Mode Crossing

BENJAMIN PETERS, MPI-SWS, Germany
JULES JACOBS, Jane Street, USA
DIANA KALINICHENKO, Jane Street, USA
LIAM STEVENSON, Jane Street, USA
ASPEN SMITH, Jane Street, USA
DEREK DREYER, MPI-SWS, Germany
RICHARD A. EISENBERG, Jane Street, USA

OxCaml extends the OCaml type system with support for safe low-level systems programming via *modes*. For example, OxCaml's modal *portability* and *contention* axes ensure that concurrent OxCaml programs have no data races. In practice, however, modal tracking can reject programs that are obviously safe, such as when the data shared across threads is immutable. To remedy this problem, we introduce *mode crossing*—the ability to automatically strengthen modes (*e.g.*, from **`nonportable`** to **`portable`**) for values of certain types. Mode crossing significantly reduces the annotation burden associated with modal types. To support mode crossing in the presence of abstract type specifications, we further introduce a new type system feature, *modal kinds*.

We present a type-theoretic account of modal kinds, interpret them as monotone functions on a lattice of modes, and extend this interpretation to recursive and abstract types. We verify soundness of the modal kind system in Rocq on top of Iris. We design an inference procedure that reduces kind checking and subsumption to constraints solved by a dedicated lattice solver. Our design is implemented in the OxCaml compiler and deployed in a large industrial codebase, demonstrating practical usability.

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

Additional Key Words and Phrases: Modal types, modal kinds, OxCaml, Iris, Rocq

ACM Reference Format:
Benjamin Peters, Jules Jacobs, Diana Kalinichenko, Liam Stevenson, Aspen Smith, Derek Dreyer, and Richard A. Eisenberg. 2026. Mode Crossing. *Proc. ACM Program. Lang.* 10, ICFP, Article 150 (August 2026), 54 pages.

1 Introduction

OxCaml is a backward-compatible extension of OCaml for safe concurrent and systems programming [12, 13, 18]. It aims to provide strong safety guarantees (notably data-race freedom and safe stack allocation) while preserving OCaml's ergonomics and ecosystem compatibility. As the name OxCaml ("Oxidized Caml") suggests, the language is inspired by Rust and its support for safe, low-level, performance-oriented programming, but OxCaml makes some very different design choices. First, whereas Rust supports manual memory management and ensures safety by prohibiting aliased mutable state by default, OxCaml (building on OCaml) is garbage-collected and safely allows arbitrarily aliased mutable state by default. Second, whereas Rust starts from an

Authors' Contact Information: Benjamin Peters, MPI-SWS, Saarland Informatics Campus, Germany, bpeters@mpi-sws.org; Jules Jacobs, Jane Street, New York, USA, julesjacobs@gmail.com; Diana Kalinichenko, Jane Street, New York, USA, dkalinichenko@janestreet.com; Liam Stevenson, Jane Street, New York, USA, lstevenson@janestreet.com; Aspen Smith, Jane Street, New York, USA, aspsmith@janestreet.com; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org; Richard A. Eisenberg, Jane Street, New York, USA, reisenberg@janestreet.com.

ACM 2475-1421/2026/8-ART150
https://doi.org/

ownership-based type system (with rigid control over aliasing) and makes it more flexible through the use of *lifetimes* and *borrowing*, OxCaml starts from a very flexible language and supports more careful control over aliasing through the use of *modal types*.

This paper addresses a practical usability problem in the previously presented OxCaml design. We describe a language extension—already deployed in the OxCaml compiler—that supports what we call *mode crossing* and *modal kinds*. To illustrate the problem, consider a simple OxCaml fork-join example: a recursive divide-and-conquer summation that splits an immutable input array in half, processes both halves in parallel, and maps each integer with `f`.

```
let rec sum_map (f : (int → int) @ portable) (xs : int iarray) =
  match Iarray.length xs with
  | 0 → 0
  | 1 → f xs.(0)
  | _ → let s_left, s_right =
          fork_join2
            (fun () → sum_map f (Iarray.first_half_slice xs))
            (fun () → sum_map f (Iarray.second_half_slice xs))
        in
          s_left + s_right
```

OxCaml ensures race freedom by requiring callbacks passed to `fork_join2` to be `@ portable`:[1]

```
val fork_join2 : (unit → 'a) @ portable → (unit → 'b) @ portable → 'a * 'b
```

Functions are portable if they can be executed in parallel without introducing data races: they neither access unsynchronized shared mutable state themselves nor call functions that do. (In this example, we assume that top-level functions, such as `fork_join2` and the functions in `Iarray`, are themselves portable.) Accordingly, `sum_map` requires `f : (int → int) @ portable` in order to allow `f` to be called in child threads. Under that caller-side precondition, the remaining data flow involves only immutable arrays and integers, so `sum_map` is race-free.

The problem. Georges et al. [12] presented a semantic foundation for OxCaml's modal type system. Under their typing rules, because `xs` in the above program flows to other threads via `fork_join2`, the program type-checks only if the `xs` argument of `sum_map` is also `@ portable`:

```
let rec sum_map (f : (int → int) @ portable) (xs : int iarray @ portable) = ...
                                                    redundant requirement in baseline OxCaml
```

One would similarly have to establish portability-annotated types for the auxiliary functions called by `sum_map`:

```
val first_half_slice, second_half_slice : int iarray @ portable → int iarray @ portable
```

For immutable `int iarray`, however, these additional annotations impose a redundant requirement because *every* value of type `int iarray` is portable! Unfortunately, the modal type system in Georges et al. [12] does not exploit that type-specific property, as it treats modes and types *independently*: its mode-checking rules must be uniformly sound for *all* types and are therefore pessimistic, requiring both `f` and `xs` to be portable.

In our experience with a large industrial OxCaml codebase at Jane Street (see §5), this is a major usability issue: redundant annotations pollute type signatures and burden callers. For example, in practice, although all `int iarray` values are `@ portable`, it is surprisingly non-trivial to convince

[1] OxCaml has multiple modal axes, including portability, contention, affinity, uniqueness, locality, etc. We use portability as the running example axis in this introduction and generalize later to include contention also.

the type-checker of this fact. Doing so requires burdensome annotations, nested modalities [12], and mode polymorphism across the library ecosystem used to produce those values. It also makes it much harder for code that uses modes to interoperate with code that does not.

Mode crossing and modal kinds. In this paper, we show how to solve this usability problem in OxCaml's modal type system using what we call *mode crossing*. Mode crossing lets the type checker strengthen the mode of a value based on its type, thereby eliminating redundant annotations such as `@ portable` on `int iarray` while still soundly tracking mode-sensitive types. For instance, `int iarray` values can always be treated as `portable`, but `int → int` values generally cannot; in this case, we say that the type `int iarray` "crosses" portability. We track the mode-crossing properties of types using *modal kinds*.[2] For example, we would say that `int iarray` has modal kind `value mod portable`, whereas `int → int` has only modal kind `value`.[3]

To reiterate, mode crossing and modal kinds *reduce the annotation burden of a modal type system* by moving annotations where they belong. Instead of annotating every function argument and result (including mode-polymorphism annotations on type-polymorphic functions), we place the burden on abstract type specifications. For concrete types, we infer the most accurate modal kind. This reduction is crucial in practice: without mode crossing, adopting OxCaml in a large industrial codebase like Jane Street's would have been infeasible.

Although the basic idea of mode crossing is simple, its implementation and metatheory in terms of modal kinds is surprisingly subtle. In particular, we identify the following design highlights:

With-bounds (§2.5) Modal kinds of parameterized types should be computed compositionally: the kind of `int iarray` should follow from the kinds of `int` and `iarray`, which requires a kind language for type constructors. We express constructor dependence via `with`-bounds (*e.g.*, `'a iarray : value mod portable with 'a`) and model type constructors semantically as functions between modal kinds.

Recursive types (§2.6) Recursive type constructors, especially those containing non-uniform recursion, cannot be handled by naive unfolding, as that approach can fail to terminate. We instead give recursive kinds a principled lattice-theoretic least fixpoint semantics to support sound and terminating checking.

Modalities (§2.7) Modalities create axis-specific dependence: a type constructor may be insensitive to an argument on one axis while still depending on it on another. We capture this by allowing modality-qualified with-bounds such as `with 'a @@ portable`, so kinding tracks how modality application transforms crossing behavior.

Abstract types (§2.8) Kinding must remain sound across module abstraction boundaries, where concrete representations are hidden. We therefore track dependence on abstract types and abstract constructors (*e.g.*, `with t` and `with 'a t`), and use this information both for type checking and for sub-kinding obligations that arise in signature inclusion and substitution.

Lattice solver (§2.9) The compiler must solve kind inclusion checks, including kinds that are fixpoints of recursive equations, under hypotheses induced by abstract types with with-bounds. We reduce this to a decision problem in lattice functions and solve it with a dedicated solver for practical kind inference and sub-kinding checks.

[2] An alternative approach would be to track such properties in the way that Rust tracks modal properties of types (*e.g.*, portability), using so-called *marker traits* like Send and Sync. We employ kinds, in part so that we can build on OxCaml's existing kind infrastructure for unboxed types (OxCaml does not support traits or type classes). Furthermore, Rust's marker traits do not address a number of the design challenges described in this section (see §6 for details).

[3] The actual kinds are more complex due to the presence of other modal axes.

Contributions. We make the following contributions:

Design (§2) We motivate *mode crossing* and introduce *modal kinds*, giving a detailed account of modal kinds for base types, modalities, recursive types, and abstract types, together with sub-kinding and inclusion principles.

Soundness (§3) We present SMoKi (a System of Modal Kinds), a core modal type system with an explicit kind calculus that formalizes kinding constraints and their interaction with typing. We prove memory safety and data-race freedom for SMoKi (Thm. 6, in §3.6) using a logical relation mechanized in Rocq with Iris.

Inference (§4) We describe an inference procedure that reduces kind checking and sub-kinding to lattice constraints solved by a dedicated solver, and incorporate it into the OxCaml compiler.

Implementation The ideas in this paper are all implemented in open-source and are used in production to compile Jane Street's codebase.

We conclude our paper with some discussion of how mode crossing has manifested in practice (§5) and with a review of related work (§6).

Non-goals and limitations. Our core type system SMoKi does not handle all features of OxCaml. In particular, it does not formalize all OxCaml modal axes, although it does handle all axes considered by Georges et al. [12]. It also does not directly model a full-blown module system, although it supports typechecking in the presence of abstract type specifications. Lastly, SMoKi does not handle GADTs, for which the construction of a semantic model in Iris is an independently challenging problem [25]. In contrast, our OxCaml implementation handles all modal axes of OxCaml, supports the full-blown OxCaml module system, and deals with GADTs conservatively (inferring a sound modal kind but not necessarily a principal one).

2 Key Ideas

This section introduces *modal kinds*: kinds that describe which modal axes a type can safely ignore (*mode crossing*). We first recap portability and contention, then show by example that plain mode checking is too conservative for immutable data. Next, we interpret kinds as lattice-valued functions, use that view for recursion and modalities, and end with the normal-form view used by the implementation.

2.1 Background: The Portability and Contention Modes

The central idea behind OxCaml's data-race freedom guarantee concerns control of mutable state. Unless we use synchronization primitives (which are outside the scope of this paper), we must never mutate state using a reference that has passed from one thread to another. We thus assign all values that pass from one thread to another the **contended** mode, and we prevent reading or writing mutable fields from **contended** values. This might seem sufficient, except that a closure could *capture* an **uncontended** reference, then be transferred to another thread and executed there, defeating this protection. We thus say that a function capturing **uncontended** references is **nonportable**; a function capturing only **contended** references is **portable**. Functions that are **portable** are allowed to be passed to other threads; **nonportable** functions may not be. (A **portable** function must additionally capture only other **portable** functions.) For example, the function `fun () → r := !r + 1` (for some captured `r : int ref @ uncontended`) is **nonportable**, because it needs **uncontended** access to `r` to mutate it. On the other hand, `fun r → r := !r + 1` is fully **portable**: it captures only **portable** operations, including addition and reference reads and writes. (Intuitively, the notion of portability is similar to that of Rust's `Send`, in that it determines whether values can move between threads or not.)

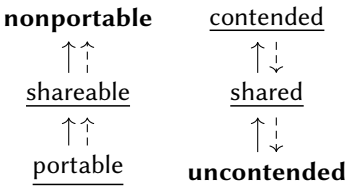


Fig. 1. The portability and contention modes (§2.1). *Sub-moding* follows solid arrows, and *sub-kinding* (§2.5) follows dashed arrows. Bolded modes are default, and underlined modes have a modality (§2.7) associated with them.

```
module Stack : sig @@ portable
  type 'a t
  val empty : 'a t
  val push : 'a → 'a t → 'a t
  val pop : 'a t → ('a * 'a t) option
end = struct
  type 'a t = | Nil | Cons of 'a * 'a t
  let empty = Nil
  let push x st = Cons (x, st)
  let pop st = match st with ...
end
val print_int_stack : (int Stack.t → unit)
    @@ portable
```

Fig. 2. An API for an immutable stack.

```
    int : value mod portable contended
'a list : value mod portable contended with 'a
 'a ref : value mod portable with 'a
'a * 'b : value mod portable contended
                with 'a with 'b
'a → 'b : value mod contended
 'a @@ m : value mod portable contended
                with 'a @@ m
```

Fig. 3. Some examples of types and their modal kinds.

We have now met two *modal axes*: **portable**/ **nonportable** on the *portability* axis, and **contended**/ **uncontended** on the *contention* axis. These two axes are summarized in Fig. 1. Each axis also has a third element: a **shared** value allows read-only access to mutable fields, and a **shareable** function may capture only read-only state (**shared** or **contended**) and read-only functions (**shareable** or **portable**).

All values in OxCaml are assigned a mode from each of the modal axes. Accordingly, the function type `arg_ty @ arg_mode → res_ty @ res_mode` specifies the expected input and output modes of the function, separate from the mode of the function value itself. Omitted modal axes are set to their **default** value, chosen for backward compatibility with OCaml.

OxCaml also supports *sub-moding*. Each axis arranges its modes along a lattice, and we can always consider a value with a stronger (lower) mode to instead have a weaker (higher) mode. For example, if we have an `int → string` function at **portable**, we can use that in a context expecting a **nonportable** value, because **portable** < **nonportable**. With sub-moding, we can refine our capture rules slightly: when a **portable** function captures an **uncontended** value, that value can be downgraded (that is, upcast) to **contended** before capture rather than becoming inaccessible.

Finally, modes in OxCaml are by default *deep*. That is, if a record is at mode **m**, then projecting a field from that record produces a value of mode **m** as well.

2.2 Motivation for Mode Crossing

Portability and contention are expressive enough to type-check interesting programs, especially in conjunction with concurrency APIs using other OxCaml modes [12]. However, limitations

appear quickly in real-world use. Consider an interface for immutable stacks (Fig. 2) and a function `print_int_stack` defined separately from the module. The `@@ portable` annotation on the signature states that all values in the module are **`portable`**. Similarly, the `@@ portable` annotation on `print_int_stack` states that it, too, is **`portable`**.

Intuitively, the implementation's immutability guarantee makes it safe to use an `int Stack.t` from another thread without risking data races. OxCaml, however, rejects the following program:

```
let st = Stack.(empty |> push 4 |> push 2) in
  fork_join2 (fun _ → print_int_stack st) (fun _ → ...)
                                      ^^
Error: The value st is nonportable but is expected to be portable
```

We can see why this is so by examining the types and modes. The `Stack` module signature does not actually expose the fact that `int Stack.t` is safe to access concurrently. Specifically, `st` is **`nonportable`**: the return mode of `Stack.push` is just the default mode **`nonportable uncontended`**. (Note that the function `Stack.push` itself is **`portable`**, but not its inputs and output.) Hence, `st` cannot be captured by the **`portable`** closure passed to `fork_join2`.

How do we generalize the `Stack` module while exposing the thread-safety guarantees that come from immutability? After all, an `int Stack.t` has no functions or mutability—there is no way it could cause trouble when used in different threads.

2.3 Mode Crossing and Kinds

We exploit the insight that some types are naturally agnostic to some modal axes. We say that a type *crosses* (or *mode-crosses*) an axis when all of its values have no interaction with that axis. In our case, `int Stack.t` crosses both portability and contention. This holds because `int` crosses both axes and because `Stack.t` preserves this property. We can encode this information in the stack signature in Fig. 2 by adding a kind annotation to the second line:

```
type 'a t : value mod portable contended with 'a
```

This specifies the *kind* of abstract `'a t`, restricting admissible instantiations while giving users of `'a t` additional capabilities (the ability to mode-cross). Such a kind has three components:

(1) The layout **`value`**. In OxCaml, layouts support unboxed representations [4, 6], inspired by [5] and [7]. Layouts are orthogonal to modal kinds and beyond this paper's scope.
(2) The modal bound **`mod portable contended`** states that values cross portability and contention. (Why do we specify modes instead of specifying whole axes? Read on to §2.7.)
(3) The list of with-bounds **`with`** `'a` states that the type can only cross axes that are also crossed by `'a`. With-bounds can mention arbitrary types.

A kind annotation on an abstract type describes a bound on admissible instantiations of that type. Any instantiation must have a kind at least as strong as this bound. The default kind annotation, if none is given, is just **`value`**. It expresses that the type does not cross any modal axes. Adding the above kind (**`value mod portable contended with`** `'a`) to the signature of the `Stack` module allows OxCaml to compile the example from before without any other changes.

2.4 Inferring Modal Kinds for Concrete Type Definitions

OxCaml automatically infers modal kinds of concrete types. Computing these kinds is sometimes challenging. In this section, we describe the rules for inferring kinds for some basic types and present an example where inferring the kind requires more care. We list examples of kinds in Fig. 3.

The kind of a record with only immutable fields is easy to infer:

```
type t = { a : string ; b : int }
(* Inferred kind: value mod portable contended with string with int,
    equivalent to value mod portable contended *)
```

The inferred kind includes two with-bounds, because `string` and `int` are contained in `t`. In many cases, the with-bounds of a type's kind are simple: they list all of the (potential) components of a type. However, users need not reason about with-bounds to understand `t`, because this kind simplifies to **`value mod portable contended`**: both `string` and `int` cross portability and contention.

What about the kind of a record with a mutable field?

```
type 'a ref = { mutable contents : 'a } (* Inferred kind: value mod portable with 'a *)
```

A record with a mutable field can never cross contention, because contention is used to track whether it can be accessed/mutated or not (but it may cross portability, if `'a` does so). The modal bound on its kind is thus **`value mod portable`**, and because it contains an `'a`, we need to add a with-bound **`with`** `'a`.

Functions, on the other hand, carry *computation* and no value components: their kind has no with-bounds. Functions never cross portability (because that's how we track their thread-safety), but they do cross contention. So, their overall kind is always **`value mod contended`**.

We have not yet talked about the kinds of composite types like `'a ref ref`. The syntax of kinds does not make it obvious how to compose and simplify them. These remaining gaps in our understanding of kinds become more pronounced when we consider recursive types, like this one:

```
type 'a nested_refs =
| Nil of 'a
| Cons of 'a ref nested_refs
```

This recursion is *non-uniform*: each recursive step adds another `ref` to the argument. Computing its kind requires reasoning about unbounded nesting.

2.5 Key Idea I: Kinds as Functions

Having seen several examples of kinds in action, we now formalize their meaning. So far we have explained kinds informally as a way to express the mode-crossing behavior of types. Our first key idea is to interpret the kinds of type constructors as functions that map the modes crossed by their type arguments to the modes crossed by their results. This yields a principled way to handle recursive types and later check *sub-kinding*.

Base types like `int` or `string ref` have *modal base kinds*, which specify their mode-crossing behavior along each axis—for example, **`mod portable contended`**. Modal base kinds do not have with-bounds. The kinds of type constructors are more interesting. For instance, recall the kind of `'a ref` from §2.4:

```
type 'a ref : value mod portable with 'a
```

The kind of `'a ref` depends only on the kind of `'a`, and not on its concrete instantiation. This means that we can interpret the kinds of type constructors as functions that take the modal base kind of `'a` and compute the modal base kind of `'a ref`. What should this function be, given the kind of `'a`? Specifically, what is the meaning of the with-bound **`with`** `'a`?

A with-bound **`with`** `t` *trims* the possible mode-crossing behaviors; It can only reduce the number of axes that are crossed. We intend to set this up as a join in a lattice, but we first need a more precise representation of modal base kinds. We define modal base kinds as elements of the set of modes, but with a different (partial) order on this set compared to the sub-moding order: We flip the order on the contention axis, as shown in Fig. 1. We call this the *sub-kinding* order. (Overall, a "larger"

kind comprises more types and permits less mode-crossing behavior.) The bottom modes with respect to sub-kinding (**portable** and **contended**) represent that a type can cross the respective axis, like modal bound annotations `mod portable contended` (§2.3). Conversely, top modes (**nonportable** and **uncontended**) represent that the respective axis is not crossed. The reason for this choice is that mode crossing is intimately connected to modalities (§2.7), and their behavior determines which mode sits at $\bot$.

Let us get back to the kind of `'a ref`. The modal-bound `value mod portable` (represented by (**portable**, **uncontended**)) states the "floor" of mode-crossing behavior, that is, that `'a ref` could potentially cross portability, but never crosses contention. However, its mode-crossing behavior also depends on `'a`: the with-bound `with 'a` potentially *removes* per-axis mode-crossing behavior, based on whether `'a` crosses **portable** or not. So `with 'a` means taking the least upper bound of the rest of the kind with the kind of `'a`. This is precisely the join operation on the sub-kinding lattice. Given the base kind of `'a`, call it $\hat{\alpha}$, we get the following so-called *kind function* for `'a ref`:

$$f(\hat{\alpha}) = (\textbf{portable}, \textbf{uncontended}) \sqcup \hat{\alpha}$$

2.6 Key Idea II: Fixpoints of Kind Functions for Recursive Types

Recall the `'a nested_refs` type from §2.4. It is not obvious what its mode-crossing behavior is. We use kind functions as a tool to analyze the mode-crossing behavior of such complicated recursive types: we set up a *recursive equation* that must hold for its kind function. Using the type's definition and the kind function of `'a ref` we just computed, we can set up the following equation on the kind function f of `'a nested_refs`:

$$f(\hat{\alpha}) = (\textbf{portable}, \textbf{contended}) \sqcup \hat{\alpha} \sqcup f((\textbf{portable}, \textbf{uncontended}) \sqcup \hat{\alpha})$$

Note that we use composition of f with the kind function of `'a ref` to interpret the with-bound `with 'a ref nested_refs` that mentions a composed type. There are many monotone solutions for f that satisfy the equation. But since we are working in a finite and hence complete lattice, we know that there always exists a least solution (a least fixpoint) to such equations.[4] This least solution corresponds to the strongest kind function we can choose. In this case, the solution is:

$$f(\hat{\alpha}) = (\textbf{portable}, \textbf{uncontended}) \sqcup \hat{\alpha}$$

It corresponds to the kind `value mod portable with 'a`.

2.7 Key Idea III: Invariant Modalities and Mode Crossing

We have discussed how several of OxCaml's features (records, mutable state, and recursive types) interact with modal kinds. So far, we have ignored one important feature of OxCaml: modalities. Modalities are used to adjust the underlying mode of selected record fields. Mode crossing and modalities are closely related: A type crosses a modal axis if and only if it is *invariant* under a certain modality. In this section, we connect mode crossing and modalities, use that connection to justify our sub-kinding lattice from the previous section, and then describe the mode-crossing behavior of modalities themselves.

Consider the following record:

```
type 'a port = { x : 'a @@ portable }
```

Its only field is equipped with a *modality annotation* `@@ portable`, expressing that the value in the `x` field is always **portable**, regardless of the mode of the record. (Note that the `@@ portable` annotation itself is not a type former, just an annotation on a record field.) We refer to `'a port` as

[4] To be pedantic: We are forming a least fixpoint on the lattice of functions from modal base kinds to modal base kinds, which is complete if the sub-kinding lattice itself is.

the **portable** *modality* because it wraps its only field with `@@ portable`. OxCaml also supports the **contended** modality, among others.

