1;\ y.e_2\} \\ \Gamma &::= \cdot \mid \Gamma, x : A \mathbin{@} q \end{aligned}$$

Modes are annotated on function arrows, where the return mode of the function corresponds to the mode of F^q in the core calculus. The syntax is standard for a call-by-value language.

The inference calculus is defined in Figure 3. We annotate each judgement as either using synthesis $\Rightarrow$ or checking $\Leftarrow$. These annotations allow us to infer contravariant mode annotations below. For the moment, you may wish to ignore them: if all mode annotations on arrow types are given, we write just $\Gamma \vdash e : A \mathbin{@} q$ to subsume both $\Rightarrow$ and $\Leftarrow$ judgements. The rules are presented in the same order as in Figure 2 of Lorenzen et al. [2024], although we diverge from that work by adding a [seq] rule, and removing the [sub] rule.

Instead of a providing a stand-alone [sub] rule, we integrate the mode inequalities with the other rules. In an algorithmic implementation, we check against the output mode, and synthesise the mode annotations in the context. This requires us to add mode inequalities to the [app] rule, whose output mode has to match the return mode of the arrow type, and to the [lam] rule, where the mode synthesised from the body of the function needs to match the input mode of the arrow type.

Typing rules.

$$\boxed{\Gamma, x : \overset{\Uparrow}{A} @ q \vdash e \overset{\Downarrow}{\Rightarrow} \overset{\Downarrow}{A} @ q \overset{\uparrow}{}} \qquad \boxed{\Gamma, x : \overset{\Uparrow}{A} @ q \vdash e \overset{\Uparrow}{\Leftarrow} \overset{\Uparrow}{A} @ q \overset{\uparrow}{}}$$

$$\frac{}{\underline{0}\Gamma, x : A @ q, \underline{0}\Gamma' \vdash x \Rightarrow A @ q}\ \textsc{var} \qquad \frac{\Gamma_1 \vdash e_1 \Rightarrow \mathbb{1} @ q \quad \Gamma_2 \vdash e_2 \Leftarrow B @ r}{\Gamma_1 + \Gamma_2 \vdash e_1 ;_q e_2 \Leftarrow B @ r}\ \textsc{seq}$$

$$\frac{q' \geq q \quad \Gamma, x : A @ q' \vdash e \Leftarrow B @ r}{s \cdot \Gamma \vdash \lambda x.\, e \Leftarrow (A @ q \to B @ r) @ s}\ \textsc{lam} \qquad \frac{\Gamma_1 \vdash e_1 \Rightarrow (A @ q \to B @ r) @ 1 \quad \Gamma_2 \vdash e_2 \Leftarrow A @ q \quad r' \geq r}{\Gamma_1 + \Gamma_2 \vdash e_1\, e_2 \Rightarrow B @ r'}\ \textsc{app}$$

$$\frac{}{\underline{0}\Gamma \vdash () \Leftarrow \mathbb{1} @ q}\ \textsc{unit} \qquad \frac{\Gamma \vdash e \Leftarrow A @ q}{\Gamma \vdash \mathrm{inl}\ e \Leftarrow A + B @ q}\ \textsc{inl} \qquad \frac{\Gamma \vdash e \Leftarrow B @ q}{\Gamma \vdash \mathrm{inr}\ e \Leftarrow A + B @ q}\ \textsc{inr}$$

$$\frac{\Gamma_1 \vdash e_1 \Leftarrow A @ q \quad \Gamma_2 \vdash e_2 \Leftarrow B @ q}{\Gamma_1 + \Gamma_2 \vdash (e_1, e_2) \Leftarrow A \times B @ q}\ \textsc{pair} \qquad \frac{\Gamma_1 \vdash e_1 \Rightarrow A \times B @ (q_x \sqcap q_y) \quad \Gamma_2, x : A @ q_x, y : B @ q_y \vdash e_2 \Leftarrow C @ r}{\Gamma_1 + \Gamma_2 \vdash \mathrm{let}(x, y) = e_1\ \mathrm{in}\ e_2 \Leftarrow C @ r}\ \textsc{split}$$

$$\frac{\Gamma_1 \vdash e_1 \Rightarrow A @ q \quad \Gamma_2, x : A @ q \vdash e_2 \Leftarrow B @ r}{\Gamma_1 + \Gamma_2 \vdash \mathrm{let}\, x = e_1\ \mathrm{in}\ e_2 \Leftarrow B @ r}\ \textsc{let} \qquad \frac{\Gamma_1 \vdash e \Rightarrow A + B @ (1 \sqcap q_x \sqcap q_y) \quad \Gamma_2, x : A @ q_x \vdash e_1 \Leftarrow C @ r \quad \Gamma_3, y : B @ q_y \vdash e_2 \Leftarrow C @ r}{\Gamma_1 + (\Gamma_2 \sqcap \Gamma_3) \vdash \mathrm{case}\ e\ \{x.e_1;\ y.e_2\} \Leftarrow C @ r}\ \textsc{case}$$

$$\frac{\Gamma \vdash e \Leftarrow A @ q * \mu}{\Gamma \vdash \mathrm{box}_\mu\ e \Leftarrow \Box^\mu A @ q}\ \Box\mathrm{I} \qquad \frac{\Gamma \vdash e \Rightarrow \Box^\mu A @ q/\mu}{\Gamma \vdash \mathrm{unbox}_\mu\ e \Rightarrow A @ q}\ \Box\mathrm{E}$$

Fig. 3. Inference calculus

Annotation Burden. The surface syntax requires almost no mode annotation, with the exception of the [seq] rule, where we still require an annotation for the mode of the eliminated unit. We can avoid this annotation for those skew-semirings where a greatest mode exists (eg. 0 for uniqueness). This problem is also discussed by Abel et al. [2023, Section 7.3].

6.2 Locks

The original OxCaml inference algorithm uses locks to track the modes of closures in the context. Locks have no benefit in our simplified inference algorithm, but for completeness we show a variation of our rules with locks. We can phrase the [var] and [lam] rules in terms of locks:

$$\frac{q \leq \mathrm{locks}(\Gamma', r)}{\underline{0}\Gamma, x : A @ q, \underline{0}\Gamma' \vdash x \Rightarrow A @ r}\ \textsc{var} \qquad \frac{q' \geq q \quad \Gamma, \blacksquare_s, x : A @ q' \vdash e \Leftarrow B @ r}{\Gamma \vdash \lambda x.\, e \Leftarrow (A @ q \to B @ r) @ s}\ \textsc{lam}$$

where $\mathrm{locks}(\Gamma, r)$ multiplies the locks in Γ with r:

$$\mathrm{locks}(\cdot, r) = r \qquad \mathrm{locks}((\Gamma, x : A @ q), r) = \mathrm{locks}(\Gamma, r) \qquad \mathrm{locks}((\Gamma, \blacksquare_s), r) = \mathrm{locks}(\Gamma, s \cdot r)$$

Conceptually, a lock corresponds to a division of the modes in the context; and we could also implement locks $\Gamma, 🔒_s$ as an explicit division $\Gamma \setminus s$ that divides each mode annotation q in Γ, using the laws:

$$q \setminus s \leq r \quad \text{iff} \quad q \leq s \cdot r \qquad\qquad (q + r) \setminus s \leq q \setminus s + r \setminus s$$

Zero as Greatest Mode. The division operator forces 0 to be the greatest mode. When we instantiate the adjunction with $s = 0$ and $r = q \setminus 0$, we obtain $q \leq 0 \cdot (q \setminus 0)$. The beta-reduction laws then imply $q \leq 0$, so the zero mode has to be greater than any other mode. This is a big restriction in practice, since it rules out models of linearity, where $1 \not\leq 0$ as linear variables cannot be submoded to be unused. Note, however, that this restriction is only necessary when the rules above are modified to use locks, and our inference algorithm in Figure 3 could model linearity.

