Records, Variants, and Row Types

Alex Hubers
Department of Computer Science, The University of Iowa
ahubers@uiowa.edu

Abstract

Records and variants pervade programming languages. Records give a foundational model to objects (à la object-oriented programs) and variants to algebraic data types (à la functional programs). Many type systems have been proposed to ensure static type safety and other metatheoretic properties of systems with records and variants. Row types describe one approach, in which records and variants are constructed from maps associating labels with types, or *rows*. This report describes row type systems from their foundation to today. It also advertises recent work on Rω, a higher-order row type system with first-class labels.

1 Introduction

Records and variants pervade programming languages: records model objects and modules; variants, dual to records, model algebraic data types. Respectively, these are the foundational data types of object-oriented programming and functional programming. Despite their ubiquity, they have proven notoriously difficult to extend simultaneously in both their *representations* and their *behaviors*. Wadler [49] termed this the *expression problem*: adding new methods to a class interface requires refactoring its implementors; adding new cases to an algebraic data type requires refactoring its functions. *Row types* are one approach to solving this problem based on the simple observation that both records and variants are constructed from the same matter: associations of labels to types, or *rows*.

Wand [51] proposed row type systems to safely type record extensibility. Row type systems have since been used to describe many additional aspects of record and variant calculi, such as record concatenation [53], type inference and polymorphism [43], efficient compilation schemes [11], first-class mixin modules [28], and first-class labels [20]. Row type systems have also seen successful application in other active research areas, such as algebraic effects [23, 14] and session types [26]. The structure rows take in each of these type systems is not uniform, but at their intersection is the simple observation above: rows map labels to types.

This report introduces readers to row type systems modulo the numerous row theories above. In this sense, it can serve as a sort of tutorial for those unfamiliar with row type systems. We place special emphasis on row type systems with qualified types; the reader should not confuse their presence here as ubiquity in row type systems writ large. Rather, in sections 3 and 4, we offer a view of row type systems catered towards our (my) interests in two qualified type systems, ROSE [34] and Rω [15]. These systems are discussed (resp.) in sections 5 and 6. Both are type theories *parametric over row theories*—that is, they may be instantiated with theories of rows from the other works cited. Thus, despite some restricted scope, we must still cover quite a bit of the literature. We begin by motivating rows in the context of the (still) open problems they were first designed to address.

2 Background

We introduce the reader to records, variants, and open problems in the expression thereof. This motivates the formal introduction of rows in the next section. Examples are given in Haskell-style notation and analogies are drawn between row type systems and Haskell as it is implemented in GHC [1]. We presume of the reader

some familiarity with the two.

2.1 Records

Records group names, or *labels*, with data. They generalize finite products of data (à la set or category theory). Consider (in abstract) such a product now.

```
p = (1, 2)
```

What does `p` represent? Some sort of point? It is common wisdom in software design that "the ratio of time spent reading versus writing code is well over ten to one." [29] Let's let the next reader know what we are writing.

```
p = (x = 1, y = 2)
```

The term `p` is now a record that groups labels `x` and `y` with values 1 and 2. It is not ambiguous to us that the first component is `x` and the second component `y`. Further, we can group our code according to this specification: functions can be written just for points, and so on. This is good software design, and has seen widespread adoption in the form of objects, albeit in different syntax. Try Java's[1]:

```
1  class 2dPoint {
2    x , y :: float
3
4    void 2dPoint(float x, float y) {
5      this.x = x;
6      this.y = y;
7    }
8  }
```

We construct objects in Java by specifying a value for each label, as illustrated by the following in combination with lines 5 and 6 above.

```
2dPoint p = new 2dPoint(1.0, 2.0)
```

Dually, we destruct a record by projecting a value from a given label, e.g., `p.x` and `p.y`.

2.2 Variants

Variants express the choice of one label among many. The algebraic data types of e.g. Haskell are variants, whose constructors are labels. Let's consider a simple

[1] We will elide the `public`, `private`, and `static` annotations from Java's syntax, none of which have meaningful context w.r.t. the systems in this report.

example.

```
data Expr = Val Int | Plus Expr Expr
```

The `Expr` data type is an abstract syntax tree (AST) for an arithmetic language with addition over integer literals. We represent this AST as a variant with labels `Val` and `Plus` mapped (resp.) to `Int` and (recursively) `Expr` $\times$ `Expr`. We construct a variant by choosing one label to occupy, e.g., we may declare a synonym for the value one as

```
one :: Expr
one = Val 1
```

and then represent the addition of one and one as:

```
two :: Expr
two = Plus one one
```

We destruct a variant by specifying what to do in each case. This is specified by a series of equations in Haskell. For example, it is straightforward to write an evaluator in this manner below.

```
eval :: Expr -> Int
eval (Val x)    = x
eval (Plus x y) = eval x + eval y
```

Each equation handles a case. The first case returns the `Int` held in `Val x`; the second case evaluates each subtree and adds the result. Observe:

```
> eval two
  2
```

Variants are dual to records: we construct records (and destruct variants) by assigning each label a value; we construct variants (and destruct records) by specifying the value at (resp., behavior of) one label. This follows directly from the duality of products and sums.

2.3 The Duality of Products and Sums

It will be helpful for the reader to think of records, variants, products and sums not as what they *are* but as what they *do*—or, to borrow a well known maxim in category theory, "it's the arrows that matter". They are dual in the latter sense. We illustrate this now, which should aid the reader's conceptualization of the next two sections.

Consider products and disjoint sums in set theory. If we have an element $x \in A \times B$, then we may destruct x in one of two ways—we either grab the element of

set A or the element of set B.

$$A \xleftarrow{fst} A \times B \xrightarrow{snd} B$$

Dually, if we have element $x \in A \uplus B$, then x was constructed in one of two ways: as either a *left* element of set A or a *right* element of set B.

$$A \xrightarrow{left} A + B \xleftarrow{right} B$$

These constructions generalize to n-ary finite products and sums as well as to labeled records and variants—in all you will see the same duality of arrows. We may think of the arrows out of records as specifying the record's *behavior*, and the arrows into variants as specifying the variant's *cases*. As data types are variants, the arrows inward are the fixed constructors; as objects are records, the arrows outward are the fixed destructors (i.e, the *interface*). Fixing each of these permits the other to extend easily—data types can freely gain more behaviors, while records can freely gain more cases. It is difficult however to let a data type or record extend freely in both directions. This is a well known problem in programming language design, which we describe next.

2.4 The Expression Problem

Variants are not freely extensible: we cannot extend the `Expr` type with a new case without altering its definition and the definitions of its behaviors (e.g. `eval`). This was already an "old problem" when Wadler [49] termed it *the expression problem*, stating that:

> The goal is to define a data type by cases, where one can add new cases to the data type and new functions over the data type, without recompiling existing code, and while retaining static type safety.

The name is a pun: Wadler is describing a difficulty in *expressing* variant extension by benchmarking it literally with a data type for arithmetic *expressions*. The benchmark goes like this: add a new case to the `Expr` data type (def. §2.2), like multiplication.

```
data Expr = Val Int | Plus Expr Expr | Mult Expr Expr
```

The `eval` function is no longer totally defined on all cases, and applications such as the following will incur runtime errors.

Expr	eval	print
Var	...	...
Plus	...	...

Figure 1: The `Expr` data type as a table

```
eval (Mult (Val 1) (Val 2))
```

To fix this application, we must refactor `eval` to be well-defined for inputs constructed by the `Mult`. This is not too bad for just one function, but the work is multiplied by all functions over `Expr`. Dually, we may want to write nano-passes over ASTs during compilation so that AST manipulation is modular. When blown to scale, this can have serious implications in domains such as compiler design. Consider Leroy's [24] compcert compiler, which translates from C source code through eight intermediate languages—many of which have substantial syntactic overlap—on its way to assembly.