We can observe the following connection between the **portable** modality and mode crossing: A type `t` crosses portability iff it is *invariant* under the **portable** modality, i.e., if `t` and `t port` are morally "equivalent". The proof is simple: if `t` crosses portability, we can define trivial coercions between `t` and `t port`; conversely, these coercions suffice to justify mode crossing. Analogously, a type crosses contention iff it is invariant under the **contended** modality. Indeed, the modes in a modal bound `mod portable contended` do not refer to the portability and contention axes, but to the **portable** and **contended** modalities. They express that a type is invariant under the **portable** and **contended** modalities, and so crosses portability and contention. Hence we want **portable** and **contended** to be the bottom values in the sub-kinding order.

This is why we flipped the order of the contention axis in the sub-kinding lattice relative to the sub-moding lattice in §2.5. Take another look at Fig. 1: modalities behave differently on portability versus contention. The **portable** and **contended** modalities correspond to the bottom and top modes of their axes with respect to sub-moding, respectively. A **nonportable** or an **uncontended** modality would be unsound: they could be used to store **nonportable** (**uncontended**) data in a **portable** (**contended**) value and would allow it to be smuggled to another thread.

OxCaml also supports **shareable** and **shared** modalities that correspond to *middle modes* on their respective axes. Reading a record field annotated with the **shared** modality yields its value at **shared** mode, except when the surrounding record is at **contended**: then the field value is only accessible at **contended** mode (cf. [12, §4.3]). We say that a type crosses **shared** if it is invariant under the **shared** modality. In this case, values of that type can move freely between the **uncontended** and **shared** modes, but not between those and **contended**. The **shareable** modality behaves similarly: it collapses the **shareable** and **nonportable** modes.

We have discussed the interactions between mode crossing and modalities so far, but not the mode-crossing behavior of modalities themselves. The type `'a port` always crosses portability, and only crosses contention if `'a` also does so. Unfortunately, we cannot express this mode-crossing behavior using the syntax of modal kinds we have introduced so far: when interpreting with-bounds as joins on the sub-kinding lattice, we cannot "selectively apply" the mode-crossing behavior on some axes. From now on, we allow modality annotations to show up in with-bounds to express this behavior. The inferred kind of `'a port` is:

```
type 'a port : value mod portable contended with 'a @@ portable
```

Semantically, we can read **with** as a join (combining restrictions) and `@@` as a meet (weakening restrictions):

$$f(\hat{\alpha}) = (\textbf{portable}, \textbf{contended}) \sqcup (\hat{\alpha} \sqcap (\textbf{portable}, \textbf{uncontended}))$$

We interpret `@@ portable` as taking the meet with $(\textbf{portable}, \top) = (\textbf{portable}, \textbf{uncontended})$ as to only affect the portability axis.

2.8 Key Idea IV: Abstract Kinds for Abstract Types

In our kind-function account of OxCaml's kind system, we have so far handled only types whose definitions are visible. But as we have seen in our introductory example of kinds in §2.3, kinds also interact with OxCaml's module system. Using modules and functors, we can mention so-called *abstract types* in with-bounds. An abstract type is a type declaration without a definition, either because it has not been instantiated or because its definition has been hidden.

Consider the following example of a map functor with mode crossing annotations:

```
module type OrderedType = sig
  type t
  val compare : (t → t → int) @@ portable
end
module Map (Ord : OrderedType) : sig @@ portable
  type key = Ord.t
  type 'a t : value mod portable contended with 'a with Ord.t
  ...
end = struct ... end
```

Given a type with an associated `compare` function, the `Map` functor generates an implementation of an immutable map. The `compare` function is required to be **`portable`** so that it can be called concurrently from multiple threads. The functor exposes a type constructor `'a t` whose mode-crossing behavior is constrained by `'a` and `Ord.t`. Naturally, we want `'a t` to have as much mode-crossing behavior as `Ord.t` does. However, this means that the with-bound `with Ord.t` depends not on the declared kind of `Ord.t`—the default kind **`value`**—but on the as-yet-unknown kind of the *instantiation* of `Ord.t`. We need to treat `with Ord.t` as referring to an *abstract kind* that is instantiated along with `Ord`.

Checking sub-kinding in the presence of abstract kinds is challenging. Consider this:

```
type 'a t (* some abstract type in our context *)
module Foo : sig
  type 'a t2 : value mod portable contended with 'a t
end = struct
  type 'a t2 = { x : 'a t t }
end
```

The compiler has to determine whether **`value mod portable contended with`** `'a t t` is a sub-kind of **`value mod portable contended with`** `'a t`. It is. The example type-checks successfully. But why? To answer this question, we need a more precise description of how kinds can behave.

2.9 Key Idea V: Kinds as Coefficients for Inference

We now shift from semantics to implementation: we discuss how to represent kind functions by coefficients and reduce sub-kinding to coefficient constraints. Previously, we claimed that **`value mod portable contended with`** `'a t t` is a sub-kind of **`value mod portable contended with`** `'a t`. How do we show this? Let us use f to refer to the abstract kind function of `'a t`. We need to check:

$$\forall \hat{\alpha}.\ (\textbf{portable}, \textbf{contended}) \sqcup f(f(\hat{\alpha})) \leq (\textbf{portable}, \textbf{contended}) \sqcup f(\hat{\alpha})$$

Kind functions admit a normal form: for unary f, there are constants c_0, c_1 such that[5]

$$f(\hat{\alpha}) = c_0 \sqcup (c_1 \sqcap \hat{\alpha}).$$

This fact follows from the structure of kind functions: They are compositions of functions that consist of constants, arbitrary joins, meets with constants, and least fixpoints (of recursive types). All of these operations preserve this structure, because in the sub-kinding lattice, meets and joins are *distributive*. Finally, by some basic computation on lattice terms, we can justify the claim:[6]

$$f(f(\hat{\alpha})) = c_0 \sqcup (c_1 \sqcap (c_0 \sqcup (c_1 \sqcap \hat{\alpha}))) = c_0 \sqcup (c_1 \sqcap c_0) \sqcup (c_1 \sqcap c_1 \sqcap \hat{\alpha}) = c_0 \sqcup (c_1 \sqcap \hat{\alpha}) = f(\hat{\alpha})$$

For n-ary constructors, the normal form generalizes to:

$$f(\hat{\alpha}_1, \ldots, \hat{\alpha}_n) = c_0 \sqcup (c_1 \sqcap \hat{\alpha}_1) \sqcup \ldots \sqcup (c_n \sqcap \hat{\alpha}_n)$$

[5] Our actual normal form theorem(s) require additional properties c_0 and c_1 to ensure uniqueness.
[6] We actually show something stronger: the with-bounds **`with`** `'a t` and **`with`** `'a t t` are equivalent for all unary type constructors `'a t`.

Note, however, that these constants may themselves depend on the kinds of other types. Specifically, these c_i can be expressed using constants, joins, and (arbitrary) meets of coefficients of the kind functions of other ambient types (either concrete or abstract).

Generally, a sub-kinding query asks whether one kind function f is less than another g for all inputs. How would we go about checking this using coefficients? For simplicity, let us consider unary functions f and g again: let f be represented by coefficients c_0, c_1, and g be represented by coefficients d_0, d_1. We want to verify the following property:

$$\forall \hat{\alpha}.\ c_0 \sqcup (c_1 \sqcap \hat{\alpha}) \leq d_0 \sqcup (d_1 \sqcap \hat{\alpha})$$

How do we check this? Using $x \leq y \iff x \sqcup y = y$ we can rewrite it as:

$$\forall \hat{\alpha}.\ (c_0 \sqcup d_0) \sqcup ((c_1 \sqcup d_1) \sqcap \hat{\alpha}) = d_0 \sqcup (d_1 \sqcap \hat{\alpha})$$

We compute a normal form of both sides, and check for syntactic equality of the coefficient formulas!

This is promising news for our sub-kinding solver (§4): it can manipulate kind functions in terms of coefficients. It represents the kinds of concrete types using *equalities* $c_i = \ldots$ *on their coefficients* computed from type structure, and represents the kinds of abstract types using *inequalities* $c_i \leq \ldots$ on their coefficients computed from their modal kind bounds. The right-hand sides of these (in-)equalities may mention constants and arbitrary meets and joins of other coefficients. Similarly, the solver reduces sub-kinding queries themselves to inequalities on coefficients only.

To solve such inequalities on coefficients in the presence of constraints on coefficients, the sub-kinding solver iteratively *inlines* more and more constraints into the inequalities to be checked, while simultaneously tying recursive knots that show up on the way. It continues until no constraints are left, at which point subsumption is easy to check. This procedure is explained by example in §4.

3 Type System

This section presents *SMoKi*, a pen-and-paper type system for the language from §2. We describe the kind calculus in §3.2, recall OxCaml-style modal type systems in §3.3, and instantiate the calculus to model SMoKi in §3.4. In §3.5 we justify module reasoning via substitution, and in §3.6 we sketch soundness. We summarize definitions from this section in Fig. 4.

All orderings and lattice operations in this section are wrt. sub-kinding, not sub-moding.

3.1 Preliminaries

We recall some standard definitions and facts about finite lattices. A finite lattice L is a set with a partial order $\leq$ and operations meet $\sqcap$ and join $\sqcup$. The element $\ell_1 \sqcap \ell_2$ is the greatest element ℓ_3 such that $\ell_3 \leq \ell_1$ and $\ell_3 \leq \ell_2$. Join is dual to meet, working up the lattice instead of down. Our lattices are always *bounded*, i.e. they have distinguished top $\top$ and bottom $\bot$ elements. On finite lattices, all monotone functions have least fixpoints.

Recall the definition of a Bi-Heyting algebra [e.g., 2, 8]:

Definition 1. A *Bi-Heyting algebra* $(L, \sqcup, \sqcap, \top, \bot, \leq, /, \setminus)$ is a bounded lattice equipped with additional binary operations $/$ and $\setminus$ called *implication* and *subtraction*, respectively, such that

$$a \leq b \,/\, c \iff a \sqcap b \leq c \qquad\qquad a \setminus b \leq c \iff a \leq b \sqcup c.$$

Simple examples of Bi-Heyting algebras are lattices based on *bounded linear orders*. In that case, the implication and subtraction operations behave as follows:

$$a \,/\, b := \begin{cases} \top & \text{if } a \leq b \\ b & \text{if } a > b \end{cases} \qquad\qquad a \setminus b := \begin{cases} \bot & \text{if } a \leq b \\ a & \text{if } a > b \end{cases}$$

Products of Bi-Heyting algebras are again Bi-Heyting algebras under pointwise lifting of all operations. Hence, the lattices of sub-moding and of sub-kinding (Fig. 1) are Bi-Heyting algebras: they are products of bounded linear orders. Bi-Heyting algebras are distributive lattices, *i.e.*, meets and joins distribute in both directions. We use this fact implicitly throughout this text.

3.2 Kind Calculus

We now describe the SMoKi components specific to kinds: the type context Δ, a syntactic notion of kind functions, and the sub-kinding judgment $\Delta \vDash \phi \leq \psi$. All three components are agnostic to the set of types, the underlying type system, the choice of modes, and the sub-kinding lattice. For the purpose of this section, fix an *arbitrary finite sub-kinding lattice* M. Our components are agnostic to what kinds are used for: we defer the mode-crossing rule that connects kinds to typing to §3.4.

Type contexts Δ are lists of abstract types. Abstract types are either type variables α, which are always unrestricted, or abstract type constructors t with an associated arity and kind. All type variables and abstract type constructors in the context are associated with an implicit abstract base kind or abstract kind function, denoted by a hat like $\hat{\alpha}$ or $\hat{t}$, respectively. Type constructors appear in type contexts as elements $t : \overline{\hat{\alpha}}.\ \phi$, where the *kind term* ϕ describes a kind bound on t given the argument kinds $\overline{\hat{\alpha}}$. The kind term ϕ may also mention the kinds of abstract types in its context Δ.

Kind terms provide a syntactic description of kind functions. A kind term may contain base kind constants m, joins of two kind terms, and meets of a kind term with a base kind (matching the form of kind functions in §2). Finally, kind terms can form (non-uniform) fixpoints on the underlying lattice, modeled using a parameterized μ construct. Such fixpoints always exist because all kind terms correspond to monotone functions. Note that kind terms have no way of referring to non-abstract types like `int`: instead, we describe how to translate types to their kinds in §3.4.

Finally, we discuss the sub-kinding judgment $\Delta \vDash \phi \leq \psi$. Intuitively, it should mean the following: the base kind determined by ϕ should be at most that determined by ψ, under any instantiation of abstract base kinds $\hat{\alpha}$ (for type variables) and abstract kind functions $\hat{t}$ (for abstract type constructors) from Δ.

Actually defining this judgment is a little tricky. We define it *semantically* (hence the $\vDash$): we first quantify over all "valid" abstract base kinds and kind functions determined by Δ, then evaluate $\phi \leq \psi$ in the underlying lattice.

First, we define a semantic interpretation from kind terms to base kinds with respect to a *substitution* that maps variables $\hat{\alpha}$ and $\hat{t}$ to base kinds or kind functions, respectively:

Definition 2. A *substitution* is a function $\rho : (\mathrm{TyCon} \cup \mathrm{TyVar}) \rightharpoonup \mathrm{Mon}(M^*, M)$, where $A \rightharpoonup B$ denotes the set of partial functions and $\mathrm{Mon}(A, B)$ denotes the set of monotone functions from A to B. (Substitutions don't fix the arity of abstract kind function symbols $\hat{t}$ to simplify the notation.) Given a substitution ρ, we define the *evaluation* $[\![\phi]\!]_\rho$ of a kind term to a base kind as follows:

$$[\![\hat{\alpha}]\!]_\rho := \rho(\alpha)() \qquad [\![\phi \sqcup \psi]\!]_\rho := [\![\phi]\!]_\rho \sqcup [\![\psi]\!]_\rho$$

$$[\![\hat{t}\ \overline{\phi}]\!]_\rho := \rho(t)(\overline{[\![\phi]\!]_\rho}) \qquad [\![\phi \sqcap m]\!]_\rho := [\![\phi]\!]_\rho \sqcap m$$

$$[\![m]\!]_\rho := m \qquad [\![(\mu\ \hat{t}\ \overline{\hat{\alpha}}.\ \phi)\ \overline{\psi}]\!]_\rho := (\mu(\lambda\ f.\ \lambda\ \overline{m}.\ [\![\phi]\!]_{\rho, t \mapsto f, \overline{\alpha \mapsto m}}))\,\overline{[\![\psi]\!]_\rho}$$

Note that we use a semantic least fixpoint μF over the lattice of functions $\mathrm{Mon}(M^n, M)$ from n-tuples of base kinds to base kinds to interpret a syntactic fixpoint.

Now we define when a substitution ρ is valid for a type context Δ, by requiring that the inequalities of kinds determined by Δ hold, and use it to define sub-kinding.

$$\alpha, \beta \in \text{TyVar} := \ldots \qquad t \in \text{TyCon} := \ldots$$

$$p \in \text{Portability} := \textbf{nonportable} \mid \textbf{shareable} \mid \textbf{portable}$$

$$c \in \text{Contention} := \textbf{uncontended} \mid \textbf{shared} \mid \textbf{contended}$$

$$w \in W := \text{Portability} \qquad n \in N := \text{Contention} \qquad m \in M := W \times N$$

$$\phi, \psi, \chi \in \text{KindTerm} ::= \hat{\alpha} \mid \hat{t}\ \overline{\phi} \mid m \mid \phi \sqcup \psi \mid \phi \sqcap m \mid (\mu\ \hat{t}\ \overline{\hat{\alpha}}.\ \phi)\ \overline{\psi}$$

$$A, B, C \in \text{Type} ::= \alpha \mid t\ \overline{A} \mid A \mathbin{@} m_1 \to B \mathbin{@} m_2 \mid \square^m A \mid (\mu\ t\ \overline{\alpha}.\ A)\ \overline{B} \mid \mathbb{1} \mid \mathbb{B} \mid \mathbb{Z} \mid A \times B \mid A + B \mid \text{ref}\ A \mid \text{atomic}\ A \mid \ldots$$

$$\Delta ::= \emptyset \mid \Delta, \alpha \mid \Delta, t : \overline{\hat{\alpha}}.\ \phi$$

$$\Gamma ::= \emptyset \mid \Gamma, x : - \mid \Gamma, x : A \mathbin{@} m$$

Substitution (selected rules)

$$(t\ \overline{A})[t \mapsto \overline{\alpha}.\ C] := C[\overline{\alpha} \mapsto \overline{A[t \mapsto \overline{\alpha}.\ C]}]$$

$$(t'\ \overline{A})[t \mapsto \overline{\alpha}.\ C] := t'\ \overline{A[t \mapsto \overline{\alpha}.\ C]} \qquad (t \neq t')$$

$$((\mu\ t'\ \overline{\beta}.\ A)\ \overline{B})[t \mapsto \overline{\alpha}.\ C] := (\mu\ t'\ \overline{\beta}.\ A[t \mapsto \overline{\alpha}.\ C])\ \overline{B[t \mapsto \overline{\alpha}.\ C]}$$

The "kind of" operation $\text{kind}(A)$

$$\begin{aligned}
\text{kind}(\alpha) &:= \hat{\alpha} \\
\text{kind}(t\ \overline{A}) &:= \hat{t}\ \overline{\text{kind}(A)} \\
\text{kind}(\mathbb{1}) &:= (\textbf{portable}, \textbf{contended}) \\
\text{kind}(\mathbb{B}) &:= (\textbf{portable}, \textbf{contended}) \\
\text{kind}(\mathbb{Z}) &:= (\textbf{portable}, \textbf{contended}) \\
\text{kind}(\square^m A) &:= \text{kind}(A) \sqcap m
\end{aligned}$$

$$\begin{aligned}
\text{kind}(A_1 \times A_2) &:= \text{kind}(A_1) \sqcup \text{kind}(A_2) \\
\text{kind}(A_1 + A_2) &:= \text{kind}(A_1) \sqcup \text{kind}(A_2) \\
\text{kind}(A \mathbin{@} m_1 \to B \mathbin{@} m_2) &:= (\textbf{nonportable}, \textbf{contended}) \\
\text{kind}(\text{ref}\ A) &:= (\textbf{portable}, \textbf{uncontended}) \sqcup \text{kind}(A) \\
\text{kind}(\text{atomic}\ A) &:= (\textbf{portable}, \textbf{contended}) \\
\text{kind}((\mu\ t\ \overline{\alpha}.\ A)\ \overline{C}) &:= (\mu\ \hat{t}\ \overline{\hat{\alpha}}.\ \text{kind}(A))\ \overline{\text{kind}(C)}
\end{aligned}$$

$\boxed{\Delta; \Gamma \vdash e : A \mathbin{@} m}$

Cross
$$\frac{\Delta; \Gamma \vdash e : A \mathbin{@} (w, n) \qquad \Delta \vDash \text{kind}(A) \leq (w', n')}{\Delta; \Gamma \vdash e : A \mathbin{@} (w' \sqcap w, n' \,/\, n)}$$

Weaken
$$\frac{\Delta; \Gamma \vdash e : A \mathbin{@} (w, n) \qquad w \leq w' \qquad n' \leq n}{\Delta; \Gamma \vdash e : A \mathbin{@} (w', n')}$$

Lam
$$\frac{\Delta; \blacklock_w(\Gamma), x : A \mathbin{@} m_1 \vdash e : B \mathbin{@} m_2}{\Delta; \Gamma \vdash \lambda x.\ e : (A \mathbin{@} m_1 \to B \mathbin{@} m_2) \mathbin{@} (w, n)}$$

App
$$\frac{\Delta; \Gamma \vdash e_1 : (A \mathbin{@} m_1 \to B \mathbin{@} m_2) \mathbin{@} m_3 \qquad \Delta; \Gamma \vdash e_2 : A \mathbin{@} m_1}{\Delta; \Gamma \vdash e_1\ e_2 : B \mathbin{@} m_2}$$

Box
$$\frac{\Delta; \Gamma \vdash e : A \mathbin{@} m_1 \sqcap m_2}{\Delta; \Gamma \vdash \text{box}\ e : \square^{m_2} A \mathbin{@} m_1}$$

Unbox
$$\frac{\Delta; \Gamma \vdash e : \square^{m_2} A \mathbin{@} m_1}{\Delta; \Gamma \vdash \text{unbox}\ e : A \mathbin{@} m_1 \sqcap m_2}$$

Roll
$$\frac{\forall i.\ \Delta \vdash B_i\ \text{ok} \qquad \Delta; \Gamma \vdash e : A[\overline{\alpha} \mapsto \overline{B}, t \mapsto \overline{\alpha}.(\mu\ t\ \overline{\alpha}.\ A)\ \overline{\alpha}] \mathbin{@} m}{\Delta; \Gamma \vdash \text{roll}\ e : (\mu\ t\ \overline{\alpha}.\ A)\ \overline{B} \mathbin{@} m}$$

Unroll
$$\frac{\Delta; \Gamma \vdash e : (\mu\ t\ \overline{\alpha}.\ A)\ \overline{B} \mathbin{@} m}{\Delta; \Gamma \vdash \text{unroll}\ e : A[\overline{\alpha} \mapsto \overline{B}, t \mapsto \overline{\alpha}.(\mu\ t\ \overline{\alpha}.\ A)\ \overline{\alpha}] \mathbin{@} m}$$

Fig. 4. Selected SMoKi rules. The remaining substitution rules are standard. Parallel substitution of type variables into types $A[\overline{\alpha} \mapsto \overline{B}]$ and substitutions on lattice terms are defined analogously. The judgement $\Delta \vdash A$ ok is defined as usual to check whether type constructors are applied correctly. All lattice operations are with respect to *sub-kinding*.

Definition 3. We write $\rho \in [\![\Delta]\!]$ if

(1) $\mathrm{dom}\ \rho = \mathrm{dom}\ \Delta$.
(2) for all $(t : \overline{\hat{\alpha}}.\ \phi) \in \Delta$, for all $\overline{m}$, $\rho\ t\ \overline{m} \leq [\![\phi[\overline{\hat{\alpha}} \mapsto \overline{m}]]\!]_\rho$.
(3) ρ is valid: for all $(t : \overline{\hat{\alpha}}.\ \phi) \in \Delta$, there exists χ such that for all $\overline{m}$, $\rho\ t\ \overline{m} = [\![\chi[\overline{\hat{\alpha}} \mapsto \overline{m}]]\!]_\varnothing$.

Definition 4. We say that $\Delta \vDash \phi \leq \psi$ if for every $\rho \in [\![\Delta]\!]$ we have $[\![\phi]\!]_\rho \leq [\![\psi]\!]_\rho$.

There is a subtle validity condition (3) in Def. 3. For sub-kinding, intuitively, we need to check $[\![\phi]\!]_\rho \leq [\![\psi]\!]_\rho$ only for ρ where all functions $\rho\ t$ could have come from a valid kind term, *i.e.*, are "realizable" kind functions. However, some functions in $\mathrm{Mon}(M^*, M)$ don't correspond to any such realizable kind. (Specifically, only those with a normal form as presented in §2.9 do.) The condition ensures that we only need to check $[\![\phi]\!]_\rho \leq [\![\psi]\!]_\rho$ for realistic substitutions ρ. It does so by requiring $\rho\ t$ be representable by a kind term χ wrt. the empty substitution, which can be shown equivalent to the normal form condition from §2.9.

3.3 Modal Type Systems

We now instantiate §3.2 to a concrete model of OxCaml. Key rules are in Fig. 4; the supplementary material gives the remaining definitions [23]. DRFCaml background appears in [12, §4].

We distinguish between the lattices of comonadic modes $w \in W :=$ Portability and monadic modes $n \in N :=$ Contention. Comonadic modes are those that track properties of functions, and monadic modes are those that track properties of references. To accommodate more modal axes, the W and N lattices can also be instantiated with (the products of) multiple modal axis lattices. Modes themselves are drawn from $m \in M := W \times N$. *We order these sets as the sub-kinding lattice, rather than by sub-moding as in previous work.* The choice of ordering affects, for instance, the Weaken rule: It allows *weakening* from mode (w, n) to (w', n') when $w \leq w'$ and $n' \leq n$ (note the flipped direction for the monadic component).

Our typing judgment $\Delta; \Gamma \vdash e : A\ @\ m$ can mention modes in three places: as part of the output (alongside the type), within types themselves, and in variable bindings in the term context Γ. Binders in Γ are either associated with a type and a mode, or *inaccessible*, denoted with $x : -$.