Division on Finite Modes. If the set of modes is finite, we may define division as $\bigsqcap\{s : q \leq r \cdot s\}$, where the empty set is mapped to the greatest mode 0. All laws above follow from this definition.

OxCaml. In OxCaml, the division operator $q \setminus r$ can be expressed as a combination of meets, joins, and a monotone function $\cdot^\dagger$. This allows the solver to work purely on lattices, without considering multiplication-constraints at all. In that setting, it is possible to infer *all* mode annotations, even on arrow types. However, this trick does not work for all skew-semirings, and not even for our instantiation of uniqueness and affinity due to the additional `erased` mode.

6.3 Soundness and Completeness of Inference

For the purpose of the next two theorems, we write $\vdash_{\text{gr}}$ for the core calculus of Section 3 and $\vdash_{\text{inf}}$ for the inference calculus of this section. In the Appendix, we define a translation $[\![-]\!]$ from the contexts, types, and terms of the inference calculus to those of the graded calculus. Our translation of types is exactly the usual embedding of call-by-value into call-by-push-value [Levy 1999], while our translation of terms also needs to insert let-bindings for those expressions that appear in positions where a value is required. A similar translation and proof was given by Jang et al. [2024, Section 7] and sketched by Lorenzen et al. [2024, Section 5.5] and Lorenzen et al. [2026].

Theorem 6.1 (Inference is Sound). *If* $\Gamma \vdash_{inf} e : A \mathbin{@} q$, *then* $[\![\Gamma]\!] \vdash_{gr} [\![e]\!] : F^q\,[\![A]\!]$.

To show that inference is complete, we would like to give a translation in the opposite direction and prove that the translations form a Galois Connection [Sabry and Wadler 1997]. However, this does not seem to be easily possible, since CBPV includes types like $U\,(A \to B \to F\,C)$, that do not have a CBV interpretation. Instead, we state the completeness theorem on the image of our translation into CBPV. Moreover, due to the lack of a [sub] rule in the inference calculus, we cannot check the inference derivation against any translated context directly, but the inference algorithm synthesises a context Γ_0 that is as great as possible:

Theorem 6.2 (Inference is Complete). *If* $[\![\Gamma]\!] \vdash_{gr} [\![e]\!] : F^q\,[\![A]\!]$, *then there is a* $\Gamma_0 \geq \Gamma$ *with* $\Gamma_0 \vdash_{inf} e : A \mathbin{@} q$.

The theorems hold both for the inference calculus with locks and without. When locks are used, we must assume the left distributivity law, which holds for OxCaml, even though it is a beta-expansion law.

6.4 Inference of Contravariant Mode Annotations

To infer the mode annotations on contravariant positions of arrow types, we use an analogy to bidirectional type checking [Odersky et al. 2001; Pierce and Turner 2000]. Before we explain it in detail, we want to point out though, that we assume in this calculus that the type structure is

already inferred: Since types do not depend on modes (Theorem 3.1), we can infer the types first using standard Hindley-Milner type inference. Our calculus then works on skeletons, where the type structure is fully present, and only the mode annotations are unification variables. We colour a type in red if its mode annotations in contravariant position are fully resolved, and in blue if its mode annotations in covariant position are fully resolved. In that setting, we define:

$$\Gamma \vdash e \Leftarrow A \mathbin{@} q \qquad \text{check against the covariant mode annotations of } A$$
$$\Gamma \vdash e \Rightarrow A \mathbin{@} q \qquad \text{synthesise the contravariant mode annotations of } A$$

The intuition behind this definition is that in the standard presentation of bidirectional type checking, the direction of checking and synthesis flips exactly with the variance of the type. We use the convention that all types in the context are red, and so the [var] rule is synthesising.

The [lam] rule is checking, so we know the covariant mode annotations of $(A \mathbin{@} q \to B \mathbin{@} r)$. As a result, we know the *covariant* mode annotations of B and the *contravariant* mode annotations of A. This allows us to put the type A into the context and continue checking against B. Furthermore, we know r but not q, which ties in perfectly with the rest of the calculus, which requires us to pass r to the premise, but infers q.

In the [app] rule, we assume that e_1 synthesises the type $(A \mathbin{@} q \to B \mathbin{@} r)$, so we know the *covariant* mode annotations of A and the *contravariant* mode annotations of B. This allows us to check e_2 against A and continue synthesising B. Furthermore, we know q but not r, which again ties in with the rest of the calculus, which receives r as an input, and passes q to the premise.

For all other rules, we follow exactly the Pfenning recipe of Dunfield and Krishnaswami [2021].

6.5 Limitations

When checking a top-level function, our bidirectional approach needs to guess modes in two places: at covariant positions in the type of the function, and at redexes in the body of the function (as is usual for bidirectional type checking). In these places, we may use a mode annotation given by the programmer, or we default to the legacy mode (`aliased many`) to ensure backwards compatibility.

A main difference to the mode solver of OxCaml is that we check each function in isolation, while the OxCaml compiler infers the modes of all functions in a module simultaneously. As an example, if the user writes a function that returns a unique value but does not annotate it as such, both our approach and OxCaml will default the functions return mode to aliased. However, if the function is called and its return value is used uniquely in another function of the same module, OxCaml will correctly infer the return mode of the first function to be unique, while our approach throws a type error.

A related limitation is that some programs do not type check unless the user provides additional mode annotations. For example, consider the program:

```
val openFile : string @ many -> File.t @ once
...
let foo () = openFile "foo.txt"
```

This program would throw a mode error in our system, since we first guess the (unannotated) output mode of `foo` as `aliased many`, which then requires the result of `openFile` to be available many times. In this case, the user would have to explicitly write `let foo () @ once = ...` for our system to accept the program. This mode annotation is completely obvious and indeed inferred by OxCaml's mode solver. Our system is thus less convenient than the existing inference algorithm, and it would be interesting to explore if we can augment our rules with specialised constraint solvers for selected skew-semirings.