Wadler illustrates the expression problem as a table in which the rows and columns are (resp.) the cases and functions of the data type (Figure 1). When programming with variants, the rows are fixed, and so adding new columns (functions) is easy but new rows (cases) difficult. Dually, when programming with records, the columns are fixed, and so adding rows (inheritors) is easy but columns (new methods) difficult.[2]

Consider the dual in Java (Figure 2). As we are declaring a record (not variant), construction is dual to the Haskell case: rather than specify the variant constructors, we specify the types of eliminators in the `Expr` interface (Figure 2A, lines 1-3). The interface says that valid expressions must evaluate to integers. We declare each case as implementing the `Expr` interface (lines 5-15 and 17-28). The logic in each class is functionally equivalent to that of the Haskell implementation above: the literal case stores int literal `x` (lines 8-10) and returns it upon evaluation (lines 12-15); the addition case stores left and right `Expr` subtrees (lines 20-23) and evaluates to the sum of each tree's evaluation (lines 25-27). Adding a new case for multiplication (Figure 2B) is easy and the other cases remain well-typed. However, changing the interface (Figure 2C) makes each class ill-typed:

```
> javac Expr.java
  Expr.java error: Lit does not override method print()
  Expr.java error: Plus does not override method print()
```

[2]This is quite literally the case with relational databases, which is no coincidence—think of table schema as record types and table rows as record terms.

So, dual to the functional case, the `Expr` interface permits the free extension of its cases (`Lit` and `Plus`) while fixing the behaviors (`eval`). The challenge, when approached from this side, is to permit the free extension of behaviors. We consider next such an approach.

```
interface Expr {
  public int eval();
}

class Lit implements Expr {
  int x;

  Lit(int x) {
    this.x = x;
  }

  int eval() {
    return this.x;
  }
}

class Plus implements Expr {
  Expr left , right;

  Plus(Expr left, Expr right) {
    this.left = left;
    this.right = right;
  }

  int eval() {
    return
     left.eval() +
     right.eval();
  }
}
```

(a) Integer literal and addition ASTs in Java (`public` annotations elided for space)

```
class Mult implements Expr {
  Expr left , right;

  Mult(Expr left, Expr right) {
    this.left = left;
    this.right = right;
  }

  int eval() {
    return left.eval() *
           right.eval();
  }
}
```

(b) Multiplication AST in Java

```
interface Expr {
  int eval();
  String print();
}
```

(c) The `Expr` interface of Figure 2A with new method `print()`

Figure 2: The expression problem in Java

2.5 Structural Inheritance

Structural inheritance (or, *structural subtyping*) is one solution to the expression problem's dual.[3] The aim of structural inheritance is to type object inheritance by field sets, not names. Let us illustrate with a simple example.

```
class 2dPoint {
  float x , y;
}
```

```
class 3dPoint {
  float x , y, z;
}
```

It is natural to consider the `3dPoint` class an *extension* of `2dPoint`, as it overlaps at the `x` and `y` fields. There exist two main styles of expressing this relationship in the type system. *Nominal* inheritance expresses the relationship as a strictly declared hierarchy: `3dPoint` inherits from `2dPoint` if we declare it so. This is standard practice in Java.

```
class 3dPoint extends 2dPoint {
   float z;
}
```

Alternatively, a *structural* treatment of inheritance describes when one object inherits from another if and only if the subobject has all of the fields (and, perhaps others) of its parent. So, a `3dPoint` is not a `2dPoint` because we declared so, but automatically so: a `3dPoint` has all the fields of a `2dPoint`. This is the behavior of e.g. objects in OCaml, which are typed by their methods [35]. For example, in the OCaml code below, we expect object `q` to be an inheritor of object `p`.

```
 let p =
     object
       method x = 1
     end;;
let q =
  object
    method x = 1
    method y = 2
  end;;
```

These terms have types

```
p : < x : int >
q : < x : int; y : int >
```

and so we should expect two scenarios to hold: that `q` may be coerced to the type of `p`

[3] Although declared along both OOP and functional axes, it is common to let the "expression problem" refer to only the latter unless otherwise specified.

```
# (q :> < x : int >)
- : < x : int >
```

and, by consequence, a function expecting an input with the type of `p` accepts `q` as well:

```
let get_x r = r#x;;
val get_x : < x : 'a ; .. > -> 'a = <fun>

# get_x q
- : int = 1
```

The types of these terms have included lists of label-type associations, e.g., the type `< x : int; y : int >` denotes an object mapping `x` and `y` to `int`. We call these rows.

3 Row Types — Basics

We have offered some intuition to long-open problems in record and variant extensibility. We used object-oriented programming (via Java) to illustrate records and functional programming (via Haskell) to illustrate variants. In this section, we consider formal systems for record and variant calculi, all of which are purely functional. So, we drop the parlance of OOP—we have records, not objects; functions, not methods; labels, not fields; and terms, not variables.

3.1 Rows à la Wand and Rémy

Wand [51] introduced row typing to formally capture structural inheritance of objects. A *row*, in (most) row type systems, describes an association of labels to types. The precise shape of rows otherwise varies. Wand shapes rows most simply as lists of pairs. We give the syntax of rows in Wand's [51] system below.

$$\text{Rows} \quad \rho \ ::= \ \alpha \mid \varepsilon \mid (\ell \triangleright \tau, \rho)$$

A Row can be either: a *row variable* α; the *empty row* ε, analogous to `nil`; or $(\ell \triangleright \tau, \rho)$, the extension of a row ρ with ℓ labeling the type τ, analogous to `cons`. The following row captures the labels and types of the `3dPoint` record

$$(x \triangleright \texttt{Float}, (y \triangleright \texttt{Float}, (z \triangleright \texttt{Float}, \varepsilon))).$$

But this is quite bloated. Where possible, we instead write

$$(x \triangleright \texttt{Float}, y \triangleright \texttt{Float}, z \triangleright \texttt{Float}).$$

Both record and variant types are constructed from rows, but do dual things with the row information. Rows constructing record types describe the names and output types of record eliminators; rows constructing variant types describe the names and input types of variant constructors. This duality is captured in the type system: $\Pi\rho$ constructs a record from row ρ and $\Sigma\rho$ a variant. We give a type and term syntax for each of these below.

$$\begin{array}{lrcl} \text{Types} & \tau & ::= & \Pi\rho \mid \Sigma\rho \mid ... \\ \text{Terms} & M, N & ::= & \emptyset \mid \{\ell \triangleright M, N\} \mid M.\ell \mid \\ & & & \texttt{inj}\,\ell\,M \mid \texttt{case}\,\ell\,M\,N\,P \mid ... \end{array}$$

The term syntax describes record introduction (called *extension*), record elimination (called *selection*), variant introduction (called *injection*), and variant elimination (called *case distinction*). All record and variant calculi have some mix of these operations. Let's define each now.

Definition 1 (Record extension)**.** Record *extension*, denoted $\{\ell \triangleright M, N\}$, refers to the extension of record N by the singleton row $(\ell \triangleright M)$.

By *extension*, we often (but not always) mean to add only one entry whose label is not present. This distinction is emphasized so as to not confuse extension with *update*.

Definition 2 (Record update)**.** A record is *updated* when we modify the value of a label already present in a record. So, we update $\{\ell \triangleright M, N\}$ to be $\{\ell \triangleright M', N\}$.

So, extension requires the absence of a label and update requires the presence of. The operation of doing the former when the label is present but the latter otherwise is always referred to as *simultaneous* extension and update. Confusingly, this is the actual behavior of extension in Wand's semantics (*cf.* §3.1.3).

Definition 3 (selection)**.** Record *selection*, denoted $M.\ell$, projects the value at ℓ in record M. So, $\{x \triangleright 1\}.x$ should equal 1, and so forth.

Definition 4 (injection)**.** Variant *injection*, denoted $\texttt{inj}\,\ell\,M$, constructs a variant of type $\Sigma(\ell \triangleright M, \rho)$. Note that ρ may be any row—we may always inject a singleton case into a larger variant.

Definition 5 (case distinction)**.** Variant *case distinction*, denoted $\texttt{case}\,\ell\,M\,N\,P$, destructs the variant $N : \Sigma\rho$ by checking if N was constructed with label ℓ; if so, pass

N to $M : \rho(\ell) \to \tau$ and, if not, default to $P : \tau$. The metasyntax $\rho(\ell)$ denotes the type τ such that $(\ell \triangleright \tau)$ occurs somewhere in ρ.

We will note when the syntax or semantics of these operations later differs. The record operations stay pretty consistent between systems; the variant operations vary more greatly. In particular, most systems have their own flavor of variant destruction.