Inaccessible binders model the capture behavior of closures. Consider the Lam rule for lambda expressions $\lambda x.\ e$. It has type $A\ @\ m_1 \to B\ @\ m_2$ if, given $x : A\ @\ m_1$, the body e has type B at mode m_2. The mode of the closure itself, (w, n), also affects typing, because it determines what the closure can capture from its environment, and at what mode it can access it. This depends only on the comonadic component (so portability). The Lam rule expresses this constraint by applying a *lock* $🔒_w$ to the term context Γ before the binder $x : A\ @\ m_1$. Locks are operations on contexts that change the modes of binders in the context or render them inaccessible, depending on w.

$$
\begin{aligned}
🔒_w(\emptyset) &:= \emptyset \\
🔒_w(\Gamma, x : -) &:= 🔒_w(\Gamma), x : - \\
🔒_{w_1}(\Gamma, x : A\ @\ (n_2, w_2)) &:= \begin{cases} 🔒_{w_1}(\Gamma), x : A\ @\ (n_2, w_2) & \text{if } w_1 = \texttt{nonportable} \\ 🔒_{w_1}(\Gamma), x : A\ @\ (n_2, \texttt{contended}) & \text{if } w_1 = w_2 = \texttt{portable} \\ 🔒_{w_1}(\Gamma), x : - & \text{otherwise} \end{cases}
\end{aligned}
$$

Since locks are an operation (and not syntactically part of contexts), the variable rule of our system is simple: it just looks up a variable in the context. (Keeping inaccessible binders in the context instead of just removing them ensures that locks and variable shadowing interact well together.)

3.4 Instantiation to OxCaml

We now instantiate the kind system from §3.2. The sub-kinding lattice M is the sub-kinding lattice over the set of modes from § 2.5. We present OxCaml-specific types and their kinds in Fig. 4, including integers, products, sums, references, and iso-recursive types, among others.

The kind of a type A is computed by a function $\mathrm{kind}(A)$ from types to kind terms in Fig. 4. The key typing rule in SMoKi is Cross. Its antecedent $\Delta \vDash \mathrm{kind}(A) \leq (w', n')$ requires that the kind of A, has at least the mode-crossing behavior (w', n'). Recall from §2.7 that for a type with mode-crossing behavior (w', n'), all modes *above* (w', n') in sub-kinding collapse; they are equivalent and can be crossed between. We need to determine the "best" mode a value currently at mode (w, n) can cross to, that is, the least mode *with respect to sub-moding*.

On comonadic axes, sub-kinding and sub-moding coincide: for each axis, if the crossing mode w' is less than the current mode w, we want to cross to w', and otherwise stay at w. This is the meet $w \sqcap w'$. The situation for monadic axes is different: for each axis, if the crossing mode n' is less than or equal to the current mode n, we want to cross to the *top mode* on that axis. This top mode corresponds to the bottom ("strongest") mode with respect to sub-moding. Otherwise, if $n' > n$, we want to stay at mode n. We actually have an operation that does this: it is the implication $/$ from a Bi-Heyting algebra! Hence, the monadic mode to cross to is n' / n.

For instance, if we have a $n = \textbf{contended}$ value that crosses $n' = \textbf{contended}$, then it should cross to $\textbf{contended} / \textbf{contended} = \textbf{uncontended}$. (Recall the definition of $/$ in bounded linear orders from §3.1.) If it only crossed $n' = \textbf{shared}$, then it would cross to $\textbf{shared} / \textbf{contended} = \textbf{contended}$.

The modality type $\Box^m$ is used to model record fields annotated with a modality (§ 2.7). It is annotated with an arbitrary mode m, and its typing rules take a meet (on the sub-kinding lattice) of the ambient mode with m. We can recover all modalities considered in §2.7 by choosing m as follows, for instance for `@@ portable`, `@@ contended`, and `@@ shared`, respectively:

$$(\textbf{portable}, \textbf{uncontended}) \qquad (\textbf{nonportable}, \textbf{contended}) \qquad (\textbf{nonportable}, \textbf{shared})$$

Our formalization supports non-uniform recursive type constructors. To avoid syntactic overhead, we desugar mutually recursive types to non-mutually recursive types using *Bekić's theorem* [1]. The kind of a recursive type is defined as a fixpoint of its underlying type function, as expected.

3.5 Modules

SMoKi does not explicitly formalize a module system: it only formalizes abstract types, and so disregards many aspects of a full module system. Instead, the examples of modules in this paper should be viewed through the lens of abstract types, as formalized by typing contexts Δ.

For example, an ambient opaque module signature

```
module Foo : sig
  type t : value mod portable contended
  val check : t → bool
end
```

can be read as SMoKi context assumptions

$$\Delta = t : (\textbf{portable}, \textbf{contended}) \qquad \Gamma = \mathsf{check} : t \to \mathsf{bool}.$$

A client may use the fact that `Foo.t` crosses portability and contention, but it cannot inspect the representation of `Foo.t`. Correctness of module instantiation, then, corresponds to a form of substitution. For instance, a concrete implementation may define `Foo.t` as `int`:

```
module Foo : sig ... end = struct
  type t = int
```

```
  let check x = x = 0
end
```

The *module inclusion check* requires the usual sub-kinding obligation:

$$\mathrm{kind}(\mathbb{Z}) \leq (\textbf{portable}, \textbf{contended}).$$

After this check, replacing the abstract type by its implementation should preserve typing. This is the role of the substitution theorem:

Theorem 5 (Substitution)**.** *The following rules are admissible, where all objects must be closed in their respective contexts:*

$$\frac{\Delta, \alpha; \Gamma \vdash e : A \ @\ m}{\Delta[\hat{\alpha} \mapsto \mathrm{kind}(B)]; \Gamma[\alpha \mapsto B] \vdash e : A[\alpha \mapsto B] \ @\ m}$$

$$\frac{\Delta, t : \overline{\hat{\alpha}}.\ \phi; \Gamma \vdash e : A \ @\ m \qquad \Delta, \overline{\alpha} \vDash \mathrm{kind}(B) \leq \phi}{\Delta[\hat{t} \mapsto \overline{\hat{\alpha}}.\ \mathrm{kind}(B)]; \Gamma[t \mapsto \overline{\alpha}.\ B] \vdash e : A[t \mapsto \overline{\alpha}.\ B] \ @\ m}$$

The first rule substitutes a concrete type for an abstract variable ($\alpha \mapsto B$, $\hat{\alpha} \mapsto \mathrm{kind}(B)$). The second does the analogous higher-order substitution for abstract constructors. Both follow from similar substitution lemmas for $\Delta \vDash \phi \leq \psi$. Although we leave a formal treatment of OxCaml modules to future work, we believe the above substitution theorem is the key sanity check needed to ensure soundness of modes in the presence of modules.

3.6 Correctness

We build on the DRFCaml semantic model in Rocq with Iris [16, 17] to prove the soundness of SMoKi. The mechanized proof is part of the supplementary materials [23]. Here we restate the soundness theorem and outline the changes required to support mode crossing.

DRFCaml defines an operational semantics for a language with an explicit separation between heap and stack, concurrency, and data race detection à la RustBelt [15]. Its logical relation supports the portability, contention, locality, affinity, and uniqueness modal axes. Although we present SMoKi only with the portability and contention axes, this is only a projection used for exposition: our soundness proof handles all of DRFCaml's axes. The complete rule set appears in the supplementary material [23]. When instantiated with all modal axes, SMoKi strictly extends DRFCaml. We extend the DRFCaml logical relation to prove the following theorem in Rocq:

Theorem 6. *SMoKi is sound; that is, if a program type-checks in the empty context $\emptyset; \emptyset \vdash e : A \ @\ m$, then no execution of e crashes or has a data race.*

The mechanized soundness theorem is in the style of RustBelt [15], where contexts (specifically Δ) are not modeled explicitly. Instead, RustBelt models such contexts as assumptions in the meta-level context. Similarly, we do not explicitly formalize the function that computes the kind of a type $\mathrm{kind}(A)$; instead, we define a predicate $\mathrm{crosses}(\tau, m)$ stating that a semantic type τ crosses modal base kind m and prove properties of this predicate for all type constructors.

Conceptually, the $\mathrm{crosses}(\tau, m)$ predicate is defined as follows:

$$\mathrm{crosses}(\tau, m) := \forall m' : M, v : \mathrm{val}.\ \tau(m', v) \Leftrightarrow \tau(m \sqcap m', v)$$

In DRFCaml's semantic model, semantic types τ are predicates not only over values, but also over the mode m of the value. (They are also parameterized over a *world* Δ that we omit here for simplicity.) The $\mathrm{crosses}(\tau, m)$ predicate states that, if a type τ has modal base kind m, then for its

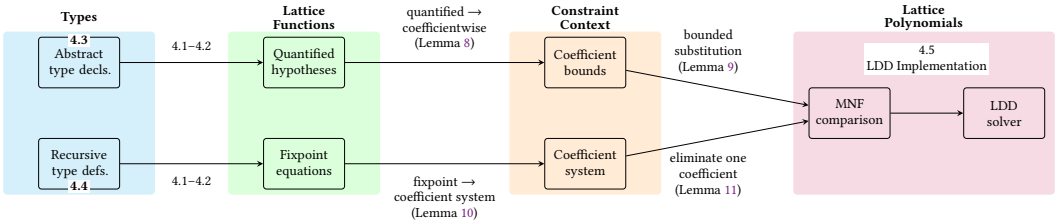


Fig. 5. Reduction architecture of inference.

inhabitants, all modes above m collapse. Indeed, if we know crosses(τ, m) and have two modes $m_1 \geq m$ and $m_2 \geq m$, then for all v:

$$\tau(m_1, v) \overset{\text{cross}}{\Longleftrightarrow} \tau(m \sqcap m_1, v) \overset{\text{absorb}}{\Longleftrightarrow} \tau(m, v) \overset{\text{absorb}}{\Longleftrightarrow} \tau(m \sqcap m_2, v) \overset{\text{cross}}{\Longleftrightarrow} \tau(m_2, v)$$

For many types, such as products and functions, establishing their mode-crossing behavior in the semantic model is straightforward. Proving the correct mode-crossing rules for mutable references and recursive types required some innovation; see the supplementary material for details [23].

Interestingly, the proof that the mode-crossing behavior of a recursive type corresponds to a least fixpoint is almost trivial: it follows directly by induction on the fixpoint used to define the semantic model of recursive types.

4 Inference

This section describes the part of the compiler that infers and checks the modal kinds of type expressions and decides sub-kinding obligations. It does not cover ordinary type or mode inference; those phases use the resulting mode-crossing information, but are separate from the kind inference problem studied here. The supplementary material gives the formal reduction and solver details [23]. We cover non-recursive coefficients, abstract-kind constraints, recursive fixpoint equations, and solver architecture.

Inference is coefficient-driven: sub-kinding obligations and recursive kind equations reduce to coefficient constraints. These coefficients are then treated symbolically for normalization and solving. See Fig. 5 for an overview.

4.1 Computing the Kind of a Type Expression

This subsection sets up the representation used by the rest of the algorithm: the modal kind of every type expression is interpreted as a lattice expression and then normalized to coefficient form. The later subsections all use this same representation of modal kinds.

Recall the kinds in Fig. 3. In general, the "standard form" of the kind of a type constructor `t` is:

```
('a, 'b) t : value mod t.0 with 'a @@ t.1 with 'b @@ t.2
```

where the "kind coefficients" `t.0, t.1, t.2` range over the sub-kinding lattice:

- `t.0` is the base contribution of `('a, 'b) t`. It records which modes are crossed by `t` itself, independent of its arguments. For example, `list` crosses portability and contention, while `ref` crosses only portability.
- `t.1, t.2` describe how arguments `'a, 'b` affect the mode-crossing behavior of `('a, 'b) t`. For example, `'a → 'b` ignores its arguments entirely, while `'a @@ m` weakens `'a`'s restrictions by `m`.

This yields the following kind coefficients for the base type constructors, where $\bot$ is `portable contended` and $\top$ includes no modes:

`int.0` $= \bot$		
`list.0` $= \bot$	`list.1` $= \top$	
`ref.0` $=$ `portable`	`ref.1` $= \top$	
`(*).0` $= \bot$	`(*).1` $= \top$	`(*).2` $= \top$
`(→).0` $=$ `contended`	`(→).1` $= \bot$	`(→).2` $= \bot$
`(@@ m).0` $= \bot$	`(@@ m).1` $= m$	

Here we have used that a plain `with 'a` bound is equivalent to `with 'a @@` $\top$, and `with 'a @@` $\bot$ is equivalent to no with-bound at all. Semantically, read a surface-syntax kind as a lattice expression: interpret `with` as join (combine restrictions) and `@@` as meet (weaken restrictions):

`('a,'b) t : value mod t.0 with 'a @@ t.1 with 'b @@ t.2`

means

$$\mathsf{kind}(\texttt{('a, 'b) t}) = \texttt{t.0} \sqcup (\texttt{t.1} \sqcap \mathsf{kind}(\texttt{'a})) \sqcup (\texttt{t.2} \sqcap \mathsf{kind}(\texttt{'b}))$$

Computing the kind of a compound type expression is straightforward: apply the rule above repeatedly. For example, for `(int list) ref`, we have:

$$\begin{aligned}\mathsf{kind}(\texttt{(int list) ref}) &= \texttt{ref.0} \sqcup (\texttt{ref.1} \sqcap \mathsf{kind}(\texttt{int list})) \\ &= \texttt{ref.0} \sqcup (\texttt{ref.1} \sqcap (\texttt{list.0} \sqcup (\texttt{list.1} \sqcap \texttt{int.0}))) \\ &= \texttt{portable} \sqcup (\top \sqcap (\bot \sqcup (\top \sqcap \bot))) \\ &= \texttt{portable}\end{aligned}$$

This gives a systematic way to compute the kind of any type expression.

4.2 Non-recursive Type Definitions

For non-recursive definitions, inference computes the kind of the right-hand side, normalizes it to coefficient form, and stores the resulting coefficients for later uses of the constructor.

For a non-recursive type definition

```
type t = (right-hand side)
```

we store the kind of `t` by setting `t.0` $= \mathsf{kind}($`right-hand side`$)$. For type constructor definitions:

```
type ('a,'b) t = (right-hand side mentioning 'a and 'b)
```

we compute the right-hand-side kind as a *lattice expression*. Because the right-hand side can mention `'a` and `'b`, the resulting lattice expression need not be a ground lattice element; it may contain $\mathsf{kind}(\texttt{'a})$ and $\mathsf{kind}(\texttt{'b})$ as subexpressions. We simplify it to coefficient form

$$\mathsf{kind}(\texttt{right-hand side mentioning 'a and 'b}) = A \sqcup (B \sqcap \mathsf{kind}(\texttt{'a})) \sqcup (C \sqcap \mathsf{kind}(\texttt{'b}))$$

then extract `t.0` $:= A$, `t.1` $:= B$, `t.2` $:= C$. (Note how this coefficient form never has a term containing $\mathsf{kind}(\texttt{'a}) \sqcap \mathsf{kind}(\texttt{'b})$ because no type has this kind.) For example:

```
type ('a,'b) t = ('a * ('b @@ portable)) ref
```

Here we have

$$\mathsf{kind}(\texttt{('a * ('b @@ portable)) ref}) = \texttt{portable} \sqcup (\top \sqcap \mathsf{kind}(\texttt{'a})) \sqcup (\texttt{portable} \sqcap \mathsf{kind}(\texttt{'b}))$$

and extract the kind coefficients for `t` from this formula:

$$\texttt{t.0} := \textbf{portable}, \quad \texttt{t.1} := \top, \quad \texttt{t.2} := \textbf{portable}$$

Later uses of `t` reuse these coefficients directly.

4.3 Abstract Types

Abstract type declarations introduce the need for a solver for sub-kinding checks. The reason is that unknown coefficients from abstract type declarations and with-bound annotations produce hypotheses in the context; the algorithm must eliminate those hypotheses and reduce each check to comparison of normalized lattice polynomials.

With abstract types in scope, coefficients need not be ground constants; they become lattice polynomials over abstract kind symbols. For an abstract unary type constructor `u`, we use unknown coefficients `u.0`, `u.1` and write

$$\mathsf{kind}(\texttt{'a u}) = \texttt{u.0} \sqcup (\texttt{u.1} \sqcap \mathsf{kind}(\texttt{'a})).$$

If present, a kind annotation constrains these unknowns. First consider no annotation:

```
type elt
type 'a u
type 'a t = ('a u * elt) ref
```

Normalizing the right-hand side gives

$$\mathsf{kind}(\texttt{('a u * elt) ref}) = \textbf{portable} \sqcup \texttt{u.0} \sqcup \texttt{elt.0} \sqcup (\texttt{u.1} \sqcap \mathsf{kind}(\texttt{'a})),$$

so we extract

$$\texttt{t.0} := \textbf{portable} \sqcup \texttt{u.0} \sqcup \texttt{elt.0}, \qquad \texttt{t.1} := \texttt{u.1}.$$

As before, later uses of `t` consult these coefficients directly; the difference is that coefficients are now symbolic *lattice polynomials*.

Sub-kind checks with abstract types. A sub-kinding check $\Delta \vDash \phi \le \psi$ quantifies over all abstract-type instantiations in Δ. We reduce it to equality via

$$\phi \le \psi \iff \phi \sqcup \psi = \psi$$

then decide equality by reducing both sides to *minimal normal form*.

Minimal normal form. To introduce minimal normal form, consider checking that the kind of `'a u u` equals the kind of `'a u`:

$$\begin{aligned}
\mathsf{kind}(\texttt{'a u}) &= \texttt{u.0} \sqcup (\texttt{u.1} \sqcap \mathsf{kind}(\texttt{'a})) \\
\mathsf{kind}(\texttt{'a u u}) &= \texttt{u.0} \sqcup (\texttt{u.1} \sqcap \mathsf{kind}(\texttt{'a u})) \\
&= \texttt{u.0} \sqcup (\texttt{u.1} \sqcap \texttt{u.0}) \sqcup (\texttt{u.1} \sqcap \mathsf{kind}(\texttt{'a})) \\
&= \texttt{u.0} \sqcup (\texttt{u.1} \sqcap \mathsf{kind}(\texttt{'a}))
\end{aligned}$$

These kinds normalize to the same formula, so they are equal. Here is how we normalize in general:

- First, we distribute meets over joins, use idempotence, and combine like terms to obtain a formula in *polynomial form*, which looks like this for two abstract kind symbols x and y:

  $$p(x, y) = c_0 \sqcup (c_1 \sqcap x) \sqcup (c_2 \sqcap y) \sqcup (c_3 \sqcap x \sqcap y)$$

  where c_0, c_1, c_2, c_3 are constants and x, y are abstract kind symbols. ("polynomial" by analogy with ordinary polynomials over numbers, where + is join and · is meet)

- Second, we reduce the polynomial to minimal form by subtracting coefficients of all smaller terms from coefficients of larger terms, which gives:

$$p(x, y) = c_0 \sqcup ((c_1 \backslash c_0) \sqcap x) \sqcup ((c_2 \backslash c_0) \sqcap y) \sqcup ((c_3 \backslash (c_0 \sqcup c_1 \sqcup c_2)) \sqcap x \sqcap y)$$

  where $\backslash$ is the co-Heyting subtraction operation on the base lattice.

The second step is essential: distribution alone does not yield a canonical form. For example:

$$x \sqcup (y \sqcap x) = x$$
$$(\texttt{portable} \sqcap x) \sqcup (y \sqcap x) = (\texttt{portable} \sqcap x) \sqcup (\texttt{contended} \sqcap y \sqcap x)$$

That is, we choose the minimal value for coefficients of the polynomial. The next example illustrates this concretely.

Example. Consider the following type definitions:

```
type 'a x
type 'a y
type 'a f = 'a x y * int x
type 'a g = 'a y x * int y
```

After computing kind coefficients, we obtain:

$$\texttt{f.0} = \underbrace{\texttt{y.0} \sqcup (\texttt{y.1} \sqcap \texttt{x.0})}_{\text{from 'a x y}} \sqcup \underbrace{\texttt{x.0}}_{\text{from int x}} \qquad \texttt{g.0} = \underbrace{\texttt{x.0} \sqcup (\texttt{x.1} \sqcap \texttt{y.0})}_{\text{from 'a y x}} \sqcup \underbrace{\texttt{y.0}}_{\text{from int y}}$$
$$\texttt{f.1} = \underbrace{\texttt{x.1} \sqcap \texttt{y.1}}_{\text{from 'a x y}} \qquad \texttt{g.1} = \underbrace{\texttt{x.1} \sqcap \texttt{y.1}}_{\text{from 'a y x}}$$

The linear coefficients already match $\texttt{f.1} = \texttt{g.1} = \texttt{x.1} \sqcap \texttt{y.1}$. Absorption ($a \sqcup (b \sqcap a) = a$) simplifies only the base coefficients, yielding:

$$\texttt{f.0} = \texttt{x.0} \sqcup \texttt{y.0} \qquad \texttt{g.0} = \texttt{x.0} \sqcup \texttt{y.0}$$

Thus, these two types have the same kind. Note that if we instead had `type 'a f = 'a x y` and `type 'a g = 'a y x`, we would get different kinds:

$$\texttt{f.0} = \texttt{y.0} \sqcup (\texttt{y.1} \sqcap \texttt{x.0}) \qquad \texttt{g.0} = \texttt{x.0} \sqcup (\texttt{x.1} \sqcap \texttt{y.0})$$
$$\texttt{f.1} = \texttt{x.1} \sqcap \texttt{y.1} \qquad \texttt{g.1} = \texttt{x.1} \sqcap \texttt{y.1}$$

`f.0` and `g.0` are genuinely different; unlike above, there are no absorbed terms. This example used absorption in an ad-hoc way; the following lemma gives the general normalization rule.

Lemma 7 (Minimal normal form). *For every polynomial $p(x_1, \ldots, x_n)$, define*

$$\mathrm{MNF}(p) \;=\; \bigsqcup_{S \subseteq \{1,\ldots,n\}} \Big(p(\chi_S) \setminus \bigsqcup_{T \subset S} p(\chi_T) \Big) \sqcap \prod_{i \in S} x_i .$$

Here χ_S is the valuation with $\chi_S(x_i) = \top$ if $i \in S$, and $\chi_S(x_i) = \bot$ otherwise (with $\prod_{i \in \varnothing} x_i = \top$). A valuation ρ maps each variable x_i to an element of the base lattice. Then $\forall \rho.\ \mathrm{MNF}(p)(\rho) = p(\rho)$, and for all polynomials p, q:

$$\mathrm{MNF}(p) \text{ and } \mathrm{MNF}(q) \text{ are syntactically equal} \iff \forall \rho.\ p(\rho) = q(\rho).$$

The supplementary material contains the proof [23]. The subset-based formulation of MNF is exponential in arity; in practice we use LDD-based normalization and solver operations, which are described in the supplementary material [23].

With-bounds and abstract annotations. With-bounds that appear in kind annotations of abstract types are compiled into solver hypotheses. For

```
type ('a1,...,'an) u : value mod m0 with 'a1 @@ m_1 ... with 'an @@ m_n
```

we keep `u.0`, `u.1`, …, `u.n` symbolic and register

$$\forall a_1, \ldots, a_n.\ \mathtt{u.0} \sqcup (\mathtt{u.1} \sqcap a_1) \sqcup \cdots \sqcup (\mathtt{u.n} \sqcap a_n) \\ \leq \mathtt{m0} \sqcup (\mathtt{m\_1} \sqcap a_1) \sqcup \cdots \sqcup (\mathtt{m\_n} \sqcap a_n).$$

This is reduced using the following lemma.