7 Comparison with OxCaml

We started our overview by asking the reader to take a leap of faith and consider a fictional version of OxCaml that implements exactly the mode theory of Lorenzen et al. [2024] and Georges et al. [2025]. The reason for this requirement is that mode crossing [Peters et al. 2026], an unrelated change to OxCaml's mode theory, has recently repaired (most of) the equational theory of the language. [2]

7.1 Mode Crossing

Peters et al. [2026] propose mode crossing as a way to increase the expressivity of OxCaml's mode system. They identify that some data types do not interact with the mode system in a semantically-meaningful way, and allow these types to cross mode boundaries. For this paper, we focus on their [cross] rule together with the kinding declaration for function types and modalities. We can specialise these to our setting as follows:

$$\frac{\gamma\Gamma \vdash v :^{q} U\,\underline{B} \qquad q' \cdot r = q \cdot r \text{ for all } r}{\gamma\Gamma \vdash v :^{q'} U\,\underline{B}}\ \textsc{cross-fun} \qquad \frac{\gamma\Gamma \vdash v :^{q} \Box^{\mu} A \qquad q' * \mu = q * \mu}{\gamma\Gamma \vdash v :^{q'} \Box^{\mu} A}\ \textsc{cross-box}$$

For our instantiation, the multiplication and aliasing modality do not consider the uniqueness of their first argument. This allows us to lift a function or aliased-box from an aliased mode to a unique mode. Similarly, the many modality does not consider the affinity of its first argument, so we can lift a many-box from mode once to mode many.

It turns out that for every law that holds as a strict inequality in the original mode system, there is way to apply mode crossing to recover the law as an equality. We believe it would be possible to prove that OxCaml with mode crossing enjoys all laws of the equational theory we propose. However, this does not hold in general, since it is possible to construct skew-semirings that do not enjoy the full equational theory even with the mode crossing rules above.

It should be pointed out that the [cross-fun] rule is trivial for semirings, where from $r = 1$ we can conclude $q = q'$. In contrast, the [cross-box] rule could also be useful when modes form a semiring, and could be useful in other graded modal type systems as well.

7.2 Recovering Beta Expansion with the Always-Unique Mode

In Appendix B.6, we propose a new skew-semiring instantiation inspired by mode crossing, that also restores most of the equational theory of OxCaml. The main idea is to add a mode for "always unique" elements, which can be used uniquely more than once. This could include integers, unboxed tuples and closures: exactly the values to which mode crossing applies. We change the addition and multiplication to return the `always-unique` mode instead of an error.

Our extended skew-semiring restores almost all of the semiring laws even without the mode crossing rules. Only two laws do not hold as equalities. Firstly, the law for "Abstracting out of Case Expressions" cannot hold for uniqueness modes (and also does not hold with mode crossing). Secondly, the eta-reduction law $q \cdot 1 \leq q$ is false when q is `unique` or `always-unique`.

We believe that the mode crossing rules are preferable to the extended skew-semiring, though. With our `always-unique` mode, we can indicate that a function can be used uniquely more than once, but we have to maintain this invariant throughout the program (for example, by using the `always-unique` modality). Once we lose this invariant, we cannot recover it. In contrast, mode crossing relies on the kind of a value, which allows it to turn aliased functions back to unique

[2] Not all laws are repaired in OxCaml at the time of this writing, since mode crossing has not yet been integrated with the uniqueness analysis. However, we believe it would be straightforward to do so.

functions at any point. Furthermore, the `always-unique` mode would apply to all variables, even if they are of a type that is never introduced as `always-unique` (like the `Floatarray.t` of the overview). This makes it possible to infer "impossible" mode signatures, when the user might expect an error instead. Since mode crossing checks the types directly, it will likely produce much better error messages.

7.3 The Erased Mode

The definition of skew-semirings has led us naturally to include the erased mode 0 in our mode system. This mode is not present in OxCaml, and our design may help with its implementation in future versions of OxCaml. It has several semantic subtleties, which we want to highlight:

The erased mode is only sound if erasure is guaranteed. The definition of addition allows us to use a variable at mode 0 in a program after it has been used uniquely. But this can lead to a soundness issue if the use at mode 0 is not erased at compile-time. For example, the unique use might free a resource, but if the variable remains in scope, it could be accidentally accessed by intrusive tools like the garbage collector or a debugger.

We must accept mode errors in erased terms. We define multiplication by 0 to absorb the error mode: $0 \cdot \bot = 0$. This is necessary to ensure that beta-reduction continues to hold. Consider the following program, which checks independently of this assumption:

```
let y = (fun _ -> use_unique x)
in use_erased (fun _ -> use_many y)
```

Here x is used at mode $(0 \cdot \text{many}) \cdot \text{unique} = 0$. After beta-reduction, x is used at mode $0 \cdot (\text{many} \cdot \text{unique}) = 0 \cdot \bot$. If we defined $0 \cdot \bot = \bot$, beta-reduction could introduce an error.

Closed terms may be used at the error mode. Conversely, we also define $\bot \cdot 0 = 0$. This choice is necessary for several properties of our calculus to hold. For example, our calculus currently accepts the following program, which checks independently of this assumption:

```
let y = (fun z -> z) in
use_unique y; use_unique y
```

Here y is used uniquely twice and is thus at mode $\bot$. But because y has no free variables, the program is accepted regardless in our calculus. [3] However, if we defined $\bot \cdot 0 = \bot$, then any unused variable in the scope would cause the program to be rejected in our calculus, which would cause weakening to fail.

8 Related Work

Skew-Semirings. Wood and Atkey [2020] show that the full semiring laws are not necessary to ensure the soundness of graded modal type systems. They propose to use a skew-semiring instead (inspired by skew-monoidal categories [Szlachányi 2012]), and show that this preserves the substitution lemma and soundness of the type system. They do not discuss the effect that this choice has on the equational theory of the type system, and dropped their insight from their later work [Wood 2024; Wood and Atkey 2022] due to a lack of examples for skew-semirings.

Value Trick. Lorenzen et al. [2024] argue that a crucial difference between their system and other graded modal type systems lies in the use of call-by-value semantics. Under a call-by-value semantics, the usual modal axiom K becomes unsound, and the box introduction rule must be restricted to values. Lorenzen et al. [2026] expand on this idea and show how to apply it to several

[3] The program would be rejected in the original OxCaml calculus, but could be accepted in the OxCaml compiler because of mode crossing.

calculi in the literature. Jang et al. [2024] show that in their system the usual IS4 box can be decomposed into a value-restricted box and a thunk. Curien et al. [2016] already remarked that it is possible to perform this decomposition, but do not design a type system around this idea. Mangel et al. [2026] and Caspar and Munch-Maccagnoni [2026] propose semantic models for the value-restricted box in certain settings.

Inference. There is little work on syntax-directed inference for graded modal type systems, and it is not uncommon for implementations to require explicit mode annotations in types [Brady 2021]. An exception to this is Abel et al. [2023, Appendix B], who provide a syntax-directed Usage Inference. They show how to infer mode annotations provided that all elimination forms are explicitly annotated with the mode of the scrutinee. This is a strong requirement, and in fact it would be easy to extend their scheme to infer the annotations on most elimination forms from the mode of the bound variables. This trick only fails for function applications where no bound variable exists. This annotation seems to have a similar purpose as our requirement that covariant mode annotations are given on arrow types, and our bidirectional approach makes it possible to infer these mode annotations from type signatures instead of having to annotate terms. Tejiščák [2020] proposes an inference algorithm for the two-element semiring $\{0, 1\}$ using an approach based on Horn-clauses to infer all modes, but also show that it is quite slow even on small programs.

