Extensible Data Types with Ad-Hoc Polymorphism

MATTHEW TOOHEY, University of Toronto, Canada
YANNING CHEN, University of Toronto, Canada
ARA JAMALZADEH, University of Toronto, Canada
NINGNING XIE, University of Toronto, Canada

This paper proposes a novel language design that combines extensible data types, implemented through row types and row polymorphism, with ad-hoc polymorphism, implemented through type classes. Our design introduces several new constructs and constraints useful for generic operations over rows. We formalize our design in a source calculus $\lambda_{\rho}^{\Rightarrow}$, which elaborates into a target calculus $\mathrm{F}_{\omega}^{\otimes\oplus}$. We prove that the target calculus is type-safe and that the elaboration is sound, thus establishing the soundness of $\lambda_{\rho}^{\Rightarrow}$. All proofs are mechanized in the Lean 4 proof assistant. Furthermore, we evaluate our type system using the Brown Benchmark for Table Types, demonstrating the utility of extensible rows with type classes for table types.

CCS Concepts: • **Theory of computation → Type theory**; **Type structures**; • **Software and its engineering → Data types and structures**; **Constraints**; *Polymorphism*; *Functional languages*.

Additional Key Words and Phrases: Row polymorphism, Row constraints, Type classes, Elaboration

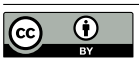
ACM Reference Format:
Matthew Toohey, Yanning Chen, Ara Jamalzadeh, and Ningning Xie. 2026. Extensible Data Types with Ad-Hoc Polymorphism. *Proc. ACM Program. Lang.* 10, POPL, Article 20 (January 2026), 29 pages. https://doi.org/10.1145/3776662

1 Introduction

Modularity stands as a fundamental property in software development, making systems easier to understand, maintain, and evolve. One significant challenge in achieving modularity in data types has traditionally been the difficulty of extending them in a type-safe manner without breaking existing code. This is where *row types* [Wand 1987] offer a compelling approach, which facilitates the creation of *extensible data types* by allowing developers to add new fields to a data type without compromising type safety or requiring widespread modifications across the codebase.

Rows are, at their core, a mapping from labels to types, effectively capturing the structure of records and variants. Row types have been extensively studied in the literature [Cardelli and Mitchell 1990; Harper and Pierce 1991; Leijen 2005; Rémy 1989, 1992; Shields and Meijer 2001; Wand 1991], and have found various applications, particularly through *row polymorphism*, a form of *parametric polymorphism* that enables abstraction over possible row extensions. Row types form the basis for the object-oriented features [Rémy and Vouillon 1998] and *polymorphic variants* [Garrigue 1998] in OCaml, and are employed to express *effect types* [Hillerström and Lindley 2016; Leijen 2017; Lindley and Cheney 2012] and extensible choices in *session types* [Lindley and Morris 2017].

Authors' Contact Information: Matthew Toohey, University of Toronto, Toronto, Canada, mtoohey@cs.toronto.edu; Yanning Chen, University of Toronto, Toronto, Canada, yanning@cs.toronto.edu; Ara Jamalzadeh, University of Toronto, Toronto, Canada, a.jamalzadeh@mail.utoronto.ca; Ningning Xie, University of Toronto, Toronto, Canada, ningningxie@cs.toronto.edu.



ACM 2475-1421/2026/1-ART20
https://doi.org/10.1145/3776662

While parametrically polymorphic functions are useful for functions whose behaviours are entirely uniform in its type argument, there often arises a need to leverage specific knowledge about that type. Consider, for instance, defining an equality function that compares two values. Such a function is not universally definable; more crucially, its behaviour may differ fundamentally depending on the type. To address this, *type classes* [Wadler and Blott 1989] offer a powerful approach for achieving *ad-hoc polymorphism* by allowing functions to be overloaded, with *qualified types* [Jones 2003a] expressing type class constraints. For example, Haskell's equality function (==) has type $\forall a.\ \mathit{Eq}\ a \Rightarrow a \to a \to \mathit{Bool}$, indicating that any instantiation of a must be an instance of the *Eq* type class. Type classes have been widely adopted in numerous languages including Haskell [Jones 2003b], Rocq [Sozeau and Oury 2008], and Idris [Brady 2013], among others.

Now, let us consider defining an equality function for a record type. In scenarios where all types within the record are fully specified and known to be comparable, the record's equality function (==) can be implemented by a pairwise comparison of corresponding fields. However, explicitly defining such a type class instance for records can quickly become tedious. Moreover, more significant challenges arise when defining such an equality function for *extensible* record types. First, extending a record with additional fields requires programmers to re-provide a type class instance (e.g. *Eq*) for the extended record, imposing an unnecessary and burdensome overhead. Moreover, and more fundamentally, the inherent nature of row polymorphism prevents us from fully specifying all types within a record. Consequently, we are unable to formally express that all fields in the extended row must support the necessary type class constraint. This restricts the practical extensibility of row types in the presence of ad-hoc polymorphism, as the inability to express type class constraints on extended rows prevents the application of ad-hoc polymorphic functions to these extensible data types. As a result, programmers are now forced to write significant boilerplate code for functions operating on each record extension, and, as the record schemas evolve, maintaining and updating the code becomes increasingly complex and error-prone.

This paper aims to address these challenges by providing a novel combination of extensible data types, implemented through row types, with ad-hoc polymorphism, through type classes. Specifically, we offer the following contributions:

- We contribute a novel design for a type system that features row polymorphism, records, variants, and type classes, which allows us to express type class constraints over polymorphic rows (§2):
  - We contribute a new form of **All** constraints, where a specific property holds across all fields.
  - We propose **ind**, a new language construct for folding over rows.
  - We introduce *row commutativity annotations*, allowing for strict row ordering when necessary. Our system also supports commutativity-polymorphic functions, and we further establish a *commutativity hierarchy* by treating non-commutativity as a subtype of commutativity.
  - We support **Lift**, a type-level mapping of rows, which is particularly useful in systems where rows can be higher-kinded [Hubers and Morris 2023].
  - Additionally, we introduce a novel unlifting constraint, **Split**, which is useful for splitting rows based on their type information.
- We formalize our design in $\lambda^{\Rrightarrow}_{\rho}$, a source calculus featuring row constraints (following *abstracting extensible data types* [Morris and McKinna 2019]), *first-class labels* [Leijen 2004], the row folding construct **ind**, and type classes, along with all the aforementioned features (§3).
- We present $\mathrm{F}^{\otimes\oplus}_{\omega}$, a target calculus extending System Fω with *type-level lists* and mappings (§4), and prove its syntactic type soundness (Thms. 4.4 and 4.5). We then present a type-directed elaboration of $\lambda^{\Rrightarrow}_{\rho}$ into the target calculus $\mathrm{F}^{\otimes\oplus}_{\omega}$, through *dictionary-passing elaboration* of type classes, interpretations of row constraints, and a constraint-based strategy for elaborating **ind** (§5). We prove elaboration soundness (Thms. 5.1 and 5.2).

- We have mechanized all lemmas and proofs in the Lean 4 proof assistant (§6).
- We evaluate our type system using the benchmark for *table types* [Lu et al. 2021], demonstrating that constraints over records can effectively express types for operations over table data (§7).

Our formalism is detailed, and some rules are elided for space reasons. The complete set of rules is included in the appendix, which can be found in the artifact [Toohey et al. 2025] along with the Lean 4 proofs.

2 Overview

This section introduces the features of our calculi: §2.1 explores row types that capture the structures of records and variants, highlighting their utility in defining extensible data types; and §2.2 provides background on type classes, and details our design that integrates them with extensible rows. For clarity, we use Haskell-like syntax for examples throughout this section.

2.1 Extensible Rows, Records, and Variants

Row types, originally introduced by Wand [1987] to model inheritance, provide an approach to typing extensible records and variants. Intuitively, rows define a mapping from labels to types. As an example, consider the following row:

$pet \triangleq \langle name \triangleright String, age \triangleright Int, weight \triangleright Float \rangle$

This *pet* row specifies three fields: *name* of type *String*, *age* of type *Int*, and *weight* of type *Float*.

Records. Row types are fundamental to representing the structure of records. Writing $\{ l \triangleright e \}$ for record expressions, and Π for record types constructed from rows, we can define an expression:

$alice = \{ name \triangleright \text{“}alice\text{”}, age \triangleright 2, weight \triangleright 2.4 \} : \Pi\, pet$

Here, *alice* is a record of type $\Pi\, pet$. The Π acts as a type constructor that takes a row (in this case, *pet*) and denotes a record type.

We use $(+\!\!+)$ for record concatenation, which comes in handy for extending records with new fields. For example, we can extend *alice* with an additional *favourite_food* field:

$alice +\!\!+ \{ favourite\_food \triangleright \text{“}fish\text{”} \} : \Pi \langle name \triangleright String, age \triangleright Int, weight \triangleright Float, favourite\_food \triangleright String \rangle$

We leave the details of which labels can appear together in a row abstract for now, as different row theories exist [Morris and McKinna 2019].

To access a field within a record, we use the r/l operator, which projects field l out of record r.[1] For instance, the function:

$getName = \lambda x.\, x/name$

takes a record and returns the value of its *name* field. Thus, (*getName alice*) would return “*alice*”.

Variants. Similarly, the Σ type constructor builds variant types from rows. Writing $[l \triangleright e]$ for variant expressions, we can define:

$shape \triangleq \langle rectangle \triangleright \Pi \langle length \triangleright Float, width \triangleright Float \rangle, circle \triangleright \Pi \langle radius \triangleright Float \rangle \rangle$
$r = [rectangle \triangleright \{ length \triangleright 2.0, width \triangleright 3.0 \}] : \Sigma\, shape$

Here, r is a rectangle with a length and a width, which is then injected into a variant type $\Sigma\, shape$. A $\Sigma\, shape$ variant can be either a *rectangle* or a *circle*.

To calculate the area of such a variant type, we define the *area* function:

[1] For clarity, we use r/l to project a field from a record that may contain multiple fields. In the formalism (§3), we require explicit projections (*prj*), and r/l is always used to project from singleton records. Similarly for variants, $[l \triangleright e]$ can have a type with multiple entries here, but the formalism requires explicit injections (*inj*) for creating non-singleton variants.

$area : \Sigma\, shape \to Float$
$area = (\lambda x.\ (x/rectangle/length) * (x/rectangle/width)) \bigtriangledown (\lambda y.\ 3.14 * (y/circle/radius) ** 2)$

The *area* function effectively matches the actual value of its input. It does so by combining ($\bigtriangledown$) two functions: one for *rectangle* values ($\Sigma\langle rectangle \triangleright \Pi\langle length \triangleright Float, width \triangleright Float\rangle\rangle \to Float$) and another for *circle* values ($\Sigma\langle circle \triangleright \Pi\langle radius \triangleright Float\rangle\rangle \to Float$). We again use the ($/$) operator to access a field from a variant. Therefore, we have ($area\ r$) return 6.0.

Row polymorphism. Now, consider: what type should we give to *getName*? A simple choice like $\Pi\, pet \to String$ would correctly type ($getName\ alice$). However, *getName* fundamentally only requires its argument to have a *name* field, regardless of any other fields the argument may contain.

Row polymorphism offers a more explicit and powerful approach to typing such polymorphic functions. Following Morris and McKinna [2019], we express two key relations on rows, *containment* and *combination*, as predicates in *qualified types* [Jones 2003a].

Specifically, since the argument to *getName* must contain a *name* field of some type a, we represent the input row as a variable r and impose a constraint on it:

$getName : \forall(a : \star)\ (r : R).\ \langle name \triangleright a\rangle \lesssim r \Rightarrow \Pi\, r \to a$

Here, *getName* is polymorphic over two type variables a and r with different *kinds*: $\star$ is the base kind for types, while R is the kind for rows; we say r is a *row variable*. Moreover, $\langle name \triangleright a\rangle \lesssim r$ is a *row containment* constraint, signifying that the row r must contain $\langle name \triangleright a\rangle$. The function can thus take any record $\Pi\, r$ with a *name* field, and return a value of type a.

Row polymorphism is useful for preserving information about the original argument. For example, consider a function that returns both the argument's *name* field and the original record itself:

$getNameAndRecord : \forall(a : \star)\ (r : R).\ \langle name \triangleright a\rangle \lesssim r \Rightarrow \Pi\, r \to (a, \Pi\, r)$
$getNameAndRecord = \lambda x.\ (x/name, x)$

In this case, *getNameAndRecord*'s return type accurately preserves the full type of the argument x. If we were to solely rely on structural subtyping with a type like $\forall a.\ \Pi\langle name \triangleright a\rangle \to (a, \Pi\langle name \triangleright a\rangle)$ for the function, we would lose information about any additional fields in the record.

Now we turn to the *row combination* constraint.[2] We have seen how the ($+\!\!+$) operator is useful for concatenating records for extension, but this operation can also introduce complexities. Consider the following function adapted from Wand [1991]:

$\lambda x\ y.\ (x +\!\!+ y)/name$

This function first concatenates two records, x and y, and then accesses the *name* field from the resulting concatenated record. The challenge lies in determining its type: specifically, we know that either x or y must contain a *name* field, but we do not know which one.

A row combination constraint ($r1 \odot r2 \sim r3$) states that concatenating two rows, $r1$ and $r2$, yields $r3$. Expressing row combination this way allows for various distinct interpretations of record extensions, such as how duplicate labels are handled, which can be implemented as different constraint resolution approaches. With this, we can precisely express the function's type as:[3]

$\forall(a : \star)\ (r1\ r2\ r3 : R).\ (r1 \odot r2 \sim r3, \langle name \triangleright a\rangle \lesssim r3) \Rightarrow \Pi\, r1 \to \Pi\, r2 \to a$

[2]While $r1 \lesssim r$ is often only satisfiable when there exists a $r2$ such that $r1 \odot r2 \sim r$, $r1 \lesssim r$ more closely reflects the term structure and thus the constraint elaboration (§5) [Morris and McKinna 2019]. Moreover, as we will see in §2.2, non-commutative row containment does not straightforwardly correspond to row combination.
[3]We often write $(a\ b\ c : k)$ as a shorthand for $(a : k)\ (b : k)\ (c : k)$.

This type incorporates two key constraints: $(r1 \odot r2 \sim r3)$ expresses that $r3$ is the result of concatenating rows $r1$ and $r2$, and $(\langle name \triangleright a \rangle \lesssim r3)$ expresses that the combined row $r3$ must have a *name* field of type a. The function then takes two records $\Pi\, r1$ and $\Pi\, r2$, and produces a value of a.

First-class labels. So far we have been working with concrete labels; e.g. *getName* always gets the *name* field. Our system supports *first-class labels* [Leijen 2004], thus allowing labels to be passed as arguments to functions or returned as results, just like any other value.

First-class labels are useful for defining functions that are generic over labels. For example, it lets us assign the following type to the record field access operator $(/)$:

$$(/) : \forall(l : L)\ (a : \star)\ (r : R).\ \langle l \triangleright a \rangle \lesssim r \Rightarrow \Pi\, r \to \lfloor l \rfloor \to a$$

Here, the function is polymorphic over three type variables: a *label variable* l of kind L, a row variable r of kind R, and a type variable a of kind $\star$. The constraint $\langle l \triangleright a \rangle \lesssim r$ indicates that the row r must contain a field l with type a. The function then accepts a record of type $\Pi\, r$, a label of type $\lfloor l \rfloor$, and returns a value of type a. Note that $\lfloor l \rfloor$ is the *singleton type* for label l.[4] Thus if $l : L$, then $\lfloor l \rfloor : \star$. For instance, we have $name : \lfloor name \rfloor$. From this, we can derive the type we have seen for $getName : \forall(a : \star)\ (r : R).\ \langle name \triangleright a \rangle \lesssim r \Rightarrow \Pi\, r \to a$.

We can extend first-class labels to *first-class rows* [Paszke and Xie 2023], denoting singleton row types by $\lfloor r \rfloor$. Often, though, we only need the label information from these rows, as the type information can typically be retrieved from the records or variants where the row is used, much like how a label l is used in the $(/)$ operator. Therefore, in our system, we model first-class rows simply by records whose fields always map to the unit type. For example, $\lfloor \langle name \triangleright String, age \triangleright Int \rangle \rfloor$ denotes $\Pi \langle name \triangleright Unit, age \triangleright Unit \rangle$. Thus, if $r : R$, then $\lfloor r \rfloor : \star$. On the term level, correspondingly, we write $\{name, age\}$ to denote $\{name \triangleright (), age \triangleright ()\}$. Instead of treating these as primitive constructs, we will see how they can be defined in §2.2.

2.2 Extensible Rows with Ad-hoc Polymorphism

Having discussed extensible rows, we now turn to ad-hoc polymorphism via type classes, exploring their various forms of combination with row polymorphism.

Ad-hoc polymorphism à la type classes. Consider defining a function that compares whether one value is greater than another. Using *type classes* [Wadler and Blott 1989], we can write:

$$compare : \forall(a : \star).\ Ord\ a \Rightarrow a \to a \to Bool$$
$$compare = \lambda x\ y.\ (x > y)$$

Here, $(Ord\ a)$ is a type class constraint which requires that any instantiation of a must be an instance of the *Ord* type class, which provides the $(>)$ method. For example, we expect *Int* and *String* to be instances of *Ord*, but not functions, as directly comparing two functions is generally challenging. For illustration purposes, we will use common type classes in our examples, including *Ord*, *Eq*, etc. Type classes are typically implemented via *dictionary-passing elaboration*, where evidence (called *dictionaries*) for type classes is explicitly propagated during a program elaboration process which translates the source into a simpler target language without type classes.

Putting row and type class constraints together. By combining row polymorphism and ad-hoc polymorphism, we can define useful ad-hoc polymorphic functions over row types such as:

$$compare\_at : \forall(l : L)\ (a : \star)\ (r : R).\ (Ord\ a, \langle l \triangleright a \rangle \lesssim r) \Rightarrow \lfloor l \rfloor \to \Pi\, r \to \Pi\, r \to Bool$$
$$compare\_at = \lambda l\ x\ y.\ (x/l) > (y/l)$$

[4] Avoid confusing singleton types, i.e. types with a single inhabitant, with singleton rows, i.e. rows with a single field.