Lemma 8 (Quantifier elimination for with-bounds). *The quantified inequality*

$$\forall a_1, \ldots, a_n.\ \mathtt{u.0} \sqcup (\mathtt{u.1} \sqcap a_1) \sqcup \cdots \sqcup (\mathtt{u.n} \sqcap a_n) \leq \mathtt{m0} \sqcup (\mathtt{m\_1} \sqcap a_1) \sqcup \cdots \sqcup (\mathtt{m\_n} \sqcap a_n)$$

is equivalent to the following set of quantifier-free inequalities on the `u.i` *symbols:*

$$\mathtt{u.0} \leq \mathtt{m0} \qquad \textit{and} \qquad \mathtt{u.i} \leq \mathtt{m\_i} \sqcup \mathtt{m0} \ \ (1 \leq i \leq n).$$

Lemma 9 (Bounded-quantifier substitution). *For predicate P on lattice elements and lattice-polynomial bound $b(x)$,*

$$\forall x.\ (x \leq b(x) \Rightarrow P(x)) \quad \Longleftrightarrow \quad \forall x.\ P(x \sqcap b(x)).$$

Together, these lemmas reduce a judgment $\Delta \vDash \phi \leq \psi$ to $\vDash \phi' \leq \psi'$. First, quantified inequalities on lattice functions in Δ are reduced to quantifier-free inequalities on coefficient symbols. Second, bounded implications are rewritten by substitution. The resulting simple judgment is then checked by testing whether $\mathrm{MNF}(\phi' \sqcup \psi') \stackrel{?}{=} \mathrm{MNF}(\psi')$.

Example: Map functor signature inclusion. We reuse the map functor from §2.8:

```
module type OrderedType = sig
  type t
  val compare : t → t → int
end
module Map (Ord : OrderedType) = struct
  type 'a t : value mod portable contended with 'a with Ord.t = ...
end
```

Now build maps on key pairs by currying two map applications:

```
module PairMap (A : OrderedType) (B : OrderedType) : sig
  type 'v t : value mod portable contended with 'v with A.t with B.t
end = struct
  module MB = Map(B) module MA = Map(A)
  type 'v t = ('v MB.t) MA.t
end
```

We must check that the implementation kind is a sub-kind of the signature kind. In the functor body, `A.t` and `B.t` are abstract, with symbols `A.t.0`, `B.t.0`. The signature requires:

$$\mathrm{kind}(\mathtt{'v\ t})_{\mathrm{sig}} = \mathrm{kind}(\mathtt{'v}) \sqcup \mathtt{A.t.0} \sqcup \mathtt{B.t.0}.$$

The implementation defines `'v t = ('v MB.t) MA.t`, so:

$$\mathrm{kind}(\mathtt{'v\ t})_{\mathrm{impl}} = \mathtt{MA.t.0} \sqcup (\mathtt{MA.t.1} \sqcap (\mathtt{MB.t.0} \sqcup (\mathtt{MB.t.1} \sqcap \mathrm{kind}(\mathtt{'v}))))$$

By the definition of the `Map` functor, we have:

$$\mathtt{MA.t.0} = \mathtt{A.t.0}, \qquad \mathtt{MA.t.1} = \top, \qquad \mathtt{MB.t.0} = \mathtt{B.t.0}, \qquad \mathtt{MB.t.1} = \top.$$

So the implementation kind simplifies to $\mathsf{kind}(\texttt{'v t})_{\text{impl}} = \texttt{A.t.0} \sqcup \texttt{B.t.0} \sqcup \mathsf{kind}(\texttt{'v})$ This is precisely as required by the signature.

Map functor inclusion with an abstract signature. Now consider the same construction as above, but where `Map` is abstracted behind a signature:

```
module Map (Ord : OrderedType) : sig
  type 'a t : value mod portable contended with 'a with Ord.t
end = struct ... end
module PairMapViaSig (A : OrderedType) (B : OrderedType) : sig
  type 'v t : value mod portable contended with 'v with A.t with B.t
end = struct
  module MB = Map(B) module MA = Map(A)
  type 'v t = ('v MB.t) MA.t
end
```

This is identical to the previous `PairMap` example, except that `Map` is abstract, so we only know its behavior through its signature. For `MA = Map(A)` and `MB = Map(B)`, this yields quantified hypotheses:

$$\forall x.\ \texttt{MA.t.0} \sqcup (\texttt{MA.t.1} \sqcap x) \leq \texttt{A.t.0} \sqcup x \qquad \forall x.\ \texttt{MB.t.0} \sqcup (\texttt{MB.t.1} \sqcap x) \leq \texttt{B.t.0} \sqcup x.$$

Applying Lemma 8 to each yields coefficient bounds:

$$\texttt{MA.t.0} \leq \texttt{A.t.0}, \quad \texttt{MA.t.1} \leq \top, \quad \texttt{MB.t.0} \leq \texttt{B.t.0}, \quad \texttt{MB.t.1} \leq \top.$$

The implementation and signature kinds are

$$\mathsf{kind}(\texttt{'v t})_{\text{impl}} = \texttt{MA.t.0} \sqcup (\texttt{MA.t.1} \sqcap (\texttt{MB.t.0} \sqcup (\texttt{MB.t.1} \sqcap \mathsf{kind}(\texttt{'v})))),$$
$$\mathsf{kind}(\texttt{'v t})_{\text{sig}} = \mathsf{kind}(\texttt{'v}) \sqcup \texttt{A.t.0} \sqcup \texttt{B.t.0}.$$

So inclusion reduces to the following universally quantified implication (omitting trivial $\top$-bounds):

$$(\texttt{MA.t.0} \leq \texttt{A.t.0}) \wedge (\texttt{MB.t.0} \leq \texttt{B.t.0}) \Longrightarrow \mathsf{kind}(\texttt{'v t})_{\text{impl}} \leq \mathsf{kind}(\texttt{'v t})_{\text{sig}}$$

where the free coefficient symbols (and $\mathsf{kind}(\texttt{'v})$) are universally quantified. Using Lemma 9 under this universal closure, we substitute bounded coefficients in the conclusion, reducing to

$$(\texttt{MA.t.0} \sqcap \texttt{A.t.0}) \sqcup (\texttt{MA.t.1} \sqcap ((\texttt{MB.t.0} \sqcap \texttt{B.t.0}) \sqcup (\texttt{MB.t.1} \sqcap \mathsf{kind}(\texttt{'v})))) \leq$$
$$\mathsf{kind}(\texttt{'v}) \sqcup \texttt{A.t.0} \sqcup \texttt{B.t.0},$$

which is then discharged by MNF equality.

4.4 Recursive Types

Recursive definitions require a least-fixpoint calculation. The solver computes the right-hand side in coefficient form with recursive occurrences left symbolic, reduces the resulting fixpoint to equations on coefficients, and eliminates those equations one coefficient at a time.

For a recursive type constructor

```
type ('a1,...,'an) t = (right-hand side mentioning 'a1,...,'an and t)
```

we write its unknown kind in coefficient form as

$$\mathsf{kind}(\texttt{('a1,...,'an) t}) = \texttt{t.0} \sqcup (\texttt{t.1} \sqcap a_1) \sqcup \cdots \sqcup (\texttt{t.n} \sqcap a_n)$$

Here a_i abbreviates the kind of the i-th type parameter. Unlike the non-recursive case, the right-hand side may mention `t` itself. So, after computing $\mathsf{kind}(\texttt{right-hand side})$ compositionally with `t` treated abstractly, every recursive occurrence $\mathsf{kind}(t(\phi_1, \ldots, \phi_n))$ is expanded as

$$\texttt{t.0} \sqcup (\texttt{t.1} \sqcap \mathsf{kind}(\phi_1)) \sqcup \cdots \sqcup (\texttt{t.n} \sqcap \mathsf{kind}(\phi_n)).$$

After substitution and normalization, we collect the base part and the a_i-parts; $R_0(\texttt{t.0}, \ldots, \texttt{t.n})$ is the resulting base coefficient, and $R_i(\texttt{t.0}, \ldots, \texttt{t.n})$ is the coefficient of a_i. Thus

$$\mathsf{kind}(\texttt{right-hand side}) = R_0(\texttt{t.0}, \ldots, \texttt{t.n}) \sqcup (R_1(\texttt{t.0}, \ldots, \texttt{t.n}) \sqcap a_1) \sqcup \cdots \sqcup (R_n(\texttt{t.0}, \ldots, \texttt{t.n}) \sqcap a_n)$$

To make this equal to the left-hand side, we need to solve the system of equations:

$$\texttt{t.0} = R_0(\texttt{t.0}, \ldots, \texttt{t.n}) \quad \texttt{t.1} = R_1(\texttt{t.0}, \ldots, \texttt{t.n}) \quad \cdots \quad \texttt{t.n} = R_n(\texttt{t.0}, \ldots, \texttt{t.n})$$

In fact, we want the *least* solution to this system of equations.

Lemma 10 (Coefficientwise reduction of recursive kinds). *For a recursive constructor* `t` *with right-hand side matching coefficient form* R_i*, the lattice function fixpoint is equivalent to finding the least solution of the following system of coefficient equations:*

$$\texttt{t.i} = R_i(\texttt{t.0}, \ldots, \texttt{t.n}) \qquad (0 \le i \le n).$$

The supplementary material validates this coefficientwise reduction [23].

To find the least solution, we use that the $R_i(\texttt{t.0}, \ldots, \texttt{t.n})$ are *lattice polynomials* in the unknowns $\texttt{t.0}, \ldots, \texttt{t.n}$, meaning expressions involving only meets, joins, constants, and unknowns.[7] This means that as a function of `t.0`, R_0 can be written as:

$$R_0(\texttt{t.0}, \texttt{t.1}, \ldots, \texttt{t.n}) = A(\texttt{t.1}, \ldots, \texttt{t.n}) \sqcup (B(\texttt{t.1}, \ldots, \texttt{t.n}) \sqcap \texttt{t.0})$$

for some A and B involving only $\texttt{t.1}, \ldots, \texttt{t.n}$. Now note that $A(\texttt{t.1}, \ldots, \texttt{t.n})$ is a fixpoint of R_0 with respect to `t.0`, because:

$$\begin{aligned} R_0(A(\texttt{t.1}, \ldots, \texttt{t.n}), \texttt{t.1}, \ldots, \texttt{t.n}) &= A(\texttt{t.1}, \ldots, \texttt{t.n}) \sqcup (B(\texttt{t.1}, \ldots, \texttt{t.n}) \sqcap A(\texttt{t.1}, \ldots, \texttt{t.n})) \\ &= A(\texttt{t.1}, \ldots, \texttt{t.n}) \end{aligned}$$

This is also the *least* fixpoint: if $A'(\texttt{t.1}, \ldots, \texttt{t.n})$ is another fixpoint, then:

$$\begin{aligned} A'(\texttt{t.1}, \ldots, \texttt{t.n}) &= R_0(A'(\texttt{t.1}, \ldots, \texttt{t.n}), \texttt{t.1}, \ldots, \texttt{t.n}) \\ &= A(\texttt{t.1}, \ldots, \texttt{t.n}) \sqcup (B(\texttt{t.1}, \ldots, \texttt{t.n}) \sqcap A'(\texttt{t.1}, \ldots, \texttt{t.n})) \\ &\ge A(\texttt{t.1}, \ldots, \texttt{t.n}) \end{aligned}$$

Because these equations are monotone over a complete lattice, Bekić-style decomposition applies: having found the least fixpoint of the first equation with respect to `t.0` in terms of the other unknowns, we substitute the solution into the remaining equations and repeat this process for the other unknowns, thus finding the least solution to the system.

Lemma 11 (One-coefficient elimination). *The least fixpoint of* $x = A \sqcup (B \sqcap x)$ *is* $x = A$.

Example: simple recursive type. Consider the type definition:

```
type rlist = Nil | Cons of int ref * rlist
```

There is one coefficient, `rlist.0`. Computing the right-hand side kind gives

$$\mathsf{kind}(\texttt{int ref * rlist}) = \textbf{portable} \sqcup \texttt{rlist.0},$$

so the fixpoint equation is `rlist.0` = **portable** $\sqcup$ `rlist.0`. Its least solution is `rlist.0` = **portable**.

[7] The constants here may be formulas involving abstract kind symbols in scope, but they are constant w.r.t. the `t.is`.

Example: non-uniform parameterized recursion. Consider the following non-uniform recursive type from §2.6:

```
type 'a nref = Nil of 'a | Cons of 'a ref nref
```

The recursive occurrence is non-uniform because it appears at `'a ref`. Write

$$\mathsf{kind}(\texttt{'a nref}) = \texttt{nref.0} \sqcup (\texttt{nref.1} \sqcap \mathsf{kind}(\texttt{'a})).$$

Computing the right-hand side kind while keeping `nref` abstract yields

$$\mathsf{kind}(\texttt{'a}) \sqcup \mathsf{kind}(\texttt{'a ref nref}) = \texttt{nref.0} \sqcup (\texttt{nref.1} \sqcap \textbf{portable}) \sqcup ((\top \sqcup \texttt{nref.1}) \sqcap \mathsf{kind}(\texttt{'a})).$$

Matching coefficients gives the system

$$\texttt{nref.0} = \texttt{nref.0} \sqcup (\texttt{nref.1} \sqcap \textbf{portable}) \qquad \texttt{nref.1} = \top \sqcup \texttt{nref.1}.$$

The least solution is $\texttt{nref.1} = \top$ and $\texttt{nref.0} = \textbf{portable}$, so `'a nref` gets kind **value mod portable with** `'a`.

4.5 Implementation

The preceding subsections reduce modal-kind inference and sub-kinding to operations on abstract lattice polynomials. To implement this efficiently, our implementation uses a new *LDD* (lattice decision diagram) datatype, a variation on BDD-style decision diagrams for lattice formulas rather than Boolean formulas [3, 21]. This shared DAG representation supports canonicalization and memoization of the normalization and solving steps used by inference. Details of the reduction, correctness-oriented formulation, and concrete algorithmic pseudocode for the LDD operations are given in the supplementary material [23].

From OxCaml inference to SMoKi sub-kinding. We do not relate this section's algorithm to SMoKi's semantic sub-kinding judgement $\Delta \vDash \phi \leq \psi$ directly. This is intentional: this algorithm implicitly operates on a representation of types closer to the internal one in OxCaml's compiler, as opposed to the idealized version in §3. We can, however, assemble the results from this section into a sound and complete decision procedure for $\Delta \vDash \phi \leq \psi$, as described in Appendix B.

5 Mode Crossing in Practice

Mode crossing and modal kinds are essential for practical use of OxCaml in Jane Street's codebase of 80 million lines of code. Uptake has been fast: as of February 2026, the codebase had roughly 6,000 modal kind annotations, though the feature has been available (in a preliminary, less principled implementation) only since early 2025. These annotations were not added to decorate the code; they were added only when needed to fix concrete type errors. Furthermore, our colleagues urgently wanted a more expressive kind system than the initial versions, holding up important (closed-source) features until the type system was ready.

Here we explore examples inspired by real code where these annotations are necessary.

5.1 Regular Expressions

Consider a regular-expression library with abstract compiled-regex type `t`. A simplified API is:

```
type t : value mod portable contended
val create : string → t Or_error.t
val find_all : t → string → string list Or_error.t
val matches : t → string → bool
```

Compiled regexes are immutable, so cross-thread use is safe; mode crossing exposes this without API noise. Without mode crossing, the signatures would need explicit mode decorations:

```
val create : string → t Or_error.t @ portable
val find_all : t @ contended → string → string list Or_error.t
val matches : t @ contended → string → bool
```

Without these decorations, mode errors would arise when capturing a compiled regular expression in a **portable** function. Many other functions in this API and many similarly shaped libraries (defining an abstract type with no functions or mutable state) would also need annotations, along with any polymorphic functions that might be instantiated with these types. The annotation burden would simply be impractical in a large codebase. Instead, mode crossing allows us to avoid this work, enabling parallel programming with a lower annotation burden.

5.2 Finite Maps

Recall the functorial `Map` example from §2.8. There, we aimed to ensure that immutable maps over immutable values should cross portability, and we achieved that by requiring the `compare` function used by the map (carried in `OrderedType`) to be **portable**. This is appropriate for maps meant to cross portability, but when we interact with legacy OCaml code, comparison functions might only be available at **nonportable** mode, in which case the corresponding maps should *not* cross portability. One way to accommodate such maps would be to require them to be constructed using a different `Map` functor, but that would result in completely different types of **portable** and **nonportable** maps. Here, we discuss an alternative, unifying interface for maps, deployed at Jane Street, by which the distinction between **portable** and **nonportable** maps can be encoded via a *phantom type* parameter, thus providing an interesting showcase for how modal kinds enable the expression of novel APIs.

The key idea of this API is to piggyback on a solution to a different problem, namely how to differentiate between maps that share the same key type but use different comparison functions. It is important that such maps *not* be treated as interoperable since the internal representation invariant of the map depends crucially on the comparison function—*e.g.*, merging maps built from different comparison functions would result in a broken invariant. Rossberg et al. [24] suggest a technique for how to enforce this dependency soundly and systematically by introducing phantom type declarations as static representatives of all value declarations, and then using a variant of OCaml's applicative functor types to effectively parameterize the map type by the static representative of its corresponding comparison function. Our API achieves the same effect in a less systematic but more lightweight manner, via explicit type parameterization rather than applicative functors.

Figure 6 shows an excerpt of the API. First, observe that we index the type of maps not only by the types of keys and values that they operate on (as usual) but also by a *comparator witness* type `'cmp`: This `'cmp` type is a *phantom type* parameter: it is not associated with runtime data, but merely serves as a static representative of the comparison function on which the map type depends. (We will return to the with-bounds on this type declaration in a moment.)

Comparator witness phantom types, in turn, are generated by the `make` functions in the `Comparator` module (Figure 6). These functions return first-class modules [10], so that each returned module generates a fresh abstract `comparator_witness` type. Because this type is abstract, clients are forced to distinguish the witnesses from different comparators.

Having explained how we encode the dependency of map types on their comparison functions, let us now return to the original problem: how can we ensure that map types using **portable** comparison functions cross portability whereas map types using **nonportable** functions do not? .

The trick is to employ *two* `make` functions, one for **portable** comparators and another for **nonportable** comparators. In the former case the freshly generated abstract `comparator_witness` type is declared to cross portability, whereas in the latter case it is not. Then, thanks to the with-bounds on `Comparator.t` and `Map.t`, the mode-crossing behavior of `comparator_witness` determines

```
module Map : sig
  type ('key, 'val, 'cmp) t : value mod portable contended
      with 'key with 'val with ('key, 'cmp) Comparator.t
end
module Comparator : sig
  type ('a, 'cmp) t : value mod portable contended with 'cmp @@ contended
  module type S = sig
    type element     type comparator_witness
    val comparator : (element, comparator_witness) t
  end
  val make : ('a → 'a → int) → (module S with type element = 'a)
  module type S_portable = sig
    type element     type comparator_witness : value mod portable
    val comparator : (element, comparator_witness) t
  end
  val make_portable : ('a → 'a → int) @ portable → (module S_portable with type element = 'a)
end
```

Fig. 6. Excerpt of the phantom type-based `Map` and `Comparator` APIs.

that of `('key, comparator_witness) Comparator.t` and `('key, 'val, comparator_witness) Map.t`, ensuring that the map type crosses portability only if the comparison function did.

The implementation of `Comparator.t` itself is simple:

```
type ('a, 'cmp) t : value mod portable contended with 'cmp @@ contended =
  { compare : 'a → 'a → int } [@@unsafe_allow_any_mode_crossing]
```

The payload is just the comparison function. However, as is common with phantom type APIs, the type checker cannot verify the associated invariants: it has no way of knowing that (thanks to module-level data abstraction) the function is guaranteed to be **`portable`** if the phantom `'cmp` parameter crosses portability. Hence, we use the unsafe annotation `[@@unsafe_allow_any_mode_crossing]` to work around the type system. (See Georges et al. [12] for another example of a modal, phantom type-based API, whose implementation also relies on unsafe code.) In summary, this example shows a more advanced application for modal kinds in a real API; without them we would need either modality-parameterized types or duplicated portable/nonportable variants of the map type.

5.3 Cost of Mode Crossing

The examples in this paper illustrate why mode crossing is useful, but the compiler must now compute crossing information while type checking. The cost of computing modal kinds for type definitions is small, but the Cross typing rule (Fig. 4) is potentially expensive: the compiler repeatedly queries a type's mode-crossing behavior when propagating modes through values, patterns, applications, field accesses, subtyping, and signature inclusion.

We tested the cost of mode crossing in our implementation on plain OCaml files from the standard library and the compiler implementation. They do not need modes to type check, but the OxCaml compiler still runs the full mode-crossing machinery on them because mode information is used in later optimization passes. To reduce measurement noise, we added a compiler flag to repeat mode-crossing checks multiple times, and normalized the numbers afterwards. The estimates may slightly understate ordinary compilation cost because repeating the same checks can improve cache and branch-predictor behavior. Even so, on the files we tested, the approximate cost of mode crossing stays below 1% of total native compile time, and about 0–6% of the type-checking phase. The full measurements are shown in Appendix A.

6 Related Work

Modal type systems for safe concurrency in OCaml. OxCaml's mode system, formalized in DRFCaml, establishes memory safety and data-race freedom by enforcing safety restrictions through modes [12, 13, 18]. Our work extends this line of work: we do not change the underlying safety story, but add *mode crossing* and a kind system that relaxes certain modes based on value types. (Specifically, SMoKi is an extension of DRFCaml.) This addresses a common source of false negatives: client code need not satisfy modal constraints when the type's representation and operations make those axes semantically irrelevant. From a technical standpoint, SMoKi builds on the DRFCaml logical relation by extending it with mode crossing, iso-recursive types, and the `shareable` mode.

Marker traits and protocols (Rust/Swift). The closest conceptual analogues to modal kinds are *marker traits* such as Rust's `Send` and `Sync` and Swift's `Sendable`, classifying types as safe for transfer or sharing across threads [28, 29]. Both languages structurally analyze types to automatically determine whether they satisfy these traits. For instance, Rust can use coinduction to infer whether a (uniform) recursive type implements `Sync` [27]. We also infer type properties, but our setting is more complex because of abstract types (from ML-style modules) and non-uniform recursive type constructors. In particular, the combination of recursion and abstraction forces us to reason about least fixpoints in the presence of unknown (abstract) kind functions.

Many languages also have "deriving" mechanisms (Haskell's `deriving`, Rust's derive macros, OCaml's `ppx_compare`) to derive properties or implementations from type structure. These mechanisms often handle recursive types and, in the case of Haskell, even non-uniform recursive types. However, these mechanisms cannot express interactions with abstract type constructors like we can using with-bounds. Simultaneously, our use of kind functions over lattices helps us solve these more complicated constraints.

Qualified types and type classes. Type qualifiers are a classic framework for tracking type properties, often with constraint-based inference and extensible checking frameworks [9, 22]. Modes and modal kinds resemble constraint-based polymorphism (*qualified types*) and type classes: they record obligations that must hold when instantiating type arguments. Our setting differs: our constraints range over a lattice of mode-crossing behaviors and are discharged by lattice-specific entailment rather than instance resolution, and they interact well with non-uniform recursion and module abstraction, as described above. More broadly, similar phenomena appear in nullable and non-nullable types, universe hierarchies (e.g., *Prop* vs. *Type*), and linear/non-linear modalities (e.g., session-typed languages such as GV/Par), though with different semantics than mode crossing.

Boolean and lattice-valued constraint solving for type inference. Our inference procedure reduces kind checking and subsumption to constraints over a distributive lattice, represented as lattice polynomials and solved via a dedicated solver (§ 4). This connects to prior work relating effect inference to Boolean constraints and Boolean unification [19], and more broadly to type parameters over lattices in Boolean-kinded type systems [30]. Both extend HM-style type systems with algebraic type arguments from *Boolean lattices*. They rely on results in Boolean unification to compute *most general unifiers* on *Boolean formulas*. Our work goes in a different direction: We consider lattice-based properties of types, and not lattice *arguments* to types. Additionally, it is unclear to us whether the unification approach generalizes to Bi-Heyting lattices.

Decision diagrams. Our LDD representation (§ 4) is analogous to Boolean decision diagrams (BDDs/ZDDs) [3, 21]: a shared canonical DAG, but for lattice polynomials. Our contribution is a solver architecture specialized to modal-kind algebra: co-Heyting-normalized lattice polynomials plus a disciplined fragment of monotone linear functions and fixpoints, sufficient in practice.

Table 1. Mode-crossing cost on selected OxCaml files. TC abbreviates type checking.