(Inferring) Polymorphism. For simplicity, we elide polymorphism from this work. Polymorphism over types is easy to support in our system, and does not affect inference, since types do not depend on modes [Lorenzen et al. 2024, Section 3.8]. Polymorphism over modes is more difficult. It should be possible to support it in our graded core calculus, following the design of Abel and Bernardy [2020], but it seems much harder to infer polymorphic mode annotations. In particular, our mode inference system could generate arbitrary inequality constraints between expressions involving both mode variables and the skew-semiring operations. As all skew-semiring operations commute with meets, the set of solutions will always form a sub-semilattice; but it is unclear how to display the constraints to the user and how to ensure that the constraints remain satisfiable by non-error modes. It might be possible to solve the latter issue by invoking an SMT solver similar to the inference algorithm of Granule [Moon et al. 2021; Orchard et al. 2019].

9 Conclusion

Our work establishes the relationship between OxCaml-style mode systems and graded modal type systems. We envisage our effort benefitting both communities.

OxCaml can benefit from the well-defined meta-theory of graded modal type systems. We showed how the equational theory of graded modal type systems reveals which refactorings are sound in OxCaml's mode systems. We also explored alternative designs for OxCaml that allow us to recover more of the equational theory. Graded modal type systems can also be extended to dependent types, as explored by [Abel et al. 2023; Doré 2025a,b], and this work opens up a path towards implementing an OxCaml-style mode system in a dependently typed language such as Lean.

Conversely, graded modal type system can benefit from stronger inference algorithms. OxCaml's mode inference algorithm can infer all mode annotations in a program. Our work casts this inference algorithm in the language of general skew-semirings, providing a new perspective to inform the design of strong inference algorithms for other instantiations of graded modal type systems.

Data-Availability Statement

We provide a Lean mechanisation covering all the claims in this document, which we intend to submit for Artifact Evaluation. The Lean artifact states the calculi and proofs in about 8000 lines of code, and was generated entirely using AI. It is organised into folders corresponding to the sections

of this paper, and each folder has a `Main.lean` file that contains the main claims of that section. It is intended as a sanity-check for subtle calculations involving skew-semiring laws. We have taken special care to ensure that the preconditions of our lemmas match the Lean proofs.

Acknowledgments

We thank Wenhao Tang for suggesting several useful improvements to the inference algorithm.

References

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

Andreas Abel, Nils Anders Danielsson, and Oskar Eriksson. 2023. A graded modal dependent type theory with a universe and erasure, formalized. *Proceedings of the ACM on Programming Languages* 7, ICFP (2023), 920–954.

Danel Ahman. 2023. When programs have to watch paint dry. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, 1–23.

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

Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. 2023. Resource-Aware Soundness for Big-Step Semantics. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 267 (Oct. 2023), 29 pages. https://doi.org/10.1145/3622843

Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. 2026. Don't exhaust, don't waste. *Journal of Functional Programming* (2026). https://doi.org/10.46298/jfp.17800

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

Jean Caspar and Guillaume Munch-Maccagnoni. 2026. S4 modal sequent calculus as intermediate logic and intermediate language. arXiv:2601.08071 [cs.LO] https://arxiv.org/abs/2601.08071

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

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

Stephen Dolan and Leo White. 2022. Stack allocation for OCaml. Talk. In *OCaml workshop 2022*. ICFP.

Maximilian Doré. 2025a. Dependent Multiplicities in Dependent Linear Type Theory. *arXiv preprint arXiv:2507.08759* (2025).

Maximilian Doré. 2025b. Linear Types with Dynamic Multiplicities in Dependent Type Theory (Functional Pearl). *Proceedings of the ACM on Programming Languages* 9, ICFP (2025), 763–783.

Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional typing. *ACM Computing Surveys (CSUR)* 54, 5 (2021), 1–38.

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

Jane Street. 2026. OxCaml. Project website. https://oxcaml.org/

Junyoung Jang, Sophia Roshal, Frank Pfenning, and Brigitte Pientka. 2024. Adjoint natural deduction. In *9th International Conference on Formal Structures for Computation and Deduction (FSCD 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 15–1.

PaulBlain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Information and computation* 185, 2 (2003), 182–210.

Paul Blain Levy. 1999. Call-by-push-value: A subsuming paradigm. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 228–243.

Anton Lorenzen, Wenhao Tang, and Sam Lindley. 2026. A Value Trick for Modal Type Systems. In *Intuitionistic Modal Logic and Applications*.

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

Éléonore Mangel, Paul-André Melliès, and Guillaume Munch-Maccagnoni. 2026. Syntax and semantics of focalisation with relative monads and comonads. arXiv:2606.14652 [cs.LO] https://arxiv.org/abs/2606.14652

Luke Maurer, Paul Downen, Zena M Ariola, and Simon Peyton Jones. 2017. Compiling without continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 482–494.

Conor McBride. 2016. I got plenty o'nuttin'. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Springer, 207–233.

Benjamin Moon, Harley Eades III, and Dominic Orchard. 2021. Graded modal dependent type theory. In *European Symposium on Programming*. Springer International Publishing Cham, 462–490.

Martin Odersky, Christoph Zenger, and Matthias Zenger. 2001. Colored local type inference. *ACM SIGPLAN Notices* 36, 3 (2001), 41–53.

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

Benjamin Peters, Jules Jacobs, Diana Kalinichenko, Liam Stevenson, Aspen Smith, Derek Dreyer, and Richard A Eisenberg. 2026. Mode Crossing. *Proceedings of the ACM on Programming Languages* 10, ICFP (2026), 236–266.

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

Benjamin C Pierce and David N Turner. 2000. Local type inference. *Acm transactions on programming languages and systems (toplas)* 22, 1 (2000), 1–44.

Amr Sabry and Philip Wadler. 1997. A reflection on call-by-value. *ACM transactions on programming languages and systems (TOPLAS)* 19, 6 (1997), 916–941.

Kornel Szlachányi. 2012. Skew-monoidal categories and bialgebroids. *Advances in Mathematics* 231, 3-4 (2012), 1694–1730.

Wenhao Tang and Sam Lindley. 2026. Rows and Capabilities as Modal Effects. *Proceedings of the ACM on Programming Languages* 10, POPL (2026), 923–950.

Wenhao Tang, Leo White, Stephen Dolan, Daniel Hillerström, Sam Lindley, and Anton Lorenzen. 2025. Modal effect types. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 1130–1157.

Matúš Tejiščák. 2020. A dependently typed calculus with pattern matching and erasure inference. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–29.

Cassia Torczon, Emmanuel Suárez Acevedo, Shubh Agrawal, Joey Velez-Ginorio, and Stephanie Weirich. 2024. Effects and coeffects in call-by-push-value. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 1108–1134.