Notably, the function incorporates *both* row constraints and type class constraints: the $(\mathit{Ord}\ a)$ constraint requires a to be an instance of *Ord*, while $(\langle l \triangleright a\rangle \lesssim r)$ requires r to contain $\langle l \triangleright a\rangle$. The function then takes a first-class label l of type $\lfloor l \rfloor$, and two records x and y of type $\Pi\, r$, and returns a *Bool* indicating whether x's l field is greater than y's.

As an example, recall that *alice* has type $\Pi\langle \mathit{name} \triangleright \mathit{String}, \mathit{age} \triangleright \mathit{Int}, \mathit{weight} \triangleright \mathit{Float}\rangle$ (§2.1). Supposing *Float* is an instance of *Ord*, and *bob* is another record of the same type as *alice*, we can compare *alice* and *bob* by their *weight*s using:

$\mathit{compare\_at}\ \mathit{weight}\ \mathit{alice}\ \mathit{bob}$

Combining type class constraints and row polymorphism. Now, let's consider how we may compare pets for equivalence. A straightforward approach involves defining a function that compares two pets field by field:

$$\mathit{eq\_pet} : \Pi\, \mathit{pet} \to \Pi\, \mathit{pet} \to \mathit{Bool}$$
$$\mathit{eq\_pet} = \lambda x\ y.\ (x/\mathit{name}) == (y/\mathit{name})\ \ \&\&\ (x/\mathit{age}) == (y/\mathit{age})\ \ \&\&\ (x/\mathit{weight}) == (y/\mathit{weight})$$

However, defining such a function for every record type can quickly become cumbersome. Instead, we would prefer a general *eq* function that can compare fields within any given record type. This presents challenges, especially with row polymorphism, where the exact fields within a row may not be known. Moreover, we may expect some fields in a row to be comparable, but not others.

Our system supports ad-hoc polymorphism over row polymorphism. Specifically, assuming *Eq* is a type class that provides the equality operator (==), we can define a generalized equality function for row polymorphism with the following type:

$$\mathit{eq}_\Pi : \forall(\mathit{r1}\ r : R).\ (\mathit{r1} \lesssim r,\ \mathit{All}\ \mathit{Eq}\ \mathit{r1}) \Rightarrow \lfloor \mathit{r1} \rfloor \to \Pi\, r \to \Pi\, r \to \mathit{Bool}$$

Here, the constraint $(\mathit{r1} \lesssim r)$ indicates that r contains *r1*. Additionally, the constraint $(\mathit{All}\ \mathit{Eq}\ \mathit{r1})$ requires all fields within *r1* to be instances of *Eq*. The function then receives an argument of type $\lfloor \mathit{r1} \rfloor$, two records of type $\Pi\, r$, and returns a *Bool*. As an example:

$\mathit{eq}_\Pi\ \{\mathit{name}, \mathit{age}\}\ \mathit{alice}\ \mathit{bob}$

compares *alice* and *bob* based on their names and ages, but not their weights.

This form of constraint, $(\mathit{All}\ \mathit{Eq}\ \mathit{r1})$, is novel in our system, allowing us to express type class constraints over a polymorphic row. This constraint is satisfiable when all fields within *r1* individually satisfy *Eq*, thus allowing us to define functions polymorphic over rows where a specific property holds across all fields, as exemplified by eq_Π. It is crucial to distinguish between a constraint like $(\mathit{All}\ \ C\ \ r)$ and a constraint like $C\ \ (\Pi\ r)$. The former is derived automatically from individual *C* instances for each field in r, a process that becomes evident during program elaboration. In contrast, the latter represents a single type class instance for the type $\Pi\, r$. For instance, if *Ord* is defined across r using $(\mathit{All}\ \mathit{Ord}\ r)$, many different ordering functions can be implemented—such as component-wise, lexicographical, or those based on specific distance metrics. Conversely, $\mathit{Ord}\ \ (\Pi\, r)$ would provide only a single, monolithic ordering definition.

Row constraints and commutativity. We have provided the type signature of *eq*; now, let's try to define it. As a first step, we can introduce a primitive that lifts a function (e.g. (==)) from operating on a type to operating on all fields within a row (e.g. *r1*), accumulating the results. However, before we introduce the primitive, a question is immediately raised: for a given row r, what order should we use to apply the function to its fields? While in the case of *Eq*, the order may not matter, since equivalence comparisons are order-independent, consider a *print* function below:

$$\mathit{print} : \forall(r : R).\ (\mathit{All}\ \mathit{Show}\ r) \Rightarrow \Pi\, r \to \mathit{String}$$

This function prints a record, where all fields in row r are instances of the *Show* type class. For printing records, establishing a fixed order and thus *deterministic* behaviour would be valuable.

To this end, our system supports explicit *commutativity* annotations. Specifically, the type system incorporates commutativity c and non-commutativity n as types with a kind U. While rows are easier to work with when handled commutatively, since they are considered equivalent up to reordering, rows explicitly carry ordering information when treated as non-commutative.[5]

We then generalize row constraints by incorporating these commutativity annotations. For example, the following holds, where row combination is commutative:

$$\langle name \triangleright String \rangle \odot_c \langle age \triangleright Int, weight \triangleright Float \rangle \sim \langle weight \triangleright Float, name \triangleright String, age \triangleright Int \rangle$$

while the following does not, as non-commutative combination requires *name* to appear before both *age* and *weight*, and *age* to appear before *weight*:

$$\langle name \triangleright String \rangle \odot_n \langle age \triangleright Int, weight \triangleright Float \rangle \sim \langle weight \triangleright Float, name \triangleright String, age \triangleright Int \rangle$$

Notably, non-commutative containment does not require the contained row to be a *continuous* subset of the larger one, meaning some non-commutative containments cannot be expressed with a single non-commutative concatenation. For example, the following holds:

$$\langle name \triangleright String, weight \triangleright Float \rangle \lesssim_n \langle name \triangleright String, age \triangleright Int, weight \triangleright Float \rangle$$

while the following does not, since the entries in the contained row must appear in the same order as they do in the larger one:

$$\langle age \triangleright Int, name \triangleright String \rangle \lesssim_n \langle name \triangleright String, age \triangleright Int, weight \triangleright Float \rangle$$

Moreover, records, as well as variants, can also carry explicit commutativity annotations. Since we incorporate commutativities as types, we can express a commutativity-polymorphic function:

$$splitName : \forall (a : \star)\ (r\ r1 : R)\ (\mu : U).\ (\langle name \triangleright a \rangle \odot_n r1 \sim r) \Rightarrow \Pi_\mu\, r \to (a, \Pi_\mu\, r1)$$

This function requires that concatenating $\langle name \triangleright a \rangle$ and $r1$, as non-commutative rows, yields r. Therefore, if μ is instantiated to c (commutative), commutativity can automatically rearrange the record to bring *name* to the front. On the other hand, if μ is instantiated to n (non-commutative), this function only applies when *name* is precisely the first field in the record.

Furthermore, we consider non-commutative records (or variants, respectively) as *subtypes* of commutative ones. Intuitively, non-commutative records carry more information due to the inherent ordering between fields, and this additional information can be "forgotten" through subtyping. To our knowledge, our system is the first extensible row type system to feature both the commutativity hierarchy and polymorphism over commutativity.

Lastly, we note that first-class rows always have their orders fixed. For instance, $\lfloor \langle name \triangleright String, age \triangleright Int \rangle \rfloor$ denotes $\Pi_n \langle name \triangleright Unit, age \triangleright Unit \rangle$. This fixed ordering often aligns with the intended use of first-class rows, which can also be made commutative through subtyping.

Folding over rows. We are now ready to introduce our operator *ind* (short for "induction"), which intuitively takes a base value and folds over a row's entries. Using *ind*, we define eq_Π as follows:

$$eq_\Pi : \forall (r1\ r : R).\ (r1 \lesssim_c r,\ All\ Eq\ r1) \Rightarrow \lfloor r1 \rfloor \to \Pi_c\, r \to \Pi_c\, r \to Bool$$
$$eq_\Pi = \lambda (w : \lfloor r1 \rfloor)\ x\ y.\ ind\ (\lambda a : R.\ Bool)\ r1\ (\lambda l\ acc.\ acc\ \&\&\ (x/l == y/l))\ True$$

[5] Alternatively, instead of using annotations on the *type* level, we may track commutativity in the *kind* of the row, which would allow folding to be limited to non-commutative rows. However, that would then require a non-commutative row even when the ordering does not matter. Also, commutativity polymorphism would likely require kind polymorphism, as the row kind would depend on commutativity.

Note that the function takes a w of type $\lfloor r1 \rfloor$ and two records of type $\Pi_c\ r$, returning a *Bool*. Commutativity polymorphism is unnecessary here since row types do not appear in the output, so the function can still be used with non-commutative inputs via subtyping. The heavy lifting is performed by *ind*, which takes four arguments in the form $ind\ (\lambda a : R.\ t)\ r\ e1\ e2$. Here, $(\lambda a : R.\ t)$ is a *type-level function*, r is a row type of kind R, $e1$ is the folding function, and $e2$ is the initial value, with the following types:[6]

$$e1 : \forall (r1\ r2\ r3 : R)\ (l : L)\ (b : \star).\ (r1 \odot_n \langle l \triangleright b \rangle \sim r2, r2 \odot_n r3 \sim r) \Rightarrow \lfloor l \rfloor \to t\ [r1/a] \to t\ [r2/a]$$
$$e2 : t\ [\langle\rangle/a]$$

At each step, $e1$ receives a label $\lfloor l \rfloor$, as well as the accumulated value of type $t\ [r1/a]$ (where $[r1/a]$ denotes type substitution). It then returns a new value of type $t\ [r2/a]$, where $r2$ is the result of concatenating $r1$ and $\langle l \triangleright b \rangle$. The second constraint states that $r2$ and $r3$ concatenate to r, so $r3$ contains the entries in r which have not yet been processed. Notably, we allow the type t to depend on the type of the row that has been folded. In the case of *eq*, t is simply *Bool*; we will explore examples where this substitution proves useful later. Thus, $e2$ as the initial value has type $t\ [\langle\rangle/a]$, where a is substituted by the empty row. Moreover, the row constraints in $e1$ are non-commutative. This means that once r is fixed, *ind* will fold over rows in a predetermined order. Lastly, we remark that the argument w is not used, as its primary purpose is to provide the singleton type $\lfloor r1 \rfloor$, which is a common pattern with singleton types. Note that w is explicitly annotated to allow the type variable $r1$ to be passed as an argument to *ind*. For the remainder of this section, we will often omit type annotations and assume type variables from the signature are automatically brought into scope; this behaviour is consistent with our formalism.

As another example, we define *print* as follows:

$$print : \forall (r : R).\ (All\ Show\ r) \Rightarrow \Pi_n\ r \to String$$
$$print = \lambda x.\ ind\ (\lambda r : R.\ String)\ r$$
$$(\lambda l\ acc.\ \textbf{if}\ acc == ""\ \textbf{then}\ show\ (x/l)\ \textbf{else}\ acc +\!\!+ ",\ " +\!\!+ show\ (x/l))\ ""$$

Here, *print* takes a non-commutative record $\Pi_n\ r$, and provides *ind* with the order of r, a function that accumulates the strings, and an empty string as the initial value.

The *ind* construct is highly expressive, allowing us to fold over not just records, but also variants. For instance, we can define a general equality function eq_Σ for variants as follows:

$$eq_\Sigma : \forall (r : R).\ All\ Eq\ r \Rightarrow \Sigma_c\ r \to \Sigma_c\ r \to Bool$$
$$eq_\Sigma = ind\ (\lambda a : R.\ \Sigma_c\ a \to \Sigma_c\ r \to Bool)\ r$$
$$(\lambda l\ acc.\ acc\ \triangledown\ (\lambda x.\ (\lambda y.\ False)\ \triangledown\ (\lambda y.\ x/l == y/l)\ \triangledown\ (\lambda y.\ False)))\ (\lambda x\ y.\ True)$$

Here, we construct a function that compares x with y by folding over all possible cases of x. At each step, $(\lambda a : R.\ \Sigma_c\ a \to \Sigma_c\ r \to Bool)$ constructs a function. This function takes cases of x corresponding to the subrow $\Sigma_c\ a$, along with y of type $\Sigma_c\ r$, and returns a *Bool*. The initial value corresponds to when x is an empty variant, which should never occur, so it simply returns *True*. The folding function, at each step, takes a label l and the accumulated function *acc*, and extends the function to handle the case when x is l. The body is another function of type $\Sigma_c\ r \to Bool$. The inner function is also composed using $(\triangledown)$: it returns $(x/l == y/l)$ when y is also l, and *False* otherwise. Intuitively, the final result is a function with nested matches: first on x, and then on y. The function returns *True* only if x and y have the same label, and that label maps to equivalent values.

Lastly, we note that commutative records or variants can be made non-commutative using *ind*. In particular, we define *order* r which simply expands to the following definitions:

[6] With higher kinds, $r1, r2, r3$ will all be of kind R^k for some k; see §3 for the complete typing rules.

$(order_{\Pi}\ r) :: \Pi_c\, r \to \Pi_n\, r \triangleq \lambda x.\ ind\ (\lambda a : R.\ \Pi_n\, a)\ r\ (\lambda l\ acc.\ acc +\!\!+ \{\, l \triangleright x/l \,\})\ \{\,\}$

$(order_{\Sigma}\ r) :: \Sigma_c\, r \to \Sigma_n\, r \triangleq ind\ (\lambda a : R.\ \Sigma_n\, a)\ r\ (\lambda l\ acc.\ acc \bigtriangledown (\lambda x.\ [\, l \triangleright x/l \,]))\ (\lambda x.\ x)$

Notably, these functions must be used with care: the r parameter should be fixed, e.g. through its occurrence in a non-commutative record, to ensure that it can be instantiated deterministically.

Now that we have covered type classes over simple types, we turn our focus to type classes over type constructors. Before that, we need type-level lifting.

Mapping, lifting, and higher kinds. Consider a row $\langle name \triangleright String, age \triangleright Int \rangle$. Suppose we would like to apply *Maybe* to each field, effectively turning each field into a maybe value: *name* maps to type *Maybe String*, and *age* maps to type *Maybe Int*.

We provide an explicit constructor, called *Lift*, which acts as a type-level mapping [Chlipala 2010; Hubers and Morris 2023]. For example, *Lift Maybe* $\langle name \triangleright String, age \triangleright Int \rangle$[7] returns $\langle name \triangleright Maybe\ String, age \triangleright Maybe\ Int \rangle$. We can implement a function with such a type as:

$lift\_maybe : \forall (r : R)\ (\mu : U).\ \Pi_\mu\, r \to \Pi_\mu\, (Lift\ Maybe\ r)$
$lift\_maybe = \lambda x.\ ind\ (\lambda a : R.\ \Pi_\mu\, (Lift\ Maybe\ a))\ r\ (\lambda l\ acc.\ acc +\!\!+ \{\, l \triangleright Just\ (x/l) \,\})\ \{\,\}$
$lift\_maybe\ \{\, name \triangleright \text{“}alice\text{”}, age \triangleright 2 \,\}$ -- $\{\, name \triangleright Just \text{“}alice\text{”}, age \triangleright Just\ 2 \,\}$

While we have primarily discussed rows with fields of kind $\star$ so far, the *Lift* operator also makes it useful to support *higher-kinded* rows. Specifically, given an *Int* and a higher-kinded row:

$\langle stack \triangleright List, optional \triangleright Maybe \rangle : R^{\star \to \star}$

applying $Lift\ (\lambda a.\ a\ Int) \langle stack \triangleright List, optional \triangleright Maybe \rangle$ yields $\langle stack \triangleright List\ Int, optional \triangleright Maybe\ Int \rangle$.

There are two notable things. First, the row's fields are of kind $\star \to \star$. We denote the kind of such a row as $R^{\star \to \star}$. In other words, the kind R now explicitly carries a kind annotation; we often omit this annotation when it can be inferred from the context. Second, $(\lambda a.\ a\ Int)$ is a type-level function[8], which can be fully annotated as $(\lambda a : \star \to \star.\ a\ Int)$. More generally, *Lift* takes a type function of kind $k_1 \to k_2$, a row of kind R^{k_1}, and returns a row of kind R^{k_2}.

We are now ready to define the first-class row operator $\lfloor r \rfloor$ (§2.1). The type-level operator $\lfloor r \rfloor$ can be expressed as $\Pi_n\ (Lift\ (\lambda a.\ Unit)\ r)$. Similarly, we can define a corresponding term-level operator that applies to non-commutative records and turns each field into a $()$.[9]

Functors and monads. We now present the definition of *fmap*, which maps a function f over a record whose fields are types where the type constructors are functors:

$fmap_{\Pi} : \forall (r : R^{\star \to \star})\ (a\ b : \star)\ (\mu : U).\ (All\ Functor\ r)$
$\quad \Rightarrow (a \to b) \to \Pi_\mu\, (Lift\ (\lambda c.\ c\ a)\ r) \to \Pi_\mu\, (Lift\ (\lambda c.\ c\ b)\ r)$
$fmap_{\Pi} = \lambda f\ x.\ ind\ (\lambda a.\ \Pi_\mu\, (Lift\ (\lambda c.\ c\ b)\ a))\ r\ (\lambda l\ acc.\ acc +\!\!+ \{\, l \triangleright fmap\ f\ (x/l) \,\})\ \{\,\}$

Here, r is a row of kind $\star \to \star$. The input record x has type $\Pi_\mu\, (Lift\ (\lambda c.\ c\ a)\ r)$, meaning each field's type is a field from r applied to a. The resulting record's fields are then the corresponding fields from r applied to b. The definition simply uses *ind* to apply *fmap f* to each field.