File	Total (s)	TC (s)	TC/total	Cross (ms)	Cross/TC	Cross/total
Compiler files						
`utils/misc.ml`	0.735	0.175	23.85%	2.7	1.53%	0.36%
`parsing/ast_mapper.ml`	0.625	0.120	19.22%	3.4	2.83%	0.54%
`lambda/lambda.ml`	0.765	0.159	20.72%	5.3	3.34%	0.69%
`lambda/simplif.ml`	0.619	0.097	15.74%	3.4	3.51%	0.55%
`lambda/translcore.ml`	1.293	0.169	13.09%	7.3	4.34%	0.57%
`lambda/matching.ml`	1.628	0.278	17.10%	11.5	4.13%	0.71%
`driver/compile_common.ml`	0.161	0.042	25.86%	0.3	0.83%	0.21%
`typing/typecore.ml`	4.531	0.645	14.23%	35.4	5.50%	0.78%
`typing/ctype.ml`	2.456	0.466	18.97%	23.2	4.99%	0.95%
Standard library files						
`stdlib/list.ml`	0.197	0.056	28.63%	0.7	1.24%	0.35%
`stdlib/array.ml`	0.213	0.074	34.78%	0.2	0.21%	0.07%
`stdlib/map.ml`	0.321	0.088	27.45%	2.2	2.44%	0.67%
`stdlib/hashtbl.ml`	0.253	0.064	25.44%	1.6	2.49%	0.63%
`stdlib/format.ml`	0.467	0.088	18.84%	1.8	2.04%	0.38%
`stdlib/buffer.ml`	0.174	0.055	31.58%	0.4	0.75%	0.24%
`stdlib/string.ml`	0.135	0.046	33.78%	0.4	0.98%	0.33%

APPENDIX

A Mode-Crossing Benchmark Table

See Table 1 for our measurements on the cost of mode crossing as described in §5.3. We measured the compilation time of different OCaml standard library and compiler files, how much of compilation time was spend type checking, and how much of type checking was spent checking for mode crossing. These measurements are taken over three runs.

To better assess the effect of mode crossing on compilation time, we added a compiler flag that repeats mode-crossing checks `N` times. We measure total compilation time with `ocamlopt.opt -c`, type checking time with `ocamlopt.opt -i`, and estimate the time for one mode-crossing pass as $(T_{10} - T_1)/9$, where T_N is type checking time with `-mode-crossing-repetitions N`.

On the compiler files, mode crossing takes about 1–6% of overall type checking time; on the standard-library files, it is about 0–3%. In this setup, type checking itself accounts for roughly 13–35% of total native compilation time.

B A Decision Procedure for SMoKi Sub-Kinding

The inference algorithm of §4 does not directly decide the semantic sub-kinding judgment $\Delta \vDash \phi \leq \psi$ of Def. 4, for the reasons discussed in §4.5. Here we show how its results compose into a sound and complete decision procedure for $\Delta \vDash \phi \leq \psi$.

To be precise: we prove that the *overall procedure* underlying §4 is sound and complete for $\Delta \vDash \phi \leq \psi$. This procedure represents each kind by coefficients, reduces the constraints of Δ to coefficient bounds, inlines them, and decides the residual inequality by minimal-normal-form comparison. For some subroutines, though, we establish only that the subproblem is *decidable* rather than verifying the more efficient algorithm §4 uses in its place. The recursive case is the one

such spot: Lemma 12 computes second-order fixpoints by iteration over the finite function lattice, whereas §4.4 solves the same fixpoint more efficiently via a coefficient system.

Unfolding Def. 4, the judgment asks whether

$$\forall \rho \in [\![\Delta]\!].\ [\![\phi]\!]_\rho \le [\![\psi]\!]_\rho \qquad (\star)$$

The procedure represents every abstract kind function by coefficients, rewrites both the constraints that Δ places on those coefficients and the inequality itself into symbolic form, and discharges the result with the minimal-normal-form test of §4.3. Throughout, we treat each type variable α in Δ as a nullary abstract constructor t with trivial bound $\top$.

The reduction rests on a notion of *join-linearity* for lattice functions. Call a function $g : M^k \to M$ *join-linear* if it can be written in *coefficient form* $g(m_1, \ldots, m_k) = g_0 \sqcup (g_1 \sqcap m_1) \sqcup \ldots \sqcup (g_k \sqcap m_k)$ for some coefficients $g_0, \ldots, g_k$. Recall from §4.3 that a *lattice polynomial* in some variables is a function built from those variables and constants by arbitrary meets and joins. We write c for mode constants (or, equivalently, constant polynomials without variables), and Φ or Ψ for arbitrary lattice polynomials.

Lemma 12 (Coefficient form of kind terms). *Let χ be a kind term, let $\overline{\hat\alpha}$ be distinguished type variables, and let ρ be a substitution (in the sense of Def. 2) that assigns a join-linear function with constant coefficients $\overline{c}$ to every other free variable in χ:*

$$(\rho\, t)(\overline{m}) = c_0^t \sqcup \bigsqcup_i (c_i^t \sqcap m_i)$$

Then $\overline{m} \mapsto [\![\chi[\overline{\hat\alpha} \mapsto \overline{m}]]\!]_\rho$ is join-linear with coefficient form

$$[\![\chi[\overline{\hat\alpha} \mapsto \overline{m}]]\!]_\rho = \Phi_0 \sqcup \bigsqcup_i (\Phi_i \sqcap m_i),$$

and each of its coefficients Φ_i can be written as a lattice polynomial in $\overline{c}$.

Proof. By induction on χ. Most cases preserve join-linearity and the required coefficient form by distributivity. The only case that needs more than distributivity is the second-order μ-fixpoint

$$\chi = (\mu\, t\, \overline{\hat\beta}.\, \phi)\, \overline{\psi}.$$

Note that we are working in a *finite* lattice, and so that $M^n \to M$ is also finite. Hence, we can compute the second-order fixpoint

$$\overline{m} \ \mapsto\ [\![\chi[\overline{\hat\alpha} \mapsto \overline{m}]]\!]_\rho = \mu(\lambda\, f.\ \lambda\, \overline{m'}.\ [\![\phi[\overline{\hat\alpha} \mapsto \overline{m}]]\!]_{\rho, t \mapsto f, \overline{\beta \mapsto m'}})\overline{[\![\psi[\overline{\hat\alpha} \mapsto \overline{m}]]\!]_\rho}$$

by iterating from $\overline{m} \mapsto \bot$ until it stabilizes, which takes at most $|M^n \to M|$ iterations. Each step of the iteration preserves join-linearity and the required coefficient form by induction. □

For instance, for unary abstract t, u with coefficient forms $(\rho\, t)(\hat\alpha) = g_0^t \sqcup (g_1^t \sqcap \hat\alpha)$ and $(\rho\, u)(\hat\alpha) = g_0^u \sqcup (g_1^u \sqcap \hat\alpha)$, the kind term $\hat u\ (\hat t\ \hat\alpha)$ has coefficient form

$$g_0^u \sqcup (g_1^u \sqcap (g_0^t \sqcup (g_1^t \sqcap \hat\alpha))) \rightsquigarrow g_0^u \sqcup (g_1^u \sqcap g_0^t) \sqcup ((g_1^u \sqcap g_1^t) \sqcap \hat\alpha)$$

with coefficients $g_0^u \sqcup (g_1^u \sqcap g_0^t)$ and $g_1^u \sqcap g_1^t$.

We use Lemma 12 twice, once to bring an arbitrary kind function into coefficient form, and here in degenerate form to represent entire substitutions in terms of constant coefficients:

Lemma 13 (Coefficient representation of valid substitutions). *For every $\rho \in [\![\Delta]\!]$ and every $t \in \mathrm{dom}\,\Delta$, the function $\rho\, t$ is join-linear, with constant coefficients $c_0^t, \ldots, c_k^t$.*

Proof. By clause (3) of Def. 3, $(\rho\, t)(\overline{m}) = [\![\chi[\overline{\hat{\alpha}} \mapsto \overline{m}]]\!]_\varnothing$ for a kind term χ with only free variables $\overline{\hat{\alpha}}$. We apply Lemma 12 to χ with the empty substitution $\varnothing$, yielding coefficients Φ_i^t such that

$$(\rho\, t)(\overline{m}) = [\![\chi[\overline{\hat{\alpha}} \mapsto \overline{m}]]\!]_\varnothing = \Phi_0^t \sqcup \bigsqcup_i (\Phi_i^t \sqcap m_i).$$

Note that the Φ_i^t are constant coefficients c_i^t because we applied Lemma 12 to the empty substitution, and hence the Φ_i^t are polynomials in no variables. □

With these two lemmas, the reduction proceeds in four steps.

Represent substitutions by coefficients By Lemma 13, each $\rho \in [\![\Delta]\!]$ is represented by a finite family of constants $\overline{c} = (c_i^t)_{t,i}$:

$$(\rho\, t)(\overline{m}) = c_0^t \sqcup \bigsqcup_i (c_i^t \sqcap m_i) \tag{1}$$

Quantifying over $\rho \in [\![\Delta]\!]$ thus becomes quantifying over $\overline{c}$, subject to the constraints of the next step.

To make sure that the procedure described here is complete, we also need to argue that any such family of constants gives rise to a substitution $\rho \in [\![\Delta]\!]$. Clause (1) of Def. 3 is trivial, clause (2) is taken care of by the next step, and clause (3) follows because $\rho\, t$ can be represented as a kind term χ such that $(\rho\, t)(\overline{m}) = [\![\chi]\!]_\varnothing$ by following the structure in (1).

Reduce the constraints on ρ to constraints on $\overline{c}$ For each abstract t with declared bound χ_t, clause (2) of Def. 3 requires

$$\forall \overline{m}.\ (\rho\, t)(\overline{m}) \leq [\![\chi_t[\overline{\hat{\alpha}} \mapsto \overline{m}]]\!]_\rho$$

The left side has a coefficient form by the previous step, and the right side has a coefficient form by Lemma 12, with coefficients that are lattice polynomials Ψ_i^t in $\overline{c}$:

$$\forall \overline{m}.\ c_0^t \sqcup \bigsqcup_i (c_i^t \sqcap m_i) \leq \Psi_0^t \sqcup \bigsqcup_i (\Psi_i^t \sqcap m_i)$$

Comparing the two reshaped sides of the equation, Lemma 8 reduces the quantified inequality to finitely many bounds of the form $c_0^t \leq \Psi_0^t$ or $c_i^t \leq \Psi_0^t \sqcup \Psi_i^t$ (for $i \geq 1$). Applying Lemma 12 to ϕ and ψ likewise turns the conclusion $[\![\phi]\!]_\rho \leq [\![\psi]\!]_\rho$ into a lattice-polynomial inequality $\Phi \leq \Psi$ in $\overline{c}$. So putting it all together, ($\star$) is equivalent to

$$\forall \overline{c}.\ \left(\bigwedge_{t,i} c_i^t \leq \Phi_i^t\right) \implies \Phi \leq \Psi$$

for finitely many polynomials Φ_i^t derived from the Ψ_i^t.

Inline the bounds We inline the coefficient bounds iteratively: each application of Lemma 9 discharges one hypothesis $c_i^t \leq \Phi_i^t$ by substituting $c_i^t \mapsto c_i^t \sqcap \Phi_i^t$ until none remain. (Note that Lemma 9 holds even if Φ_i^t mentions c_i^t.) This leaves an unconditional inequality $\forall \overline{c}.\ \Phi' \leq \Psi'$ between lattice polynomials equivalent to ($\star$).

Decide The residual inequality $\forall \overline{c}.\ \Phi' \leq \Psi'$ is a symbolic query that can be decided by the minimal-normal-form comparison of §4.3.

By composing these different results, we obtain:

Theorem 14 (Decision procedure for SMoKi sub-kinding)**.** *For every well-formed SMoKi context Δ and kind terms ϕ, ψ, the procedure above accepts $\phi \leq \psi$ under Δ exactly when $\Delta \vDash \phi \leq \psi$ holds.*

References

[1] Hans Bekić. 1984. Definable Operations in General Algebras, and the Theory of Automata and Flowcharts. In *Programming Languages and Their Definition*. Lecture Notes in Computer Science, Vol. 177. Springer, 30–55. doi:10.1007/BFb0048939

[2] T. S. Blyth. 2005. *Lattices and Ordered Algebraic Structures*. Springer.

[3] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* C-35, 8 (1986). doi:10.1109/TC.1986.1676819

[4] Stephen Dolan. 2019. Unboxed Types for OCaml. Jane Street Tech Talk. https://www.janestreet.com/tech-talks/unboxed-types-for-ocaml/

[5] Paul Downen, Zena M. Ariola, Simon Peyton Jones, and Richard A. Eisenberg. 2020. Kinds are calling conventions. In *ICFP*. doi:10.1145/3408986

[6] Richard A. Eisenberg, Stephen Dolan, and Leo White. 2022. Unboxed types for OCaml. In *ML Workshop*. https://icfp22.sigplan.org/details/mlfamilyworkshop-2022-papers/13/Unboxed-types-for-OCaml

[7] Richard A. Eisenberg and Simon Peyton Jones. 2017. Levity polymorphism. In *PLDI*. doi:10.1145/3062341.3062357

[8] Leo Esakia, Guram Bezhanishvili, Wesley H. Holliday, and Anton Evseev. 2019. *Heyting Algebras: Duality Theory* (1st ed.). Springer.

[9] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. 1999. A Theory of Type Qualifiers. In *PLDI*. doi:10.1145/301618.301665

[10] Alain Frisch and Jacques Garrigue. 2010. First-Class Modules and Composable Signatures in Objective Caml 3.12. In *ACM SIGPLAN Workshop on ML*. Baltimore, MD.

[11] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. *PACMPL* 8, PLDI (2024). doi:10.1145/3656422

[12] 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. *PACMPL* 9, POPL (2025). doi:10.1145/3704859

[13] Jane Street. 2026. OxCaml. Project website. https://oxcaml.org/ (accessed February 2, 2026).

[14] Ralf Jung. 2020. *Understanding and evolving the Rust programming language*. Ph. D. Dissertation. Saarland University. doi:10.22028/D291-31946

[15] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (2017), 34 pages. doi:10.1145/3158154

[16] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018). https://doi.org/10.1017/S0956796818000151

[17] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. https://doi.org/10.1145/2676726.2676980

[18] Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. *PACMPL* 8, ICFP (2024). doi:10.1145/3674642

[19] Magnus Madsen and Jaco van de Pol. 2020. Polymorphic types and effects with Boolean unification. *PACMPL* 4, OOPSLA (2020). doi:10.1145/3428222

[20] Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *ESOP*. doi:10.1007/978-3-030-17184-1_1

[21] Shin-ichi Minato. 1993. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *DAC*. doi:10.1145/157485.164890

[22] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical pluggable types for Java. In *ISSTA*. doi:10.1145/1390630.1390656

[23] Benjamin Peters, Jules Jacobs, Diana Kalinichenko, Liam Stevenson, Aspen Smith, Derek Dreyer, and Richard A. Eisenberg. 2026. Supplementary Materials for Mode Crossing. Zenodo. doi:10.5281/zenodo.20272263

[24] Andreas Rossberg, Claudio Russo, and Derek Dreyer. 2014. F-ing modules. *Journal of Functional Programming* 24, 5 (2014), 529–607. doi:10.1017/S0956796814000264

[25] Filip Sieczkowski, Sergei Stepanenko, Jonathan Sterling, and Lars Birkedal. 2024. The Essence of Generalized Algebraic Data Types. *Proc. ACM Program. Lang.* 8, POPL, Article 24 (Jan. 2024), 29 pages. doi:10.1145/3632866

[26] Thomas Somers, Jonas Kastberg Hinrichsen, Lennard Gäher, and Robbert Krebbers. 2026. Building Blocks for Step-Indexed Program Logics. In *CPP*. doi:10.1145/3779031.3779095

[27] The Rust Project Developers. 2026. Coinduction. Rust Compiler Development Guide. https://rustc-dev-guide.rust-lang.org/solve/coinduction.html

[28] The Rust Project Developers. 2026. Send and Sync. The Rustonomicon. https://doc.rust-lang.org/nomicon/send-and-sync.html

[29] The Swift Project Authors. 2026. The Swift Programming Language: Concurrency. Swift.org Documentation. https://docs.swift.org/swift-book/documentation/the-swift-programming-language/concurrency/

[30] Joseph A. Zullo. 2026. Let Generalization, Polymorphic Recursion, and Variable Minimization in Boolean-Kinded Type Systems. *PACMPL* 10, POPL (2026). doi:10.1145/3776644

C Inference Solver Details

Inference reduces kind checking and sub-kinding to constraints on a lattice of mode crossings, solved by a dedicated solver. We present the solver as four layers over a shared base lattice.

Layer 0 (Appendix C.1) The base lattice
Layer 1 (Appendix C.2) Lattice polynomials and operations
Layer 2 (Appendix C.3) Fixpoint constraints
Layer 3 (Appendix C.4) Linear lattice functions
Layer 4 (Appendix C.5) From kinds to constraints

Connection to simplified LDD operations. This section gives the layer-by-layer reduction used by inference. The concrete Layer 1 data structure and operations are documented in Appendix D, which also covers the operations reused by Layer 2 after inlining solved variables.

C.1 Layer 0: The base lattice

We work over a distributive lattice $(L, \leq, \sqcup, \sqcap)$. A *lattice* is a poset in which every pair of elements has a join $a \sqcup b$ (their least upper bound) and a meet $a \sqcap b$ (their greatest lower bound). The lattice is *distributive* when joins and meets distribute over each other:

$$a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c) \qquad \text{and} \qquad a \sqcup (b \sqcap c) = (a \sqcup b) \sqcap (a \sqcup c).$$

In a finite lattice, the top and bottom elements exist and are given by $\top = \bigsqcup L$ and $\bot = \bigsqcap L$.

Bi-Heyting structure. A lattice is *Heyting* if for all $a, b \in L$ the implication $a \,/\, b$ exists and is characterized by the adjunction

$$x \leq (a \,/\, b) \quad \text{iff} \quad x \sqcap a \leq b.$$

Equivalently, $a \,/\, b$ is the greatest c such that $a \sqcap c \leq b$. Dually, a lattice is *co-Heyting* if for all $a, b \in L$ the subtraction $a \setminus b$ exists and satisfies

$$(a \setminus b) \leq x \quad \text{iff} \quad a \leq b \sqcup x,$$

i.e. $a \setminus b$ is the least c such that $a \leq b \sqcup c$. A *bi-Heyting lattice* is both Heyting and co-Heyting.

Finite distributive lattices are bi-Heyting. Let L be finite and distributive. Define the implication by the finite join

$$a \,/\, b \;=\; \bigsqcup \{\, x \in L \mid x \sqcap a \leq b \,\}.$$

Finiteness ensures the join exists. By distributivity,

$$(a \,/\, b) \sqcap a = \bigsqcup_{x \sqcap a \leq b} (x \sqcap a) \leq b,$$

so $a \,/\, b$ itself satisfies the defining inequality, and it is the greatest such element because every x in the set is below the join. Similarly, define the subtraction by the finite meet

$$a \setminus b \;=\; \bigsqcap \{\, x \in L \mid a \leq b \sqcup x \,\}.$$

Using the dual distributive law, we get $a \leq b \sqcup (a \setminus b)$, and minimality follows because the meet is below every x in the defining set. Hence every finite distributive lattice is bi-Heyting.

Conversely. Every Heyting lattice is distributive (and dually for co-Heyting). Indeed, let $d = (a \sqcap b) \sqcup (a \sqcap c)$. The inequality $d \leq a \sqcap (b \sqcup c)$ holds in any lattice. For the other direction, the adjunction gives $b \leq a \,/\, d$ and $c \leq a \,/\, d$, hence $b \sqcup c \leq a \,/\, d$, so $a \sqcap (b \sqcup c) \leq d$. Thus $a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$.

Examples. *Finite chains.* The linear order $L_n = \{0, 1, \ldots, n\}$ is a distributive lattice with order $0 \le 1 \le \cdots \le n$, join $\sqcup = \max$, and meet $\sqcap = \min$. The implication and subtraction have a simple closed form:

$$a \,/\, b = \begin{cases} n & \text{if } a \le b \\ b & \text{otherwise} \end{cases} \qquad \text{and} \qquad a \setminus b = \begin{cases} 0 & \text{if } a \le b \\ a & \text{otherwise.} \end{cases}$$

Intuitively, if $a \le b$ then $a \,/\, b = \top$ because *any* x satisfies $x \sqcap a \le b$; otherwise the largest x with $x \sqcap a \le b$ is exactly b. Dually, if $a \le b$ then $a \setminus b = \bot$, and otherwise the least x with $a \le b \sqcup x$ is a. For instance in L_3, $2 \,/\, 1 = 1$ while $1 \,/\, 2 = 3$, and $2 \setminus 1 = 2$ while $1 \setminus 2 = 0$.

Products. If L_1 and L_2 are distributive lattices, then so is $L_1 \times L_2$ with componentwise order, join, and meet; thus $(a_1, a_2) \le (b_1, b_2)$ iff $a_i \le b_i$ for each component. Top and bottom are $(\top_1, \top_2)$ and $(\bot_1, \bot_2)$, and when L_1, L_2 are Heyting/co-Heyting the operations are also componentwise: $(a_1, a_2) \,/\, (b_1, b_2) = (a_1 \,/\, b_1, a_2 \,/\, b_2)$ and $(a_1, a_2) \setminus (b_1, b_2) = (a_1 \setminus b_1, a_2 \setminus b_2)$. For example in $L_3 \times L_3$, $(2,1) \,/\, (1,0) = (1,0)$ and $(2,1) \setminus (1,0) = (2,1)$, while $(1,0) \,/\, (2,1) = (3,3)$.

Running concrete lattice. Concrete product-lattice examples below use $L = L_3 \times L_3$.

C.2 Layer 1: Lattice polynomials and operations

We now introduce lattice polynomials, written in algebraic notation to mirror familiar numeric polynomials.

Algebraic notation. To streamline formulas, we will write $a + b$ for the join $a \sqcup b$ and ab (or $a \cdot b$) for the meet $a \sqcap b$. We also write 0 for $\bot$ and 1 for $\top$, so $a + 0 = a$ and $a1 = a$. With this notation, join and meet are commutative, associative, and idempotent, and meet distributes over join (and vice versa):

$$\begin{aligned} a + b &= b + a & ab &= ba, \\ a + (b + c) &= (a + b) + c & a(bc) &= (ab)c, \\ a + a &= a & aa &= a, \\ a(b + c) &= ab + ac \end{aligned}$$

The dual distributive law

$$a + bc = (a + b)(a + c)$$

also holds in distributive lattices, but we will not use it explicitly. The order is recovered by $a \le b$ iff $a + b = b$ (equivalently $ab = a$).

Polynomials. We work with polynomials over such a base lattice L, by analogy to ordinary polynomials over numbers, using $+$ for join and multiplication for meet. Polynomials are generated by

$$p, q ::= c \mid x \mid p + q \mid pq$$

for constants $c \in L$ in the base lattice and variables x. An example of such a polynomial is:

$$p(x, y) = (1,0) + (2,3)x + (3,1)y + (1,2)xy$$

with coefficients $(1,0)$, $(2,3)$, $(3,1)$, and $(1,2)$ in $L = L_3 \times L_3$.

Normal forms. By repeatedly distributing meet over join and then using commutativity and associativity to flatten, any term from the grammar can be expanded into the form above: a finite sum of monomials of the form $cx_1 \cdots x_k$ (with $c \in L$) where the variables are distinct. Duplicate variables are removed using idempotence, explicitly $x \cdot x = x$, and duplicate monomials (same variable set) can be combined by joining their coefficients. *This expansion is not a normal form:* for instance, $x + xy$ and x denote the same polynomial (since $xy \le x$, so $x + xy = x$), yet they are

syntactically distinct. Moreover, even after removing absorbed monomials, coefficients are not canonical: $(1,0)x + (0,1)xy$ equals $(1,0)x + (1,1)xy$ because $(1,1)xy = (1,0)xy + (0,1)xy$ and the extra $(1,0)xy$ is absorbed by $(1,0)x$. However, the $(1,1)xy$ term cannot be removed entirely. The following develops two normal forms that are canonical and correct.