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

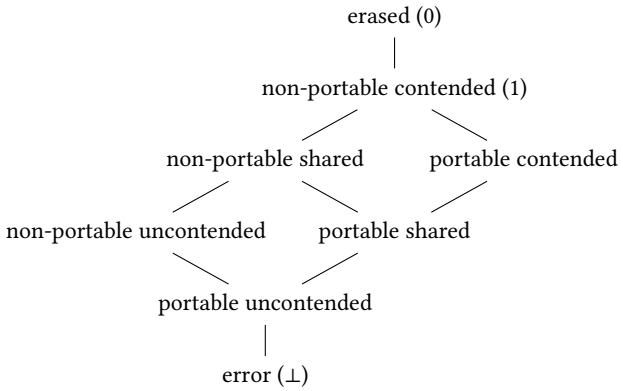
James Wood. 2024. A framework for semiring-annotated type systems. (2024).

James Wood and Robert Atkey. 2020. A linear algebra approach to linear metatheory. *arXiv preprint arXiv:2005.02247* (2020).

James Wood and Robert Atkey. 2022. A Framework for Substructural Type Systems. In *ESOP (Lecture Notes in Computer Science, Vol. 13240)*. Springer, 376–402. https://doi.org/10.1007/978-3-030-99336-8_14

A Portability and Contention

Georges et al. [2025] introduce the portability and contention modes to control whether the mutable fields of a value may be accessed from another domain. [4] A closure is *portable* if it may be sent to another domain, and *non-portable* otherwise. A value is *contended* if another domains may write its mutable fields (thus preventing the current domain from reading them without risking data-races), *shared* if no domain may write, but multiple domains may read its mutable fields, and *uncontended* if it is only accessible by the current domain.



As before, we include the erased mode (0) and the error mode ($\bot$) in the lattice.

Addition. Adding two uses is just their meet: $q + r = q \sqcap r$.

Multiplication. For the portability and contention modes, the multiplication operation is defined as follows:

$$q \cdot r = \begin{cases} 0 & \text{if } q = 0 \text{ or } r = 0 \\ r & \text{if } q \text{ is non-portable} \\ \text{portable contended} & \text{if } q \text{ is portable and } r \text{ is contended} \\ \bot & \text{otherwise} \end{cases}$$

This is the key duality condition of the parallelism axis: a portable closure may be sent to another domain, and so it may only capture its variables at the contended mode.

Modalities.

$$q * \text{portable} = \begin{cases} 0 & \text{if } q = 0 \\ \text{portable contended} & \text{if } q \text{ is contended} \\ \text{portable shared} & \text{if } q \text{ is shared} \\ \text{portable uncontended} & \text{if } q \text{ is uncontended} \\ \bot & \text{if } q = \bot \end{cases}$$

$$q * \text{contended} = \begin{cases} 0 & \text{if } q = 0 \\ \text{non-portable contended} & \text{if } q \text{ is non-portable} \\ \text{portable contended} & \text{if } q \text{ is portable} \\ \bot & \text{if } q = \bot \end{cases}$$

[4] If you have not heard of domains before, you may think of them as threads.

$$q * \text{shared} = \begin{cases} 0 & \text{if } q = 0 \\ \text{non-portable contended} & \text{if } q \text{ is non-portable contended} \\ \text{portable contended} & \text{if } q \text{ is portable contended} \\ \text{non-portable shared} & \text{if } q \text{ is non-portable shared/uncontended} \\ \text{portable shared} & \text{if } q \text{ is portable shared/uncontended} \\ \bot & \text{if } q = \bot \end{cases}$$

The portable modality plays the role of the many modality, and the contended modality that of the aliased modality.

A.1 The Failure of Beta Expansion

As for uniqueness and affinity, we can translate each law of this algebra into two expressions in the source language that are related by beta-reduction. All laws involving addition hold as equalities, as $+ = \sqcap$ and multiplication and modalities distribute over meets.

Associativity of Scaling. Again, this law only holds as an inequality. Consider the program:

```
let scale_assoc_success (x @ contended) @ portable =
  (fun _ -> (fun _ -> x) :@ shared)
```

In this program, we create a portable closure that may be sent to another domain. Each time it runs, it creates a fresh shared closure containing a contended x. We might like to abstract this closure out and write:

```
let scale_assoc_failure (x @ aliased) @ portable =
  let y @ shared = (fun _ -> x) in
  (fun _ -> y :@ shared)
```

but this program is not well-typed: because the resulting closure is portable, it may not capture the shared variable y.

Beta Expansion for Contended Modalities. For the portability and shared modalities, the beta-expansion laws hold as equalities, but they fail for the contended modality. Given

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

we can store a contended variable within a shared block in a portable closure

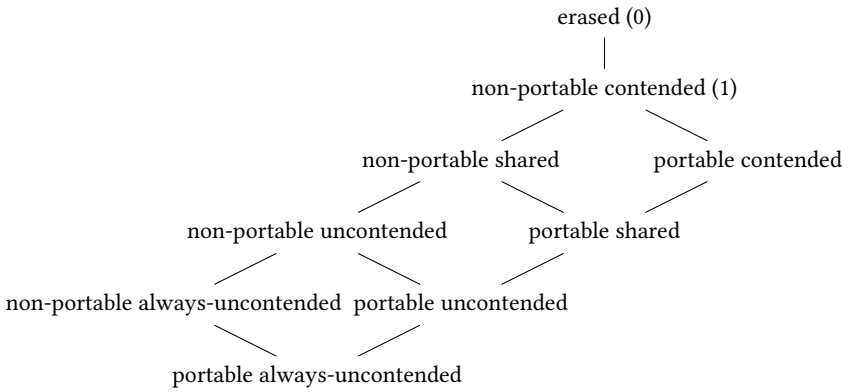
```
let scale_box_assoc_success (x @ contended) @ portable =
  (fun _ -> { contended = x } :@ shared)
```

but we can not extract the block into a let-binding, since the portable closure can not capture a shared variable:

```
let scale_box_assoc_failure (x @ contended) @ portable =
  let y @ shared = { contended = x } in
  (fun _ -> y :@ shared)
```

A.2 Recovering Beta Expansion with the Always-Uncontended Mode

As in Section 7.2, we can extend the algebra to make all laws but eta-reduction hold as equalities. The main idea is to add a mode for always-uncontended values, which can be written to from several domains at the same time. There is no more error mode $\bot$: whenever a value is captured by a portable closure at more than the contended mode, it has to be always-uncontended.



As before, addition is defined as the meet. For multiplication, we return always-uncontended instead of an error:

$$q \cdot r = \begin{cases} 0 & \text{if } q = 0 \text{ or } r = 0 \\ r & \text{if } q \text{ is non-portable} \\ \text{portable contended} & \text{if } q \text{ is portable and } r \text{ is contended} \\ \text{portable always-uncontended} & \text{otherwise} \end{cases}$$

The associativity of scaling now holds as an equality, and so the beta-expansion above becomes sound. As for the uniqueness and affinity modes, eta-reduction still fails: $q \cdot 1$ is contended for all $q \neq 0$, so $q \geq q \cdot 1$ is false when q is shared, uncontended or always-uncontended.