Similarly, we can define $fmap_{\Sigma}$ that maps over a variant:

$fmap_{\Sigma} : \forall (r : R^{\star \to \star})\ (a\ b : \star)\ (\mu : U).\ (All\ Functor\ r)$
$\quad \Rightarrow (a \to b) \to \Sigma_\mu\, (Lift\ (\lambda c.\ c\ a)\ r) \to \Sigma_\mu\, (Lift\ (\lambda c.\ c\ b)\ r)$

[7] We could make the lifting operation implicit, writing $Maybe\ \langle name \triangleright String, age \triangleright Int \rangle$ directly [Hubers and Morris 2023]. However, we choose to keep lifting explicit for clarity as well as consistency with other constructs like *ind*.
[8] Similarly, *Maybe* can be eta-expanded to $\lambda a.\ Maybe\ a$.
[9] We can define the term-level operator $\lambda (x : \Pi_n\, r).\ ind\ (\Pi_n\, (Lift\ (\lambda a.\ Unit)))\ r\ (\lambda l\ acc.\ acc +\!\!+ \{\, l \triangleright () \,\})\ \{\,\}$.

$$\begin{aligned}&\mathit{fmap}_{\Sigma} = \lambda f\ x.\ \mathit{ind}\ (\lambda d.\ \Sigma_{\mu}\,(\mathit{Lift}\ (\lambda c.\ c\ a)\ d) \to \Sigma_{\mu}\,(\mathit{Lift}\ (\lambda c.\ c\ b)\ d))\\ &\qquad r\ (\lambda l\ \mathit{acc}.\ \mathit{acc} \triangledown (\lambda y.\ [\,l \triangleright \mathit{fmap}\ f\ (y/l)\,]))\ (\lambda y.\ y)\ x\end{aligned}$$

Lastly, let us consider monads as another example, here focusing on their definitions for records.

$$\begin{aligned}&\mathit{return}_{\Pi} : \forall (r : R^{\star\to\star})\ (a : \star).\ (\mathit{All}\ \mathit{Monad}\ r) \Rightarrow a \to \Pi_c\,(\mathit{Lift}\ (\lambda b.\ b\ a)\ r)\\ &\mathit{return}_{\Pi} = \lambda x.\ \mathit{ind}\ (\lambda c.\ \Pi_c\,(\mathit{Lift}\ (\lambda b.\ b\ a)\ c))\ r\ (\lambda l\ \mathit{acc}.\ \mathit{acc} +\!\!+ \{\,l \triangleright \mathit{return}\ x\,\})\ \{\,\}\end{aligned}$$

The return_{Π} function takes a value x of type a, and constructs a record where each field maps to $(\mathit{return}\ x)$, where $r : R^{\star\to\star}$ and all its fields are *Monad* instances. As examples, we have:

$$r \triangleq \mathit{return}_{\Pi}\ 2 : \Pi_c \langle \mathit{maybe} \triangleright \mathit{Maybe}\ \mathit{Int}, \mathit{list} \triangleright \mathit{List}\ \mathit{Int} \rangle \quad \text{-- } \{\,\mathit{maybe} \triangleright \mathit{Just}\ 2, \mathit{list} \triangleright [\,2\,]\,\}$$

$$\mathit{fmap}_{\Pi}\ (+1)\ r \quad \text{-- } \{\,\mathit{maybe} \triangleright \mathit{Just}\ 3, \mathit{list} \triangleright [\,3\,]\,\}$$

The definition of bind_{Π} is given as follows, where $(\gg\!=)$ is the bind operator of *Monad*:

$$\begin{aligned}&\mathit{bind}_{\Pi} : \forall (r : R^{\star\to\star})\ (a\ b : \star)\ (\mu : U).\ (\mathit{All}\ \mathit{Monad}\ r)\\ &\qquad \Rightarrow \Pi_{\mu}\,(\mathit{Lift}\ (\lambda c.\ c\ a)\ r) \to \Pi_{\mu}\,(\mathit{Lift}\ (\lambda c.\ a \to c\ b)\ r) \to \Pi_{\mu}\,(\mathit{Lift}\ (\lambda c.\ c\ b)\ r)\\ &\mathit{bind}_{\Pi} = \lambda x\ f.\ \mathit{ind}\ (\lambda a.\ \Pi_{\mu}\,(\mathit{Lift}\ (\lambda c.\ c\ b)\ a))\ r\ (\lambda l\ \mathit{acc}.\ \mathit{acc} +\!\!+ \{\,l \triangleright ((x/l) \gg\!= (f/l))\,\})\ \{\,\}\end{aligned}$$

Unlifting row types. We have seen how *lift_maybe* converts a record of values into a record of *Maybe* values. Now, consider a reverse operator that converts a record of *Maybe* values back into a record of concrete values. This is not always possible though, since if a field maps to *Nothing*, we simply cannot recover a value of its original type. Therefore, we instead consider a function that takes a list of records with *Maybe* values, filters out entries that contain *Nothing* in one of their fields, and then returns a list containing only the remaining records with unwrapped values:

$$\begin{aligned}&\mathit{unlift\_maybe} : \forall (r : R)\ (\mu : U).\ \mathit{List}\ (\Pi_{\mu}\,(\mathit{Lift}\ \mathit{Maybe}\ r)) \to \mathit{List}\ (\Pi_{\mu}\,r)\\ &\mathit{unlift\_maybe} = \lambda x.\ \mathit{map}\ (\lambda y.\ \mathit{ind}\ \_\ \_\ (\lambda l\ \mathit{acc}.\ \mathit{acc} +\!\!+ \{\,l \triangleright \mathit{fromJust}\ y/l\,\})\ \{\,\})\ \$\ \mathit{filter}\ \mathit{complete}\ x\\ &\mathit{complete} : \forall (r : R)\ (\mu : U).\ (\Pi_{\mu}\,(\mathit{Lift}\ \mathit{Maybe}\ r)) \to \mathit{Bool}\\ &\mathit{complete} = \lambda y.\ \mathit{ind}\ (\lambda a.\ \mathit{Bool})\ r\ (\lambda l\ \mathit{acc}.\ \mathit{acc}\ \&\&\ \mathit{isJust}\ (y/l))\ \mathit{True}\end{aligned}$$

Here, *complete* checks if a record contains *Nothing* in any of its fields. After filtering out these records, *unlift_maybe* maps over the list, applying *ind* to extract the field values from their *Just* wrapper.

However, *unlift_maybe* requires a record to contain *Maybe* values in all of its fields. More commonly, only some fields may have missing data, while others are guaranteed to be present. For example, we may always expect a *pet* to have a *name*, but its *weight* information could be missing. Therefore, it can be useful to split rows into two parts: those with *Maybe* fields and those without. This way, *unlift_maybe* can be applied specifically to the fields containing *Maybe* values.

To this end, our system introduces an additional constraint of the form $\mathit{Split}\ (\lambda a : k.\ t)\ r1\ r2\ r$. Specifically, a $\mathit{Split}\ (\lambda a{:}k.\ t)\ r1\ r2\ r$ constraint looks at the types of r's fields. If a field's type matches t after substituting a with some type t', then the field mapping to t' is placed in *r1*. Otherwise the original field is placed in *r2*. As an example, we have:

$$\mathit{Split}\ \mathit{Maybe}\ \langle \mathit{weight} \triangleright \mathit{Float} \rangle\ \langle \mathit{name} \triangleright \mathit{String} \rangle\ \langle \mathit{name} \triangleright \mathit{String}, \mathit{weight} \triangleright \mathit{Maybe}\ \mathit{Float} \rangle$$

Notice that first row contains $\langle \mathit{weight} \triangleright \mathit{Float} \rangle$, instead of $\langle \mathit{weight} \triangleright \mathit{Maybe}\ \mathit{Float} \rangle$. In other words, $\mathit{Split}\ (\lambda a : k.\ t)\ r1\ r2\ r$ implies $((\mathit{Lift}\ (\lambda a : k.\ t)\ r1) \odot r2 \sim r)$, rather than $(r1 \odot r2 \sim r)$.

This allows us to define a generalized version of *unlift_maybe*:

$$\begin{aligned}&\mathit{unlift\_maybe'} : \forall (r\ r1\ r2\ r' : R)\ (\mu : U).\ (\mathit{Split}\ \mathit{Maybe}\ r1\ r2\ r, r1 \odot_c\ r2 \sim r')\\ &\quad \Rightarrow \mathit{List}\ (\Pi_{\mu}\,r) \to \mathit{List}\ (\Pi_{\mu}\,r')\end{aligned}$$

The constraint *Split Maybe r1 r2 r* splits *r*, according to whether the field's type matches *Maybe*, into *r1* and *r2*. The function then takes a list of the original record type $\Pi_\mu\, r$, and produces a list of type $\Pi_\mu\, r'$. For example:

```
pets = [ { name ▷ "alice", weight ▷ Just 2.4 }, { name ▷ "bob", weight ▷ Nothing }
       , { name ▷ "carol", weight ▷ Just 3.6 } ]
unlift_maybe' pets    -- [ { name ▷ "alice", weight ▷ 2.4 }, { name ▷ "carol", weight ▷ 3.6 } ]
```

The *Split* constraint differs from the *All* constraint in a few key ways. *All C r* takes a constraint *C* and requires all fields in *r* to satisfy it. In contrast, *Split* $(\lambda a : k.\ t)$ *r1 r2 r* takes a type abstraction, and splits *r* based on its type information. In languages with expressive type systems, where type equivalence can be expressed as constraints, we could potentially express *Split* $(\lambda a : k.\ t)$ *r1 r2 r* as (*Lift* $(\lambda a : k.\ t)$ *r1* $\odot_c$ *r2* $\sim$ *r*), (*All* $(\lambda c.\ \nexists\ t'.\ subst\ a\ t'\ t == c)$ *r2*), assuming (*subst a t' t*) substitutes *a* for *t'* in *t*, and (==) denotes type-level equivalence. Notably, *Split* also conveys negative information, since *r2* must not match *t*.

We can define the corresponding split operator, with which we define *unlift_maybe'*:

```
split_Π ϕ : ∀(r r1 r2 : R). Split ϕ r1 r2 r ⇒ Π_c r → { match ▷ Π_c (Lift ϕ r1), rest ▷ Π_c r2 }
  ≜ (λr. { match ▷ (r : Π_c (Lift ϕ r1)), rest ▷ (r : Π_c r2) })
unlift_maybe' r = let r' = split_Π Maybe r in (unlift_maybe (r'/match)) ++ (r'/rest)
```

3 Declarative Type System

This section presents the type system of the source calculus $\lambda^{\Rightarrow}_{\rho}$, which incorporates row polymorphism and type classes. The dynamic semantics will be defined later in §5.

3.1 Syntax

Fig. 1 presents the syntax of $\lambda^{\Rightarrow}_{\rho}$. A program *pgm* consists of a sequence of **class** and **instance** declarations, followed by a term. A type class declaration defines a type class for types of kind κ, with the overline notation denoting that any number of superclasses TC'_i are allowed. For simplicity, we assume each type class has a single method *m*, though the system could easily be extended. Instances can be qualified with any number of prerequisite constraints ψ_i.

Terms M include term variables x, type class methods m, lambdas $\lambda\, x.\, M$, applications $M\, N$, let expressions $\mathbf{let}\, x : \sigma = M\, \mathbf{in}\, N$, expressions with type annotations $M : \sigma$, first-class labels ℓ, singleton products $\{M \triangleright N\}$ and sums $[M \triangleright N]$, unlabel operations M/N, projection $\mathbf{prj}\, M$[10], concatenation $M{+\!\!+}N$, injection $\mathbf{inj}\, M$, elimination $M \triangledown N$, and **ind** expressions.

Types are stratified: type schemes σ include polymorphic types $\forall\, a : \kappa.\, \sigma$ and qualified types γ. Qualified types γ include types $\psi \Rightarrow \gamma$ with a constraint ψ, and monotypes τ.

Monotypes τ represent types across various kinds. For clarity, we often use specific symbols to informally refer to types of particular kinds: we write ϕ for type-level functions, ρ for rows, ψ for constraints, ξ for labels, and μ for commutativities. Monotypes include type variables a, type applications $\phi\, \tau$, function types $\tau_0 \to \tau_1$, labels ℓ, singleton label types $\lfloor \xi \rfloor$, commutativities u, and rows $\langle \overline{\xi_i \triangleright \tau_i} \rangle$. Records or variants are denoted by $\Xi_\mu\, \rho$, using Π or Σ, respectively. The Ξ notation is associated with commutativity information μ, which can be polymorphic. A concrete commutativity u is either $\mathfrak{c}$ for commutative or $\mathfrak{n}$ for non-commutative. The **Lift** operation applies a type-level function over rows. For simplicity, we do not include general type-level lambdas, avoiding the

[10]The formalism includes explicit projection and injection, following Morris and McKinna [2019].

$$
\begin{array}{rcll}
\mathit{pgm} & ::= & \textbf{class}\ \overline{TC'_i\ a_i} \Rightarrow TC\ a : \kappa\ \textbf{where}\ \{m : \sigma\}; \mathit{pgm} \mid & \textit{program} \\
 & & \textbf{instance}\ \overline{\psi_i} \Rightarrow TC\ \tau\ \textbf{where}\ \{m = M\}; \mathit{pgm} \mid M & \\
M, N & ::= & x \mid m \mid \lambda\, x.\, M \mid M\, N \mid \textbf{let}\ x : \sigma = M\ \textbf{in}\ N \mid M : \sigma \mid \ell \mid & \textit{term} \\
 & & \{M \triangleright N\} \mid [M \triangleright N] \mid M/N \mid \textbf{prj}\, M \mid M +\!\!+ N \mid \textbf{inj}\, M \mid M \triangledown N \mid & \\
 & & \textbf{ind}\ (\lambda\, a : \kappa.\, \tau)\ \rho\ M\, N & \\
 & & & \\
\sigma & ::= & \forall\, a : \kappa.\, \sigma \mid \gamma & \textit{type scheme} \\
\gamma & ::= & \psi \Rightarrow \gamma \mid \tau & \textit{qualified type} \\
\tau, \phi, \rho, \psi, \xi, \mu & ::= & a \mid \phi\ \tau \mid \tau_0 \to \tau_1 \mid \ell \mid \lfloor \xi \rfloor \mid u \mid \langle \overline{\xi_i \triangleright \tau_i} \rangle \mid \Xi_\mu\, \rho \mid & \textit{monotype} \\
 & & \textbf{Lift}\ (\lambda\, a : \kappa.\, \tau)\ \rho \mid \rho_0 \lesssim_\mu \rho_1 \mid \rho_0 \odot_\mu \rho_1 \sim \rho_2 \mid TC\ \tau \mid & \\
 & & \textbf{All}\ (\lambda\, a : \kappa.\, \tau)\ \rho \mid \textbf{Split}\ (\lambda\, a : \kappa.\, \tau)\ \rho_0\ \rho_1\ \rho_2 & \\
u & ::= & \mathfrak{c} \mid \mathfrak{n} & \textit{commutativity} \\
\Xi & ::= & \Pi \mid \Sigma & \textit{prod or sum} \\
 & & & \\
\Gamma & ::= & \epsilon \mid \Gamma, a : \kappa \mid \Gamma, x : \sigma \mid \Gamma, \psi & \textit{type environment} \\
\Gamma_C & ::= & \epsilon \mid \Gamma_C, (\overline{TC'_i\ a_i} \Rightarrow TC\ a : \kappa) \mapsto m : \sigma & \textit{class environment} \\
\Gamma_I & ::= & \epsilon \mid \Gamma_I, (\forall\, \overline{a_i : \kappa_i}.\, \overline{\psi_i} \Rightarrow TC\ \tau) & \textit{instance environment} \\
 & & & \\
\kappa & ::= & \star \mid \kappa_0 \mapsto \kappa_1 \mid \mathbf{R}^\kappa \mid \mathbf{C} \mid \mathbf{L} \mid \mathbf{U} & \textit{kind}
\end{array}
$$

Fig. 1. Syntax

$\boxed{\Gamma_C; \Gamma \vdash \sigma : \kappa}$ *(Kinding)*

$$\textsc{label}\ \frac{}{\Gamma_C; \Gamma \vdash \ell : \mathbf{L}} \qquad \textsc{floor}\ \frac{\Gamma_C; \Gamma \vdash \xi : \mathbf{L}}{\Gamma_C; \Gamma \vdash \lfloor \xi \rfloor : \star} \qquad \textsc{comm}\ \frac{}{\Gamma_C; \Gamma \vdash u : \mathbf{U}}$$

$$\textsc{row}\ \frac{\overline{\Gamma_C; \Gamma \vdash \xi_i : \mathbf{L}} \qquad \vdash_{\mathcal{T}} \overline{\xi_i} \qquad \overline{\Gamma_C; \Gamma \vdash \tau_i : \kappa}}{\Gamma_C; \Gamma \vdash \langle \overline{\xi_i \triangleright \tau_i} \rangle : \mathbf{R}^\kappa}$$

$$\textsc{prodOrSum}\ \frac{\Gamma_C; \Gamma \vdash \mu : \mathbf{U} \qquad \Gamma_C; \Gamma \vdash \rho : \mathbf{R}^\star}{\Gamma_C; \Gamma \vdash \Xi_\mu\, \rho : \star} \qquad \textsc{lift}\ \frac{\Gamma_C; \Gamma, a : \kappa_0 \vdash \tau : \kappa_1 \qquad \Gamma_C; \Gamma \vdash \rho : \mathbf{R}^{\kappa_0}}{\Gamma_C; \Gamma \vdash \textbf{Lift}\ (\lambda\, a : \kappa_0.\, \tau)\ \rho : \mathbf{R}^{\kappa_1}}$$

$$\textsc{contain}\ \frac{\Gamma_C; \Gamma \vdash \mu : \mathbf{U} \qquad \Gamma_C; \Gamma \vdash \rho_0 : \mathbf{R}^\kappa \qquad \Gamma_C; \Gamma \vdash \rho_1 : \mathbf{R}^\kappa}{\Gamma_C; \Gamma \vdash \rho_0 \lesssim_\mu \rho_1 : \mathbf{C}} \qquad \textsc{concat}\ \frac{\Gamma_C; \Gamma \vdash \mu : \mathbf{U} \qquad \Gamma_C; \Gamma \vdash \rho_0 : \mathbf{R}^\kappa \qquad \Gamma_C; \Gamma \vdash \rho_1 : \mathbf{R}^\kappa \qquad \Gamma_C; \Gamma \vdash \rho_2 : \mathbf{R}^\kappa}{\Gamma_C; \Gamma \vdash \rho_0 \odot_\mu \rho_1 \sim \rho_2 : \mathbf{C}}$$

$$\textsc{tc}\ \frac{(\overline{TC'_i\ a} \Rightarrow TC\ a : \kappa) \mapsto m : \sigma \in \Gamma_C \qquad \Gamma_C; \Gamma \vdash \tau : \kappa}{\Gamma_C; \Gamma \vdash TC\ \tau : \mathbf{C}}$$

$$\textsc{all}\ \frac{\Gamma_C; \Gamma, a : \kappa \vdash \psi : \mathbf{C} \qquad \Gamma_C; \Gamma \vdash \rho : \mathbf{R}^\kappa}{\Gamma_C; \Gamma \vdash \textbf{All}\ (\lambda\, a : \kappa.\, \psi)\ \rho : \mathbf{C}} \qquad \textsc{split}\ \frac{\Gamma_C; \Gamma \vdash (\textbf{Lift}\ (\lambda\, a : \kappa.\, \tau)\ \rho_0) \odot_{\mathfrak{c}} \rho_1 \sim \rho_2 : \mathbf{C}}{\Gamma_C; \Gamma \vdash \textbf{Split}\ (\lambda\, a : \kappa.\, \tau)\ \rho_0\ \rho_1\ \rho_2 : \mathbf{C}}$$

Fig. 2. Kinding

need to handle them in arbitrary positions.[11] Monotypes also include several forms of constraints: containment $\lesssim$ and concatenation $\odot$ constraints which are also associated with commutativities, type class constraints $TC\ \tau$, **All** constraints, and **Split** constraints.