Maximal normal form. We define the *maximal normal form* for a multivariate polynomial $p(x_1, \ldots, x_n)$ as

$$\sum_{S \subseteq \{1,\ldots,n\}} p(\chi_S) \prod_{i \in S} x_i,$$

where χ_S assigns $x_i = 1$ for $i \in S$ and $x_i = 0$ otherwise. Here sums and products are join and meet, with the empty sum 0 and empty product 1. For the bivariate case this specializes to

$$p(0,0) + p(1,0)x + p(0,1)y + p(1,1)xy.$$

This is a normal form for two reasons:

(A) *Correctness.* Without loss of generality write $p(x,y)$ in multilinear form as $c_{00} + c_{10}x + c_{01}y + c_{11}xy$ using distributivity, commutativity, associativity, and idempotence. Then

$$\begin{aligned} p(0,0) &= c_{00} & p(1,0) &= c_{00} + c_{10} \\ p(0,1) &= c_{00} + c_{01} & p(1,1) &= c_{00} + c_{10} + c_{01} + c_{11}, \end{aligned}$$

so substituting into the normal form and expanding with distributivity yields terms like $c_{00}x$ and $c_{10}xy$, which are absorbed by c_{00} and $c_{10}x$ respectively, leaving $c_{00} + c_{10}x + c_{01}y + c_{11}xy$.

(B) *Canonicity.* Semantically equivalent polynomials agree on $(0,0)$, $(1,0)$, $(0,1)$, and $(1,1)$, so they produce the same normal form.

Examples.

(a) *Bivariate formula.* The definition specializes to

$$p(0,0) + p(1,0)x + p(0,1)y + p(1,1)xy.$$

(b) *Worked example.* For $p(x,y) = (1,0)x + (0,1)y$ we have $p(0,0) = (0,0)$, $p(1,0) = (1,0)$, $p(0,1) = (0,1)$, and $p(1,1) = (1,1)$, so the maximal normal form is $(1,0)x + (0,1)y + (1,1)xy$.

Minimal normal form. The *minimal normal form* uses the same basis of monomials, but replaces each coefficient $p(\chi_S)$ by a co-Heyting subtraction of all smaller terms:

$$\sum_{S \subseteq \{1,\ldots,n\}} \Big(p(\chi_S) \setminus \sum_{S' \subset S} p(\chi_{S'}) \Big) \prod_{i \in S} x_i.$$

This uses the co-Heyting subtraction $\setminus$ on the base lattice. *Examples.*

(a) *Bivariate formula.* The definition specializes to

$$\begin{aligned} p(0,0) + \big(p(1,0) \setminus p(0,0)\big)x + \big(p(0,1) \setminus p(0,0)\big)y \\ + \big(p(1,1) \setminus (p(1,0) + p(0,1) + p(0,0))\big)xy. \end{aligned}$$

(b) *Coefficient reduced.* For $p(x,y) = (1,0)x + (1,1)xy$, the xy coefficient reduces to $(1,1) \setminus (1,0) = (0,1)$, giving minimal normal form $(1,0)x + (0,1)xy$.

(c) *Term removed.* For $p(x,y) = (1,0)x + (1,0)xy$, we have $p(1,1) = (1,0)$ and $(1,0) \setminus (1,0) = (0,0)$, so the xy term disappears and the minimal normal form is just $(1,0)x$.

(d) *Term removed (mixed).* For $p(x,y) = (0,1)x + (1,0)y + (1,1)xy$, we have $p(1,0) = (0,1)$, $p(0,1) = (1,0)$, and $p(1,1) = (1,1)$, so the xy coefficient reduces to $(1,1) \setminus ((0,1) + (1,0)) = (0,0)$, and the minimal normal form is $(0,1)x + (1,0)y$.

(e) *Coefficient reduced (mixed).* In $L_3 \times L_3$, for $p(x,y) = (2,0)x + (0,3)y + (3,3)xy$, we have $p(1,0) = (2,0)$, $p(0,1) = (0,3)$, and $p(1,1) = (3,3)$, so the xy coefficient reduces to $(3,3) \setminus ((2,0) + (0,3)) = (3,0)$, giving minimal normal form $(2,0)x + (0,3)y + (3,0)xy$.

(f) *No xy term.* For $p(x,y) = (1,0)x + (0,1)y$ we have $p(0,0) = (0,0)$, $p(1,0) = (1,0)$, $p(0,1) = (0,1)$, $p(1,1) = (1,1)$, so the minimal normal form is $(1,0)x + (0,1)y$.

Pointwise order. For polynomials p, q, we write $p \leq q$ iff $p(\rho) \leq q(\rho)$ for every valuation ρ of the variables into L.

Lemma 15 (One-sided maximal normal form suffices). *Let*

$$p = \sum_{S \subseteq \{1,\ldots,n\}} c_S \prod_{i \in S} x_i \qquad q = \sum_{S \subseteq \{1,\ldots,n\}} d_S \prod_{i \in S} x_i,$$

where missing monomials have coefficient 0*. Assume only that q is in maximal normal form, i.e. $d_S = q(\chi_S)$ for all $S \subseteq \{1,\ldots,n\}$. Then $p \leq q$ iff $c_S \leq d_S$ for all S.*

Proof. For any S, evaluation at χ_S gives

$$p(\chi_S) = \sum_{T \subseteq S} c_T,$$

since $x_T(\chi_S) = 1$ iff $T \subseteq S$ and 0 otherwise. Hence $c_S \leq p(\chi_S)$.

($\Leftarrow$) Assume $c_S \leq d_S$ for all S. For any valuation ρ, monotonicity of meet gives $c_S \prod_{i \in S} \rho(x_i) \leq d_S \prod_{i \in S} \rho(x_i)$, and taking the join over all S preserves inequality, so $p(\rho) \leq q(\rho)$. Thus $p \leq q$.

($/$) Assume $p \leq q$. Fix $S \subseteq \{1,\ldots,n\}$ and show $c_S \leq d_S$.

1. From the corner-evaluation formula above, $p(\chi_S) = \sum_{T \subseteq S} c_T$, so in particular $c_S \leq p(\chi_S)$ because c_S is one of the join-summands.
2. By the pointwise assumption $p \leq q$, we have $p(\chi_S) \leq q(\chi_S)$.
3. By maximality of q, $q(\chi_S) = d_S$.

Chaining these yields $c_S \leq d_S$, and since S was arbitrary we get coefficientwise inequality for all S. □

If q is not maximal, coefficientwise comparison can fail: $x + xy \leq x$ but the right-hand side has $d_{\{1,2\}} = 0$. Also note that coefficientwise comparison on minimal normal forms is unsound: $p(x,y) = xy$ and $q(x,y) = x + y$ are both minimal and satisfy $p \leq q$, but the xy coefficient is 1 for p and 0 for q.

Operations and interface. The first solver layer exposes a small set of operations on polynomials:

- *Introduce constants and rigid variables.* Constants $c \in L$ and rigid variables $r \in R$ can be used as polynomials directly.
- *Combine by addition and multiplication.* Polynomials are closed under $p + q$ and pq.
- *Create a fresh flexible variable.* We may introduce a fresh flexible variable $x \in X$ to stand for an unknown subterm.
- *Order testing for ground polynomials.* When p and q contain no unsolved flexible variables, we check $p \leq q$ by computing the residual $p \setminus q$ and testing whether its rounded-up value is $\bot$.
- *Co-Heyting subtraction.* We can compute $p \setminus q$ directly in the DAG representation; this is the core normalizing operation.
- *Rounding to a lattice element.* Given a ground polynomial p, we can compute its least upper approximation in L, written $\lceil p \rceil$, by setting all rigid variables to 1.

Here $p \setminus q$ is co-Heyting subtraction in the lattice of positive lattice polynomials represented by the DAGs: it is the least positive polynomial r such that $p \le q + r$. It is not pointwise base-lattice subtraction. Thus

$$p \le q \iff p \setminus q = 0 \iff \lceil p \setminus q \rceil = \bot.$$

The last equivalence uses monotonicity: a positive polynomial is 0 exactly when its value with all rigid variables set to 1 is $\bot$. Operationally, the reference implementation checks

$$p \le q \iff \lceil p \setminus q \rceil = \bot$$

by computing `diff = sub_subsets (inline_solved_vars p) (inline_solved_vars q)` and then `witness = round_up diff`; the check succeeds exactly when `witness` is bottom. These operations form the interface of the first layer; later layers will encode kind inference and subsumption constraints into this language. Fixpoint assignments are handled in Layer 2. The simplified reference implementation of these operations (including `canonicalize`, `sub_subsets`, `round_up`, and `leq_with_reason`) is documented in Appendix D.

C.2.1 Efficient representation in minimal normal form.

Representation. The solver represents polynomials in a form reminiscent of binary decision diagrams (BDDs). We pick an order on the variables $v \in R \cup X$, and represent polynomials as a tree (or DAG) on these variables:

$$p, q ::= \; c \;\mid\; p + (vq) \qquad \text{where } c \in L \text{ and } v \in R \cup X$$

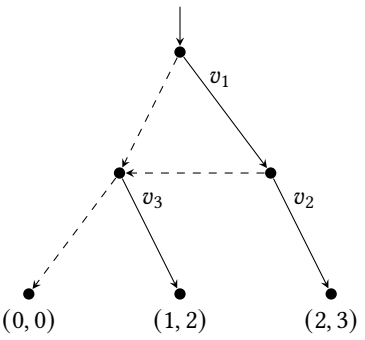
i.e.,

$$\textbf{type}\ \text{poly} = \text{Leaf}\ \textit{of}\ L \mid \text{Node}\ \textit{of}\ (R \cup X) \times \text{poly} \times \text{poly}$$

Leaves of the DAG are constants $c \in L$, and a node for variable v has two children: p and q, and represents the polynomial $p + vq$. We draw the p-edge as a dashed edge and the q-edge as a solid edge labeled by the variable.

Each root-to-leaf path contributes a product term: when a path follows the q-edge out of a variable node, we include that variable in the product, and then multiply the result with the leaf constant. The polynomial is the sum of all these path terms.

Example. For the base lattice $L = L_3 \times L_3$ and variable order $v_1 < v_2 < v_3$, consider the following DAG with a shared subtree:



This DAG denotes the polynomial

$$t + v_1\big(t + v_2(2,3)\big) \quad \text{where} \quad t = (0,0) + v_3(1,2).$$

For the DAG above, the path terms are:

$$(0,0), \quad (v_3(1,2)), \quad (v_1(0,0)), \quad (v_1v_3(1,2)), \quad (v_1v_2(2,3)).$$

DAG normal form. The solver maintains these polynomial DAGs in a normal form that corresponds to minimal normal form with respect to the chosen variable order:

Variable Order: When walking down the DAG, variables always appear according to the chosen variable order: the variables of children are always later in the order than the variable of the parent. Note that different paths down the DAG may visit different sets of variables.

Split Irredundancy: The q-child is never 0. If it is, the split is redundant and we can remove the node.

Normalization: The q-child is irredundant with the p-child according to co-Heyting subtraction: $q \setminus p = q$.

Normalization example. In the running product lattice $L = L_3 \times L_3$ with componentwise order, consider the two terms

$$(1,0)v_1 \quad \text{and} \quad (1,2)v_1v_2.$$

Since $v_1v_2 \leq v_1$, the second term is only relevant on the subregion where the first already applies, so normalization subtracts the overlap by replacing the coefficient $(1,2)$ with

$$(1,2) \setminus (1,0) = (0,2),$$

yielding the normalized polynomial

$$(1,0)v_1 + (0,2)v_1v_2.$$

If the coefficients are equal, *e.g.* $(1,0)v_1$ and $(1,0)v_1v_2$, then subtraction yields $(1,0) \setminus (1,0) = 0$ and the more specific term disappears; in the DAG this corresponds to a node whose q-child is 0, so the split is removed by split irredundancy.

Canonicity. For a fixed variable order, this normal form is canonical: if two polynomials in normal form are equal for all valuations of the variables, then they are syntactically identical (and conversely).

Operations on DAGs. Multiplication, addition, and co-Heyting subtraction are defined by recursion on the tree structure, followed by normalization to restore the invariants above.

C.3 Layer 2: Fixpoint constraints

This layer adds fixpoint assignments on top of lattice polynomials.

Interface.

- *Assign a flexible variable by a fixpoint.* We allow asserting the value of a flexible variable by $x := p$ where p is a polynomial. Because p may itself contain x, we have two variants with different fixpoint equations:

  $$x :=_{\text{lfp}} p \quad \text{and} \quad x :=_{\text{gfp}} p$$

  which take the least or greatest fixpoint of the induced monotone function.

Operationally, solving is done in two phases. First we force solved substitutions in the right-hand side using `inline_solved_vars`; this avoids reasoning about stale references to already-solved flexible variables. Then we install the solved definition of the assigned variable. In the simplified reference solver, this assignment step is:

$$x :=_{\text{lfp}} p : \ x \mapsto p[x := 0] \qquad \text{and} \qquad x :=_{\text{gfp}} p : \ x \mapsto p[x := 1],$$

implemented by `assign_bot` and `assign_top` after inlining. This one-step rule is valid because p is a positive lattice polynomial. Viewed as a polynomial in x, it has the form $A + Bx$ for polynomials A and B that do not mention x. Hence $\mu x.\, p(x) = A = p[0/x]$ and $\nu x.\, p(x) = A + B = p[1/x]$.

For systems of variables, the solver applies this rule by sequential elimination in the usual Bekic style. This is the same solver interface used by the optimized implementation; the appendix gives pseudocode and discusses pending GFP queues and forcing points (Appendix D).

C.4 Layer 3: Linear lattice functions

Layer 3 lifts Layer 1 from lattice elements to join-linear functions between products of lattices. We use the form

$$f(x_1, \ldots, x_n) = c_0 + c_1 x_1 + \cdots + c_n x_n,$$

where each coefficient c_i is itself a Layer 1 polynomial (possibly over rigid variables). Coefficients are extracted from corner evaluations. With e_i the valuation that maps $x_i \mapsto 1$ and all other arguments to 0, we have:

$$c_0 = f(0, 0, \ldots, 0),$$
$$c_i = f(e_i) \setminus c_0 \quad (1 \le i \le n).$$

Equivalently, one may use the redundant maximal coefficients

$$d_0 = f(0, 0, \ldots, 0),$$
$$d_i = f(e_i),$$

with $d_i = c_0 + c_i$. We use this redundant view when comparing functions: after rewriting the right-hand side into maximal normal form, Layer 1's coefficientwise comparison lemma applies directly.

Fixpoints at this layer reduce to Layer 2 by representing function coefficients as Layer 1 polynomials and solving their assignment constraints. The next subsection presents the construction for one ternary function symbol; the implementation uses the same construction for arbitrary arities, with one coefficient per basis point. Other abstract type constructors in scope are treated as fixed coefficient parameters while solving the distinguished recursive function.

C.4.1 Coefficient-based fixpoints for linear lattice terms.

C.4.2 Setting. Let $(L, \le, \sqcup, \sqcap, \bot, \top)$ be a *bounded distributive lattice*. We order function spaces pointwise: for $g, h : L^3 \to L$, $g \le h$ iff $\forall (x, y, z) \in L^3,\ g(x, y, z) \le h(x, y, z)$.

Definition 16 (Linear functions)**.** A function $f : L^3 \to L$ is *linear* (in the sense used in this paper) if there exist coefficients $c_0, c_1, c_2, c_3 \in L$ such that

$$f(x, y, z) \;=\; c_0 \;\sqcup\; (c_1 \sqcap x) \;\sqcup\; (c_2 \sqcap y) \;\sqcup\; (c_3 \sqcap z). \tag{2}$$

Write $\mathsf{Lin}_3(L)$ for the set of all such functions.

C.4.3 Canonical coefficient representation. A naive coefficient tuple (c_0, c_1, c_2, c_3) for (2) is not unique. For fixpoint computation it is convenient to work with a canonical representation.

Define the *basis points*

$$p_0 = (\bot, \bot, \bot), \quad p_1 = (\top, \bot, \bot), \quad p_2 = (\bot, \top, \bot), \quad p_3 = (\bot, \bot, \top).$$

Definition 17 (Coefficient domain, abstraction, concretization)**.** Define

$$\mathsf{Coef}_3(L) \;=\; \{(a_0, a_1, a_2, a_3) \in L^4 \mid a_0 \le a_1,\ a_0 \le a_2,\ a_0 \le a_3\},$$

ordered componentwise.

Define $\gamma : \mathsf{Coef}_3(L) \to \mathsf{Lin}_3(L)$ by

$$\gamma(a_0, a_1, a_2, a_3)(x, y, z) \;=\; a_0 \;\sqcup\; (a_1 \sqcap x) \;\sqcup\; (a_2 \sqcap y) \;\sqcup\; (a_3 \sqcap z). \tag{3}$$

Define $\delta : \mathsf{Lin}_3(L) \to \mathsf{Coef}_3(L)$ by evaluation on basis points:

$$\delta(f) \;=\; \big(f(p_0), f(p_1), f(p_2), f(p_3)\big). \tag{4}$$

Lemma 18 (Normalization and uniqueness via basis points). *For every $f \in \mathsf{Lin}_3(L)$ we have $\delta(f) \in \mathsf{Coef}_3(L)$ and*

$$\gamma(\delta(f)) \;=\; f.$$

Conversely, for every $a \in \mathsf{Coef}_3(L)$,

$$\delta(\gamma(a)) \;=\; a.$$

Hence γ and δ are mutually inverse order-isomorphisms between $\mathsf{Coef}_3(L)$ and $\mathsf{Lin}_3(L)$.

Proof. Let f be linear, so f has the form (2). Evaluating at p_0 gives $f(p_0) = c_0$. Evaluating at p_1 gives $f(p_1) = c_0 \sqcup (c_1 \sqcap \top) = c_0 \sqcup c_1$. Similarly $f(p_2) = c_0 \sqcup c_2$ and $f(p_3) = c_0 \sqcup c_3$. Thus $\delta(f) = (c_0, c_0 \sqcup c_1, c_0 \sqcup c_2, c_0 \sqcup c_3) \in \mathsf{Coef}_3(L)$.

Now compute $\gamma(\delta(f))$:

$$\gamma(\delta(f))(x,y,z) = c_0 \sqcup ((c_0 \sqcup c_1) \sqcap x) \sqcup ((c_0 \sqcup c_2) \sqcap y) \sqcup ((c_0 \sqcup c_3) \sqcap z).$$

Using distributivity, $(c_0 \sqcup c_1) \sqcap x = (c_0 \sqcap x) \sqcup (c_1 \sqcap x)$. After distributing similarly for y, z and using absorption $c_0 \sqcup (c_0 \sqcap x) = c_0$ (and likewise for y, z), we obtain $\gamma(\delta(f))(x,y,z) = c_0 \sqcup (c_1 \sqcap x) \sqcup (c_2 \sqcap y) \sqcup (c_3 \sqcap z) = f(x,y,z)$.

Conversely, let $a = (a_0, a_1, a_2, a_3) \in \mathsf{Coef}_3(L)$. Then $\gamma(a)(p_0) = a_0$. Moreover $\gamma(a)(p_1) = a_0 \sqcup (a_1 \sqcap \top) = a_0 \sqcup a_1 = a_1$ because $a_0 \le a_1$. Similarly $\gamma(a)(p_2) = a_2$ and $\gamma(a)(p_3) = a_3$. Hence $\delta(\gamma(a)) = a$.

Monotonicity of γ and δ under componentwise/pointwise orders is immediate, and mutual inverses yield an order-isomorphism. □

C.4.4 A grammar of lattice expressions with a function symbol. Fix a constant set of lattice elements $K \subseteq L$ (the set of allowed constants). We consider expressions generated by

$$e \;::=\; k \mid x \mid y \mid z \mid f(e_1, e_2, e_3) \mid (e_1 \sqcup e_2) \mid (e \sqcap k), \qquad (k \in K).$$

Given a function $g : L^3 \to L$, the standard denotation $[\![e]\!]_g : L^3 \to L$ is defined as usual:

$$[\![k]\!]_g(x,y,z) = k,$$
$$[\![x]\!]_g(x,y,z) = x, \quad [\![y]\!]_g(x,y,z) = y, \quad [\![z]\!]_g(x,y,z) = z,$$
$$[\![e_1 \sqcup e_2]\!]_g = [\![e_1]\!]_g \sqcup [\![e_2]\!]_g, \qquad [\![e \sqcap k]\!]_g = [\![e]\!]_g \sqcap k,$$
$$[\![f(e_1,e_2,e_3)]\!]_g(x,y,z) = g\big([\![e_1]\!]_g(x,y,z),\ [\![e_2]\!]_g(x,y,z),\ [\![e_3]\!]_g(x,y,z)\big).$$

C.4.5 Coefficient semantics (four-point evaluation). The key observation is that, when g is linear, every expression denotes a linear function, and it suffices to track the four basis-point values.

Definition 19 (Coefficient semantics)**.** For $a \in \mathsf{Coef}_3(L)$, define $[\![e]\!]^\#_a \in \mathsf{Coef}_3(L)$ by structural recursion:

Variables.

$$[\![x]\!]^\#_a = (\bot, \top, \bot, \bot), \quad [\![y]\!]^\#_a = (\bot, \bot, \top, \bot), \quad [\![z]\!]^\#_a = (\bot, \bot, \bot, \top).$$

Constants.

$$[\![k]\!]^\#_a = (k, k, k, k).$$

Join and meet-with-constant (componentwise). If $u = (u_0, u_1, u_2, u_3)$ and $v = (v_0, v_1, v_2, v_3)$, define $u \sqcup^4 v = (u_0 \sqcup v_0, \ldots, u_3 \sqcup v_3)$ and $u \sqcap^4 k = (u_0 \sqcap k, \ldots, u_3 \sqcap k)$. Then

$$[\![e_1 \sqcup e_2]\!]^\#_a = [\![e_1]\!]^\#_a \sqcup^4 [\![e_2]\!]^\#_a, \qquad [\![e \sqcap k]\!]^\#_a = [\![e]\!]^\#_a \sqcap^4 k.$$

Application. Let $b^i = [\![e_i]\!]^\#_a = (b^i_0, b^i_1, b^i_2, b^i_3)$ for $i \in \{1,2,3\}$. Write $a = (a_0, a_1, a_2, a_3)$. Define $\mathrm{app}(a, b^1, b^2, b^3) \in L^4$ componentwise by

$$\big(\mathrm{app}(a, b^1, b^2, b^3)\big)_j = a_0 \sqcup (a_1 \sqcap b^1_j) \sqcup (a_2 \sqcap b^2_j) \sqcup (a_3 \sqcap b^3_j), \qquad j \in \{0,1,2,3\}. \tag{5}$$

Set

$$[\![f(e_1, e_2, e_3)]\!]^\#_a = \mathrm{app}(a, b^1, b^2, b^3).$$

Lemma 20 (Soundness of coefficient semantics). *For all $a \in \mathsf{Coef}_3(L)$ and all expressions e,*

$$\gamma\big([\![e]\!]^\#_a\big) = [\![e]\!]_{\gamma(a)}. \tag{6}$$

Equivalently, $[\![e]\!]^\#_a = \delta([\![e]\!]_{\gamma(a)})$.

Proof. By structural induction on e.

Case $e = k$. Immediate from the definitions: $\gamma(k,k,k,k)$ is the constant function with value k.

Case $e \in \{x, y, z\}$. Immediate from (3) and the definitions in Definition 19.

Case $e = e_1 \sqcup e_2$. By induction hypothesis, $\gamma([\![e_i]\!]^\#_a) = [\![e_i]\!]_{\gamma(a)}$ for $i = 1, 2$. Since γ is defined by joins/meets and $\sqcup^4$ is componentwise, $\gamma(u \sqcup^4 v) = \gamma(u) \sqcup \gamma(v)$ pointwise. Hence

$$\gamma([\![e]\!]^\#_a) = \gamma([\![e_1]\!]^\#_a \sqcup^4 [\![e_2]\!]^\#_a) = \gamma([\![e_1]\!]^\#_a) \sqcup \gamma([\![e_2]\!]^\#_a) = [\![e_1]\!]_{\gamma(a)} \sqcup [\![e_2]\!]_{\gamma(a)} = [\![e]\!]_{\gamma(a)}.$$

Case $e = e' \sqcap k$. Similarly, componentwise meets satisfy $\gamma(u \sqcap^4 k) = \gamma(u) \sqcap k$ pointwise, using distributivity of $\sqcap$ over $\sqcup$ in L.