3.1.1 Structural Inheritance With Rows

Structural inheritance in Wand's system can be demonstrated by repeating the OCaml examples above. Actually (by no real coincidence [43]), Wand's calculus is sufficient to type these examples. In the finite-row case (i.e, where the row is an extension of the empty row), we had the OCaml object of type

```
< x : int ; y : int >
```

which may be written in Wand's calculus as the record type

$$\Pi(x \triangleright \texttt{int}, y \triangleright \texttt{int}).$$

In the extensible case (where the row extends a type variable), rewrite the OCaml row type

```
< x : 'a ; .. >
```

to

$$\Pi(x \triangleright \alpha, \rho).$$

In both case, the coercion of records with more fields to records with less permits functions expecting the latter to accept the former.

3.1.2 Rows & The Expression Problem

Early row systems (i.e, all those discussed in §3) omit any substantial treatment of variants. Correspondingly also omitted is any treatment of the expression problem—understandably so, as it had yet to be popularized by Wadler [49] to the degree it is today by (although Reynolds [44] had already described the problem as early as 1975). Nevertheless, we can demonstrate some progress thereof. Recall the data type for arithmetic expressions,

```
data Expr = Val Int | Plus Expr Expr
```

and rewrite in the language of rows as an extensible variant.

$$\texttt{Expr} := \Sigma(\texttt{Val} \triangleright \texttt{Int}, \texttt{Plus} \triangleright \texttt{Expr} \times \texttt{Expr}, \rho)$$

Now, retype and rewrite evaluation

```
eval :: Σ(Val ▷ Int, Plus ▷ Expr × Expr, ρ) → Int
eval v =
  case Val (λ x. x) v
    (case Plus (λ (x, y). x + y) v 0)
```

where the `case` primitive is as described in Definition 5. If we let ρ equal $(\texttt{Mult} \triangleright \texttt{Expr} \times \texttt{Expr})$ then a variant constructed as the `Mult` case is an acceptable input to `eval`. This is the gist of the dual story, with some complexity swept under the rug—namely, `eval` permits the multiplication case but does not define it (the default case, 0, would instead be used). Additionally, the `Expr` type is recursive, which we do not support in any of our presented calculi. The former is remedied by composing `eval` above with a handler for multiplication, i.e.:

```
eval' v = case Mult (...) v (eval v)
```

The latter is, to this author's knowledge, not described formally in any of the row type literature; consequently, nor is a proper stab at the expression problem as it is described by Wadler. But well-founded recursive variants are in fact inductive data, which themselves have been the object of extensive inquiry [12]. So, there may be some merit in this line of future research.

3.1.3 Label Overlaps

Wand [51] interprets rows as partial functions from some set $\mathcal{L}$ of labels to the set of all types generated by the type-level grammar. He interprets row extension then as function composition: Suppose N denotes to a partial function f and M denotes to the value m. Then the interpretation of $\{\ell \triangleright M, N\}$ is the extension of f by the singleton function mapping ℓ to m.

$$\{\ell \mapsto m\} \circ f$$

By consequence of this interpretation, extending a row ρ by a label ℓ which is already present in ρ overwrites the type to which ℓ maps. Hence, we should really refer to Wand's row extension as simultaneous extension and update. For example, the following extension results in a record in which `x` maps to `String` rather than `Float` (as we *extend* on the left).

$$\{x \triangleright \texttt{String}, x \triangleright \texttt{Float}\}$$

There are, of course, other ways to resolve the ambiguity introduced by label overlap. We might just as well have chosen to

- overwrite from the right, rather than left (choosing `Float` over `String`, opposite to Wand [51]);
- report an error (as in Rémy [43]);
- introduce non-deterministic semantics (as considered, but prevented, by [45]);
- or even temporarily *scope* the type `String` over `Float` (as in Leijen [21]).

Rémy [43] asserts that row type systems may be distinguished by their choice of *free* extension versus *strict* extension. In the former, extension of a row by a label already present is well-defined; in the latter, it is not. Rémy's system (discussed next) follows the latter by way of *presence polymorphism.* Here it is not simply a matter of taste—free extension in Wand's system in fact introduced incompleteness to the decidability of type inference [52].

3.1.4 Presence Polymorphism

To address incompleteness in Wand's system, Rémy [43] introduces two *presence flags*, `Pre` and `Abs`, which track (resp.) the presence or absence of labels in rows.

Type Variables	θ
Types	$\tau ::= \ldots$
Presence Types	$\Delta ::= \texttt{Abs} \mid \texttt{Pre}\,\tau \mid \theta$

Presence types indicate whether a label is absent (`Abs`), present with type τ (`Pre`τ), or polymorphic in its presence (as type variable θ). To illustrate, we may retype record selection as

$$\lambda r.r.\ell \quad : \quad \Pi(\ell : \texttt{Pre}\,\alpha, \rho) \to \alpha$$

which states that, to select ℓ from record r, ℓ must be present in the input record's row. To resolve cases of label overlap,we may retype record extension as follows

$$\lambda ar.\{\ell \triangleright a, r\} \quad : \quad \alpha \to \Pi(\ell : \texttt{Abs}, \rho) \to \Pi(\ell : \texttt{Pre}\,\alpha, \rho)$$

which forces ℓ to be absent from the input record. Record inputs which would incur an overwrite of ℓ are now rejected as type errors. We may type record update, in which an overwrite is to occur, as

$$\alpha \to \Pi(\ell : \mathtt{Pre}\,\beta, \rho) \to \Pi(\ell : \mathtt{Pre}\,\alpha, \rho),$$

which forces the input record to have the ℓ label present. We must introduce a new record operation to inhabit this term.

Definition 6 (restriction)**.** Record *restriction*, denoted $r - \ell$, describes the record obtained by removing row $(\ell \triangleright \tau)$ from record r.

Now we may type strict record update as:

$$\lambda\, ar.\{\ell \triangleright a, (r - \ell)\}$$

To decide absence, Rémy presumes some finite enumeration of labels, as one would expect at point of compilation. However, he does not give directly (or a semantics lending itself easily to) an efficient compilation method, which Gaster and Jones [11] sought to address.

3.2 Qualified Row Types

Gaster and Jones [11] frame absence polymorphism in the framework of qualified types [17]. Qualified types generalize Haskell typeclasses [50]. For example, we say that the type of the Haskell term `sum`

```
sum :: Num a => [a] -> Int
```

is *qualified* by the *predicate* `Num a`. The predicate is then (at runtime) interpreted as evidence of its claim [39]—in this case, the predicate `Num a` claims that addition is well-defined for type `a`, the type contained by the input list. In other words, the predicate asserts the existence of the following infix operator

```
(+) :: a -> a -> a
```

for the runtime instantiation of `a`.

The relevant syntax necessary to add qualified types to a type system is given below.

$$\text{Types} \quad \tau ::= \pi \Rightarrow \tau \mid \forall \alpha.\tau \mid \ldots$$

The syntax $\forall\alpha.\tau$ denotes the quantification of type variable α over term τ. This abstraction is routine for Hindley-Milner type systems and was heretofore done (by Wand [51], Rémy [43], and us) implicitly. For continuity, we will continue to quantify row and type variables implicitly when doing so adds no ambiguities. We write $\pi \Rightarrow \tau$ to type a term which has type τ given the assumption of predicate π. This notation is identical to Haskell typeclass qualification. However, base Haskell (as implemented in GHC, its most prominent compiler compiler) restricts the shape of predicates to just the language of typeclasses. Qualified type systems, as originally proposed by Jones [18], generalize qualification to arbitrary predicates. Gaster and Jones define in particular the *lacks* predicate, written $\rho\backslash\ell$, to denote that row ρ lacks label ℓ.

$$\text{Predicates} \quad \pi ::= (\rho\backslash\ell) \mid ...$$

The lacks predicate replaces the presence flags of Rémy. The following types strict record extension

$$(\rho\backslash z) \Rightarrow \Pi\rho \to \alpha \to \Pi(z \triangleright \alpha, \rho)$$

while the following types record update

$$(\rho\backslash z) \Rightarrow \beta \to \Pi(z \triangleright \alpha, \rho) \to \Pi(z \triangleright \beta, \rho).$$

Each of these types are inhabited by more or less equivalent terms to that of Rémy [43] above.