Moreover, we have three environments: a type environment Γ maps type variables to their kinds and term variables to their types, as well as keeping track of constraints. A class environment Γ_C stores type class declarations, and an instance environment Γ_I stores type class instance declarations.

Lastly, we employ a kind system to distinguish between types: kinds κ include $\star$ for the base kind, $\kappa_0 \mapsto \kappa_1$ for type-level functions and type constructors, $\mathbf{R}^\kappa$ for rows, $\mathbf{C}$ for constraints, $\mathbf{L}$ for labels, and $\mathbf{U}$ for commutativities. Fig. 2 presents selected kinding rules related to rows and constraints; most of them are straightforward. Rule ROW checks the labels and their types. Here, $\mathcal{T}$ represents

[11] Type inference for type-level lambdas is known to be generally *undecidable*; we leave type inference for future work. We also believe solving **Split** constraints in the presence of type-level lambdas would introduce similar challenges.

a *row theory* [Morris and McKinna 2019] which allows a language to be defined generically with respect to various aspects of rows. More concretely, we rely on the row theory to specify:

(1) a validity check for labels within a row (e.g. ROW in Fig. 2);
(2) a predicate which can restrict reordering for commutative rows (e.g. COMM in Fig. 4); and
(3) a constraint solver (e.g. QUALE in Fig. 3), allowing for varied interpretations of row constraints.

For rule PRODORSUM, the row must have kind $\mathbf{R}^\star$ since its entries describe terms within products or variants. In contrast, CONTAIN and CONCAT allow rows of any kind. Lastly, rule SPLIT checks **Split** simply by checking a concatenation constraint, as they share the same kinding requirements.

3.2 Typing

Fig. 3 presents the term typing rules. Program typing rules (along with their elaborations) are omitted; these rules are standard, simply extending the class and instance environments, and can be found in the appendix. The judgement $\Gamma_I; \Gamma_C; \Gamma \vdash M : \sigma$ reads that under contexts Γ_I, Γ_C and Γ, term M has type σ. Readers are advised to disregard any $\rightsquigarrow$ blue and highlighted parts, as these are relevant to the elaboration process that will be explained in §5. The first six rules are standard. Rule METHOD states that a type class method can be invoked whenever the corresponding class constraint can be solved (§3.3).

Rule QUALI adds qualifiers to types by putting the constraint in the environment (with the $\rightsquigarrow x$ part needed for elaboration) to type-check the term, while QUALE eliminates them by invoking the row theory's constraint solving relation. Similarly, SCHEMEI generalizes the term's type, while SCHEMEE instantiates polymorphic types.

In LABEL, first-class labels are assigned a singleton type of the same label. Rules PROD and SUM create singletons of their respective row types, requiring the label term to have a singleton label type. UNLABEL applies to both singleton products and sums, when the label in the type of N matches the label in the row of M's type. Rules PRJ, CONCAT, INJ, and ELIM all require the corresponding row constraints to hold. Rule SUB allows terms to be implicitly cast to a supertype, using the subtyping judgement. Finally, IND precisely specifies the **ind** primitive, as described in §2.2.

Subtyping. Fig. 4 presents selected subtyping rules. The subtyping judgement $\Gamma_C; \Gamma \vdash \sigma_0 <: \sigma_1$ reads that under the contexts Γ_C and Γ, the type σ_0 is a subtype of σ_1. For clarity, we also informally write $\Gamma_C; \Gamma; \psi_0 \vDash \psi_1$ for subtyping between constraints. The subtyping relation supports a partial ordering between commutativity annotations, type equivalence between commutative rows, and type equivalence for **Lift** types and their applied forms. Relations between rows, however, are handled by constraint solving, as their interpretation can vary depending on the specific row theory.

Subtyping is reflexive and transitive, and is co-variant over both product and sum constructors. Thus, rule PRODORSUM simply checks the subtyping relation between the components. Rules PRODORSUMROW, CONTAIN, and ALL relate equivalent rows inside constructors. Row equivalence $\Gamma_C; \Gamma \vdash \rho_0 \equiv_\mu \rho_1$ is reflexive, symmetrical, and transitive. It is parameterized by the commutativity μ, which is passed in from subtyping. Row equivalence relates commutative rows, as in rule COMM (where the row permutation is parameterized by the row theory, and $[:n]$ denotes the natural numbers up to, but not including n), as well as lifts of concrete rows regardless of the commutativity, as in rules LIFTL and LIFTR.

Rule DECAY allows a record or variant with commutativity μ_0 to be used as one with commutativity μ_1, provided $\mu_0 \leqslant \mu_1$. Rule NEVER converts an empty sum, which has no inhabitants, to any other well-formed type, which is especially handy for writing initial values when using **ind** with sums.

As previously discussed, commutativity has a partial ordering, denoted as $\mu_0 \leqslant \mu_1$, which is reflexive. Non-commutativity $\mathfrak{n}$ is the strongest form, as it retains row order information. This

$\boxed{\Gamma_I; \Gamma_C; \Gamma \vdash M : \sigma \rightsquigarrow E}$ *(Typing And Elaboration)*

VAR
$$\frac{x : \sigma \in \Gamma}{\Gamma_I; \Gamma_C; \Gamma \vdash x : \sigma \rightsquigarrow x}$$

METHOD
$$\frac{(\overline{TC'_i\ a \rightsquigarrow A'_i} \Rightarrow TC\ a : \kappa) \mapsto m : \sigma \rightsquigarrow A \in \Gamma_C \qquad \Gamma_I; \Gamma_C; \Gamma \vDash_{\mathcal{T}} TC\ \tau \rightsquigarrow E}{\Gamma_I; \Gamma_C; \Gamma \vdash m : \sigma\ [\tau/a] \ \rightsquigarrow \pi_0\ E}$$

LAM
$$\frac{\Gamma_I; \Gamma_C; \Gamma, x : \tau_0 \vdash M : \tau_1 \rightsquigarrow E \qquad \Gamma_C; \Gamma \vdash \tau_0 : \star \rightsquigarrow A}{\Gamma_I; \Gamma_C; \Gamma \vdash \lambda\, x.\, M : \tau_0 \to \tau_1 \rightsquigarrow \lambda\, x : A.\, E}$$

APP
$$\frac{\Gamma_I; \Gamma_C; \Gamma \vdash M : \tau_0 \to \tau_1 \rightsquigarrow F \qquad \Gamma_I; \Gamma_C; \Gamma \vdash N : \tau_0 \rightsquigarrow E}{\Gamma_I; \Gamma_C; \Gamma \vdash M\ N : \tau_1 \rightsquigarrow F\ E}$$

LET
$$\frac{\Gamma_I; \Gamma_C; \Gamma \vdash M : \sigma_0 \rightsquigarrow E \qquad \Gamma_C; \Gamma \vdash \sigma_0 : \star \rightsquigarrow A \qquad \Gamma_I; \Gamma_C; \Gamma, x : \sigma_0 \vdash N : \sigma_1 \rightsquigarrow F}{\Gamma_I; \Gamma_C; \Gamma \vdash \mathbf{let}\ x : \sigma_0 = M\ \mathbf{in}\ N : \sigma_1 \rightsquigarrow (\lambda\, x : A.\, F)\ E}$$

ANNOT
$$\frac{\Gamma_I; \Gamma_C; \Gamma \vdash M : \sigma \rightsquigarrow E}{\Gamma_I; \Gamma_C; \Gamma \vdash M : \sigma : \sigma \rightsquigarrow E}$$

QUALI
$$\frac{\Gamma_C; \Gamma \vdash \psi : \mathbf{C} \rightsquigarrow A \qquad \Gamma_I; \Gamma_C; \Gamma, \psi \rightsquigarrow x \vdash M : \gamma \rightsquigarrow E}{\Gamma_I; \Gamma_C; \Gamma \vdash M : \psi \Rightarrow \gamma \rightsquigarrow \lambda\, x : A.\, E}$$

QUALE
$$\frac{\Gamma_I; \Gamma_C; \Gamma \vDash_{\mathcal{T}} \psi \rightsquigarrow E \qquad \Gamma_I; \Gamma_C; \Gamma \vdash M : \psi \Rightarrow \gamma \rightsquigarrow F}{\Gamma_I; \Gamma_C; \Gamma \vdash M : \gamma \rightsquigarrow F\ E}$$

SCHEMEI
$$\frac{\Gamma_I; \Gamma_C; \Gamma, a : \kappa \vdash M : \sigma \rightsquigarrow E \qquad \vdash \kappa \rightsquigarrow K}{\Gamma_I; \Gamma_C; \Gamma \vdash M : \forall\, a : \kappa.\, \sigma \rightsquigarrow \Lambda\, a : K.\, E}$$

SCHEMEE
$$\frac{\Gamma_I; \Gamma_C; \Gamma \vdash M : \forall\, a : \kappa.\, \sigma \rightsquigarrow E \qquad \Gamma_C; \Gamma \vdash \tau : \kappa \rightsquigarrow A}{\Gamma_I; \Gamma_C; \Gamma \vdash M : \sigma\ [\tau/a] \ \rightsquigarrow E\ [A]}$$

LABEL
$$\frac{}{\Gamma_I; \Gamma_C; \Gamma \vdash \ell : \lfloor \ell \rfloor \rightsquigarrow ()}$$

PROD
$$\frac{\Gamma_I; \Gamma_C; \Gamma \vdash M : \lfloor \xi \rfloor \qquad \Gamma_I; \Gamma_C; \Gamma \vdash N : \tau \rightsquigarrow E}{\Gamma_I; \Gamma_C; \Gamma \vdash \{M \triangleright N\} : \Pi_{\mathfrak{n}}\ \langle \xi \triangleright \tau \rangle \rightsquigarrow (E)}$$

UNLABEL
$$\frac{\Gamma_I; \Gamma_C; \Gamma \vdash M : \Xi_{\mathfrak{c}}\ \langle \xi \triangleright \tau \rangle \rightsquigarrow E \qquad \Gamma_I; \Gamma_C; \Gamma \vdash N : \lfloor \xi \rfloor \qquad \Gamma_C; \Gamma \vdash \tau : \star \rightsquigarrow A \qquad F = \begin{cases} \pi_0\ E & \Xi = \Pi \\ \mathbf{case}\ E\ \{\lambda\, x : A.\, x\} & \Xi = \Sigma \end{cases}}{\Gamma_I; \Gamma_C; \Gamma \vdash M / N : \tau \rightsquigarrow F}$$

SUM
$$\frac{\Gamma_I; \Gamma_C; \Gamma \vdash M : \lfloor \xi \rfloor \qquad \Gamma_I; \Gamma_C; \Gamma \vdash N : \tau \rightsquigarrow E}{\Gamma_I; \Gamma_C; \Gamma \vdash [M \triangleright N] : \Sigma_{\mathfrak{n}}\ \langle \xi \triangleright \tau \rangle \rightsquigarrow \iota_0\ E}$$

PRJ
$$\frac{\Gamma_I; \Gamma_C; \Gamma \vdash M : \Pi_\mu\ \rho_0 \rightsquigarrow E \qquad \Gamma_I; \Gamma_C; \Gamma \vDash_{\mathcal{T}} \rho_1 \lesssim_\mu \rho_0 \rightsquigarrow F}{\Gamma_I; \Gamma_C; \Gamma \vdash \mathbf{prj}\ M : \Pi_\mu\ \rho_1 \rightsquigarrow (\pi_0\ F)\ [\lambda\, a : \star.\, a]\ E}$$

CONCAT
$$\frac{\Gamma_I; \Gamma_C; \Gamma \vdash M : \Pi_\mu\ \rho_0 \rightsquigarrow E_0 \qquad \Gamma_I; \Gamma_C; \Gamma \vdash N : \Pi_\mu\ \rho_1 \rightsquigarrow E_1 \qquad \Gamma_I; \Gamma_C; \Gamma \vDash_{\mathcal{T}} \rho_0 \odot_\mu \rho_1 \sim \rho_2 \rightsquigarrow F}{\Gamma_I; \Gamma_C; \Gamma \vdash M +\!\!+ N : \Pi_\mu\ \rho_2 \rightsquigarrow ((\pi_0\ F)\ [\lambda\, a : \star.\, a]\ E_0)\ E_1}$$

INJ
$$\frac{\Gamma_I; \Gamma_C; \Gamma \vdash M : \Sigma_\mu\ \rho_0 \rightsquigarrow E \qquad \Gamma_I; \Gamma_C; \Gamma \vDash_{\mathcal{T}} \rho_0 \lesssim_\mu \rho_1 \rightsquigarrow F}{\Gamma_I; \Gamma_C; \Gamma \vdash \mathbf{inj}\ M : \Sigma_\mu\ \rho_1 \rightsquigarrow (\pi_1\ F)\ [\lambda\, a : \star.\, a]\ E}$$

ELIM
$$\frac{\Gamma_I; \Gamma_C; \Gamma \vdash M : (\Sigma_\mu\ \rho_0) \to \tau \rightsquigarrow E_0 \qquad \Gamma_I; \Gamma_C; \Gamma \vdash N : (\Sigma_\mu\ \rho_1) \to \tau \rightsquigarrow E_1 \qquad \Gamma_I; \Gamma_C; \Gamma \vDash_{\mathcal{T}} \rho_0 \odot_\mu \rho_1 \sim \rho_2 \rightsquigarrow F \qquad \Gamma_C; \Gamma \vdash \tau : \star \rightsquigarrow A}{\Gamma_I; \Gamma_C; \Gamma \vdash M \bigtriangledown N : (\Sigma_\mu\ \rho_2) \to \tau \rightsquigarrow ((\pi_1\ F)\ [\lambda\, a : \star.\, a]\ [A]\ E_0)\ E_1}$$

SUB
$$\frac{\Gamma_I; \Gamma_C; \Gamma \vdash M : \sigma_0 \rightsquigarrow E \qquad \Gamma_C; \Gamma \vdash \sigma_0 <: \sigma_1 \rightsquigarrow F}{\Gamma_I; \Gamma_C; \Gamma \vdash M : \sigma_1 \rightsquigarrow F\ E}$$

IND
$$\frac{\begin{array}{c}\Gamma_C; \Gamma \vdash \rho : \mathbf{R}^\kappa \qquad \Gamma_C; \Gamma, a : \mathbf{R}^\kappa \vdash \tau : \star \rightsquigarrow A \\ \Gamma_I; \Gamma_C; \Gamma, a_l : \mathbf{L}, a_t : \kappa, a_p\ a_i\ a_n : \mathbf{R}^\kappa \vdash M : a_p \odot_{\mathfrak{n}} \langle a_l \triangleright a_t \rangle \sim a_i, a_i \odot_{\mathfrak{n}} a_n \sim \rho \Rightarrow \lfloor a_l \rfloor \to \tau\ \left[a_p/a\right] \ \to \tau\ [a_i/a] \ \rightsquigarrow E_0 \\ \Gamma_I; \Gamma_C; \Gamma \vdash N : \tau\ [\langle\rangle/a] \ \rightsquigarrow E_1 \end{array} \qquad \vdash \kappa \rightsquigarrow K \qquad \Gamma_I; \Gamma_C; \Gamma \vDash_{\mathcal{T}} \mathbf{Ind}\ \rho \rightsquigarrow F \qquad E = (F\ [\lambda\, a : \mathbf{L}^K.\, A]\ (\Lambda\, a_l : \star, a_t : K, a_p\ a_i\ a_n : \mathbf{L}^K.\, E_0))\ E_1}{\Gamma_I; \Gamma_C; \Gamma \vdash \mathbf{ind}\ (\lambda\, a : \mathbf{R}^\kappa.\, \tau)\ \rho\ M\ N : \tau\ [\rho/a] \ \rightsquigarrow E}$$

Fig. 3. Term typing (with elaboration detailed in §5)

information can be discarded, allowing it to "decay" into any other commutativity type. Conversely, commutativity $\mathfrak{c}$ is the weakest form, as it carries no additional information. This ordering is partial because there is no relationship between distinct polymorphic commutativity type variables. Recall that the typing rules PROD, SUM, and UNLABEL (Fig. 3) use concrete commutativities. This partial ordering allows us to apply those rules to values with other forms of commutativity without any loss of expressiveness.