Case $e = f(e_1, e_2, e_3)$. Let $b^i = [\![e_i]\!]^\#_a$ and by induction hypothesis $\gamma(b^i) = [\![e_i]\!]_{\gamma(a)}$. Let $g = \gamma(a)$. Then for any $(x, y, z) \in L^3$,

$$[\![e]\!]_g(x, y, z) = g\big(\gamma(b^1)(x, y, z), \gamma(b^2)(x, y, z), \gamma(b^3)(x, y, z)\big).$$

Expanding $g = \gamma(a)$ using (3) and expanding each $\gamma(b^i)$ again using (3), we obtain an expression built from $\sqcup, \sqcap$. Distributivity allows pushing each meet by a_1, a_2, a_3 through the joins coming from the arguments. After regrouping the constant part and the x-, y-, and z-parts, the resulting function is exactly $\gamma(c_0, c_1, c_2, c_3)$ where $(c_0, c_1, c_2, c_3) = \mathrm{app}(a, b^1, b^2, b^3)$ as defined in (5). Thus

$$[\![e]\!]_{\gamma(a)} = \gamma([\![e]\!]^\#_a),$$

as required. □

Corollary 21 (Closure). *If $a \in \mathsf{Coef}_3(L)$ then $\gamma([\![e]\!]^\#_a)$ is linear, i.e., $[\![e]\!]_{\gamma(a)} \in \mathsf{Lin}_3(L)$. In particular, every expression in the grammar denotes a linear function when f is linear.*

C.4.6 Coefficient transformer and fixpoint transfer. Fix an expression E in the grammar and define the functional $F : \mathsf{Lin}_3(L) \to \mathsf{Lin}_3(L)$ by

$$F(g) = [\![E]\!]_g.$$

(When needed for fixpoints, we assume F is monotone; this holds for the above grammar because all constructors are monotone.)

Define the induced coefficient transformer

$$\widehat{F} : \mathsf{Coef}_3(L) \to \mathsf{Coef}_3(L), \qquad \widehat{F}(a) = [\![E]\!]^\#_a. \tag{7}$$

Lemma 22 (Conjugacy). *For all $a \in \mathsf{Coef}_3(L)$,*

$$F(\gamma(a)) = \gamma(\widehat{F}(a)).$$

Equivalently, $\widehat{F} = \delta \circ F \circ \gamma$.

Proof. Immediate from Lemma 20 with $e = E$. □

Theorem 23 (Fixpoint computation over coefficients)**.** *Assume L is a complete lattice (so $\mathsf{Coef}_3(L)$ and $\mathsf{Lin}_3(L)$ are complete under the respective orders) and F is monotone. Then both least fixpoints exist and satisfy*

$$\mu F \;=\; \gamma\big(\mu\widehat{F}\big).$$

Proof. By Lemma 18, $\gamma : \mathsf{Coef}_3(L) \to \mathsf{Lin}_3(L)$ is an order-isomorphism with inverse δ. By Lemma 22, $F \circ \gamma = \gamma \circ \widehat{F}$, i.e. F and $\widehat{F}$ are conjugate through the isomorphism. Order-isomorphisms preserve least fixpoints of monotone maps (apply γ to the Knaster–Tarski characterization of μ as the least pre-fixpoint, or equivalently transport the complete lattice structure). Hence $\mu F = \gamma(\mu\widehat{F})$. □

C.4.7 Operational computation (Kleene iteration). If L has finite height (or more generally if Kleene iteration stabilizes), $\mu\widehat{F}$ can be computed by iterating in $\mathsf{Coef}_3(L)$:

$$a^{(0)} = (\bot, \bot, \bot, \bot), \qquad a^{(n+1)} = \widehat{F}(a^{(n)}).$$

Upon stabilization $a^{(n+1)} = a^{(n)} = a^\star$, we return $f_{\mathrm{fix}} = \gamma(a^\star)$. This is the generic mathematical computation for the coefficient transformer. The implementation uses the specialized Layer 2 polynomial solver described above, which eliminates coefficients one at a time instead of iterating the whole function transformer.

C.4.8 Remark: n-ary generalization. All definitions extend to arity n by using $n{+}1$ basis points $p_0 = (\bot, \ldots, \bot)$ and p_i with $\top$ in coordinate i and $\bot$ elsewhere. Then $\mathsf{Coef}_n(L) \subseteq L^{n+1}$ consists of tuples $(a_0, \ldots, a_n)$ with $a_0 \le a_i$, and $\gamma(a)(x_1, \ldots, x_n) = a_0 \sqcup \bigsqcup_{i=1}^{n}(a_i \sqcap x_i)$. The coefficient semantics evaluates expressions on these basis points and the fixpoint transfer theorem remains unchanged.

C.5 Layer 4: From kinds to constraints

We now reduce judgments from §3 to Layer 3 constraints. Given a type context Δ, we introduce one unknown join-linear function F_t for each abstract type constructor $t \in \mathrm{dom}(\Delta)$. Lattice terms are translated compositionally:

$$\begin{aligned}
[\![\hat{\alpha}]\!] &= \alpha, \\
[\![m]\!] &= m, \\
[\![\phi \sqcup \psi]\!] &= [\![\phi]\!] + [\![\psi]\!], \\
[\![\phi \sqcap m]\!] &= [\![\phi]\!]\, m, \\
[\![\hat{t}\ \overline{\phi}]\!] &= F_t(\overline{[\![\phi]\!]}), \\
[\![(\mu\ \hat{t}\ \overline{\hat{\alpha}}.\ \phi)\ \overline{\psi}]\!] &= \big(\mu X.\, \lambda\overline{a}.\, [\![\phi]\!][\hat{t} \mapsto X, \hat{\alpha}_i \mapsto a_i]\big)(\overline{[\![\psi]\!]}).
\end{aligned}$$

Here X is a fresh unknown function symbol replacing $\hat{t}$ in the body. The fixpoint is taken at the function level and is applied to $\overline{[\![\psi]\!]}$ only after solving the recursive function; the substitutions $\hat{\alpha}_i \mapsto a_i$ are simultaneous. For each declaration $(t : \overline{\alpha}.\phi_t) \in \Delta$, we add the hypothesis constraint

$$F_t(\overline{\alpha}) \le [\![\phi_t]\!].$$

To decide $\Delta \vDash \phi \le \psi$, we ask the solver whether

$$[\![\phi]\!] \le [\![\psi]\!]$$

is entailed under these hypotheses. Before calling the Layer 1 order test, these hypothesis constraints are eliminated by bounded substitution: a hypothesis $x \le b(x)$ is discharged by replacing x with $x \sqcap b(x)$ in the conclusion. After this step, the final Layer 1 query is unconditional. This is the constraint check used by the Cross rule (Fig. 4) and therefore by kind checking and subsumption in the type system.

$$\begin{aligned}(\text{Layer 4 kinds}) &\Longrightarrow (\text{Layer 3 function constraints})\\ &\Longrightarrow (\text{Layer 2 fixpoint assignments})\\ &\Longrightarrow (\text{Layer 1 polynomial DAG operations}).\end{aligned}$$

Detailed pseudocode for the solver operations used by this reduction is given in Appendix D.

D Simplified LDD Operations

These supplementary materials include the complete OxCaml compiler used in the paper, including both an optimized LDD implementation and a pedagogical simplified LDD implementation used as a reference model. This section documents the simplified version, whose purpose is to expose the mathematical structure directly, without low-level representation tricks.

Placement in the inference pipeline. This appendix section is the concrete operational companion to Layer 1 and parts of Layer 2 in Appendix C. In particular, `sub_subsets` is the Layer 1 co-Heyting subtraction operation on DAGs, and Layer 2 fixpoint assignments reuse these normalized DAG operations.

Design goal. The simplified implementation keeps exactly the semantic invariants used by the optimized solver, but removes engineering optimizations that obscure the core algorithm. In particular, it does not use unboxed leaves, cached down-edges, pointer-equality short-cuts, or mutation-based fast paths. As a result, each operation is written as direct structural recursion over the LDD syntax tree.

Representation and invariant. An LDD node has the form

$$\mathrm{Node}(v, \ell, h) \quad \text{representing} \quad \ell \sqcup (v \sqcap h).$$

Variables are totally ordered and each path follows increasing variable ids. The key normalization invariant is:

$$h = h \backslash \ell.$$

Intuitively, the high branch stores only the contribution that is not already present in the low branch. Normalization is enforced by a dedicated constructor `node v ~lo ~hi`, which computes canonicalize(hi, lo) before building the node.

Core operations. The algebraic operations `join`, `meet`, and `sub_subsets` follow one common pattern: they compare the top variables of the two inputs and recurse according to the variable order. When roots coincide, recursion proceeds on corresponding subtrees; when one root is smaller, recursion descends only on that side. This is the same merge discipline used in ordered BDD-style structures, and it preserves variable ordering by construction.

The only non-trivial operation is canonicalization (co-Heyting subtraction). Its recursive cases mirror the ordered merge above and maintain $h = h\backslash\ell$ at each rebuilt node. In the simplified code this procedure is intentionally explicit rather than factored through specialized fast paths.

```
(* Node(v, lo, hi) denotes lo ⊔ (v ⊓ hi). *)

let rec canonicalize hi lo =
  match hi, lo with
  | _, _ when hi == lo → bot
  | _, _ when is_bot lo → hi
  | Leaf a, Leaf b → Leaf (a \ b)
```

```
    | Leaf a, Node _ → Leaf (a \ down0 lo)
    | Node _, Leaf b → map_leaves (fun a → a \ b) hi
    | Node (v,l1,h1), Node (w,l2,h2) →
        let order = compare v w in
        if order = 0 then
          let lo' = canonicalize l1 l2 in
          let hi' = canonicalize (canonicalize h1 h2) l2 in
          node_raw v lo' hi'
        else if order < 0 then
          node_raw v (canonicalize l1 lo) (canonicalize h1 lo)
        else
          canonicalize hi l2

  let node v ~lo ~hi = node_raw v lo (canonicalize hi lo)
  let sub_subsets a b = canonicalize a b
```

Here `canonicalize` is the core co-Heyting subtraction routine on DAGs: it computes $hi \backslash lo$ while respecting variable order and preserving the normalization invariant. `node` is the public smart constructor; it is the only constructor that enforces normalization. `sub_subsets` is the exported subtraction operation, defined directly by canonicalization.

Why the `order = 0` case is subtle. Suppose both roots have the same variable:

$$tree_1 = \mathrm{Node}(v, l_1, h_1), \qquad tree_2 = \mathrm{Node}(v, l_2, h_2).$$

Then we must compute

$$(\, l_1 \sqcup (v \sqcap h_1)\,) \backslash (\, l_2 \sqcup (v \sqcap h_2)\,).$$

The key point is that l_2 is active in *both* worlds ($v = \bot$ and $v = \top$), while h_2 is active only when $v = \top$. Therefore:

$$\begin{aligned} lo' &= l_1 \backslash l_2, \\ hi1' &= h_1 \backslash h_2, \\ hi' &= hi1' \backslash l_2 = (h_1 \backslash h_2) \backslash l_2. \end{aligned}$$

So the result is $\mathrm{Node}(v, lo', hi')$. The second subtraction by l_2 is exactly the non-obvious step: without it, the high branch would incorrectly retain mass already covered by the low branch of b.

```
  let rec join a b =
    match a, b with
    | Leaf x, Leaf y → Leaf (x ⊔ y)
    | Leaf x, _ → join_with_leaf x b
    | _, Leaf y → join_with_leaf y a
    | Node (v,l1,h1), Node (w,l2,h2) →
        let order = compare v w in
        if order = 0 then
          node_raw v
            (join l1 l2)
            (join (canonicalize h1 l2) (canonicalize h2 l1))
        else if order < 0 then
          node_raw v (join l1 b) (canonicalize h1 b)
        else
          node_raw w (join a l2) (canonicalize h2 a)
```

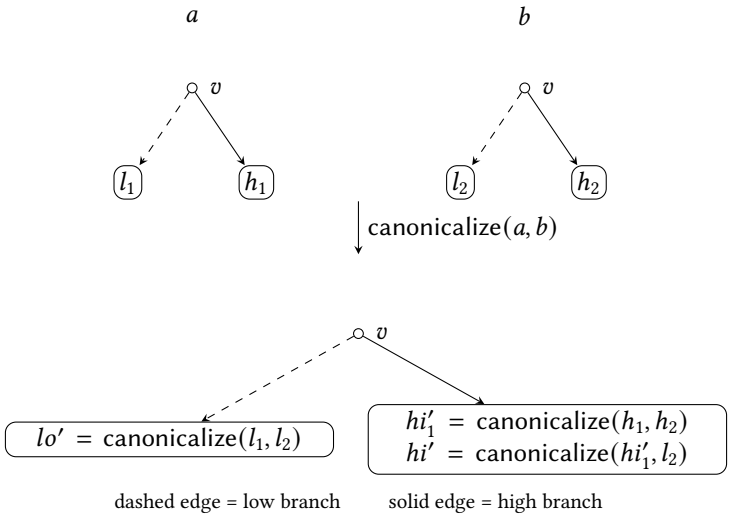


Fig. 7. Equal-root case in canonicalize: both inputs split on the same variable v, so subtraction is computed by separate low/high residuals, with the high residual requiring an additional subtraction by l_2.

```
and join_with_leaf c n =
  match n with
  | Leaf d → Leaf (c ⊔ d)
  | Node (v,lo,hi) →
      node_raw v (join_with_leaf c lo) (canonicalize hi (Leaf c))

let rec meet a b =
  match a, b with
  | Leaf x, _ → meet_with_leaf x b
  | _, Leaf y → meet_with_leaf y a
  | Node (v,l1,h1), Node (w,l2,h2) →
      let order = compare v w in
      if order = 0 then
        let lo = meet l1 l2 in
        let hi = meet (join h1 l1) (join h2 l2) in
        node v ~lo ~hi
      else if order < 0 then
        node v ~lo:(meet l1 b) ~hi:(meet h1 b)
      else
        node w ~lo:(meet a l2) ~hi:(meet a h2)

and meet_with_leaf c n =
  match n with
  | Leaf d → Leaf (c ⊓ d)
  | Node (v,lo,hi) →
      node v ~lo:(meet_with_leaf c lo) ~hi:(meet_with_leaf c hi)
```

`join` merges two normalized DAGs according to top-variable order and re-normalizes only where overlap can arise. `join_with_leaf` is the specialized branch for adding a constant to a structured

diagram; it pushes the constant down the low spine and subtracts it from the high branch. `meet` is dual in structure and combines same-variable nodes by intersecting the induced upper sets. `meet_with_leaf` is the base case for meeting with a constant.

Naive baseline and why this code is faster. A fully naive presentation would define join and meet by first constructing an unnormalized candidate and then always calling the smart constructor node, which runs canonicalization on the high branch. For example, with equal roots $a = \mathsf{Node}(v, l_1, h_1)$, $b = \mathsf{Node}(v, l_2, h_2)$, a naive join could be written as

$$\mathsf{node}(v,\ \mathsf{join}(l_1, l_2),\ \mathsf{join}(\mathsf{join}(l_1, h_1), \mathsf{join}(l_2, h_2)))\,,$$

and then rely on node to subtract the low part from the high part. The implementation above is equivalent, but faster, because it performs only the subtraction work that is actually needed and in several branches uses node_raw (no extra canonicalization pass).

The key equalities rely on canonical inputs:

$$h_1 = h_1 \backslash l_1, \qquad h_2 = h_2 \backslash l_2.$$

These invariants are used in the following non-obvious places.

(1) **Equal-root** join **(`order = 0`).** The naive high branch is

$$hi_{\text{naive}} = (l_1 \sqcup h_1) \sqcup (l_2 \sqcup h_2).$$

After subtracting $lo' = \mathsf{join}(l_1, l_2)$, we get

$$hi_{\text{naive}} \backslash lo' = (h_1 \backslash l_2) \sqcup (h_2 \backslash l_1),$$

exactly $\mathsf{join}(\mathsf{canonicalize}(h_1, l_2), \mathsf{canonicalize}(h_2, l_1))$. This is why the code can directly build with node_raw: the residual is already low-disjoint.

(2) **Unequal-root** join **(`order < 0/> 0`).** In the `order < 0` branch, the naive high would be $\mathsf{join}(h_1, b)$, then normalized against $\mathsf{join}(l_1, b)$. Also, because inputs are variable-ordered and $v < w$, b cannot contain a test on v; this is why recursion may keep b unchanged while descending under a's v-node. Using $h_1 \backslash l_1 = h_1$, this reduces to $h_1 \backslash b$, i.e. $\mathsf{canonicalize}(h_1, b)$, so the code avoids constructing and re-normalizing $\mathsf{join}(h_1, b)$. The `order > 0` case is symmetric.

(3) join_with_leaf**.** Naively, one would normalize $\mathsf{join}(c, hi)$ against $\mathsf{join}(c, lo)$. By canonicality of (lo, hi), this simplifies to $hi \backslash c$, so the code uses $\mathsf{canonicalize}(hi, \mathsf{Leaf}(c))$ directly.

(4) meet**.** The implementation already follows the denotational split, but still depends on canonical representation: for equal roots, the high branch must use activated values $(l_i \sqcup h_i)$, not just h_i, because $\mathsf{Node}(v, l, h)$ denotes $l \sqcup (v \sqcap h)$. For unequal roots, ordering of canonical DAGs ensures that if $v < w$, then b contains no test on v, so descending only on a's side is sound.

If inputs were not canonical, these identities would fail and the node_raw-based fast paths in join would no longer be correct.

Why the Node–Node case of `join` is subtle. Let

$$a = \mathsf{Node}(v, l_1, h_1), \qquad b = \mathsf{Node}(w, l_2, h_2).$$

The `join` recursion splits into three structurally different cases.

(1) $v = w$ **(`order = 0`).** Both diagrams branch on the same variable, so low branches combine directly:

$$lo' = \mathsf{join}(l_1, l_2).$$

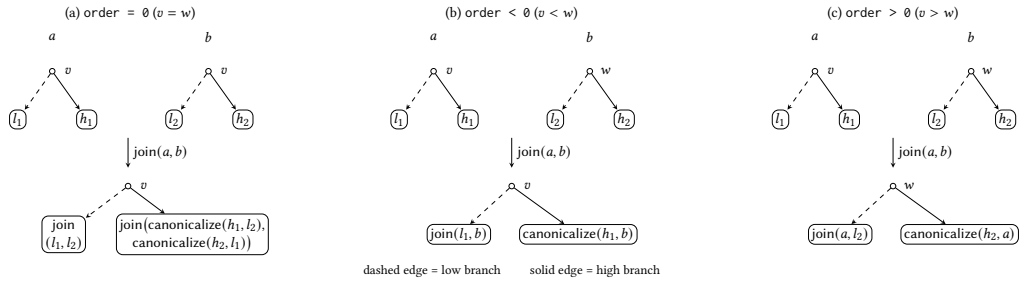


Fig. 8. Case split for join on two non-leaf roots. The three cases are determined by the top-variable order and differ in where normalization (canonicalize) is required to maintain $hi = hi \backslash lo$.

The high branch is subtler: each side's high contribution must be normalized against the other side's low contribution before being joined:

$$hi' = \mathsf{join}\big(\mathsf{canonicalize}(h_1, l_2), \mathsf{canonicalize}(h_2, l_1)\big).$$

This is what preserves the invariant $hi = hi \backslash lo$.

(2) $v < w$ **(`order < 0`).** Variable v must remain above w, so we keep v at the root and push all of b under both branches:

$$lo' = \mathsf{join}(l_1, b), \qquad hi' = \mathsf{canonicalize}(h_1, b).$$

(3) $v > w$ **(`order > 0`).** This is the symmetric case:

$$lo' = \mathsf{join}(a, l_2), \qquad hi' = \mathsf{canonicalize}(h_2, a),$$

and the output root is w.

Why the Node–Node case of `meet` is subtle. Again let

$$a = \mathsf{Node}(v, l_1, h_1), \qquad b = \mathsf{Node}(w, l_2, h_2).$$

The recursion has the same three-way split by variable order, but the `order = 0` branch differs fundamentally from `join`. When $v = w$, expanding the denotation $\mathsf{Node}(v, l, h) = l \sqcup (v \sqcap h)$ gives

$$\begin{aligned} a \sqcap b &= (l_1 \sqcup (v \sqcap h_1)) \sqcap (l_2 \sqcup (v \sqcap h_2)) \\ &= (l_1 \sqcap l_2) \ \sqcup \ \big(v \sqcap ((l_1 \sqcup h_1) \sqcap (l_2 \sqcup h_2))\big). \end{aligned}$$

So the high branch must combine the *activated* values $(l_i \sqcup h_i)$, not just h_i.

(1) $v = w$ **(`order = 0`).** The low branch is pointwise:

$$lo' = \mathsf{meet}(l_1, l_2).$$

For the high branch, each side can contribute either via its low branch or via its high branch once v is true, so we first form the activated values $\mathsf{join}(h_1, l_1)$ and $\mathsf{join}(h_2, l_2)$, then meet them:

$$hi' = \mathsf{meet}\big(\mathsf{join}(h_1, l_1), \mathsf{join}(h_2, l_2)\big).$$

The result is rebuilt as $\mathsf{node}(v, lo', hi')$, which re-normalizes.

(2) $v < w$ **(`order < 0`).** Keep v at the root and meet both branches of a with all of b:

$$lo' = \mathsf{meet}(l_1, b), \qquad hi' = \mathsf{meet}(h_1, b).$$

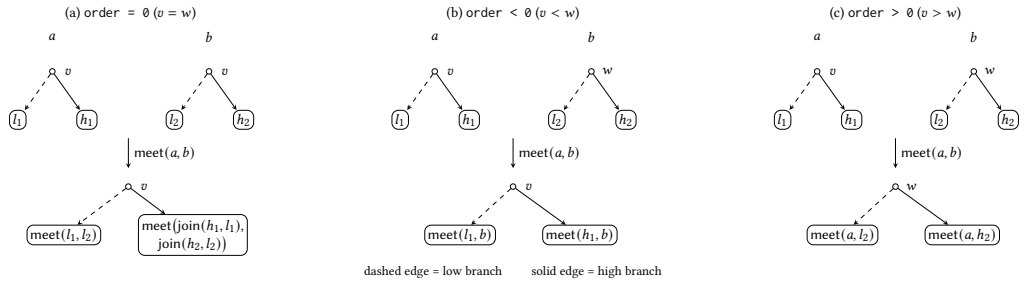


Fig. 9. Case split for meet on two non-leaf roots. In the `order = 0` case, the high branch meets $\text{join}(h_1, l_1)$ with $\text{join}(h_2, l_2)$ before re-normalization via node.

(3) $v > w$ **(`order > 0`).** Symmetrically:

$$lo' = \text{meet}(a, l_2), \qquad hi' = \text{meet}(a, h_2),$$

and the output root is w.

```
let rec assign_bot ~var n =
  match n with
  | Leaf _ → n
  | Node (v,lo,hi) →
      let order = compare var v in
      if order < 0 then n
      else if order = 0 then lo
      else node v ~lo:(assign_bot ~var lo) ~hi:(assign_bot ~var hi)

let rec assign_top ~var n =
  match n with
  | Leaf _ → n
  | Node (v,lo,hi) →
      let order = compare var v in
      if order < 0 then n
      else if order = 0 then join lo hi
      else node v ~lo:(assign_top ~var lo) ~hi:(assign_top ~var hi)

let rec inline_solved_vars n =
  match n with
  | Leaf _ → n
  | Node (v,lo,hi) →
      match v.state with
      | Rigid _ → n
      | Solved d →
          let lo' = inline_solved_vars lo in
          let hi' = inline_solved_vars hi in
          let d'  = inline_solved_vars d in
          v.state <- Solved d';
          join lo' (meet hi' d')
      | Unsolved →
          node v ~lo:(inline_solved_vars lo) ~hi:(inline_solved_vars hi)
```

`assign_bot` computes substitution $var := \bot$ by removing the high branch at the matching variable. `assign_top` computes substitution $var := \top$ by joining low and high branches at the matching variable. `inline_solved_vars` eliminates solved flexible variables by replacing each solved occurrence with its definition, while leaving rigid variables untouched.