Modalities. The contended modality is unchanged, and the portable modality sends the always-uncontended modes to portable always-uncontended. The shared modality is refined to send the always-uncontended mode to the always-uncontended mode:

$$q * \text{shared} = \begin{cases} 0 & \text{if } q = 0 \\ \text{non-portable contended} & \text{if } q \text{ is non-portable contended} \\ \text{portable contended} & \text{if } q \text{ is portable contended} \\ \text{non-portable shared} & \text{if } q \text{ is non-portable shared/uncontended} \\ \text{portable shared} & \text{if } q \text{ is portable shared/uncontended} \\ \text{non-portable always-uncontended} & \text{if } q \text{ is non-portable always-uncontended} \\ \text{portable always-uncontended} & \text{if } q \text{ is portable always-uncontended} \end{cases}$$

B Omitted Rules and Definitions

B.1 Congruence Rules

To state the full congruence rules, we need to extend our equivalence relation to values. Following the equivalence of expressions of Section 5, we write $\Gamma \vdash v \equiv v' :^q A$ if, for every γ, the judgement $\gamma\Gamma \vdash v :^q A$ implies $\gamma\Gamma \vdash v' :^q A$, and vice versa.

Congruence for Values.

$$\frac{\Gamma \vdash e \equiv e' : \underline{B}}{\Gamma \vdash \mathrm{thunk}\ e \equiv \mathrm{thunk}\ e' :^q U\,\underline{B}}\ \textsc{cong-thunk} \qquad \frac{\Gamma \vdash v \equiv v' :^{q*\mu} A}{\Gamma \vdash \mathrm{box}_\mu\ v \equiv \mathrm{box}_\mu\ v' :^q \Box^\mu A}\ \textsc{cong-box}$$

$$\frac{\Gamma \vdash v_1 \equiv v_1' :^q A \qquad \Gamma \vdash v_2 \equiv v_2' :^q B}{\Gamma \vdash (v_1, v_2) \equiv (v_1', v_2') :^q A \times B}\ \textsc{cong-pair} \qquad \frac{\Gamma \vdash v \equiv v' :^q A}{\Gamma \vdash \mathrm{inl}_B\ v \equiv \mathrm{inl}_B\ v' :^q A + B}\ \textsc{cong-inl}$$

$$\frac{\Gamma \vdash v \equiv v' :^q B}{\Gamma \vdash \mathrm{inr}_A\ v \equiv \mathrm{inr}_A\ v' :^q A + B}\ \textsc{cong-inr}$$

Congruence for Expressions.

$$\frac{\Gamma \vdash v \equiv v' :^q A}{\Gamma \vdash \mathrm{return}_q\ v \equiv \mathrm{return}_q\ v' : F^q A}\ \textsc{cong-return} \qquad \frac{\Gamma \vdash v \equiv v' :^1 U\,\underline{B}}{\Gamma \vdash \mathrm{force}\ v \equiv \mathrm{force}\ v' : \underline{B}}\ \textsc{cong-force}$$

$$\frac{\Gamma, x :^q A \vdash e \equiv e' : \underline{B}}{\Gamma \vdash \lambda x :^q A.\, e \equiv \lambda x :^q A.\, e' :\ ^qA \to \underline{B}}\ \textsc{cong-lam}$$

$$\frac{\Gamma \vdash e \equiv e' :\ ^qA \to \underline{B} \qquad \Gamma \vdash v \equiv v' :^q A}{\Gamma \vdash e\ (v :^q A) \equiv e'\ (v' :^q A) : \underline{B}}\ \textsc{cong-app}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : F^q A \qquad \Gamma, x :^q A \vdash e_2 \equiv e_2' : \underline{B}}{\Gamma \vdash \mathrm{let}\, x :^q A \leftarrow e_1\, \mathrm{in}\, e_2 \equiv \mathrm{let}\, x :^q A \leftarrow e_1'\, \mathrm{in}\, e_2' : \underline{B}}\ \textsc{cong-bind}$$

$$\frac{\Gamma \vdash v \equiv v' :^q \Box^\mu A \qquad \Gamma, x :^{q*\mu} A \vdash e \equiv e' : \underline{B}}{\Gamma \vdash \mathrm{let}_q\, \mathrm{box}_\mu(x : A) = v\, \mathrm{in}\, e \equiv \mathrm{let}_q\, \mathrm{box}_\mu(x : A) = v'\, \mathrm{in}\, e' : \underline{B}}\ \textsc{cong-}\Box\textsc{e}$$

$$\frac{\Gamma \vdash v \equiv v' :^q \mathbb{1} \qquad \Gamma \vdash e \equiv e' : \underline{B}}{\Gamma \vdash v;_q\ e \equiv v';_q\ e' : \underline{B}}\ \textsc{cong-drop}$$

$$\frac{\Gamma \vdash v \equiv v' :^q A_1 \times A_2 \qquad \Gamma, x :^q A_1, y :^q A_2 \vdash e \equiv e' : \underline{B}}{\Gamma \vdash \mathrm{let}_q(x : A_1, y : A_2) = v\, \mathrm{in}\, e \equiv \mathrm{let}_q(x : A_1, y : A_2) = v'\, \mathrm{in}\, e' : \underline{B}}\ \textsc{cong-split}$$

$$\frac{\Gamma \vdash v \equiv v' :^q A_1 + A_2 \qquad \Gamma, x :^q A_1 \vdash e_1 \equiv e_1' : \underline{B} \qquad \Gamma, y :^q A_2 \vdash e_2 \equiv e_2' : \underline{B}}{\Gamma \vdash \mathrm{case}_q\ v\ \{x : A_1.\, e_1;\ y : A_2.\, e_2\} \equiv \mathrm{case}_q\ v'\ \{x : A_1.\, e_1';\ y : A_2.\, e_2'\} : \underline{B}}\ \textsc{cong-case}$$

B.2 Internalised Substitution

We extend the substitution rule to simultaneously substitute two values for two variables:

$$\frac{\gamma_1\Gamma \vdash v_1 :^q A_1 \qquad \gamma_2\Gamma \vdash v_2 :^q A_2 \qquad \gamma_3\Gamma, x :^q A_1, y :^q A_2, \gamma_3'\Gamma' \vdash e : \underline{B}}{(\gamma_1+\gamma_2+\gamma_3)\Gamma, \gamma_3'\Gamma' \vdash e[v_1/_q x, v_2/_q y] : \underline{B}}\ \textsc{subst2}$$

B.3 Translation from Inference to Graded Judgements

We only include the translation from the inference calculus with locks, which is more interesting than the translation without locks.