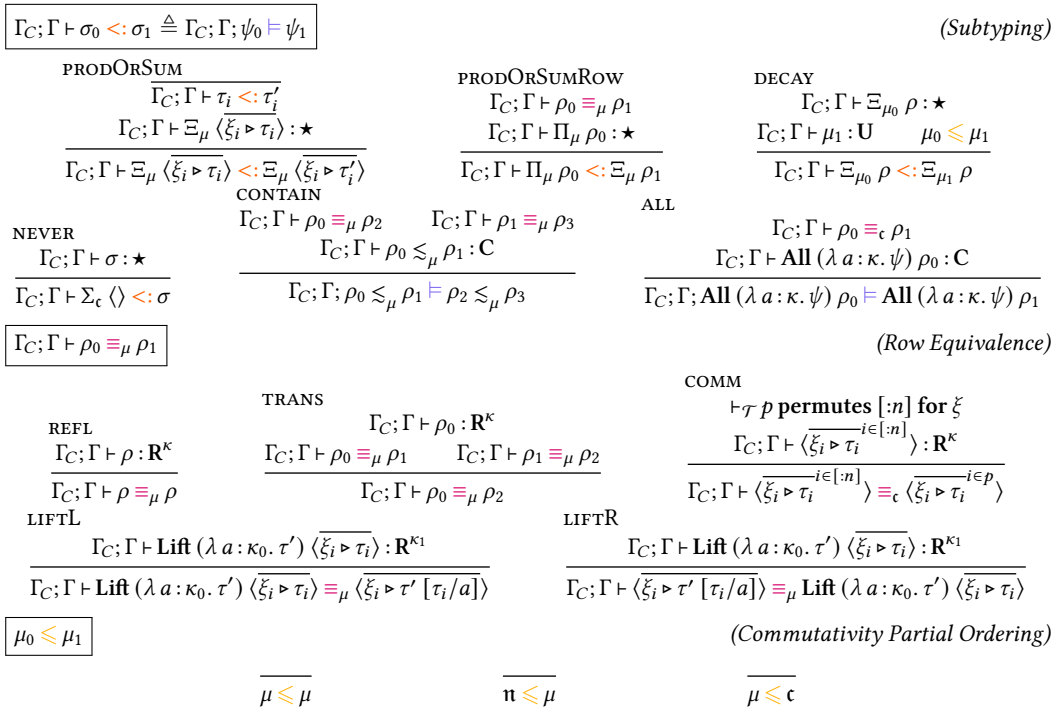


Fig. 4. Subtyping

3.3 Constraint Solving

Up to this point, we have left the row theory abstract (§3.1). To illustrate constraint solving (and subsequent elaboration in §5.2), we provide an example interpretation $\mathcal{T}_s$ of *simple rows*, similar to those from Morris and McKinna [2019]. For this row theory, the label check in the ROW kinding rule is instantiated as a uniqueness check, and row permutations in the COMM row equivalence rule are unrestricted.[12] Fig. 5 presents selected rules for the constraint solving relation of $\mathcal{T}_s$. The first rule LOCAL simply solves constraints by using those already available in the context.

Row constraints. The next few rules are for rows. Rule CONTAINTRANS states that row containment is transitive. Rule CONTAINDECAY uses the commutativity partial ordering to solve row containment constraints, deriving weaker commutativities from stronger ones.

Rule CONTAINCONCAT permits the combination of multiple contained (possibly non-adjacent) rows into a larger containment constraint. Rule CONCATCONCRETE is the only rule for solving concrete, non-commutative rows constraints; the constraint well-kindedness ensures the uniqueness of labels. The non-commutativity requires the combined row to be made up of all entries of the first row in order, followed by all entries of the second, meaning no "interleaving" is allowed. However, elements can be interleaved in containment constraints via CONTAINCONCAT.

Rule CONCATSWAP states that commutative row combination is symmetric. Rule CONCATDECAY, similarly to rule CONTAINDECAY, allows containment constraints with stronger commutativity to prove those with weaker commutativity. Lastly, containment can also be derived from concatenation using rules CONCATCONTAINL and CONCATCONTAINR.

[12] It is possible to instantiate $\mathcal{T}$ with other row theories, such as *scoped rows* [Berthomieu and De Sagazan 1995; Leijen 2005]. In that case, rule ROW allows duplicate labels, and COMM permits permutations between non-duplicate labels.

$\boxed{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \psi}$ *(Constraint Solving)*

Local

$$\textsc{local}\ \frac{\psi \in \Gamma}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \psi}$$

Row

$$\textsc{containTrans}\ \frac{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \lesssim_\mu \rho_1 \qquad \Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_1 \lesssim_\mu \rho_2}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \lesssim_\mu \rho_2}$$

$$\textsc{containDecay}\ \frac{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \lesssim_{\mu_0} \rho_1 \qquad \mu_0 \leqslant \mu_1 \qquad \Gamma_C; \Gamma \vdash \mu_1 : \mathbf{U}}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \lesssim_{\mu_1} \rho_1}$$

$$\textsc{containConcat}\ \frac{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \odot_\mu \rho_1 \sim \rho_2 \qquad \Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_3 \odot_\mu \rho_4 \sim \rho_5 \qquad \Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \lesssim_\mu \rho_3 \qquad \Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_1 \lesssim_\mu \rho_4}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_2 \lesssim_\mu \rho_5}$$

$$\textsc{concatConcrete}\ \frac{\Gamma_C; \Gamma \vdash \langle \overline{\xi_i \triangleright \tau_i}^{\,i \in [:m]}, \overline{\xi'_j \triangleright \tau'_j}^{\,j \in [:n]} \rangle : \mathbf{R}^\kappa}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \langle \overline{\xi_i \triangleright \tau_i}^{\,i \in [:m]} \rangle \odot_\pi \langle \overline{\xi'_j \triangleright \tau'_j}^{\,j \in [:n]} \rangle \sim \langle \overline{\xi_i \triangleright \tau_i}^{\,i \in [:m]}, \overline{\xi'_j \triangleright \tau'_j}^{\,j \in [:n]} \rangle}$$

$$\textsc{concatSwap}\ \frac{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \odot_\mathfrak{c} \rho_1 \sim \rho_2}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_1 \odot_\mathfrak{c} \rho_0 \sim \rho_2}$$

$$\textsc{concatDecay}\ \frac{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \odot_{\mu_0} \rho_1 \sim \rho_2 \qquad \mu_0 \leqslant \mu_1 \qquad \Gamma_C; \Gamma \vdash \mu_1 : \mathbf{U}}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \odot_{\mu_1} \rho_1 \sim \rho_2}$$

$$\textsc{concatContainL}\ \frac{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \odot_\mu \rho_1 \sim \rho_2}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \lesssim_\mu \rho_2}$$

$$\textsc{concatContainR}\ \frac{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \odot_\mu \rho_1 \sim \rho_2}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_1 \lesssim_\mu \rho_2}$$

Lift

$$\textsc{liftContain}\ \frac{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \lesssim_\mu \rho_1 \qquad \Gamma_C; \Gamma \vdash \rho_0 : \mathbf{R}^{\kappa_0} \qquad \Gamma_C; \Gamma, a : \kappa_0 \vdash \tau : \kappa_1}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} (\mathbf{Lift}\ (\lambda\, a : \kappa_0.\ \tau)\ \rho_0) \lesssim_\mu (\mathbf{Lift}\ (\lambda\, a : \kappa_0.\ \tau)\ \rho_1)}$$

$$\textsc{liftConcat}\ \frac{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \odot_\mu \rho_1 \sim \rho_2 \qquad \Gamma_C; \Gamma \vdash \rho_0 : \mathbf{R}^{\kappa_0} \qquad \Gamma_C; \Gamma, a : \kappa_0 \vdash \tau : \kappa_1}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} (\mathbf{Lift}\ (\lambda\, a : \kappa_0.\ \tau)\ \rho_0) \odot_\mu (\mathbf{Lift}\ (\lambda\, a : \kappa_0.\ \tau)\ \rho_1) \sim (\mathbf{Lift}\ (\lambda\, a : \kappa_0.\ \tau)\ \rho_2)}$$

Type classes

$$\textsc{TCInst}\ \frac{(\forall\, \overline{a_k : \kappa_k}.\ \overline{\psi_j} \Rightarrow TC\ \tau) \in \Gamma_I \qquad \overline{\Gamma_C; \Gamma \vdash \tau'_k : \kappa_k} \qquad \overline{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \psi_j[\tau'_k/a_k]}}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} TC\ \tau[\tau'_k/a_k]}$$

$$\textsc{TCSuper}\ \frac{(\overline{TC'_j\ a}^{\,j \in [:n]} \Rightarrow TC\ a : \kappa) \mapsto m : \sigma \in \Gamma_C \qquad \Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} TC\ \tau \qquad i \in [:n]}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} TC'_i\ \tau}$$

All

$$\textsc{allEmpty}\ \frac{\Gamma_C; \Gamma, a : \kappa \vdash \psi : \mathbf{C}}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \mathbf{All}\ (\lambda\, a : \kappa.\ \psi)\ \langle\rangle}$$

$$\textsc{allSingletonIntro}\ \frac{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \psi\ [\tau/a] \qquad \Gamma_C; \Gamma, a : \kappa \vdash \psi : \mathbf{C} \qquad \Gamma_C; \Gamma \vdash \xi : \mathbf{L} \qquad \Gamma_C; \Gamma \vdash \tau : \kappa}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \mathbf{All}\ (\lambda\, a : \kappa.\ \psi)\ \langle \xi \triangleright \tau \rangle}$$

$$\textsc{allSingletonElim}\ \frac{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \mathbf{All}\ (\lambda\, a : \kappa.\ \psi)\ \langle \xi \triangleright \tau \rangle}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \psi\ [\tau/a]}$$

$$\textsc{allContain}\ \frac{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \lesssim_\mathfrak{c} \rho_1 \qquad \Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \mathbf{All}\ (\lambda\, a : \kappa.\ \psi)\ \rho_1}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \mathbf{All}\ (\lambda\, a : \kappa.\ \psi)\ \rho_0}$$

$$\textsc{allConcat}\ \frac{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \odot_\mathfrak{c} \rho_1 \sim \rho_2 \qquad \Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \mathbf{All}\ (\lambda\, a : \kappa.\ \psi)\ \rho_0 \qquad \Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \mathbf{All}\ (\lambda\, a : \kappa.\ \psi)\ \rho_1}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \mathbf{All}\ (\lambda\, a : \kappa.\ \psi)\ \rho_2}$$

Split

$$\textsc{splitEmpty}\ \frac{\Gamma_C; \Gamma, a : \kappa \vdash \tau : \star}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \mathbf{Split}\ (\lambda\, a : \kappa.\ \tau)\ \langle\rangle\ \langle\rangle\ \langle\rangle}$$

$$\textsc{splitSingletonMatch}\ \frac{\Gamma_C; \Gamma, a : \kappa \vdash \tau_0 : \star \qquad \Gamma_C; \Gamma \vdash \tau_1 : \kappa \qquad \Gamma_C; \Gamma \vdash \xi : \mathbf{L}}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \mathbf{Split}\ (\lambda\, a : \kappa.\ \tau_0)\ \langle \xi \triangleright \tau_1 \rangle\ \langle\rangle\ \langle \xi \triangleright \tau_0\ [\tau_1/a] \rangle}$$

$$\textsc{splitSingletonRest}\ \frac{\nexists\, \tau_2, \Gamma_C; \Gamma \vdash \tau_1 <: \tau_0\ [\tau_2/a] \qquad \Gamma_C; \Gamma, a : \kappa \vdash \tau_0 : \star \qquad \Gamma_C; \Gamma \vdash \tau_1 : \star \qquad \Gamma_C; \Gamma \vdash \xi : \mathbf{L}}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \mathbf{Split}\ (\lambda\, a : \kappa.\ \tau_0)\ \langle\rangle\ \langle \xi \triangleright \tau_1 \rangle\ \langle \xi \triangleright \tau_1 \rangle}$$

$$\textsc{splitConcat}\ \frac{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \mathbf{Split}\ (\lambda\, a : \kappa.\ \tau)\ \rho_0\ \rho_1\ \rho_2}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} (\mathbf{Lift}\ (\lambda\, a : \kappa.\ \tau)\ \rho_0) \odot_\mathfrak{c} \rho_1 \sim \rho_2}$$

Fig. 5. Constraint solving

$$
\begin{array}{rcll}
E, F & ::= & x \mid \lambda\, x : A.\, E \mid E\, F \mid \Lambda\, a : K.\, E \mid E\,[A] \mid (\overline{E_i}) \mid \pi_n\, E \mid \iota_n\, E \mid \mathbf{case}\, E\, \{\overline{F_i}\} & \textit{term} \\
V & ::= & \lambda\, x : A.\, E \mid \Lambda\, a : K.\, E \mid (\overline{V_i}) \mid \iota_n V & \textit{value} \\
A, B, C & ::= & a \mid \lambda\, a : K.\, A \mid A\, B \mid \forall\, a : K.\, A \mid A \to B \mid \{\overline{A_i}\} \mid A\,[\![B]\!] \mid \otimes A \mid \oplus A & \textit{type} \\
K & ::= & \star \mid K_1 \mapsto K_2 \mid \mathbf{L}^K & \textit{kind} \\
\Delta & ::= & \epsilon \mid \Delta, a : K \mid \Delta, x : A & \textit{environment}
\end{array}
$$

Fig. 6. $F_\omega^{\otimes\oplus}$ syntax

Rules liftContain and liftConcat solve containment and concatenation constraints for lifted rows by solving their unlifted forms. This also highlights the utility of allowing row constraints over rows of arbitrary kinds.

Type classes, All, and Split. There are two rules for type class constraints. Rule TCInst finds an instance in the context, instantiates its type variables, and solves the prerequisites. Rule TCSuper solves a superclass when a subclass constraint can be solved.

We have five rules for the **All** constraint. Rule allEmpty states that **All** is always satisfied for empty rows. Rules allSingletonIntro and allSingletonElim convert between **All** applied to singleton rows and the corresponding constraint on the type itself. allSingletonElim is particularly useful within the step function of **ind** expressions. Rule allContain solves **All** over smaller rows using **All** over larger rows. Finally, rule allConcat solves **All** constraints over larger rows using those over smaller rows. This rule is often used at the top level when polymorphic functions are instantiated and we need to solve constraints for concrete rows.

The remaining rules are for **Split**. Rule splitEmpty applies when the row being split is empty. Rules splitSingletonMatch and splitSingletonRest handle cases where an entry either matches or does not match the type lambda, respectively. While rule splitSingletonMatch is straightforward, rule splitSingletonRest incorporates subtyping to ensure deterministic **Split** solving: splitSingletonRest only applies when a type fails to match the type lambda, even through subtyping. Such a subtyping rule is not needed for splitSingletonMatch because such a check can be handled by the sub typing rule if necessary to solve the constraint. Larger split constraints are solved by combining smaller ones, with the corresponding rule provided in the appendix for space reasons. Finally, rule splitConcat states that **Split** implies the corresponding concatenation constraint.

Lastly, we remark that while the system incorporates numerous rules, the complexity is a worthwhile tradeoff for the simplified type signatures it enables: many rules allows for the derivation of weaker constraints from stronger ones, meaning programmers can specify only the most powerful constraints in their type signatures, leading to significantly more readable types.

4 Target Calculus

This section presents $F_\omega^{\otimes\oplus}$, our target calculus. We present its semantics (§4.1), and prove its type soundness (§4.2). $F_\omega^{\otimes\oplus}$ builds upon the $F^{\otimes\oplus}$ calculus [Morris and McKinna 2019], but extends it in several significant aspects. First, while $F^{\otimes\oplus}$ is based on System F, $F_\omega^{\otimes\oplus}$ is built on System Fω by incorporating type lambdas, type applications, and higher kinds. Second, we introduce first-class type-level lists. This allows product and sum types to be constructed with type-level lists instead of embedded entries or variants. Unlike rows, type lists are considerably simpler: they lack labels, so there are no concerns about uniqueness, and they do not support containment, concatenation constraints, or commutativity. Finally, we introduce type-level mappings, which are similar to **Lift** in the source, and apply a type lambda over type-level lists.

Syntax. The syntax of $F_\omega^{\otimes\oplus}$ is presented in Fig. 6. Expressions E include variables x, lambdas $\lambda\, x : A.\, E$, applications $E\, F$, type abstractions $\Lambda\, a : K.\, E$, type applications $E\,[A]$, products $(\overline{E_i})$ written

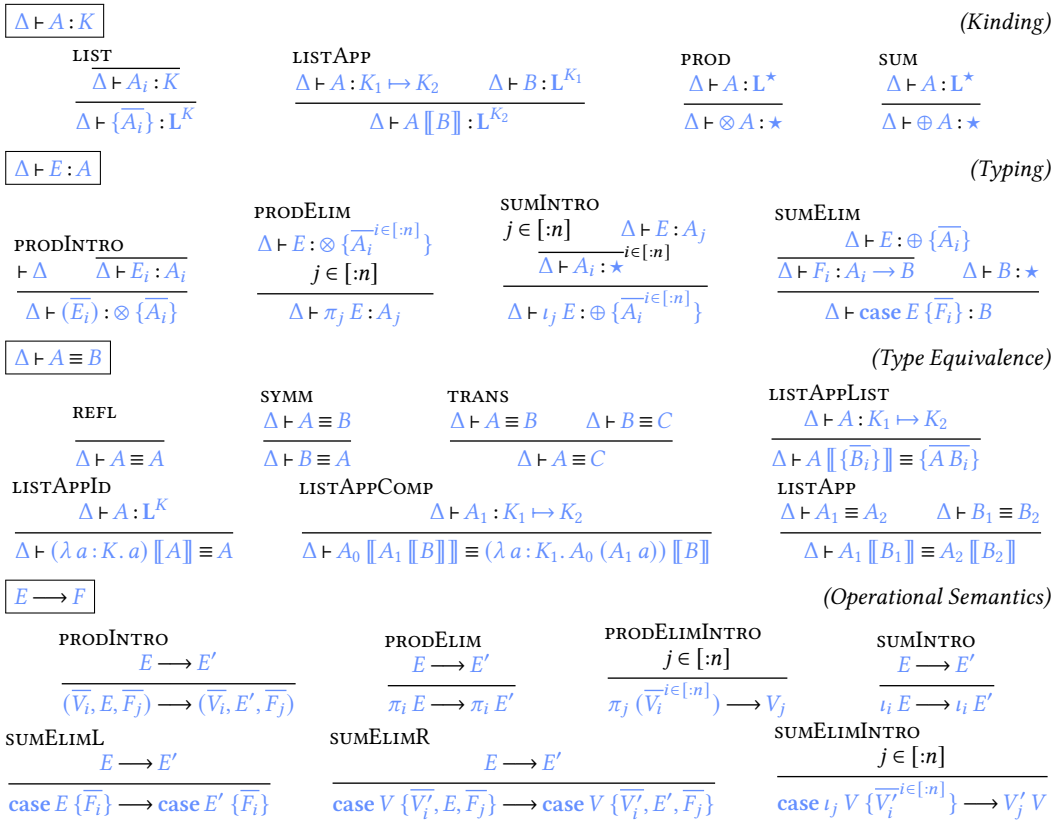


Fig. 7. $\mathrm{F}_\omega^{\otimes\oplus}$ semantics

with the tuple syntax, product projection $\pi_n\, E$, sum introduction $\iota_n\, E$, and elimination $\mathbf{case}\ E\ \{\overline{F_i}\}$. Values V are a subset of expressions.