Fixpoint variables. The simplified implementation keeps the same solver-level interface as the optimized code: `solve_lfp`, `solve_gfp`, pending GFP queues, and solved-variable inlining. The simplified module is therefore a reference model for the optimized module, covering full solver behavior rather than only basic lattice operations.

A key interface point is that solved substitutions are *lazy* with respect to existing trees. Calling `solve_lfp` or `solve_gfp` updates variable state (`var.state <- Solved ...`) but does not traverse and rewrite every LDD currently in scope. Hence multiple trees may remain "in flight" after solving, still containing references to variables that are now marked solved. The operation `inline_solved_vars` is the forcing step that pushes those substitutions into a given tree. Operationally, after solving, any tree that is reused should be forced with `inline_solved_vars`, unless the called operation already does so internally.

```
let solve_lfp var rhs =
  match var.state with
  | Unsolved →
      let rhs' = inline_solved_vars rhs in
      var.state <- Solved (assign_bot ~var rhs')
  | _ → error "invalid variable state"

let solve_gfp var rhs =
  match var.state with
  | Unsolved →
      let rhs' = inline_solved_vars rhs in
      var.state <- Solved (assign_top ~var rhs')
  | _ → error "invalid variable state"

let enqueue_gfp var rhs = pending := (var, rhs) :: !pending
let solve_pending () =
  let work = !pending in
  pending := [];
  iter (fun (v,rhs) → solve_gfp v rhs) work
```

`solve_lfp` and `solve_gfp` implement one-variable least and greatest fixpoint solving, respectively, after forcing solved substitutions in the right-hand side. `enqueue_gfp` records deferred GFP constraints, and `solve_pending` snapshots and clears the queue, then discharges the snapshot in sequence. Neither `solve_lfp` nor `solve_gfp` rewrites arbitrary external trees; they only install the solved definition on the variable.

```
let rec round_up n =
  match inline_solved_vars n with
  | Leaf c → c
  | Node (_,lo,hi) → round_up lo ⊔ round_up hi

let leq_with_reason a b =
```

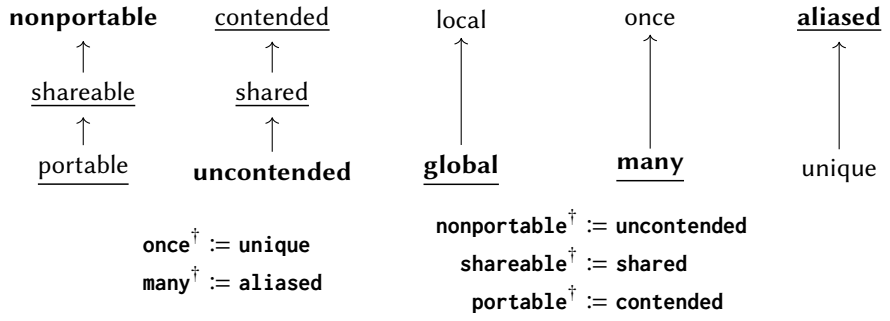


Fig. 10. The portability, contention, locality, affinity, and uniqueness modal axes, and their duality operation. Portability, locality, and affinity are comonadic, and contention and uniqueness are monadic.

```
  let diff = sub_subsets (inline_solved_vars a) (inline_solved_vars b) in
  let witness = round_up diff in
  non_bot_axes witness
```

`round_up` evaluates a ground diagram by interpreting each variable as $\top$, i.e., by joining all leaves reachable in the normalized DAG. `leq_with_reason` checks $a \leq b$ via $a \backslash b$, then extracts non-bottom lattice axes as a concrete witness when the inequality fails. Both routines explicitly force solved substitutions, so they are safe entry points when trees are still in flight.

Relationship to the optimized implementation. The code above implements the same API as the optimized implementation. The optimized module has better node representation, cached summaries, and aggressive short-circuiting. To validate this correspondence, we tested (1) concrete examples (canonicalization, assignments, LFP/GFP solving), (2) randomized algebraic/property tests, and (3) cross-checks between optimized and simplified modules.

E SMoKi Type System Definitions

The complete SMoKi with all supported modal axes (locality, affinity, uniqueness, portability, and contention) is described in Figures 10 and 12 to 16. These rules additionally use the locality, affinity, and uniqueness modes. Specifically, the typing rules make heavy use of the context joining operation + to encode substructural features. These figures are heavily based on Georges et al. [12].

F Important Changes to the Logical Relation

Proving the correct mode-crossing rules for mutable references and recursive types (§3.4) required interesting changes to the model of DRFCaml [12]. We briefly describe these changes in this section.

DRFCaml's type system distinguishes between two reference types $\mathrm{ref}_p\ A$ using a portability-mode annotation. This annotation represents the portability mode of the reference's contents: Only a $\mathrm{ref}_{\texttt{portable}}\ A$ reference can be allocated at **portable** mode, and a $\mathrm{ref}_{\texttt{nonportable}}\ A$ would always be at **nonportable** mode. This distinction was necessary precisely because DRFCaml's model of references did not support mode crossing. In OxCaml, by contrast, an `int ref` allocated at **nonportable** mode can cross portability and become **portable**. We improved DRFCaml's model of references to support this mode-crossing behavior. We achieve this by additionally requiring a reference at **portable** (**shareable**) mode to carry a witness that its inner type crosses portability (crosses **shareable**).

$$
\begin{array}{rcl}
\emptyset + \emptyset & := & \emptyset \\
(\Gamma_1, x : -) + (\Gamma_2, x : -) & := & (\Gamma_1 + \Gamma_2), x : - \\
(\Gamma_1, x : A \mathbin{@} m) + (\Gamma_2, x : -) & := & (\Gamma_1 + \Gamma_2), x : A \mathbin{@} m \\
(\Gamma_1, x : -) + (\Gamma_2, x : A \mathbin{@} m) & := & (\Gamma_1 + \Gamma_2), x : A \mathbin{@} m \\
(\Gamma_1, x : A \mathbin{@} (p, l, o_1, c, \textbf{aliased})) & & \\
+ (\Gamma_2, x : A \mathbin{@} (p, l, o_2, c, \textbf{aliased})) & := & (\Gamma_1 + \Gamma_2), x : A \mathbin{@} (p, l, \textbf{many}, c, u)
\end{array}
$$

$$
\boxed{\Gamma_1 \le \Gamma_2}
$$

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

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

$$
\begin{array}{rcl}
🔒_{(l_2,o_2,p_2)}(\emptyset), & := & \emptyset \\
🔒_{(l_2,o_2,p_2)}(\Gamma, x : -) & := & 🔒_{(l_2,o_2,p_2)}(\Gamma), x : - \\
🔒_{(l_2,o_2,p_2)}(\Gamma, x : A \mathbin{@} (p_1, l_1, o_1, c_1, u_1)) & := & \begin{cases} 🔒_{(l_2,o_2,p_2)}(\Gamma), x : (p_1, l_1, o_1, c_1 \vee p_2^\dagger, u_1 \vee o_2^\dagger) & \\ \qquad \text{if } l_1 \le l_2,\ o_1 \le o_2, \text{ and } p_1 \le p_2 & \\ 🔒_{(l_2,o_2,p_2)}(\Gamma), x : - \qquad \text{otherwise} & \end{cases}
\end{array}
$$

Fig. 11. The context ordering and join relations, and the context lock operation.

$$
A, B \in \text{Type} ::= \alpha \mid t\ \overline{A} \mid \square^m A \mid (\mu\ t\ \overline{\alpha}.\ A)\ \overline{B} \mid \mathbb{1} \mid \mathbb{B} \mid \mathbb{Z} \mid A \times B \mid A + B \mid A \mathbin{@} m_1 \to B \mathbin{@} m_2 \mid \text{ref}\ A \mid \text{box}\ A \mid \text{atomic}\ A
$$

$$
\begin{array}{rcl}
\text{kind}(\alpha) & := & \hat{\alpha} \\
\text{kind}(t\ \overline{A}) & := & \hat{t}\ \overline{\text{kind}(A)} \\
\text{kind}(\square^m A) & := & \text{kind}(A) \sqcap m \\
\text{kind}((\mu\ t\ \overline{\alpha}.\ A)\ \overline{C}) & := & (\mu\ \hat{t}\ \overline{\hat{\alpha}}.\ \text{kind}(A))\ \overline{\text{kind}(C)} \\
\text{kind}(\mathbb{1}) & := & \bot
\end{array}
\qquad
\begin{array}{rcl}
\text{kind}(\mathbb{B}) & := & \bot \\
\text{kind}(\mathbb{Z}) & := & \bot \\
\text{kind}(A_1 \times A_2) & := & \text{kind}(A_1) \sqcup \text{kind}(A_2) \\
\text{kind}(A_1 + A_2) & := & \text{kind}(A_1) \sqcup \text{kind}(A_2)
\end{array}
$$

$$
\begin{array}{rcl}
\text{kind}(A \mathbin{@} m_1 \to B \mathbin{@} m_2) & := & (\textbf{nonportable}, \textbf{local}, \textbf{once}, \textbf{contended}, \textbf{aliased}) \\
\text{kind}(\text{box}\ A) & := & (\textbf{portable}, \textbf{local}, \textbf{many}, \textbf{contended}, \textbf{unique}) \sqcup \text{kind}(A) \\
\text{kind}(\text{ref}\ A) & := & (\textbf{portable}, \textbf{local}, \textbf{many}, \textbf{uncontended}, \textbf{unique}) \sqcup \text{kind}(A) \\
\text{kind}(\text{atomic}\ A) & := & (\textbf{portable}, \textbf{local}, \textbf{many}, \textbf{contended}, \textbf{unique}) \sqcup \\
& & (\text{kind}(A) \sqcap (\textbf{portable}, \textbf{local}, \textbf{once}, \textbf{contended}, \textbf{unique}))
\end{array}
$$

Fig. 12. SMoKi types and their kind rules.

We also added recursive types to the DRFCaml semantic model. This was difficult because the laws on semantic types (cf. [12]), specifically those that use Iris updates $\Rrightarrow$, are not preserved by *laters*. Laters come up when taking fixpoints of functions on semantic types.

$\boxed{\Delta \vdash A \text{ ok}}$

$$\frac{\Delta \vdash A \text{ ok} \qquad \Delta \vdash B \text{ ok}}{\Delta \vdash A \times B \text{ ok}} \qquad \frac{\Delta \vdash A \text{ ok} \qquad \Delta \vdash B \text{ ok}}{\Delta \vdash A + B \text{ ok}} \qquad \frac{\Delta \vdash A \text{ ok}}{\Delta \vdash \Box^{m'} A \text{ ok}} \qquad \frac{(\alpha : m) \in \Delta}{\Delta \vdash \alpha \text{ ok}}$$

$$\frac{\forall i.\ \Delta \vdash A_i \text{ ok} \qquad (t : \overline{a}.\ \varphi) \in \Delta}{\Delta \vdash t\ \overline{A} \text{ ok}} \qquad \frac{\Delta, t : \overline{\alpha}.\ \top, \overline{\alpha : \top} \vdash A \text{ ok} \qquad \forall i.\ \Delta \vdash B_i \text{ ok}}{\Delta \vdash (\mu\ t\ \overline{\alpha}.\ A)\ \overline{B} \text{ ok}}$$

Fig. 13. Auxiliary SMoKi judgements.

Something similar was already observed in RustBelt, where delayed sharing is presented as a solution [14]. It is unclear how delayed sharing would be adapted to the DRFCaml logical relation, or whether it generalizes to all of the laws we require. Instead, we develop a technique based on later credits and additive time receipts for later credit generation [20] and on RefinedRust's *logical step modality* [11, 26] to work around these limitations.

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

$$\frac{\Delta;\text{🔒}_w(\Gamma), x : A \mathbin{@} m_1 \vdash e : B \mathbin{@} m_2}{\Delta;\Gamma \vdash \lambda^{w.\mathsf{l}} x, e : (A \mathbin{@} m_1 \to B \mathbin{@} m_2) \mathbin{@} (w, n)}\ \textsc{Lam}$$

$$\frac{w.\mathrm{o} = \textbf{many} \qquad \Delta;\text{🔒}_w(\Gamma), x : A \mathbin{@} m_1, f : (A \mathbin{@} m_1 \to B \mathbin{@} m_2) \mathbin{@} (w, n) \vdash e : B \mathbin{@} m_2}{\Delta;\Gamma \vdash \lambda^{w.\mathsf{l}} f\ x, e : (A \mathbin{@} m_1 \to B \mathbin{@} m_2) \mathbin{@} (w, n)}\ \textsc{Rec}$$

$$\frac{\Delta;\Gamma_1 \vdash e_1 : (A \mathbin{@} m_1 \to B \mathbin{@} m_2) \mathbin{@} m_3 \qquad \Delta;\Gamma_2 \vdash e_2 : A \mathbin{@} m_1}{\Delta;\Gamma_1 + \Gamma_2 \vdash e_1\ e_2 : B \mathbin{@} m_2}\ \textsc{App}$$

$$\frac{\Delta;\Gamma_1 \vdash e_1 : \mathbb{1} \mathbin{@} m_1 \qquad \Delta;\Gamma_2 \vdash e_2 : A \mathbin{@} m_2}{\Delta;\Gamma_1 + \Gamma_2 \vdash (e_1; e_2) : A \mathbin{@} m_2}\ \textsc{Seq} \qquad \frac{\Delta;\Gamma_1 \vdash e_1 : A \mathbin{@} m_1 \qquad \Delta;\Gamma_2, x : A \mathbin{@} m_1 \vdash e_2 : B \mathbin{@} m_2}{\Delta;\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2 : B \mathbin{@} m_2}\ \textsc{Let}$$

$$\frac{\begin{array}{c}\Delta;\Gamma_1 \vdash e_1 : A \mathbin{@} (p', l, \textbf{many}, c', \textbf{unique}) \\ \Delta;\Gamma_2, x : A \mathbin{@} (p', \textbf{local}, \textbf{many}, c', \textbf{aliased}) \vdash e_2 : B \mathbin{@} (p, \textbf{global}, o, c, u) \\ \Delta;\Gamma_3, x : A \mathbin{@} (p', l, \textbf{many}, c', \textbf{unique}), y : B \mathbin{@} (p, \textbf{global}, o, c, u) \vdash e_3 : C \mathbin{@} m\end{array}}{\Delta;\Gamma_1 + \Gamma_2 + \Gamma_3 \vdash \mathsf{borrow}\ x := e_1\ \mathsf{for}\ y := e_2\ \mathsf{in}\ e_3 : C \mathbin{@} m}\ \textsc{Borrow}$$

$$\frac{\Delta;\text{🔒}_{(\textbf{global},\textbf{once},\textbf{portable})}(\Gamma) \vdash e : A \mathbin{@} m_1}{\Delta;\Gamma \vdash \mathsf{fork}(e) : \mathbb{1} \mathbin{@} m_2}\ \textsc{Fork} \qquad \frac{\Delta;\Gamma_1 \vdash e_1 : A \mathbin{@} m \qquad \Delta;\Gamma_2 \vdash e_2 : B \mathbin{@} m}{\Delta;\Gamma_1 + \Gamma_2 \vdash (e_1, e_2) : A \times B \mathbin{@} m}\ \textsc{Pair}$$

$$\frac{\Delta;\Gamma_1 \vdash e_1 : A \times B \mathbin{@} m_1 \qquad \Delta;\Gamma_2, y : B \mathbin{@} m_1, x : A \mathbin{@} m_1 \vdash e_2 : C \mathbin{@} m_2}{\Delta;\Gamma_1 + \Gamma_2 \vdash \mathsf{unpair}\ e_1\ \mathsf{as}\ (x, y)\ \mathsf{in}\ e_2 : C \mathbin{@} m_2}\ \textsc{Unpair}$$

$$\frac{\Delta;\Gamma \vdash e : A \mathbin{@} m}{\Delta;\Gamma \vdash \mathsf{inl}(e) : A + B \mathbin{@} m}\ \textsc{Inl} \qquad \frac{\Delta;\Gamma \vdash e : B \mathbin{@} m}{\Delta;\Gamma \vdash \mathsf{inr}(e) : A + B \mathbin{@} m}\ \textsc{Inr}$$

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

$$\frac{\Delta;\Gamma \vdash e : A \mathbin{@} (p, \textbf{global}, o, c, u)}{\Delta;\Gamma \vdash \mathsf{region}(e) : A \mathbin{@} (p, \textbf{global}, o, c, u)}\ \textsc{Region} \qquad \frac{\Delta;\Gamma_1 \vdash e_1 : A \mathbin{@} m \qquad \Delta;\Gamma_2 \vdash e_2 : B \mathbin{@} m \qquad \mathit{binopTyped}(\oplus, A, B, C)}{\Delta;\Gamma_1 + \Gamma_2 \vdash e_1 \oplus e_2 : C \mathbin{@} m}\ \textsc{BinOp}$$

$$\frac{\Delta;\Gamma \vdash e : A \mathbin{@} m \qquad \mathit{unopTyped}(\oplus, A, B)}{\Delta;\Gamma \vdash \oplus(e) : B \mathbin{@} m}\ \textsc{UnOp} \qquad \textsc{Box}\ \frac{\Delta;\Gamma \vdash e : A \mathbin{@} m_1 \sqcap m_2}{\Delta;\Gamma \vdash \mathsf{box}\ e : \Box^{m_2} A \mathbin{@} m_1} \qquad \textsc{Unbox}\ \frac{\Gamma;\Delta \vdash e : \Box^{m_2} A \mathbin{@} m_1}{\Gamma;\Delta \vdash \mathsf{unbox}\ e : A \mathbin{@} m_1 \sqcap m_2}$$

$$\frac{\Delta;\Gamma \vdash e : A \mathbin{@} (w, n) \qquad w \le w' \qquad n' \le n}{\Delta;\Gamma \vdash e : A \mathbin{@} (w', n')}\ \textsc{Sub} \qquad \frac{\Gamma_1 \ge \Gamma_2 \qquad \Delta;\Gamma_1 \vdash e : A \mathbin{@} m}{\Delta;\Gamma_2 \vdash e : A \mathbin{@} m}\ \textsc{Sub-Ctx} \qquad \frac{\Delta;\Gamma \vdash e : A \mathbin{@} (w, n) \qquad \Delta \vDash \mathrm{kind}(A) \le (w', n')}{\Delta;\Gamma \vdash e : A \mathbin{@} (w' \sqcap w, n' \,/\, n)}\ \textsc{Cross}$$

Fig. 14. SMoKi typing rules.

$$\frac{\forall i.\ \Delta \vdash B_i \text{ ok} \qquad \Delta;\Gamma \vdash e : A[\overline{\alpha \mapsto B}, t \mapsto \overline{\alpha}.(\mu\ t\ \overline{\alpha}.\ A)\ \overline{\alpha}]\ @\ m}{\Delta;\Gamma \vdash \text{roll}\ e : (\mu\ t\ \overline{\alpha}.\ A)\ \overline{B}\ @\ m}\ \textsc{Roll} \qquad \frac{\Delta;\Gamma \vdash e : (\mu\ t\ \overline{\alpha}.\ A)\ \overline{B}\ @\ m}{\Delta;\Gamma \vdash \text{unroll}\ e : A[\overline{\alpha \mapsto B}, t \mapsto \overline{\alpha}.(\mu\ t\ \overline{\alpha}.\ A)\ \overline{\alpha}]\ @\ m}\ \textsc{Unroll}$$

Fig. 15. SMoKi typing rules for recursive types.

$$\frac{\Delta;\Gamma \vdash e : A\ @\ (\textbf{nonportable}, l, \textbf{many}, \textbf{uncontended}, \textbf{aliased})}{\Delta;\Gamma \vdash \text{alloc}^{l}(e) : \text{ref}(A)\ @\ (\textbf{nonportable}, l, \textbf{many}, \textbf{uncontended}, \textbf{unique})}\ \textsc{NaAlloc}$$

$$\frac{\Delta;\Gamma \vdash e : \text{ref}(A)\ @\ (p, l, o, c, u) \qquad c \neq \textbf{contended}}{\Delta;\Gamma \vdash !^{\text{NA}}e : A\ @\ (p, l, \textbf{many}, c, \textbf{aliased})}\ \textsc{NaLoad}$$

$$\frac{\Delta;\Gamma_1 \vdash e_1 : \text{ref}(A)\ @\ (p, l, o, \textbf{uncontended}, u) \qquad \Delta;\Gamma_2 \vdash e_2 : A\ @\ (p, \textbf{global}, \textbf{many}, \textbf{uncontended}, \textbf{aliased})}{\Delta;\Gamma_1 + \Gamma_2 \vdash e_1 \leftarrow^{\text{NA}} e_2 : \mathbb{1}\ @\ m_2}\ \textsc{NaStore}$$

$$\frac{\Delta;\Gamma \vdash e : A\ @\ m \qquad m.\text{l} = l}{\Delta;\Gamma \vdash \text{alloc}^{l}(e) : \text{box}(A)\ @\ m}\ \textsc{ImAlloc} \qquad \frac{\Delta;\Gamma \vdash e : \text{box}(A)\ @\ m}{\Delta;\Gamma \vdash !^{\text{NA}}e : A\ @\ m}\ \textsc{ImLoad}$$

$$\frac{\Delta;\Gamma \vdash e : A\ @\ (\textbf{portable}, l, \textbf{many}, \textbf{contended}, \textbf{unique})}{\Delta;\Gamma \vdash \text{alloc}^{l}(e) : \text{atomic}(A)\ @\ (\textbf{portable}, l, \textbf{many}, \textbf{contended}, \textbf{unique})}\ \textsc{AtAlloc}$$

$$\frac{\Delta;\Gamma \vdash e : \text{atomic}(A)\ @\ m}{\Delta;\Gamma \vdash !^{\text{AT}}e : A\ @\ (p, l, o, \textbf{contended}, \textbf{aliased})}\ \textsc{AtLoad}$$

$$\frac{\Delta;\Gamma_1 \vdash e_1 : \text{atomic}(A)\ @\ (\textbf{portable}, l, o, \textbf{contended}, u) \qquad \Delta;\Gamma_2 \vdash e_2 : A\ @\ (\textbf{portable}, \textbf{global}, \textbf{many}, \textbf{contended}, \textbf{aliased})}{\Delta;\Gamma_1 + \Gamma_2 \vdash e_1 \leftarrow^{\text{AT}} e_2 : \mathbb{1}\ @\ m}\ \textsc{AtStore}$$

$$\frac{\Delta;\Gamma_1 \vdash e_1 : \text{atomic}(A)\ @\ (\textbf{portable}, l, o, \textbf{contended}, u) \qquad \Delta;\Gamma_2 \vdash e_2 : A\ @\ m \qquad \Delta;\Gamma_3 \vdash e_3 : A\ @\ (\textbf{portable}, \textbf{global}, \textbf{many}, \textbf{contended}, \textbf{aliased}) \qquad \mathit{typeCmpSafe}(A)}{\Delta;\Gamma_1 + \Gamma_2 + \Gamma_3 \vdash \text{cmpXchg}(e_1, e_2, e_3) : A \times \mathbb{B}\ @\ (\textbf{portable}, l, \textbf{many}, \textbf{contended}, \textbf{aliased})}\ \textsc{CmpXchg}$$

$$\frac{\Delta;\Gamma_1 \vdash e_1 : \text{atomic}(A)\ @\ (\textbf{portable}, l, o, \textbf{contended}, u) \qquad \Delta;\Gamma_2 \vdash e_2 : A\ @\ (\textbf{portable}, \textbf{global}, \textbf{many}, \textbf{contended}, \textbf{aliased})}{\Delta;\Gamma_1 + \Gamma_2 \vdash \text{xchg}(e_1, e_2) : A\ @\ (\textbf{portable}, l, \textbf{many}, \textbf{contended}, \textbf{aliased})}\ \textsc{Xchg}$$

$$\frac{\Delta;\Gamma_1 \vdash e_1 : \text{atomic}(\mathbb{Z})\ @\ (\textbf{portable}, l, o, \textbf{contended}, u) \qquad \Delta;\Gamma_2 \vdash e_2 : \mathbb{Z}\ @\ m_1}{\Delta;\Gamma_1 + \Gamma_2 \vdash \text{faa}(e_1, e_2) : \mathbb{Z}\ @\ m_2}\ \textsc{Faa}$$

Fig. 16. SMoKi typing rules for state.