Qualified type systems resolve predicate satisfaction as part of type checking. For example, the following application extends the record $\{x \triangleright 1, y \triangleright 2\}$ with z labeling input n.

$$(\lambda n\, p.\{z \triangleright n, p\})\ 3\ \{x \triangleright 1, y \triangleright 2\}$$

Instantiating appropriately, the application incurs the constraint

$$(x \triangleright \texttt{Float}, y \triangleright \texttt{Float})\backslash z$$

which must be discharged for the term to type. Qualified type systems typically model predicate verification as an *entailment* relation. We write

$$P \Vdash Q$$

to denote that the predicate set P *entails* the predicate set Q. (For convenience, we often write $P \Vdash \pi$ and $\pi \Vdash Q$ in place of (resp.) $P \Vdash \{\pi\}$ and $\{\pi\} \Vdash Q$.) Entailment must be reflexive, transitive, and closed under substitution. Gaster and Jones derives satisfaction of the lacks predicate in the direct and obvious fashion: by row traversal. Particularly, we may derive that

- the empty row ε lacks all labels, and
- the row $(\ell \triangleright \tau, \rho)$ lacks label ℓ' if $\ell \neq \ell'$ and ρ lacks ℓ'.

It is easy to verify using these rules that the predicate $(x \triangleright \texttt{Float}, y \triangleright \texttt{Float}) \backslash z$ is entailed trivially.

Gaster and Jones [11] further offer both (i) a valid runtime semantics of rows and the lacks predicate, and (ii) an efficient compilation scheme to this semantics. We omit any material discussion of the semantics of this (or any) system in this report so as to focus solely on static type systems.

This ends our discussion of what we term *early* row type systems. In the next section, we shift our attention to more modern applications. All of the systems we next discuss either (i) commit to a Rémy style of strict record and row extension or (ii) are parametric over styles of extension. All systems, barring Leijen's [21] *scoped row* system, are also qualified type systems. Consequently, a disclaimer should be given: sections 5 and 6 describe qualified type systems that, by default, implement a Rémy-style of row extension. So, the features highlighted here will recur, but should not be confused with ubiquity in row type systems writ large.

4 Modern Row Types

We now shift our attention to what we term *modern* row type systems. We demarcate *modern* (roughly 2000-2020) from *early* systems (1987-2000) not just temporally, but, perhaps more importantly, in that the each of the former address wider problems than simple record and variant extensibility. So, here we see the application of row type systems fan out to a broader reach. We first consider first-class labels, which Leijen [21] shows to be remarkably expressive in a number of domains.

4.1 First-Class Labels

Labels have thus far not enjoyed first-class status: they may appear in rows (and thus types and predicates), but not terms. Consequently, the behavior of a term may not vary by the value of a particular label. It is easy to miss that record selection, $r.\ell$, and record extension, $\{\ell \triangleright x, r\}$, are in fact *families* of primitive functions parameterized by label. To demonstrate: what is the type of ℓ when ℓ is an explicit parameter to record selection?

$$\lambda r \ell.(r.\ell)$$

Type systems which can productively answer this question are said to have *first-class labels*, meaning labels that may occur as terms in the term syntax. We describe first-class labels in particular according to Leijen's [20] presentation (although first-class labels may in fact be attributed to Gaster and Jones [11] and Sulzmann [47]). We introduce labels first at the kind level.

$$\text{Kinds} \quad \kappa ::= \star \mid \kappa \to \kappa \mid \mathsf{L} \mid ...$$

All terms have types with kind $\star$, including label terms. We include the arrow constructor so that we may type terms such as the label singleton constructor below—we do not otherwise permit a type-level lambda. The syntax $\lfloor \_ \rfloor$ denotes an infix operator such that, for label $\ell : \mathsf{L}$, the application $\lfloor \ell \rfloor$ has kind $\star$ and is therefore inhabitable by a term.

$$\lfloor \_ \rfloor \quad : L \to \star$$

It is now straightforward to type record selection with explicit label input.

$$\lambda r \ell.(r.\ell) \; : \; (\rho \backslash \ell) \Rightarrow \Pi(\ell \triangleright \tau, \rho) \to \lfloor \ell \rfloor \to \tau$$

Expressing the rest of the record (and variant) primitives is routine.

First-class labels prove remarkably expressive, and may type some things that do not often statically type at all. Label-inputted record selection (above) looks more familiar in e.g. Javascript,

```
function (obj, l) {
  return obj[l]
}
```

where the type error that occurs when `obj` lacks `l` is pushed to runtime. Leijen [20] also describes how first-class labels can permit the encoding of intersection types, object function overloading, and type-discriminative functions. First-class labels are crucial in expressing the label-generic combinators of Rω (§6).

4.2 Scoped Rows

Rémy [43] opines that row type systems may be distinguished by their choice of free or strict extension. All systems considered up to now (including Wand's free extension system) have identified labels uniquely with types. We have relatedly assumed commutativity of row labels in rows—i.e., the rows $(M \triangleright \tau, N \triangleright \upsilon)$ and $(N \triangleright \upsilon, M \triangleright \tau)$ are de facto (or, with an appropriately defined equivalence relation, de jure) equivalent. This needn't be the case in other systems with free extension. Leijen [21] describe a free extension system in which labels *scope* one another. We call these *scoped rows*, which are a non-commutative row theory in which labels need not uniquely identify types. By consequence, we resolve the access of duplicate labels according to some order. Consider a record with both an `x` label mapped to `String` and an `x` label mapped to `Float`.

$$r :: \Pi(x \triangleright \texttt{String}, x \triangleright \texttt{Float}, \rho)$$
$$r = \{x = \texttt{"foo"}, x = 2.0\}$$

In a scoped row system, the extension of row $(x \triangleright \texttt{Float}, \rho)$ by row $(x \triangleright \texttt{String})$ is well-defined, with the former out-scoping the latter. That is to say, we let $r.x$ access the left-most (i.e., most recently appended) attribute, yielding

$$r.x = \texttt{"foo"}.$$

Additional x associations can be accessed by record restriction. Restriction in Leijen's scoped system strips the leftmost ℓ from its input r. Consequently, we use the order of label redundancies to guide their access.

$$(r - x).x = 2.0$$

Labels in this system cannot commute with themselves lest we are to lose this ordering—Hence it is a non-commutative theory. However, do note that distinct labels may freely commute, as we are concerned only with the access to duplicate labels.

Leijen remarks that this system is quite straightforward to implement and can "lead to new applications of records in practice." This has become quite so in the typing of algebraic effects [22, 23, 14], which are an approach to typing effectful computation. This is a rich and active area of research; see Pretnar [42] for a proper tutorial, and Leijen [22] for an example application of row types.

The next *modern* row type system is covered in enough depth to afford its own section (as it is work from which stems the author's).

5 ROSE

ROSE is a qualified type system that (like Leijen [20]) extends the system of Gaster and Jones [11]. ROSE is novel in its account of *row concatenation* over *row extension*, the former being difficult to type safely (or at all [53]) and not described by the works above. Under concatenation, ROSE observes that rows form a *partial monoid*. This insight permits the generalization of row systems modulo monoidal theories. In other words, ROSE captures the behavior of all the row theories described above—Wand's, Rémy's, and scoped rows—by letting the predicate system parametrically capture different monoidal theories.

5.1 Records By Concatenation

Existing approaches to typing record concatenation depend on fairly complex language features, such as intersection types [53], disjoint polymorphism [54], and dependent types [6]. In contrast, ROSE is able to type record extension with only predicates and type qualification.