Interpretation of Inference Types.

$$\begin{aligned}
[\![\mathbb{1}]\!] &= \mathbb{1} \\
[\![A + B]\!] &= [\![A]\!] + [\![B]\!] \\
[\![A \times B]\!] &= [\![A]\!] \times [\![B]\!] \\
[\![\Box^\mu A]\!] &= \Box^\mu [\![A]\!] \\
[\![A \mathbin{@} q \to B \mathbin{@} r]\!] &= U\,({}^q[\![A]\!] \to F^r [\![B]\!])
\end{aligned}$$

Interpretation of Inference Contexts.

$$\begin{aligned}
[\![\cdot]\!] &= \cdot \\
[\![\Gamma, x : A \mathbin{@} q]\!] &= [\![\Gamma]\!],\ x :^q [\![A]\!] \\
[\![\Gamma, \blacksquare_q]\!] &= [\![\Gamma]\!] \setminus q
\end{aligned}$$

Here $\setminus$ acts pointwise on the modes in the translated context.

Fine-Grained Call-by-Value Terms. We restrict our syntax to fine-grained call-by-value [Levy et al. 2003] as follows:

$$\begin{aligned}
v &::= x \mid \lambda x.\, c \mid () \mid (v_1, v_2) \mid \mathrm{inl}\ v \mid \mathrm{inr}\ v \mid \mathrm{box}_\mu\ v \\
c &::= v \mid v_1\, v_2 \mid \mathrm{unbox}_\mu\ v \mid \mathrm{let}\, x = c_1\, \mathrm{in}\, c_2 \mid v;\ c \mid \mathrm{let}(x, y) = v\, \mathrm{in}\, c \mid \mathrm{case}\ v\ \{x.c_1;\ y.c_2\}
\end{aligned}$$

From Inference Terms to Fine-Grained Terms. We write $[\![e]\!]^{\mathrm{fg}}$ for the translation of e into fine-grained form. The variables z, z_1 and z_2 are fresh.

$$\begin{aligned}
[\![x]\!]^{\mathrm{fg}} &= x & [\![()]\!]^{\mathrm{fg}} &= () \\
[\![\lambda x.\, e]\!]^{\mathrm{fg}} &= \lambda x.\, [\![e]\!]^{\mathrm{fg}} & [\![e_1\, e_2]\!]^{\mathrm{fg}} &= \mathrm{let}\, z_1 = [\![e_1]\!]^{\mathrm{fg}}\, \mathrm{in}\, \mathrm{let}\, z_2 = [\![e_2]\!]^{\mathrm{fg}}\, \mathrm{in}\, z_1\, z_2 \\
[\![\mathrm{box}_\mu\ e]\!]^{\mathrm{fg}} &= \mathrm{let}\, z = [\![e]\!]^{\mathrm{fg}}\, \mathrm{in}\, \mathrm{box}_\mu\ z & [\![\mathrm{unbox}_\mu\ e]\!]^{\mathrm{fg}} &= \mathrm{let}\, z = [\![e]\!]^{\mathrm{fg}}\, \mathrm{in}\, \mathrm{unbox}_\mu\ z \\
[\![\mathrm{inl}\ e]\!]^{\mathrm{fg}} &= \mathrm{let}\, z = [\![e]\!]^{\mathrm{fg}}\, \mathrm{in}\, \mathrm{inl}\ z & [\![\mathrm{inr}\ e]\!]^{\mathrm{fg}} &= \mathrm{let}\, z = [\![e]\!]^{\mathrm{fg}}\, \mathrm{in}\, \mathrm{inr}\ z \\
[\![e_1;\ e_2]\!]^{\mathrm{fg}} &= \mathrm{let}\, z = [\![e_1]\!]^{\mathrm{fg}}\, \mathrm{in}\, z;\ [\![e_2]\!]^{\mathrm{fg}} & [\![(e_1, e_2)]\!]^{\mathrm{fg}} &= \mathrm{let}\, z_1 = [\![e_1]\!]^{\mathrm{fg}}\, \mathrm{in}\, \mathrm{let}\, z_2 = [\![e_2]\!]^{\mathrm{fg}}\, \mathrm{in}(z_1, z_2)
\end{aligned}$$

$$\begin{aligned}
[\![\mathrm{let}\, x = e_1\, \mathrm{in}\, e_2]\!]^{\mathrm{fg}} &= \mathrm{let}\, x = [\![e_1]\!]^{\mathrm{fg}}\, \mathrm{in} [\![e_2]\!]^{\mathrm{fg}} \\
[\![\mathrm{let}(x, y) = e_1\, \mathrm{in}\, e_2]\!]^{\mathrm{fg}} &= \mathrm{let}\, z = [\![e_1]\!]^{\mathrm{fg}}\, \mathrm{in}\, \mathrm{let}(x, y) = z\, \mathrm{in} [\![e_2]\!]^{\mathrm{fg}} \\
[\![\mathrm{case}\ e\ \{x.e_1;\ y.e_2\}]\!]^{\mathrm{fg}} &= \mathrm{let}\, z = [\![e]\!]^{\mathrm{fg}}\, \mathrm{in}\, \mathrm{case}\ z\ \{x.[\![e_1]\!]^{\mathrm{fg}};\ y.[\![e_2]\!]^{\mathrm{fg}}\}
\end{aligned}$$

Annotated Inference Terms. For the purpose of the second translation, we annotate the terms of the inference calculus with their types and modes:

$$\begin{aligned} e ::= {} & x \mid \lambda x.\, e \mid (e_1 : A \mathbin{@} q \to B \mathbin{@} r)\ e_2 \mid \mathrm{box}_\mu\ e \mid \mathrm{unbox}_\mu\ e \mid (e_1, e_2) \mid () \mid \mathrm{inl}\ e \mid \mathrm{inr}\ e \\ & \mid \mathrm{let}\, x : A \mathbin{@} q = e_1 \,\mathrm{in}\, e_2 \mid e_1 ;_q\ e_2 \mid \mathrm{let}(x : A \mathbin{@} q_x,\ y : B \mathbin{@} q_y) = e_1 \,\mathrm{in}\, e_2 \\ & \mid \mathrm{case}\ e\, \{x : A \mathbin{@} q_x.\, e_1;\ y : B \mathbin{@} q_y.\, e_2\} \end{aligned}$$

From Fine-Grained Terms to Graded Terms. We write $[\![v : A]\!]$ for the translation of a value v at type A, and $[\![c : A \mathbin{@} q]\!]$ for the translation of a fine-grained computation c at type A and output mode q. The variable z is fresh.