Types A include variables a, type lambdas $\lambda\, a : K.\, A$, applications $A\ B$, polymorphic types $\forall\, a : K.\, A$, function types $A \to B$, type-level lists $\{\overline{A_i}\}$ and mappings $A\ [\![B]\!]$, product types $\otimes A$, and sum types $\oplus A$. Kinds K include the base kind $\star$, $K_1 \mapsto K_2$ for mappings, and $\mathbf{L}^K$ for type-level lists of kind K. A typing context Δ maps type variables to their kinds, and term variables to their types.

4.1 Typing and Operational Semantics

Fig. 7 presents selected rules for $\mathrm{F}_\omega^{\otimes\oplus}$, focusing on type-level lists, products, and sums; the complete rules can be found in the appendix. For kinding $\Delta \vdash A : K$, rule LIST requires all entries in a list to be of kind K, and returns kind $\mathbf{L}^K$. In rule LISTAPP, A is a mapping from $K_1 \mapsto K_2$, which takes B of $\mathbf{L}^{K_1}$ and returns $\mathbf{L}^{K_2}$. Both rules PROD and SUM require A to be of kind $\mathbf{L}^\star$.

For typing $\Delta \vdash E : A$, most rules are self-explanatory. The judgement ensures that Δ is well-formed (written $\vdash \Delta$), and the output type A has kind $\star$. In rule PRODINTRO, the first hypothesis ensures the context is well-formed, even if the product is empty. Similarly, in rule SUMINTRO, the last hypothesis ensures that the other cases of the sum being created have the required kind. The last hypothesis for rule SUMELIM similarly ensures the output is well-kinded, even when E is the empty sum.

The judgement $\Delta \vdash A \equiv B$ defines type equivalence, which is reflexive, symmetrical, and transitive. Rule LISTAPPLIST gives mapping its intended semantics by applying the function to each element within a concrete list. Rules LISTAPPID and LISTAPPCOMP correspond to the identity and composition

laws for mappings. These two rules are particularly useful when the list is e.g. a type variable and thus rule LISTAPPLIST does not apply. We will revisit these rules when discussing elaboration (§5).

The operational semantics $E \longrightarrow E'$ follow a call-by-value evaluation strategy. Rule PRODINTRO evaluates product components left-to-right until all are values. Rules SUMELIML and SUMELIMR evaluate the scrutinee, then the handlers in a similar way. The computational rules PRODELIMINTRO and SUMELIMINTRO perform projection and case matching on values. All type-level constructs, including lists and mappings, are purely static and do not appear in the operational semantics.

4.2 Type Soundness

We demonstrate that $\mathrm{F}_\omega^{\otimes\oplus}$ enjoys syntactic type soundness. The proofs are largely based on those for System Fω [Pierce 2002], with the exception of the typing inversion and value cannonical form lemmas, which are significantly more complex than in standard System Fω. These lemmas rely on the property that it is impossible to establish equivalence between unrelated types. This is normally established by defining a reduction relation for types, proving that the equivalence closure of this relation corresponds to type equivalence, then proving confluence for the reduction relation.

Typically, a single-step parallel reduction relation is used, with confluence following from the *diamond property*, but this turns out to be highly non-trivial in $\mathrm{F}_\omega^{\otimes\oplus}$. As an example, consider parallel reduction on $A \,[\![\, (\lambda\, a : \star.\, a) \,[\![\{B\}]\!] \,]\!]$. This type reduces to $A \,[\![\{(\lambda\, a : \star.\, a)\; B\}]\!]$ via LISTAPPLIST. Alternatively, it reduces to $(\lambda\, a : \star.\, A\; ((\lambda\, a : \star.\, a)\; a))\; [\![\{B\}]\!]$ via LISTAPPCOMP. The diamond property claims that both reduction paths converge in one step, but this is not possible because the first type reduces to $\{A\; B\}$ (LISTAPPLIST, β) or $A \,[\![\{B\}]\!]$ (LISTAPP, β), while the second type reduces to $\{(\lambda\, a : \star.\, A\; a)\; B\}$ (LISTAPPLIST, β) or $(\lambda\, a : \star.\, A\; a)\; [\![\{B\}]\!]$ (LISTAPP, β).

Therefore, instead of using the parallel reduction approach, we define non-deterministic small-step reduction semantics for types (provided in the appendix). We then apply Newman's lemma [Newman 1942] to conclude confluence from local confluence (Thm. 4.1) and strong normalization (Thm. 4.2). For more detailed explanations of our proofs, we refer the reader to §6, the appendix, and our artifact.

We first establish local confluence by induction:

Theorem 4.1 (Local Confluence). *If* $\Delta \vdash A \longrightarrow B_0$, $\Delta \vdash A \longrightarrow B_1$, $\Delta \vdash A : K$, *and* $\vdash \Delta$, *then there exists* C *such that* $\Delta \vdash B_0 \longrightarrow^* C$ *and* $\Delta \vdash B_1 \longrightarrow^* C$.

We then apply an argument based on logical relations [Skorstengaard 2019] to prove strong normalization. The logical relation for closed types is defined as follows:

$$\begin{aligned}
\mathbf{SN}_\star(A) &= \epsilon \vdash A : \star \wedge \mathbf{SN}(A) \\
\mathbf{SN}_{K_1 \mapsto K_2}(A) &= \epsilon \vdash A : K_1 \mapsto K_2 \wedge \forall B, \mathbf{SN}_{K_1}(B) \Rightarrow \mathbf{SN}_{K_2}(A\; B) \\
\mathbf{SN}_{\mathbf{L}^K}(A) &= \epsilon \vdash A : \mathbf{L}^K \wedge \mathbf{SN}^{\mathbf{L}}_{\mathbf{SN}_K}(A) \wedge \forall A'\; B', \epsilon \vdash A \longrightarrow^* A' \,[\![B']\!] \Rightarrow \exists C, \mathbf{SN}_K(A'\; C)
\end{aligned}$$

The first two cases are standard, as typically seen in logical relations for the lambda calculus [Girard et al. 1989, Chapter 6]. The $\mathbf{SN}$ predicate (which appears in the $\star$ case) is strong normalization; this means that there is no infinite reduction sequence for the type. The last case was the most challenging to define, due to complicated interactions with the LISTAPPCOMP rule. Specifically, when this rule is applied, it is often difficult to preserve the required induction hypotheses since various useful properties (most importantly $\mathbf{SN}_K$) cannot be inverted. The $\mathbf{SN}^{\mathbf{L}}_P$ relation, paramterized by a property P, states that all reduction paths of the type terminate, or encounter a concrete list whose entries satisfy P:

$$\frac{\textsc{refl}\ \overline{P(A_i)}}{\mathbf{SN}^{\mathbf{L}}_P(\{\overline{A_i}\})} \qquad \frac{\textsc{step}\ \forall B, \epsilon \vdash A \longrightarrow B \Rightarrow \mathbf{SN}^{\mathbf{L}}_P(B) \qquad \forall \overline{B_i}, A \neq \{\overline{B_i}\}}{\mathbf{SN}^{\mathbf{L}}_P(A)}$$

We establish that the logical relation implies strong normalization by induction on kinds:

Theorem 4.2 (Strong Normalization). *If* $\mathbf{SN}_K(A)$ *then* $\mathbf{SN}(A)$.

We prove that well-kinded types are in the logical relation. We lift the logical relations to open terms, with δ being the standard substitution of contexts. The judgment $\delta \vDash \Delta$ means that δ provides a substitution for each type variable entry in the environment, and each substitution type satisfies the same $\mathbf{SN}_K$ predicate:

Theorem 4.3 (Fundamental Property). *If* $\delta \vDash \Delta$ *and* $\Delta \vdash A : K$ *then* $\mathbf{SN}_K(\delta\, A)$.

Combining Thm. 4.2 and Thm. 4.3 establishes strong normalization for all well-kinded types, which in turn allows us to prove confluence.

With all these, we prove type soundness of $\mathrm{F}^{\otimes\oplus}_{\omega}$:

Theorem 4.4 (Progress). *If* $\epsilon \vdash E : A$, *then either* E *is a value, or there exists* E' *such that* $E \longrightarrow E'$.

Theorem 4.5 (Type Preservation). *If* $\Delta \vdash E : A$ *and* $E \longrightarrow E'$, *then* $\Delta \vdash E' : A$.

5 Elaboration

This section presents the elaboration rules. Following elaboration of expressions, we cover four key aspects of elaboration: dictionary-passing elaboration of type classes (§5.1), interpretation of row constraints (§5.2), elaboration of **ind** (§5.3), and elaboration of subtyping and row equivalence (§5.4). Finally, we prove that elaboration is sound (§5.5). As before, we use blue to denote target terms.

Type-directed elaboration of expressions. Fig. 3 has presented the type-directed elaboration of expressions. Rules VAR, LAM, and APP are straightforward, translating the source constructs into their corresponding target terms. Rule ANNOT simply produces the elaboration of the expression, as the target is always fully annotated.

Rules METHOD, QUALI, and QUALE handle type class constraints, and will be discussed in detail in the next section (§5.1). Rules SCHEMEI and SCHEMEE introduce type abstractions and applications, respectively. Rule LET translates let expressions into applied lambdas.

Notably, elaboration erases first-class labels, as demonstrated in rule LABEL, where a first-class label is translated to a unit term. While labels are important for records and variants in the source calculus, their corresponding information is replaced by constant indices for the target's product and sum terms. As a result, row commutativity inserts explicit conversions to ensure that the order in elaborated product and sum terms consistently matches the order from the type (§5.4).

Rules PROD and SUM convert singleton record and variant terms into unlabelled product and sum terms in the target. In rule UNLABEL, M must already have a singleton row type, meaning only one element can be extracted. In these rules, the elaboration of the label is discarded.

Lastly, we explain rules PRJ, CONCAT, INJ, and ELIM along with elaborations of constraint solving in §5.2; rule SUB with the elaboration of subtyping in §5.4; and finally, rule IND in §5.3.

Elaboration of types and constraints. Fig. 8, 9, and 10 present elaboration environments and selected rules for type and kind elaboration respectively. These elaboration rules will be explained throughout the remainder of this section as their corresponding parts are discussed.

Fig. 11 presents selected constraint solving rules with their elaboration, which we discuss in §5.1 through §5.3. The full set of elaboration rules is provided in the appendix.

$$
\begin{array}{lcll}
\Gamma & ::= & \epsilon \mid \Gamma, a:\kappa \mid \Gamma, x:\sigma \mid \Gamma, \psi \rightsquigarrow x & \textit{type environment} \\
\Gamma_C & ::= & \epsilon \mid \Gamma_C, (\overline{TC'_i\ a_i \rightsquigarrow A'_i} \Rightarrow TC\ a:\kappa) \mapsto m:\sigma \rightsquigarrow A & \textit{class environment} \\
\Gamma_I & ::= & \epsilon \mid \Gamma_I, (\forall\ \overline{a_i:\kappa_i}.\ \overline{\psi_i \rightsquigarrow x_i} \Rightarrow TC\ \tau) \rightsquigarrow E; \overline{E'_i} & \textit{instance environment}
\end{array}
$$

Fig. 8. Elaboration environments

$\boxed{\Gamma_C; \Gamma \vdash \sigma : \kappa \rightsquigarrow A}$ *(Kinding And Elaboration)*

$$
\textsc{tc}\ \frac{(\overline{TC'_i\ a \rightsquigarrow A'_i} \Rightarrow TC\ a:\kappa) \mapsto m:\sigma \rightsquigarrow A \in \Gamma_C \qquad \Gamma_C; \Gamma \vdash \tau : \kappa \rightsquigarrow B}{\Gamma_C; \Gamma \vdash TC\ \tau : \mathbf{C} \rightsquigarrow \otimes\{A\,[B/a]\,, \overline{A'_i\,[B/a]}\}}
\qquad
\textsc{row}\ \frac{\overline{\Gamma_C; \Gamma \vdash \xi_i : \mathbf{L}} \qquad \vdash_{\mathcal{T}} \overline{\xi_i} \qquad \overline{\Gamma_C; \Gamma \vdash \tau_i : \kappa \rightsquigarrow A_i}}{\Gamma_C; \Gamma \vdash \langle \overline{\xi_i \triangleright \tau_i} \rangle : \mathbf{R}^\kappa \rightsquigarrow \{\overline{A_i}\}}
$$

$$
\textsc{prod}\ \frac{\Gamma_C; \Gamma \vdash \mu : \mathbf{U} \qquad \Gamma_C; \Gamma \vdash \rho : \mathbf{R}^\star \rightsquigarrow A}{\Gamma_C; \Gamma \vdash \Pi_\mu\ \rho : \star \rightsquigarrow \otimes A}
$$

$$
\textsc{comm}\ \frac{}{\Gamma_C; \Gamma \vdash u : \mathbf{U} \rightsquigarrow \otimes\{\}}
\qquad
\textsc{all}\ \frac{\Gamma_C; \Gamma, a:\kappa \vdash \psi : \mathbf{C} \rightsquigarrow A \qquad \vdash \kappa \rightsquigarrow K \qquad \Gamma_C; \Gamma \vdash \rho : \mathbf{R}^\kappa \rightsquigarrow B}{\Gamma_C; \Gamma \vdash \mathbf{All}\ (\lambda\ a:\kappa.\ \psi)\ \rho : \mathbf{C} \rightsquigarrow \otimes\ ((\lambda\ a:K.\ A)\ [\![B]\!])}
$$

$$
\textsc{contain}\ \frac{\Gamma_C; \Gamma \vdash \mu : \mathbf{U} \qquad \Gamma_C; \Gamma \vdash \rho_0 : \mathbf{R}^\kappa \rightsquigarrow A_0 \qquad \Gamma_C; \Gamma \vdash \rho_1 : \mathbf{R}^\kappa \rightsquigarrow A_1 \qquad \vdash \kappa \rightsquigarrow K}{\Gamma_C; \Gamma \vdash \rho_0 \lesssim_\mu \rho_1 : \mathbf{C} \rightsquigarrow \otimes\{\forall\ a:K \mapsto \star.\ (\otimes\ (a\ [\![A_1]\!])) \to \otimes\ (a\ [\![A_0]\!]), \forall\ a:K \mapsto \star.\ (\oplus\ (a\ [\![A_0]\!])) \to \oplus\ (a\ [\![A_1]\!])\}}
$$

$$
\textsc{ind}\ \frac{\begin{array}{c}\Gamma_C; \Gamma \vdash \rho : \mathbf{R}^\kappa \rightsquigarrow A \qquad \vdash \kappa \rightsquigarrow K \\ \Gamma_C; \Gamma, a_l : \mathbf{L}, a_t : \kappa, a_p\ a_i : \mathbf{R}^\kappa \vdash a_p \odot_{\mathfrak{u}} \langle a_l \triangleright a_t \rangle \sim a_i : \mathbf{C} \rightsquigarrow B_l \qquad \Gamma_C; \Gamma, a_i\ a_n : \mathbf{R}^\kappa \vdash a_i \odot_{\mathfrak{u}} a_n \sim \rho : \mathbf{C} \rightsquigarrow B_r \\ A_s = \forall\ a_l : \star, a_t : K, a_p\ a_i\ a_n : \mathbf{L}^K.\ B_l \to B_r \to (\otimes\{\}) \to (a_m\ a_p) \to a_m\ a_i\end{array}}{\Gamma_C; \Gamma \vdash \mathbf{Ind}\ \rho : \mathbf{C} \rightsquigarrow \forall\ a_m : (\mathbf{L}^K \mapsto \star).\ A_s \to (a_m\ \{\}) \to a_m\ A}
$$

Fig. 9. Type elaboration

$\boxed{\vdash \kappa \rightsquigarrow K}$ *(Kind Elaboration)*

$$
\frac{}{\vdash \star \rightsquigarrow \star}
\qquad
\frac{\vdash \kappa_0 \rightsquigarrow K_0 \qquad \vdash \kappa_1 \rightsquigarrow K_1}{\vdash \kappa_0 \mapsto \kappa_1 \rightsquigarrow K_0 \mapsto K_1}
\qquad
\frac{\vdash \kappa \rightsquigarrow K}{\vdash \mathbf{R}^\kappa \rightsquigarrow \mathbf{L}^K}
\qquad
\frac{}{\vdash \mathbf{C} \rightsquigarrow \star}
\qquad
\frac{}{\vdash \mathbf{L} \rightsquigarrow \star}
\qquad
\frac{}{\vdash \mathbf{U} \rightsquigarrow \star}
$$

Fig. 10. Kind elaboration

5.1 Dictionary-Passing Elaboration for Type Classes

We implement type classes using the *dictionary-passing elaboration* [Wadler and Blott 1989]. Intuitively, each type class corresponds to a data type whose entries represent its methods. Type class instances then correspond to values of these data types (i.e. *dictionaries*), which are implemented using product types in our target calculus. Throughout this work, we refer to the elaboration result of constraint solving as *evidence*, since our constraint solver also handles row constraints.

The type elaboration rule TC in Fig. 9 elaborates type classes to a tuple type. In the first hypothesis, A represents the elaboration of the type of the class's method σ, and A'_i are the elaborated types of the superclasses. We substitute the type variable a with the elaboration of τ. Similarly, the constraint kind $\mathbf{C}$ is elaborated to the base kind $\star$ in the target (Fig. 10).