Records in ROSE are constructed by concatenation rather than extension. Correspondingly, rows are constructed by *combination*, the row-level analogue of concatenation. We write

$$\rho_1 \cdot \rho_2 \sim \rho_3$$

to denote that ρ_1 (left-) combined with ρ_2 equals ρ_3. This forms a *partial* monoid with ε, the *empty row*, as identity. To illustrate, we expect all of the following equations to hold.

$$(x \triangleright \texttt{Bool}) \cdot \varepsilon \sim (x \triangleright \texttt{Bool})$$
$$\varepsilon \cdot (x \triangleright \texttt{Bool}) \sim (x \triangleright \texttt{Bool})$$
$$(x \triangleright \texttt{Bool}) \cdot (y \triangleright \texttt{Float}) \sim (x \triangleright \texttt{Bool}, y \triangleright \texttt{Float})$$

However, not all combinations need be defined—we permit the monoidal operation to be *partial*, as we are not (always) quite sure what to do with operations such as these.

$$(x \triangleright \texttt{Bool}) \cdot (x \triangleright \texttt{Bool})$$
$$(x \triangleright \texttt{Bool}) \cdot (x \triangleright \texttt{Float})$$

The syntax of rows and predicates of ROSE is given below. Combination may express row extension: the row $(\ell \triangleright \tau, \rho)$ may equivalently be expressed by the combination $(\ell \triangleright \tau) \cdot \rho \sim z$, using the row variable z wherever one had prior used $(\ell \triangleright \tau, \rho)$. Thus we remove extension from the syntax of rows.

$$\begin{array}{lll} \text{Rows} & \rho & ::= (\ell \triangleright \tau) \\ \text{Predicates} & \pi & ::= \rho_1 \lesssim \rho_2 \mid \rho_1 \cdot \rho_2 \sim \rho_3 \mid ... \end{array}$$

We introduce the *containment* predicate, $\lesssim$, in addition to combination. Intuitively, the predicate $\rho_1 \lesssim \rho_2$ holds when the labels of ρ_1 are a subset of ρ_2. The combination predicate $\rho_1 \cdot \rho_2 \sim \rho_3$ holds when the concatenation of ρ_1 and ρ_2 is ρ_3. ROSE is parametric over the static and dynamic meaning of predicates, meaning we cannot say that the descriptions given describe each predicate in full. However, for purposes of illustration, we typically assume that predicates such as

$$(x \triangleright \texttt{Int}) \lesssim (x \triangleright \texttt{Int}, y \triangleright \texttt{Int})$$
$$(x \triangleright \texttt{Int}) \cdot (y \triangleright \texttt{Int}) \sim (x \triangleright \texttt{Int}, y \triangleright \texttt{Int})$$

hold, but predicates such as

$$(x \triangleright \texttt{Int}) \lesssim (y \triangleright \texttt{Int}, z \triangleright \texttt{Float})$$
$$(x \triangleright \texttt{Int}) \cdot (y \triangleright \texttt{Int}) \sim (z \triangleright \texttt{Int})$$

do not. Observe finally some syntactic sugar in these examples—although we have removed the row extension syntax $(\ell \triangleright \tau, \rho)$, we still used it to write e.g. $(y \triangleright \texttt{Int}, z \triangleright \texttt{Float})$ in the predicate above. As described earlier, this may equivalent be expressed as the conjunction of the predicates

$$(y \triangleright \texttt{Int}) \cdot (z \triangleright \texttt{Float}) \sim \rho, (x \triangleright \texttt{Int}) \lesssim \rho$$

but the former is much easier to parse.

5.1.1 Typing primitives

Describing rows by concatenation changes the types and behaviors of the other primitives. We introduce ROSE more formally by way of these changes. Excerpted typing rules are given.

Record concatenation. The combination predicate in ROSE is in fact informed by the typing of concatenation; that is, we include combination as a predicate firstly so that we may concatenate records. Concatenation is typed as follows.

$$\frac{\Gamma \vdash M : \Pi\rho_1 \quad \Gamma \vdash N : \Pi\rho_2 \quad \Gamma \Vdash \rho_1 \cdot \rho_2 \sim \rho_3}{M +\!\!+ N : \Pi\rho_3}$$

This rule states that if M has type $\Pi\rho_1$, N has type $\Pi\rho_2$, and the combination of ρ_1 and ρ_2 is ρ_3, then $M +\!\!+ N$ has type $\Pi\rho_3$. In other words, the concatenation of two records is given by the row formed by the concatenation of their rows. The syntax $\Gamma \Vdash P$ denotes the entailment of P by typing environment Γ; we let Γ include both predicate and typing assumptions, for convenience.

Labeled types. Rows in ROSE are either singletons or the combination of singletons. Due to the strength of its predicate system, ROSE is able to speak of rows only by way of singletons and the empty row. Observe that, without singletons, the rule

for record introduction is in fact ill-founded, as we presume terms M and N already of record type. Singletons can be introduced at both record and variant type.

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash (\ell \triangleright M) : \Pi(\ell \triangleright \tau)} \qquad \frac{\Gamma \vdash M : \tau}{\Gamma \vdash (\ell \triangleright M) : \Sigma(\ell \triangleright \tau)}$$

Of course, a singleton variant and a singleton record are isomorphic, leading us to define also the *labeled type*:

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash (\ell \triangleright M) : (\ell \triangleright \tau)}$$

These three types may be used interchangeably in ROSE.

Record projection. Just as concatenation extends record extension to the addition of *zero or more* fields, so should the inverse of concatenation extend record restriction to the removal of zero or more fields. We call this *projection*, denoted $\mathtt{prj}\, M$.

$$\frac{\Gamma \vdash M : \Pi\rho_2 \quad \Gamma \Vdash \rho_2 \lesssim \rho_1}{\Gamma \vdash \mathtt{prj}\, M : \Pi\rho_1}$$

The rule states that we may project $M : \Pi\rho_2$ to type $\Pi\rho_1$ where ρ_1 may have less labels than ρ_2.

Record restriction. Restriction, as it is defined in other systems, can be expressed immediately as a singleton case of projection. Note that, in a non-commutative theory, one would expect *two* definitions of restriction—one for left restriction and one for right. We give just the former.

$$\lambda r.\mathtt{prj}\, r \;:\; (\ell \triangleright \tau) \cdot \rho_1 \sim \rho_2 \Rightarrow \Pi\rho_2 \to \Pi\rho_1$$

Record selection. Record selection asks for the value at label ℓ in record r. It is simplest to define selection over the labeled type.

$$\frac{\Gamma \vdash M : (\ell \triangleright \tau)}{\Gamma \vdash M.\ell : \tau}$$

Selecting ℓ from record r can be done by first projecting r to the labeled term $(\ell \triangleright M)$, given that $(\ell \triangleright M) : (\ell \triangleright \tau)$ in ρ.

$$\lambda r.(\mathtt{prj}\, r).\ell \;:\; (\ell \triangleright \tau) \lesssim \rho \Rightarrow \Pi\rho \to \tau$$

Simultaneous record update & extension. Most systems with strict extension are forced to distinguish between record update and extension. ROSE is unique in that it may express the two simultaneously—as done by Wand [51], but without the resulting conflicts with decidability.

$$\lambda rt.\{\ell \triangleright t\} +\!\!+ (\mathtt{prj} r) \;::\; \rho_2 \lesssim \rho_1 \Rightarrow \Pi\rho_1 \to \tau \to \Pi((\ell \triangleright \tau) \cdot \rho_2)$$

Consider each case. When $r : \Pi\rho_1$ lacks label ℓ, then we may let ρ_2 equal ρ_1, as the addition of ℓ to ρ_1 is well-defined. In the case where ℓ is present, we project r to the record without that label (that is, with type $\Pi\rho_2$) so as to force the combination $(\ell \triangleright \tau) \cdot \rho_2$ to be well-defined.

Variant injection. Variants are introduced by injection of smaller variants rather than by cases.

$$\frac{\Gamma \vdash M : \Sigma\rho_1 \quad \Gamma \Vdash \rho_1 \lesssim \rho_2}{\mathtt{inj}\, M \;: \Sigma\rho_2}$$

Variant introduction is not particularly noteworthy, as we need only select one case, and so $M : \Sigma\rho_1$ may always be a singleton variant. In other words, injection in this fashion is not too dissimilar from injection as it has previously been described.

Variant branching. More interesting is variant destruction, which is dual to record introduction. Recall that we introduce a record of type $\Pi(\rho_1 \cdot \rho_2)$ by concatenating a record of type $\Pi\rho_1$ with a record of type $\Pi\rho_2$. Dually, suppose we have handlers $M : \Sigma\rho_1 \to \tau$ and $N : \Sigma\rho_2 \to \tau$. Then one would expect to be able to compose the two into a handler of $\rho_1 \cdot \rho_2$. This is exactly the elimination rule for variants.

$$\frac{\Gamma \vdash M : \Sigma\rho_1 \to \tau \quad \Gamma \vdash N : \Sigma\rho_2 \to \tau \quad \Gamma \Vdash \rho_1 \cdot \rho_2 \sim \rho_3}{\Gamma \vdash M \triangledown N : \Sigma\rho_3 \to \tau}$$

We call $\triangledown$ a *branching* operator.