$$\begin{aligned} [\![x : A]\!] &= x \\ [\![() : \mathbb{1}]\!] &= () \\ [\![\lambda x.\, c : A \mathbin{@} q \to B \mathbin{@} r]\!] &= \mathrm{thunk}\ (\lambda x :^q [\![A]\!].\, [\![c : B \mathbin{@} r]\!]) \\ [\![(v_1, v_2) : A \times B]\!] &= ([\![v_1 : A]\!], [\![v_2 : B]\!]) \\ [\![\mathrm{box}_\mu\ v : \Box^\mu A]\!] &= \mathrm{box}_\mu\, [\![v : A]\!] \\ [\![\mathrm{inl}\ v : A + B]\!] &= \mathrm{inl}_{[\![B]\!]}\, [\![v : A]\!] \\ [\![\mathrm{inr}\ v : A + B]\!] &= \mathrm{inr}_{[\![A]\!]}\, [\![v : B]\!] \end{aligned}$$

$$\begin{aligned} [\![v : A \mathbin{@} q]\!] &= \mathrm{return}_q\, [\![v : A]\!] \\ [\![(v_1 : A \mathbin{@} q \to B \mathbin{@} r)\ v_2 : B \mathbin{@} r']\!] &= \mathrm{let}\, z :^r [\![B]\!] \leftarrow \\ &\qquad (\mathrm{force}\, [\![v_1 : A \mathbin{@} q \to B \mathbin{@} r]\!]) \\ &\qquad\quad ([\![v_2 : A]\!] :^q [\![A]\!])\ \mathrm{in} \\ &\qquad (\mathrm{return}_{r'}\ z \\ [\![\mathrm{unbox}_\mu\ v : A \mathbin{@} q]\!] &= \mathrm{let}_{q/\mu}\, \mathrm{box}_\mu(z : [\![A]\!]) = [\![v : \Box^\mu A]\!]\, \mathrm{in}\, \mathrm{return}_q\ z \\ [\![\mathrm{let}\, x : A \mathbin{@} q = c_1 \,\mathrm{in}\, c_2 : B \mathbin{@} r]\!] &= \mathrm{let}\, x :^q [\![A]\!] \leftarrow [\![c_1 : A \mathbin{@} q]\!]\, \mathrm{in} [\![c_2 : B \mathbin{@} r]\!] \\ [\![v ;_q\ c : B \mathbin{@} r]\!] &= [\![v : \mathbb{1}]\!] ;_q\ [\![c : B \mathbin{@} r]\!] \\ [\![\mathrm{let}(x : A \mathbin{@} q_x,\ y : B \mathbin{@} q_y) = v \,\mathrm{in}\, c : C \mathbin{@} r]\!] &= \mathrm{let}_{q_x \sqcap q_y}(x : [\![A]\!], y : [\![B]\!]) = \\ &\qquad [\![v : A \times B]\!]\, \mathrm{in} [\![c : C \mathbin{@} r]\!] \\ [\![\mathrm{case}\ v\, \{x : A \mathbin{@} q_x.\, c_1;\ y : B \mathbin{@} q_y.\, c_2\} : C \mathbin{@} r]\!] &= \mathrm{case}_{1 \sqcap q_x \sqcap q_y}\, [\![v : A + B]\!] \\ &\qquad \{x : [\![A]\!].\, [\![c_1 : C \mathbin{@} r]\!];\ y : [\![B]\!].\, [\![c_2 : C \mathbin{@} r]\!]\} \end{aligned}$$

B.4 Invertible Modalities

The `aliased` and `many` modalities are invertible:

$$q/\mathrm{aliased} = \begin{cases} 0 & \text{if } q = 0 \\ q & \text{if } q \text{ is aliased} \\ \bot & \text{otherwise} \end{cases}$$

$$q/\mathrm{many} = \begin{cases} 0 & \text{if } q = 0 \\ \text{aliased once} & \text{if } q \text{ is aliased} \\ \text{unique once} & \text{if } q \text{ is unique} \\ \bot & \text{if } q \text{ is always-unique} \end{cases}$$

B.5 Bidirectional Typing

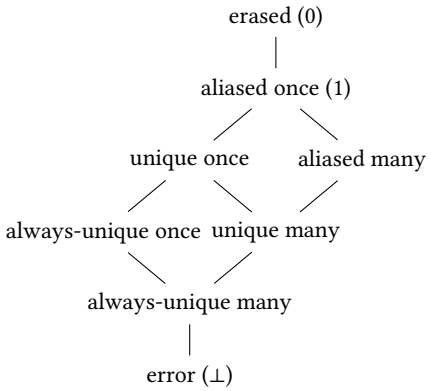
We also need to include the usual [sub] and [anno] rules for bidirectional typing:

$$\frac{\Gamma \vdash e \Rightarrow A \mathbin{@} q \qquad A = A}{\Gamma \vdash e \Leftarrow A \mathbin{@} q}\ \textsc{sub} \qquad\qquad \frac{\Gamma \vdash e \Leftarrow A \mathbin{@} q}{\Gamma \vdash (e : A) \Rightarrow A \mathbin{@} q}\ \textsc{anno}$$

In the [sub] rule, we compare the mode annotations for equality. The known mode annotations do not overlap, but the unknown mode annotations could be partially filled-in and have to be unified. In the [anno] rule, we need to provide all mode annotations in the type, so that we can check the premise against the covariant ones and synthesise the contravariant ones.

B.6 Definition of Extended Modes

To fix the equational theory of the uniqueness and affinity modes, we extend the lattice of modes with an `always-unique` mode.



In the addition and multiplication operations, we replace the error mode $\bot$ with the `always-unique many` mode:

$$q + r = \begin{cases} r & \text{if } q = 0 \\ q & \text{if } r = 0 \\ \text{aliased many} & \text{if } q \text{ and } r \text{ are aliased} \\ \text{always-unique many} & \text{if } q \text{ or } r \text{ is unique or always-unique} \end{cases}$$

$$q \cdot r = \begin{cases} 0 & \text{if } q = 0 \text{ or } r = 0 \\ r & \text{if } q \text{ is once} \\ \text{aliased many} & \text{if } q \text{ is many and } r \text{ is aliased} \\ \text{always-unique many} & \text{if } q \text{ is many and } r \text{ is unique or always-unique} \end{cases}$$

B.6.1 Invertible Modalities. We can define the following modalities (and their inverses) for the extended modes. Note that the inverse of the aliased modality can not recover uniqueness and has to return the error mode $\bot$ instead. This is the only place where the error mode is used in the extended algebra.

$$q * \text{aliased} = \begin{cases} 0 & \text{if } q = 0 \\ \text{aliased once} & \text{if } q \text{ is once} \\ \text{aliased many} & \text{if } q \text{ is many} \end{cases}$$

$$q/\text{aliased} = \begin{cases} 0 & \text{if } q = 0 \\ q & \text{if } q \text{ is aliased} \\ \bot & \text{otherwise} \end{cases}$$

$$q * \text{always-unique} = \begin{cases} 0 & \text{if } q = 0 \\ \text{always-unique once} & \text{if } q \text{ is once} \\ \text{always-unique many} & \text{if } q \text{ is many} \end{cases}$$

$$q/\text{always-unique} = \begin{cases} 0 & \text{if } q = 0 \\ \text{aliased once} & \text{if } q \text{ is once} \\ \text{aliased many} & \text{if } q \text{ is many} \end{cases}$$

$$q * \text{many} = \begin{cases} 0 & \text{if } q = 0 \\ \text{aliased many} & \text{if } q \text{ is aliased} \\ \text{unique many} & \text{if } q \text{ is unique} \\ \text{always-unique many} & \text{if } q \text{ is always-unique} \end{cases}$$

$$q/\text{many} = \begin{cases} 0 & \text{if } q = 0 \\ \text{aliased once} & \text{if } q \text{ is aliased} \\ \text{unique once} & \text{if } q \text{ is unique} \\ \text{always-unique once} & \text{if } q \text{ is always-unique} \end{cases}$$