As a result, in Fig. 3, terms with qualified types are elaborated into functions that take the corresponding evidence arguments (rule QUALI). Evidence arguments are automatically inserted after they are resolved (rule QUALE), and if a type class constraint can be resolved, the corresponding method can be accessed (rule METHOD).

Rule TCINST in Fig. 11 handles type classes. This rule applies when a corresponding entry for the required type class TC exists in the instance environment. In this rule, E corresponds to TC's method, and E'_i corresponds to the evidence for TC's superclasses. In the elaboration result, type variables are substituted with concrete types, and prerequisites are replaced with their corresponding evidence. The elaboration for constraint solving of superclasses (TCSUPER) is not presented here, but it simply projects an element out of the subclass evidence to obtain the superclass evidence.

$\boxed{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \psi \rightsquigarrow E}$ *(Constraint Solving And Elaboration)*

TCInst

$$\frac{(\forall \overline{a_k : \kappa_k}.\ \overline{\psi_j \rightsquigarrow x_j} \Rightarrow TC\ \tau) \rightsquigarrow E; \overline{E'_i} \in \Gamma_I \qquad \overline{\Gamma_C; \Gamma \vdash \tau'_k : \kappa_k \rightsquigarrow B_k} \qquad \overline{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \psi_j[\tau'_k/a_k] \rightsquigarrow F_j}}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} TC\ \tau[\tau'_k/a_k] \rightsquigarrow (E[B_k/a_k]\ \overline{[F_j/x_j]}, \overline{E'_i[B_k/a_k]\ [F_j/x_j]})}$$

concatContainL

$$\frac{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \odot_\mu \rho_1 \sim \rho_2 \rightsquigarrow E}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \lesssim_\mu \rho_2 \rightsquigarrow \pi_2\ E}$$

containDecay

$$\frac{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \lesssim_{\mu_0} \rho_1 \rightsquigarrow E \qquad \mu_0 \leqslant \mu_1 \qquad \Gamma_C; \Gamma \vdash \mu_1 : \mathbf{U}}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \lesssim_{\mu_1} \rho_1 \rightsquigarrow E}$$

splitConcat

$$\frac{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \mathbf{Split}\ (\lambda\, a : \kappa.\ \tau)\ \rho_0\ \rho_1\ \rho_2 \rightsquigarrow E}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} (\mathbf{Lift}\ (\lambda\, a : \kappa.\ \tau)\ \rho_0) \odot_\mathfrak{c} \rho_1 \sim \rho_2 \rightsquigarrow E}$$

liftContain

$$\frac{\begin{array}{c}\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \lesssim_\mu \rho_1 \rightsquigarrow E \qquad \Gamma_C; \Gamma \vdash \rho_0 : \mathbf{R}^{\kappa_0} \qquad \Gamma_C; \Gamma, a : \kappa_0 \vdash \tau : \kappa_1 \rightsquigarrow A \qquad \vdash \kappa_0 \rightsquigarrow K_0 \\ \vdash \kappa_1 \rightsquigarrow K_1 \qquad E_p = \Lambda\, a' : K_1 \mapsto \star.\ (\pi_0\ E)\ [\lambda\, a : K_0.\ a'\ A] \qquad E_i = \Lambda\, a' : K_1 \mapsto \star.\ (\pi_1\ E)\ [\lambda\, a : K_0.\ a'\ A]\end{array}}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} (\mathbf{Lift}\ (\lambda\, a : \kappa_0.\ \tau)\ \rho_0) \lesssim_\mu (\mathbf{Lift}\ (\lambda\, a : \kappa_0.\ \tau)\ \rho_1) \rightsquigarrow (E_p, E_i)}$$

allContain

$$\frac{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \rho_0 \lesssim_\mathfrak{c} \rho_1 \rightsquigarrow F \qquad \Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \mathbf{All}\ (\lambda\, a : \kappa.\ \psi)\ \rho_1 \rightsquigarrow E \qquad \Gamma_C; \Gamma, a : \kappa \vdash \psi : \mathbf{C} \rightsquigarrow A \qquad \vdash \kappa \rightsquigarrow K}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \mathbf{All}\ (\lambda\, a : \kappa.\ \psi)\ \rho_0 \rightsquigarrow (\pi_0\ F)\ [\lambda\, a : K.\ A]\ E}$$

ind

$$\frac{\begin{array}{c}\overline{\Gamma_C; \Gamma \vdash \tau_i : \kappa \rightsquigarrow A_i}^{i \in [:n]} \qquad \vdash \kappa \rightsquigarrow K \qquad \Gamma_C; \Gamma, a_l : \mathbf{L}, a_t : \kappa, a_p\ a_i : \mathbf{R}^\kappa \vdash a_p \odot_\mathfrak{n} \langle a_l \triangleright a_t \rangle \sim a_i : \mathbf{C} \rightsquigarrow B_l \\ \Gamma_C; \Gamma, a_i\ a_n : \mathbf{R}^\kappa \vdash a_i \odot_\mathfrak{n} a_n \sim \langle \overline{\ell_i \triangleright \tau_i}^{i \in [:n]} \rangle : \mathbf{C} \rightsquigarrow B_r \\ A_s = \forall\, a_l : \star, a_t : K, a_p\ a_i\ a_n : \mathbf{L}^K.\ B_l \to B_r \to (\otimes\ \{\}) \to (a_m\ a_p) \to a_m\ a_i \\ \overline{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \langle \overline{\ell_j \triangleright \tau_j}^{j \in [:i]} \rangle \odot_\mathfrak{n} \langle \ell_i \triangleright \tau_i \rangle \sim \langle \overline{\ell_k \triangleright \tau_k}^{k \in [:i+1]} \rangle \rightsquigarrow E_i}^{i \in [:n]} \\ \overline{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \langle \overline{\ell_j \triangleright \tau_j}^{j \in [:i+1]} \rangle \odot_\mathfrak{n} \langle \overline{\ell_k \triangleright \tau_k}^{k \in [i+1:n]} \rangle \sim \langle \overline{\ell_l \triangleright \tau_l}^{l \in [:n]} \rangle \rightsquigarrow E'_i}^{i \in [:n]} \\ E'' = ((x_s\ [\otimes\ \{\}]\ [A_i]\ [\{\overline{A_j}^{j \in [:i]}\}]\ [\{\overline{A_k}^{k \in [i+1]}\}]\ [\{\overline{A_l}^{l \in [i+1:n]}\}]\ E_i)\ E'_i)\ ()\qquad x_i \\ F = \Lambda\, a_m : (\mathbf{L}^K \mapsto \star).\ \lambda\, x_s : A_s.\ \lambda\, x_i : a_m\ \{\}.\ E''\end{array}}{\Gamma_I; \Gamma_C; \Gamma \vDash_{\tau_s} \mathbf{Ind}\ \langle \overline{\ell_i \triangleright \tau_i}^{i \in [:n]} \rangle \rightsquigarrow F}$$

Fig. 11. Selected elaboration rules for constraint solving

5.2 Interpretation of Row Constraints

We first discuss the elaboration of rows. Rule row in Fig. 9 elaborates rows to type-level lists in the target. As previously mentioned, $F_\omega^{\otimes\oplus}$'s type-level lists are much simpler than rows, with labels and commutativity erased. For example, rule prod discards the commutativity annotation μ. If a function is polymorphic over labels or commutativity in the source, the elaboration turns them into type abstractions taking empty products (rule comm). The corresponding kind elaboration rules elaborate $\mathbf{R}^\kappa$ to $\mathbf{L}^K$, and both $\mathbf{L}$ and $\mathbf{U}$ to $\star$ in Fig. 10.

Next, as row constraints are used primarily for records and variants, we follow Morris and McKinna [2019] and interpret row constraints as a collection of functions operating on products and sums. Therefore, the type elaboration rule contain in Fig. 9 elaborates row containment into a tuple. Compared to Morris and McKinna [2019], both components of our tuple are polymorphic over a type function. This function corresponds to the target's type-level mappings, a crucial feature in our higher-kinded system that is essential for solving **Lift** and **All** constraints. Then, the first function takes a product of the larger row and produces a product of the smaller one, effectively performing a projection. Similarly, the second function takes a sum of the smaller row and produces a sum of the larger one, performing an injection. The elaboration rule for row concatenation (given in the appendix) is similar, producing a product of four elements: the first two manage

concatenation for products and elimination for sums respectively, while the latter two provide corresponding containment evidence.

Fig. 11 presents two example rules for row constraints. Rule concatContainL simply selects the containment evidence from the combination evidence. Rule containDecay discards the commutativity information and produces the same evidence as its premise. Concrete evidence is generated in the elaboration of the constraint solving rule concatConcrete for concrete rows, which can be found in the appendix.

Returning to the term elaboration rules in Fig. 3, we can now see that rules involving row constraints (prj, concat, inj, and elim) simply select the appropriate evidence entries for the operations being performed. In these cases, the type-level mapping is just the identity type lambda, as the rows of the products and sums already match those specified in the constraints.

Lift and Split. Fig. 11 also includes constraint solving elaboration for **Lift**, where the type-level mappings in row constraints are crucial. Specifically, rule liftContain projects out the two entries from the original evidence. It then wraps them in type abstractions, applying these components to type lambdas that compose the outer mapping a' with the current mapping A.

The type elaboration for **Split** is identical to that of its associated concatenation constraint with the left side lifted. As a result, rule splitConcat simply produces the evidence from the premise.

All. The all type elaboration rule in Fig. 9 elaborates **All** to a product type. Here, the type-level mapping generates a type-level list, where each entry corresponds to evidence for ψ with a substituted by the corresponding entry in the row ρ. The resulting product allows us to access the evidence for any specific entry. Our encoding makes the elaboration for **All** constraints mostly straightforward, primarily involving the creation and projection of products.

As an example, rule allContain in Fig. 11 extracts the projection function from the containment evidence, applies it to the elaborated type lambda, and then to the **All** evidence for the larger row ρ_1, effectively projecting out the subset of fields for the smaller row ρ_0.

5.3 Elaboration of ind

We now consider the elaboration of **ind**, which performs a fold based on a row type. Elaborating this term is non-trivial, as the types of the row constraints and the accumulator depend on the portion of the row processed so far. One straightforward approach is to incorporate a similar construct directly into the target calculus, elaborating **ind** to this new construct. However, this would complicate the target calculus and make compilation to more standard target calculi more challenging.

In this work, we explore an alternative compilation strategy for **ind**. Specifically, we introduce a new form of constraint, **Ind** ρ. When **ind** is used, the **Ind** constraint is required and can be passed around just like any other constraint, as specified in the ind typing rule in Fig. 3. During elaboration, when **Ind** ρ is resolved, it produces an evidence term F that performs the fold of ρ. The term elaboration then simply applies it to the type lambda, step function, and initial value. With this elaboration strategy for **ind**, it is worth noting that the examples from §2 where we used **ind** would now require the appropriate **Ind** constraints in their types. We are interested in potentially hiding these constraints in an actual implementation using the approach of *total type classes* [Weingart et al. 2024], but we leave this for future work.

Rule ind in Fig. 9 specifies the type elaboration rule for **Ind**. The elaborated type is polymorphic over the type lambda a_m. Like the **ind** term, it accepts a step function and an initial value, then produces a final result based on the type function and elaborated row. The step function A_s is polymorphic over various components of the row and takes the left and right constraint evidence, in addition to a unit value as the elaborated label and the accumulator, producing the next accumulator.

$\boxed{\Gamma_C; \Gamma \vdash \sigma_0 <: \sigma_1 \rightsquigarrow F}$ *(Subtyping And Elaboration)*

$$\textsc{prodRow}\ \frac{\Gamma_C; \Gamma \vdash \rho_0 \equiv_\mu \rho_1 \rightsquigarrow F_p, F_s \qquad \Gamma_C; \Gamma \vdash \Pi_\mu\ \rho_0 : \star}{\Gamma_C; \Gamma \vdash \Pi_\mu\ \rho_0 <: \Pi_\mu\ \rho_1 \rightsquigarrow F_p\ [\lambda\, a : \star.\, a]} \qquad \textsc{decay}\ \frac{\Gamma_C; \Gamma \vdash \Xi_{\mu_0}\ \rho : \star \rightsquigarrow A \quad \Gamma_C; \Gamma \vdash \mu_1 : \mathbf{U} \qquad \mu_0 \leqslant \mu_1}{\Gamma_C; \Gamma \vdash \Xi_{\mu_0}\ \rho <: \Xi_{\mu_1}\ \rho \rightsquigarrow \lambda\, x : A.\, x}$$

Fig. 12. Subtyping and elaboration

$\boxed{\Gamma_C; \Gamma \vdash \rho_0 \equiv_\mu \rho_1 \rightsquigarrow F_p, F_s}$ *(Row Equivalence And Elaboration)*

$$\textsc{comm}\ \frac{\begin{array}{c}\vdash_{\mathcal{T}} p\ \textbf{permutes}\ [:n]\ \textbf{for}\ \xi \qquad \vdash_{\mathcal{T}} p'\ \textbf{permutes}\ [:n]\ \textbf{for}\ \xi \qquad p\ \textbf{inverts}\ p'\ \textbf{on}\ [:n] \\ \Gamma_C; \Gamma \vdash \langle \overline{\xi_i \triangleright \tau_i}^{\,i \in [:n]} \rangle : \mathbf{R}^\kappa \rightsquigarrow \{ \overline{A_i}^{\,i \in [:n]} \} \qquad \vdash \kappa \rightsquigarrow K \qquad F_p = \Lambda\, a : K \mapsto \star.\, \lambda\, x : \otimes\, (a\, [\![\{ \overline{A_i}^{\,i \in [:n]} \}]\!]).\, (\overline{\pi_i\, x}^{\,i \in p}) \\ F_s = \Lambda\, a : K \mapsto \star.\, \lambda\, x : \oplus\, (a\, [\![\{ \overline{A_i}^{\,i \in [:n]} \}]\!]).\, \mathbf{case}\ x\ \{ \overline{\lambda\, x' : a\, A_i.\, \iota_j\, x'}^{\,i \in [:n], j \in p'} \}\end{array}}{\Gamma_C; \Gamma \vdash \langle \overline{\xi_i \triangleright \tau_i}^{\,i \in [:n]} \rangle \equiv_{\mathfrak{c}} \langle \overline{\xi_i \triangleright \tau_i}^{\,i \in p} \rangle \rightsquigarrow F_p, F_s}$$

Fig. 13. Row equivalence and elaboration

The constraint solving rule IND in Fig. 11 resolves **Ind** for concrete rows. The first five hypotheses collect types and kinds needed when describing the evidence. Since each step of the fold performed by **ind** is provided two concatenation constraints, the next two hypotheses collect the evidence for these. E'' then assembles the body of the evidence function, in which the step function argument x_s (of type A_s) is applied once for each entry in the row, performing the fold. Each repetition within E'''s comprehension passes five type arguments: the unit type for the erased label type, the type of the current row entry, and the three sub-components of the larger row. It also passes the correct left and right concatenation constraint evidence, then a unit for the erased label term. Finally, F wraps the body E'' in a type abstraction, as well as abstractions taking the step function and the initial value term respectively.

5.4 Subtyping and Row Equivalence

Fig. 12 and 13 present selected elaborations for subtyping and row equivalence. Subtyping produces functions that convert a term from the elaborated subtype to the elaborated supertype, while row equivalence produces two functions for conversions involving products and sums, respectively.

Rule PRODROW uses the elaboration for products from the equivalence relation, and applies it to the identity type lambda. While the type-level mapping is not useful here, it is necessary for cases such as **All**. Rule DECAY simply produces an identity function, since commutativities are erased. The elaboration rule COMM for commutative rows re-orders the entries within the elaboration to match the re-ordering of the entries in the source row.

5.5 Elaboration Soundness

In this section, we prove that our elaboration is sound, specifically with row theory $\mathcal{T}_\mathbf{s}$, beginning with soundness of types and constraints:

Theorem 5.1 (Elaboration Soundness of Types and Constraints)**.**

(Kinding) *If* $\Gamma_C; \Gamma \vdash \sigma : \kappa \rightsquigarrow A$, *and* $\vdash \kappa \rightsquigarrow K$, *and* $\Gamma_C \vdash \Gamma \rightsquigarrow \Delta$, *then* $\Delta \vdash A : K$.

(Row Equivalence) *If* $\Gamma_C; \Gamma \vdash \rho_0 \equiv_\mu \rho_1 \rightsquigarrow F_p, F_s$, $\Gamma_C; \Gamma \vdash \rho_0 : \mathbf{R}^\kappa \rightsquigarrow A$, $\Gamma_C; \Gamma \vdash \rho_1 : \mathbf{R}^\kappa \rightsquigarrow B$, $\vdash \kappa \rightsquigarrow K$, *and* $\Gamma_C \vdash \Gamma \rightsquigarrow \Delta$, *then* $\Delta \vdash F_p : \forall\, a : K \mapsto \star.\, (\otimes\, (a\, [\![A]\!])) \to \otimes\, (a\, [\![B]\!])$ *and* $\Delta \vdash F_s : \forall\, a : K \mapsto \star.\, (\oplus\, (a\, [\![A]\!])) \to \oplus\, (a\, [\![B]\!])$.

(Subtyping) *If* $\Gamma_C; \Gamma \vdash \sigma_0 <: \sigma_1 \rightsquigarrow F$, $\Gamma_C; \Gamma \vdash \sigma_0 : \kappa \rightsquigarrow A$, $\Gamma_C; \Gamma \vdash \sigma_1 : \kappa \rightsquigarrow B$, $\vdash \kappa \rightsquigarrow \star$, *and* $\Gamma_C \vdash \Gamma \rightsquigarrow \Delta$, *then* $\Delta \vdash F : A \to B$.