5.1.2 Typing Wand's Problem

Record concatenation has proven challenging for good reason. Wand [53] first considered record concatenation to model multi-object inheritance, but was only able to show principal type inference modulo *sets* of most general types.[4] Wand offers this term to demonstrate the challenge.

$$\lambda\, m\, n.(m +\!\!+ n).\ell$$

This term concatenates records m and n and projects out the value at label ℓ. This corresponds to the multi-inheritance problem of object-oriented languages, i.e., when one object inherits from two parents whose fields may have non-trivial intersection. This is a tricky problem that leads Java to restrict multi-inheritance strictly to *interfaces* and not classes [37].

What makes this term difficult to type? We want to assert that ℓ is present in at least one of m or n for this term to be well-defined. However, specifying precisely which one over-specifies the behavior of the function. Rose types this term by using the containment predicate to stipulate that (i) the concatenation is well-defined and (ii) the label ℓ occurs in the result of the concatenation.

$$(\rho_1 \cdot \rho_2 \sim \rho_3, (\ell \triangleright \tau) \lesssim \rho_3) \Rightarrow \Pi\, \rho_1 \to \Pi\, \rho_2 \to \tau$$

The predicates $\rho_1 \cdot \rho_2 \sim \rho_3$ and $(\ell \triangleright \tau) \lesssim \rho_3$ qualify the type with the assumptions that (i) and (ii) hold, respectively.

5.2 Generalization Over Row Theories

ROSE is parametric over row theories. We have used the term *row theory* intuitively to describe how a row type theory resolves label overlaps. We have seen a handful of different row theories under this informal definition, e.g.

- Wand's [51] free extension system, in which label overlaps are overwritten;
- Rémy's [43] strict extension system, in which label overlaps incur a type error; and
- Leijen's [21] free extension system, in which label overlaps are scoped.

[4] Wand's approach in [53] would more accurately be described as the use of intersection types by today's parlance.

A formal definition should tell us what rows are in a particular row theory, for which rows is combination well defined, and what those combinations equal. We formalize row theories according to this specification.

5.2.1 Row Theories, Algebras, and Models

Definition 7 (Row theory)**.** A *row theory* is a 3-tuple $\langle R, \sim, \Vdash \rangle$ such that

- R is a set of syntactic rows (that is, of well-formed ground row type expressions) including at least the empty row;
- The relation $\sim$ is an equivalence relation on R, identifying syntactically distinct rows;
- $\Vdash$ is an entailment relation on the row predicates $\rho_1 \lesssim_d \rho_2$ and $\rho_1 \cdot \rho_2 \sim \rho_3$, invariant with respect to $\sim$, satisfying reflexivity, transitivity and at minimum the following rules

$$\frac{\Gamma \Vdash \rho_1 \cdot \rho_2 \sim \rho_3}{\Gamma \Vdash \rho_1 \lesssim_L \rho_3} \qquad \frac{\Gamma \Vdash \rho_1 \cdot \rho_2 \sim \rho_3}{\Gamma \Vdash \rho_2 \lesssim_R \rho_3}$$

  which relate combination to (left and right) containment.

The first two components describe what rows are in a row type system; the third component must say which equations of the form $\rho_1 \cdot \rho_2 \sim \rho_3$ can be entailed. In other words, the entailment relation must tell us which combinations are defined and what they equal. Note that the predicate for containment is now directed by $d \in \{L, R\}$ to permit non-commutative theories. The rules above specify that $\rho_1 \cdot \rho_2 \sim \rho_3$ entails that $\rho_1 \lesssim_L \lesssim \rho_3$ and $\rho_2 \lesssim_R \rho_3$; these are necessary to give a monoidal denotation of rows.

All of the type theories we have discussed can be abstracted to row theories. We give the most common example: a commutative theory of concrete rows in which the concatenation of overlapping labels is ill-defined. This abstracts the behavior of e.g. Rémy's [43] system with strict extension. We call this example the *simple row theory.*

Example 8 (Simple row theory)**.** The simple row theory is defined as $\langle R_{\texttt{simp}}, \sim_{\texttt{simp}}, \Vdash_{\texttt{simp}} \rangle$, where

- $R_{\texttt{simp}}$ is the set of uniquely-labeled sequences of types; or, the least set generated by the row grammar

$$R \quad ::= \ (\ell \triangleright \tau, R) \mid \varepsilon$$

- $\sim_{\texttt{simp}}$ identifies rows up to permutation; and

- $\Vdash_{\texttt{simp}}$ is defined by the following inference rules:

$$\frac{\{\ell_1 \triangleright \tau_1, ..., \ell_m \triangleright \tau_m\} \subseteq \{\ell'_1 \triangleright \tau'_1, ..., \ell'_n \triangleright \tau'_n\}}{\Gamma \Vdash_{\texttt{simp}} (\ell_1 \triangleright \tau_1, ..., \ell_m \triangleright \tau_m) \lesssim (\ell'_1 \triangleright \tau'_1, ..., \ell'_n \triangleright \tau'_n)}$$

$$\frac{\{\ell_1 \triangleright \tau_1, ..., \ell_k \triangleright \tau'_k\} \uplus \{\ell_{k+1} \triangleright \tau_{k+1}, ..., \ell_m \triangleright \tau_m\} = \{\ell'_1 \triangleright \tau'_1, ..., \ell'_m \triangleright \tau'_n\}}{\Gamma \Vdash_{\texttt{simpl}} (\ell_1 \triangleright \tau_1, ..., \ell_k \triangleright \tau'_k) \cdot (\ell_{k+1} \triangleright \tau_{k+1}, ..., \ell_m \triangleright \tau_m) \sim (\ell'_1 \triangleright \tau'_1, ..., \ell'_m \triangleright \tau'_m)}$$

  which state that (i) ρ_1 is contained in ρ_2 if each of the components in ρ_1 are in ρ_2, and (ii) $\rho_1 \cdot \rho_2$ is equivalent to ρ_3 if the disjoint sum of ρ_1 and ρ_2 is equal to ρ_3.

Row theories are *modeled* by partial monoids, which we call *row algebras*. This denotation validates the claim that row theories are monoidal in structure.

Definition 9 (row algebra)**.** A *row algebra* is any partial monoid $\langle M, \cdot, \varepsilon \rangle$; that is, $\cdot$ is a partial binary operator $M \times M \rightharpoonup M$ such that $m \cdot \varepsilon = m = \varepsilon \cdot m$, and $m_1 \cdot (m_2 \cdot m_3) = (m_1 \cdot m_2) \cdot m_3$ whenever $m_1 \cdot (m_2 \cdot m_3)$ is well-defined.

Definition 10 (row model)**.** Let $f : R \to M$; we write

- $f \Vdash \rho_1 \cdot \rho_2 \sim \rho_3$ if $f(\rho_1) \cdot f(\rho_2) = f(\rho_3)$;
- $f \Vdash \rho_1 \lesssim_L \rho_3$ if there exists $\rho_2 \in R$ such that $f(\rho_1) \cdot f(\rho_2) = f(\rho_3)$;
- $f \Vdash \rho_2 \lesssim_R \rho_3$ if there exists $\rho_1 \in R$ such that $f(\rho_1) \cdot f(\rho_2) = f(\rho_3)$; and
- $f \Vdash P$ if $f \Vdash \pi$ for all predicates π in P.

We say f is a *model* of row theory $\langle R, \sim, \Vdash \rangle$ in row algebra $\langle M, \cdot, \varepsilon \rangle$ if

- for all rows $\rho_1, \rho_2 \in R$, if $\rho_1 \sim \rho_2$ then $f(\rho_1) = f(\rho_2)$;
- If $\Gamma \Vdash \pi$, then for each substitution θ well-defined on the free variables of Γ and π, if $f \Vdash \theta\Gamma$ then $f \Vdash \theta\pi$; and
- there is some $\rho_0 \in R$ such that $f(\rho_0) = \varepsilon$.

We can interpret the simple row theory in the expected manner as partial label-to-type mappings.

Example 11 (simple row algebra)**.** Let $\mathcal{L}$ denote a set of labels, $\mathcal{T}$ denote a set of syntactically valid types, and consider partial functions $f, g : \mathcal{L} \to \mathcal{T}$. We define a partial union $f \sqcup g$ by

$$(f \sqcup g)(\ell) = \{ \begin{array}{ll} f(\ell) & \text{if} \ell \in \text{dom}(f) \\ g(\ell) & \text{if} \ell \in \text{dom}(g) \end{array}$$

if $\mathrm{dom}(f)$ and $\mathrm{dom}(g)$ are disjoint and undefined otherwise. $\langle \mathcal{L} \rightharpoonup \mathcal{T}, \sqcup, \emptyset \rangle$ gives a row algebra for the simple row theory by the following model.

$$f(\rho) = \{\ell \mapsto \tau\} \quad \text{for } (\ell \triangleright \tau) \in \rho$$

The simple row algebra is also a model for various row calculi with strict extension, e.g. Gaster and Jones [11], Chlipala [6] and Harper and Pierce [13].

5.2.2 Rows, By Any Other Name

The argument that ROSE in fact generalizes modulo row theories is evident in the parametricity of its syntax and typing. For row theory $\langle R, \sim, \Vdash \rangle$, we let rows be determined according to membership in R and record and variant types be syntactically valid only for $\zeta \in R$:

$$\begin{array}{ll} \text{Rows} & \zeta \in R \\ \text{Types} & \tau ::= t \mid \tau \to \tau \mid \Pi\zeta \mid \Sigma\zeta \mid \ell \triangleright \tau \end{array}$$

The instantiation of $\Vdash$ is done directly within typing. ROSE is only parameterized by row theories, meaning just its static semantics. We do not in turn parameterize the semantics of each by row algebra. Algebras serve more to (i) confirm our intuition that row theories are monoidal and (ii) show that syntactically distinct row theories may be modeled the same. Morris and McKinna [34] also show that some row algebras may be homomorphic with one another—in particular, we may inject from less expressive row algebras (e.g. the simple row theory) to more expressive (e.g. the scoped row theory).

6 Rω

Rω extends ROSE with support for generic programming over rows via first-class labels and higher-order row quantification. The extension of ROSE to Rω is analogous to the extension of System F to System $F\omega$ (hence the name). Rω is by far the most expressive of row theories presented; this section will motivate the novelties of Rω before formally introducing the calculus.

6.1 Generic Programming With Extensible Data Types

We should clarify that by *generic programming* we mean datatype-generic, viewing records and variants as the fundamental organizations of data. Let us demonstrate with an example; consider the `Eq` typeclass of Haskell.

```
class Eq where
  (==) :: a -> a -> Bool
```

If a variant $\Sigma\rho$ satisfies the predicate `Eq` $\Sigma\rho$, we should expect an instantiation of `(==)` with type $\Sigma\rho \to \Sigma\rho \to \texttt{Bool}$. Now, suppose we have a record of instantiations for each $(\ell \triangleright \tau)$ in $\Sigma\rho$. Can we produce an instantiation for $\Sigma\rho$? Such a function should have the following type

$$\Pi(\texttt{eq}\,\rho) \to \texttt{eq}\,(\Sigma\rho)$$

where $\texttt{eq} = \lambda x.x \to x \to \texttt{Bool}$ is a type operator and $\texttt{eq}\,\rho$ lifts the type application pointwise to rows. To illustrate, suppose ρ consists of boolean cases $\texttt{T} : \top$ and $\texttt{F} : \top$. Then $\Pi(eq\,\rho)$ is equal to

$$\Pi(\texttt{T} : \top \to \top \to \texttt{Bool}, \texttt{F} : \top \to \top \to \texttt{Bool}).$$

(Let $() : \top$ denote the unit term and type, respectively.) The dual, $\Sigma\rho$, represents more familiarly the boolean type.[5] If this is the case—that is, if the record consists of a finite quantity of known handlers—then the corresponding variant handler may be defined in ROSE as follows.

$$\begin{aligned}\lambda rvw.(\lambda x.(\lambda y.(r.T)\,x\,y)\triangledown(\lambda\_.\texttt{False}))\triangledown\\ (\lambda x.(\lambda y.(r.F)\,x\,y)\triangledown(\lambda\_.\texttt{False}))\end{aligned}$$

The term above uses the record of handlers d to discriminate on variants v and w. In the case that both variants are of the same case, we simply apply the case's handler; when the cases differ, we return `False`. This term in fact could apply to smaller variants, i.e, those with only one case; the type $\Sigma(T \triangleright \top, F \triangleright \top)$ denotes the greater bound of each. We can express this as the type

$$\forall z.z \lesssim (T \triangleright \top) \cdot (F \triangleright \top) \Rightarrow (\texttt{eq}\,z) \to \texttt{eq}(\Sigma z).$$

[5] For simplicity, assume the type `Bool` with constructors `True` and `False` to be primitive and distinct from the variant in discussion.

where z is constrained in the containment predicate to just those rows which inject into $(T \triangleright \top) \cdot (F \triangleright \top)$. Hubers and Morris [15] ask: can this term be expressed generically—that is, over an unconstrainted z?

$$\forall z.\Pi(\mathsf{eq}\, z) \to \mathsf{eq}(\Sigma z)$$

The authors answer in the affirmative with two novel (and dual) primitives, syn and ana, which lift label-generic operations over records and variants.

6.2 Witnessing the Duality of Records and Variants

The primitive syn is used to generically *syn*thesize records; the primitive ana is used to generically *ana*lyze variants. This section introduces the primitives iteratively, starting with simpler examples than that just described; rather than a record of boolean comparators, we will consider a record of unary operators. Suppose we have such a record, with type

$$\Pi(\rho \to \tau)$$

where $\rho \to \tau$ denotes the arrow constructor lifted over row kind. For example, if $\rho = (T : \top, F : \top)$ then $\rho \to \tau$ denotes

$$(T : \top \to \tau, F : \top \to \tau)$$

The task is to transform this record of handlers into a variant eliminator, i.e., to inhabit the type below.

$$\mathtt{reflect} : \Pi(\rho \to \tau) \to \Sigma\rho \to \tau.$$

Intuitively, for inputs $d : \Pi(\rho \to \tau)$ and $v : \Sigma\rho$, we would like to perform a case distinction on the input v and apply the correspondingly labeled handler in d—that is, we seek a sort of uniform case distinction on all cases in v. This can be performed by the ana combinator. We give our first typing of ana below as the rule (T-ana$_1$).

$$(\text{T-ana}_1)\ \frac{\Gamma \vdash \rho : \mathsf{R}^\star \quad \Gamma \vdash M : \forall l : \mathsf{L}, u : \star.(l \triangleright u) \lesssim \rho \Rightarrow \lfloor l \rfloor \to u \to \tau}{\Gamma \vdash \mathsf{ana}\, M : \Sigma\rho \to \tau}$$

The judgment $\Gamma \vdash \rho : \mathsf{R}^\star$ simply states that ρ is a row variable inhabited by types with $\star$ kinds; we will describe Rω's higher order row quantification in the next subsection. More interesting is the body M; Let us dissect this more closely. Rω commits to first class labels with kind L, so the first input $l : \mathsf{L}$ is a universally quantified label variable. Labels may be converted to star kind identically to as described in §4.1. The predicate $(l \triangleright u) \lesssim \rho$ says that the ρ in question contains the case ℓ labeling type $u : \star$. This label and type are first-class arguments to M, which is then tasked with proving a τ value. In short, the body of $\mathsf{ana}\, M$ produces (uniformly) a τ from each component in ρ. ana is used below to inhabit record reflection

$$\lambda\, d\, w.\mathsf{ana}\,(\lambda\, l\, u.\mathtt{sel}\; d\; l\; u)\; w \;:\; \Pi(\rho \to \tau) \to \Sigma\rho \to \tau.$$

where the helper sel is defined as $\lambda\, rl.(\mathtt{prj}\; r).l$, exactly as we did in §5.1.1 except with the label input $l : \mathsf{L}$ now first-class. The body of ana thus selects the handler at label l and applies it to the row contents u to get a value at type τ. This combinator is generic in that we destruct the variant v generically, i.e., without respect to its particular labels.