Table 1. Code statistics for the Lean 4 mechanization

Module	Description	LOC	#Def	#Thm
Source Language ($\lambda^{\Rrightarrow}_{\rho}$)				
Syntax	Kinds, types, terms, programs, and environments	157	21	/
Semantics	Kinding, typing, constraint solving, elaboration, etc.	1 052	54	/
Lemmas	Auxiliary lemmas and properties	3 665	4	121
Theorems	Elaboration soundness	7 430	/	14
Target Language ($\mathrm{F}^{\otimes\oplus}_{\omega}$)				
Syntax	Kinds, types, terms, and environments	175	13	/
Semantics	Kinding, typing, and operational semantics	697	59	/
Lemmas	Auxiliary lemmas and properties	4 042	4	277
Type Reduction	Strong normalization, confluence, correspondence to equivalence, etc.	4 965	8	182
Theorems	Syntactic type safety	566	/	17
Total		**22 749**	**163**	**611**

(Constraint Solving) *If* $\Gamma_I; \Gamma_C; \Gamma \vDash_{\mathcal{T}} \psi \rightsquigarrow E$, $\Gamma_C; \Gamma \vdash \psi : \mathbf{C} \rightsquigarrow A$, *and* $\Gamma_C \vdash \Gamma \rightsquigarrow \Delta$, *then* $\Delta \vdash E : A$.

This theorem has several parts for kinding, row equivalence, subtyping, and constraint solving, respectively. The judgement $\Gamma_C \vdash \Gamma \rightsquigarrow \Delta$ states that the environment Γ is well-formed under the class environment Γ_C and elaborates to the target environment Δ; for clarity, we implicitly assume all class and instance environments are well-formed. The proof of the constraint solving part is notably lengthy, primarily due to the verbosity of the row elaboration rules. The target's equivalence rules (listAppComp and listAppId) for type-level lists are essential in the proofs for **Lift**.

Finally, we prove program and term elaboration is sound; we present the statement for terms:

Theorem 5.2 (Elaboration Soundness of Terms)**.**
If $\Gamma_I; \Gamma_C; \Gamma \vdash M : \sigma \rightsquigarrow E$, *and* $\Gamma_C; \Gamma \vdash \sigma : \star \rightsquigarrow A$, *and* $\Gamma_C \vdash \Gamma \rightsquigarrow \Delta$, *then* $\Delta \vdash E : A$.

6 Mechanization

We have formalized the metatheory of $\lambda^{\Rrightarrow}_{\rho}$ and $\mathrm{F}^{\otimes\oplus}_{\omega}$ using the Lean 4 proof assistant [Moura and Ullrich 2021], and the proofs are available in the artifact [Toohey et al. 2025]. Table 1 summarizes the structure and statistics of our mechanization. Our formalization adopts the locally nameless representation [Charguéraud 2011] for handling binding and leverages the *Aesop* proof search tactic [Limperg and From 2023] for proof automation. The proof of strong normalization for $\mathrm{F}^{\otimes\oplus}_{\omega}$'s type reduction took inspiration from a strong normalization proof for the simply typed lambda calculus in Lean [Mameche 2019], and this was used to establish confluence by importing a prior proof of Newman's lemma [van Kampen 2025].

There were two notable obstacles we encountered and solved in the mechanization process. First, our formalization employs nested inductive types to define rows in $\lambda^{\Rrightarrow}_{\rho}$ and type-level lists in $\mathrm{F}^{\otimes\oplus}_{\omega}$. Unfortunately, the standard induction tactic in Lean 4 does not automatically handle nested inductive types. We addressed this by defining specialized induction principles for these constructs that explicitly deal with inductive reasoning on lists.

Second, we define the type equivalence judgement as a proposition. To prove that type equivalence ($\Delta \vdash A \equiv B$) implies the equivalence closure of type reduction ($\Delta \vdash A \leftrightarrow^* B$), we needed to establish that any type equivalence proof could be transformed into a derivation where symmetry (symm) and transitivity (trans) rules are applied only at the top level. A natural approach would be to define a proposition indexed by such a derivation to capture this property. However, this does not work because Lean's metatheory implies proof irrelevance, rendering multiple proofs of the same type equivalence judgement definitionally indistinguishable. To overcome this limitation, we

Table 2. B2T2 table API functions

Category	Supported	Description
Constructors	9/9	Creating, combining, and expanding tables.
Properties	3/3	Querying table shape information.
Access subcomponents	3/4	Extracting a row, value, or column.
Subtable	8/10	Extracting a subset of the table.
Ordering*	3/3	Sorting. *orderBy* requires existential types.
Aggregate*	4/4	Reducing multiple rows into fewer summary rows.
Missing values	3/3	Handling for optional values.
Data cleaning	2/2	Pivoting longer or wider.
Utilities*	11/11	Misc. *renameColumns* also needs more row operators.

introduced a new judgement $\Delta \vdash A \equiv_s B$ that enforces the desired property by construction, and prove equivalence between $\Delta \vdash A \equiv_s B$ and $\Delta \vdash A \equiv B$, complicating the metatheory development.

7 Evaluation

The Brown Benchmark for Table Types (B2T2) [Lu et al. 2021] provides criteria for evaluating type systems designed for tabular data. Given that record types are a well-known encoding for table types, we evaluate our type system against the B2T2 set of table API functions. For the purposes of this evaluation we assume support for lists, though they were not included in our formalism.

Table 2 lists the B2T2 benchmark's table API functions. We represent *table schemas* (column names and their corresponding types) as rows, and tables as lists of products. Our system offers several benefits: First, extensible rows readily express table schemas, while type classes provide necessary methods for operations like equality and comparison, and **All** enables defining generic functions over rows whose fields all possess a certain property. Moreover, our commutativity hierarchy allows us to use commutativity for functions where ordering is irrelevant, and non-commutativity when table columns must be ordered and folded deterministically. Lastly, our **Lift** and **Split** enable type-level updates to rows (similar to the *unlift* example in §2.2).

We discuss a few examples. The *leftJoin* constructor function takes two tables, *t1* and *t2*. It merges fields from *t2* into *t1* based on the shared columns, creating a new table. Fields are filled with *Nothing* if no matches are found in *t2*. The function has type:

$$\begin{array}{l} \mathit{leftJoin} : \forall (\mathit{rc}\ \mathit{rl'}\ \mathit{rr'}\ \mathit{rl}\ \mathit{rr}\ r : R)\ (\mu : U). \\ \quad (\mathit{All}\ \mathit{Eq}\ \mathit{rc}, \mathit{rc}\ \odot_\mu\ \mathit{rl'} \sim \mathit{rl}, \mathit{rc}\ \odot_\mu\ \mathit{rr'} \sim \mathit{rr}, \mathit{rl}\ \odot_\mu\ \mathit{Lift}\ \mathit{Option}\ \mathit{rr'} \sim r) \\ \qquad \Rightarrow \mathit{List}\ (\Pi_\mu\ \mathit{rl}) \to \mathit{List}\ (\Pi_\mu\ \mathit{rr}) \to \mathit{List}\ (\Pi_\mu\ r) \end{array}$$

Here, *All* requires that the *rc* row in the join must implement *Eq*. The row constraints describe the relationships between different variables, with the last one specifying that the output table's row *r* consists of the left row *rl*, concatenated with an *Option*-*Lift*ed version of *rr'* in the right row.

As another example, the data cleaning function *pivotWider* converts a table containing a *key* column and a *value* column into to a wider table. The wider table will have a column for each distinct case of the *key* variant. The entries in these new columns are the corresponding entries from the original *value* column, if such an entry exists. We have the function's type as:

$$\begin{array}{l} \mathit{pivotWider} : \forall (r\ \mathit{rk}\ \mathit{rr}\ r' : R)\ (\mathit{lk}\ \mathit{lv} : L)\ (t : \star)\ (\mu : U). \\ \quad (\mathit{All}\ \mathit{Eq}\ \mathit{rr}, \langle \mathit{lk} \triangleright \Sigma_\mu\ (\mathit{Lift}\ (\mathit{Const}\ \mathit{Unit})\ \mathit{rk}), \mathit{lv} \triangleright t \rangle \odot_\mu\ \mathit{rr} \sim r, \mathit{rr}\ \odot_\mu\ \mathit{Lift}\ (\mathit{Const}\ (\mathit{Option}\ t))\ \mathit{rk} \sim r') \\ \qquad \Rightarrow \mathit{List}\ (\Pi_\mu\ r) \to \lfloor \mathit{lk} \rfloor \to \lfloor \mathit{lv} \rfloor \to \mathit{List}\ (\Pi_\mu\ r') \end{array}$$

Here, row *r* consists of *lk* for the key, *lv* for the value, and *rr* for the rest. *All* requires all entries in *rr* to satisfy *Eq* so that rows, identical except for their *keys* and *values*, can be combined in the

result. This example also demonstrates the utility of rows for both products and sums: the *rk* row is *Lift*ed to units within the *lk* variant, while it is *Lift*ed to *Option t* in the *r'* product in the result.

Lastly, we note that our system cannot implement three functions that access columns based on *Int* or *Bool* inputs, as they require types to depend on terms (i.e. *dependent types*). And our implementations for three other functions (their categories are marked with *) are slightly weaker than their specified requirements. For example, while two functions in the benchmark process multiple columns simultaneously, our system can only operate on a single column at a time. Despite lacking more advanced types for these cases, we believe that this evaluation effectively demonstrates the utility of extensible rows with type classes for table types.

8 Related Work

Row types and polymorphism. Morris and McKinna [2019] propose Rose, a general framework for abstracting and unifying row theories, which Hubers and Morris [2023] later extend to support generic programming over rows. Our calculus builds upon Rose, but extends it in several ways. First, we support type classes, featuring a novel form of *All* constraints that allow us to express class constraints over polymorphic rows. In contrast, Hubers and Morris [2023] applies the same function uniformly across all fields of a row. As a result, it requires explicit passing, selection, and application of evidence, similar to our target calculus. Moreover, while their work supports three language primitives (called syn, ana, and fold) for generic operations on records and variants, our *ind* construct can be used in place of all three. Furthermore, we support row commutativity annotations over records, variants, and constraints, as well as commutativity polymorphism, and the ability to split rows according to type shapes using *Split*.

Generic programming over algebraic data types (ADTs). Generic programming has a rich literature. Of particular relevance to our work, there are several attempts to support generic folding operations over products and ADTs. Chlipala [2010] introduces a `folder` type family for type-level records, enabling generic operations over record structures. However, their approach does not specify the actual implementation of the folder function and relies on field name ordering as hints for permutations. Other relevant works include the `Typable` type class by Lämmel and Jones [2003] for generic traversals over ADTs, and an All type family over ADTs by de Vries and Löh [2014] that is semantically similar to our *All* constraint. However, these approaches cannot easily extend or mutate field types or orders due to the inherent limitations of ADTs.

Tabular types. Tabular type systems are an umbrella term for type systems supporting generic programming over tabular data. Lu et al. [2021] introduces the B2T2 benchmark suite for tabular type systems. It's a well-known approach to use polymorphic records and variants to encode tabular data. Our evaluation (§7) shows that $\lambda_{\rho}^{\Rrightarrow}$ can express a substantial portion of tabular operations in the benchmark. Another natural approach is to use *dependent types*, as demonstrated by an implementation of the benchmark in *Idris2* by Wright et al. [2022], which leverages the prover's proof search capabilities for evidence reconstruction.

Acknowledgments

This work is funded by the Natural Sciences and Engineering Research Council of Canada.

Data Availability Statement

The Lean 4 proofs of this paper are provided in the artifact [Toohey et al. 2025].

References

Bernard Berthomieu and Camille Le Monies De Sagazan. 1995. A Calculus of Tagged Types, with applications to process languages. *Types for Program Analysis* (1995), 1.

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. doi:10.1017/S095679681300018X

Luca Cardelli and John C. Mitchell. 1990. Operations on records. In *Mathematical Foundations of Programming Semantics*, M. Main, A. Melton, M. Mislove, and D. Schmidt (Eds.). Springer, New York, NY, 22–52. doi:10.1007/BFb0040253

Arthur Charguéraud. 2011. The Locally Nameless Representation. *Journal of Automated Reasoning* 49, 3 (May 2011), 363–408. doi:10.1007/s10817-011-9225-2

Adam Chlipala. 2010. Ur: statically-typed metaprogramming with type-level record computation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 122–133. doi:10.1145/1806596.1806612

Edsko de Vries and Andres Löh. 2014. True sums of products. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming* (Gothenburg, Sweden) *(WGP '14)*. Association for Computing Machinery, New York, NY, USA, 83–94. doi:10.1145/2633628.2633634

Jacques Garrigue. 1998. Programming with Polymorphic Variants. In *ML workshop*, Vol. 13. Baltimore.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and types*. Cambridge University Press, USA.

Robert Harper and Benjamin Pierce. 1991. A record calculus based on symmetric concatenation. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Orlando, Florida, USA) *(POPL '91)*. Association for Computing Machinery, New York, NY, USA, 131–142. doi:10.1145/99583.99603

Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development* (Nara, Japan) *(TyDe 2016)*. Association for Computing Machinery, New York, NY, USA, 15–27. doi:10.1145/2976022.2976033

Alex Hubers and J. Garrett Morris. 2023. Generic Programming with Extensible Data Types: Or, Making Ad Hoc Extensible Data Types Less Ad Hoc. *Proc. ACM Program. Lang.* 7, ICFP, Article 201 (Aug. 2023), 29 pages. doi:10.1145/3607843

Mark P. Jones. 2003a. *Qualified types: theory and practice*. Number 9. Cambridge University Press.

Simon Peyton Jones. 2003b. *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

Ralf Lämmel and Simon Peyton Jones. 2003. Scrap your boilerplate: a practical design pattern for generic programming. *SIGPLAN Not.* 38, 3 (Jan. 2003), 26–37. doi:10.1145/640136.604179

Daan Leijen. 2004. *First-class labels for extensible rows* (technical report uu-cs-2004-51 ed.). Technical Report UU-CS-2004-51. https://www.microsoft.com/en-us/research/publication/first-class-labels-for-extensible-rows/ UTCS Technical Report.

Daan Leijen. 2005. Extensible records with scoped labels. In *Proceedings of the 2005 Symposium on Trends in Functional Programming (TFP'05), Tallin, Estonia* (proceedings of the 2005 symposium on trends in functional programming (tfp'05), tallin, estonia ed.). https://www.microsoft.com/en-us/research/publication/extensible-records-with-scoped-labels/

Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 486–499. doi:10.1145/3009837.3009872

Jannis Limperg and Asta Halkjær From. 2023. Aesop: White-Box Best-First Proof Search for Lean. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Boston, MA, USA) *(CPP 2023)*. Association for Computing Machinery, New York, NY, USA, 253–266. doi:10.1145/3573105.3575671

Sam Lindley and James Cheney. 2012. Row-based effect types for database integration. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(TLDI '12)*. Association for Computing Machinery, New York, NY, USA, 91–102. doi:10.1145/2103786.2103798

Sam Lindley and J. Garrett Morris. 2017. Lightweight functional session types. *Behavioural Types: from Theory to Tools. River Publishers* (2017), 265–286.

Kuang-Chen Lu, Ben Greenman, and Shriram Krishnamurthi. 2021. Types for Tables: A Language Design Benchmark. *The Art, Science, and Engineering of Programming* 6, 2 (Nov. 2021). doi:10.22152/programming-journal.org/2022/6/8

Sarah Mameche. 2019. Strong Normalization of the λ-calculus in Lean.

J. Garrett Morris and James McKinna. 2019. Abstracting extensible data types: or, rows by any other name. *Proc. ACM Program. Lang.* 3, POPL, Article 12 (Jan. 2019), 28 pages. doi:10.1145/3290325

Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, André Platzer and Geoff Sutcliffe (Eds.). Springer International Publishing, Cham, 625–635. doi:10.1007/978-3-030-79876-5_37

Maxwell Herman Alexander Newman. 1942. On Theories with a Combinatorial Definition of "Equivalence". *Annals of Mathematics* 43, 2 (1942), 223–243. doi:10.2307/1968867

Adam Paszke and Ningning Xie. 2023. Infix-Extensible Record Types for Tabular Data. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Type-Driven Development* (Seattle, WA, USA) *(TyDe 2023)*. Association for Computing

Machinery, New York, NY, USA, 29–43. doi:10.1145/3609027.3609406

Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

D. Rémy. 1989. Type checking records and variants in a natural extension of ML. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 77–88. doi:10.1145/75277.75284

Didier Rémy. 1992. Typing record concatenation for free. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico, USA) *(POPL '92)*. Association for Computing Machinery, New York, NY, USA, 166–176. doi:10.1145/143165.143202

Didier Rémy and Jérôme Vouillon. 1998. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems* 4, 1 (1998), 27–50. doi:10.1002/(SICI)1096-9942(1998)4:1<27::AID-TAPO3>3.0.CO;2-4

Mark Shields and Erik Meijer. 2001. Type-indexed rows. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (London, United Kingdom) *(POPL '01)*. Association for Computing Machinery, New York, NY, USA, 261–275. doi:10.1145/360204.360230

Lau Skorstengaard. 2019. An Introduction to Logical Relations. (2019). arXiv:1907.11133 [cs.PL] doi:10.48550/arXiv.1907.11133

Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Theorem Proving in Higher Order Logics*, Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 278–293. doi:10.1007/978-3-540-71067-7_23

Matthew Toohey, Yanning Chen, Ara Jamalzadeh, and Ningning Xie. 2025. *Extensible Data Types with Ad-Hoc Polymorphism (Artifact)*. doi:10.5281/zenodo.17298033

Sam van Kampen. 2025. Abstract Rewriting Formalized in Lean.

P. Wadler and S. Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 60–76. doi:10.1145/75277.75283

Mitchell Wand. 1987. Type inference for simple objects. In *Proc., IEEE Symposium on Logic in Computer Science*. 37–44.

Mitchell Wand. 1991. Type inference for record concatenation and multiple inheritance. *Information and Computation* 93, 1 (1991), 1–15. doi:10.1016/0890-5401(91)90050-C Selections from 1989 IEEE Symposium on Logic in Computer Science.

Robert Weingart, Nicolas Wu, and Cristian Cadar. 2024. Total Type Classes: Improving the ergonomics of type-level programming in Haskell. (2024).

Robert Wright, Michel Steuwer, and Ohad Kammar. 2022. Idris2-Table: evaluating dependently-typed tables with the Brown Benchmark for Table Types.

Received 2025-07-10; accepted 2025-11-06