The reflect function naturally has a dual, which we call reify.

$$\begin{aligned}&\mathtt{reify} : (\Sigma\rho \to \tau) \to \Pi(\rho \to \tau)\\ &\mathtt{reify} = \lambda f.\mathsf{syn}(\lambda\, lx.\mathtt{con}\; l\; x)\end{aligned}$$

The helper con constructs a variant at label l and value x; it may be defined as $\lambda\, lx.\mathsf{inj}\,(l \triangleright x)$, identically defined as in §5.1.1. The dual reify abstracts the case handlers from a variant eliminator f. The primitive syn is a dual to ana: ana generically destructs a variant, and so syn generically constructs a record. Our first typing of it is given below as (T-syn_1).

$$(\text{T-syn}_1)\;\frac{\Gamma \vdash \rho : \mathsf{R}^\star \quad \Gamma \vdash M : \forall l : \mathsf{L}, u : \star.(l \triangleright u) \lesssim \rho \Rightarrow \lfloor l \rfloor \to u}{\Gamma \vdash \mathsf{syn}\; M : \Pi\rho}$$

The body of syn differs slightly from that of ana, as the latter destructs and the former constructs. Correspondingly, the body of syn is given a label and expected to construct a term at type u. We can thus think of syn as the production of a record from a uniform uniform treatment of its components.

These definitions capture the intention of each combinator. In practice, however, we found that typing more interesting terms required some additional complexity. We describe changes thereof in the following two subsections.

6.3 Lifting Functorality

Let us thus consider typing a `map` function. In particular, consider a type-preserving map over the components of ρ in a record of type $\Pi\rho$:

$$\text{map}_\Pi : \forall \rho : \mathsf{R}^\star.(\forall l : \mathsf{L}, u : \star.\, (l \triangleright u) \lesssim \rho \Rightarrow \lfloor l \rfloor \to u \to u) \to \Pi\rho \to \Pi\rho$$

The type of the input function, $(\forall l : \mathsf{L}, u : \star.\, (l \triangleright u) \lesssim z \Rightarrow \lfloor l \rfloor \to u \to u)$ is simply the type of the body of $\mathsf{ana}\, M$ when the component type, u, is preserved. As expected, we populate the type via the syn combinator.

$$\text{map}_\Pi = \lambda f\, r.\, \mathsf{syn}\, (\lambda l.\, f\, l\, (\mathtt{sel}\, r\, l))$$

Intuitively, this term applies the transformation f at each component of r to build a new record of the same type. Critically, f is type-preserving; when it is type-transforming, it becomes unclear what type this term should return.

$$\text{map}_\Pi : \forall \rho : \mathsf{R}^\star.(\forall l : \mathsf{L}, u : \star.\, (l \triangleright u) \lesssim \rho \Rightarrow \lfloor l \rfloor \to u \to \tau) \to \Pi\rho \to \Pi?$$

Try to populate the question mark with a valid type. The input $r : \Pi\rho$ has now had the types of its components each changed; thus the row itself ρ is likewise transformed. So, intuitively, we would like to express precisely this—a type-level transformation of ρ. In other words, we must introduce a higher order transformation component (ϕ below) to transform the contents of ρ. A new typing rule for syn is given below. Changes are highlighted.

$$(\text{T-syn}_2)\ \frac{\Gamma \vdash \rho : \mathsf{R}^{\boldsymbol{\kappa}} \quad \Gamma \vdash \phi : \kappa \to \star \quad \Gamma \vdash M : \forall l : \mathsf{L}, u : \boldsymbol{\kappa}.\, (l \triangleright u) \lesssim \rho \Rightarrow \lfloor l \rfloor \to \phi\, u}{\Gamma \vdash \mathsf{syn}_{\phi}\, M : \Pi(\phi\, \rho)}$$

Firstly, observe the higher order row quantification of ρ. The syntax $\rho : \mathsf{R}^\star$ denotes that ρ is a row of types (and type constructors) with kind κ. This is a novel feature of Rω. Further, and as described at the start of this section, type constructors may be lifted component-wise over rows, and, dually, rows of type constructors may be given types as arguments. We make use of both of these capabilities in typing syn. The type constructor $\phi : \kappa \to \star$ accepts an input of kind κ and returns a type at kind $\star$. This means that ϕ may (i) transform, at the type level, the row ρ into a row of types, and (ii) transform *each component* of ρ into types. This is expressed by the body of syn, which is expected to create a term at type $\phi\, u$ for each $u : \kappa$ in ρ.

To illustrate the desired type-transforming map behavior, we may simplify things slightly. Suppose that ρ is a row of types at kind $\star$ and that $f : \star \to \star$ is a unary type constructor. This is the case, for example, of a transformation which might take an input of integers and map to an output of lists of integers.

$$\mathrm{map}_\Pi : \forall \rho : \mathsf{R}^\star, f : \star \to \star . (l \triangleright u) \lesssim \rho \Rightarrow (\lfloor l \rfloor \to u \to f\, u) \to \Pi(\rho) \to \Pi(f\, \rho)$$

Changes to the original type are shaded. The term definition is omitted—it is in fact identical to the type-preserving case, disregarding some housekeeping to do with types and type constructor inputs. The shaded change simply implements what is described above—we transform the type of both ρ and its components.

6.4 Comparing Records and Variants

We return to our original example—variant comparison—to illustrate the final change to the combinators. Recall that, for input record $\Pi\rho$, we wish to compare the variants v and w of type $\Sigma\rho$. That is, we wish to inhabit the following type with something like the term below. Note that we do not perform a type transformation and therefore let $\phi : \kappa \to \star$ be the identity function. Hence ana looks and behaves as it did in §6.2.

$$\begin{aligned} &\mathrm{eq}_\Sigma : \forall z : \mathsf{R}^\star . \Pi(\mathtt{eq}\, z) \to \Sigma z \to \Sigma z \to \mathtt{Bool} \\ &\mathrm{eq}_\Sigma = \lambda\, d\, v\, w . \mathsf{ana}(\lambda\, l y . ?)\, w \end{aligned}$$

To populate the term, we must consider two cases: if v and w are constructed with the same label, we compare them with the comparator in d; if not, we return

`False` (we cannot compare terms of different type). This is the same logic that we stipulated at the start of the section, and can be implemented fairly straightforwardly. Firstly, we know we would like to branch on these cases via $\triangledown$, the branching operator.

$$\begin{aligned}&\mathrm{eq}_\Sigma : \forall z.\Pi(\mathsf{eq}\, z) \to \Sigma z \to \Sigma z \to \mathit{Bool}\\&\mathrm{eq}_\Sigma = \boldsymbol{\lambda}\, d\, v\, w.\mathsf{ana}(\boldsymbol{\lambda}\, ly.(\mathtt{case}\, l\, (\boldsymbol{\lambda} x.\mathrm{sel}\, d\, l\, x\, y)) \triangledown (\boldsymbol{\lambda} x.\mathtt{False}))\, v)\, w\end{aligned}$$

Each side of the branching operator is shaded for visual aid. The left side handles the case when the types are the same; the right side handles the case where they differ. Let us dissect the left side. The `case` operator is much the same as described in Definition 5 but destructs only singleton variants. Like `sel`, It is easily derivable in Rω.

$$\begin{aligned}&\mathrm{case} : \forall l : \mathsf{L}, t : \star, u : \star.\, \lfloor l \rfloor \to (t \to u) \to \Sigma(l \triangleright t) \to u\\&\mathrm{case} = \boldsymbol{\lambda}\, l\, f\, x.f\, (x.l)\end{aligned}$$

`case` $l\, f$ constructs a handler (whose behavior is given by f) for the variant constructed with label l. So, within eq_Σ, the subterm $(\mathtt{case}\, l\, (\boldsymbol{\lambda} x.\mathrm{sel}\, d\, l\, x\, y))$ describes a handler for the given label l that then selects the l-labeled handler in d and uses it to compare x and y, which are of the same type. The right side, $(\boldsymbol{\lambda} x.\mathtt{False}))$, is the constant function returning `False`. This behavior is as specified. This term seems reasonable but, unfortunately, does not type check.

The difficulty is in type checking the branching operator. Recall that, for $M : \Pi\rho_1 \to \tau$ and $N : \Pi\rho_2 \to \tau$, the branching operator $M \triangledown N$ is well typed at type $\Sigma\rho_3 \to \tau$ only if we can derive that $\rho_1 \cdot \rho_2 \sim \rho_3$ (refer to §5.1.1 for full rule). In effect, this is to say that we must know the totality of cases. However, in typing eq_Σ, we do not have this totality; within the body of `ana`, we have only the assumption that $(\lesssim \ell \triangleright u)\rho$. In short: we need evidence of combination, not containment. Hence we strengthen[6] the type of the bodies of `ana` and `syn`. We give the new rule for `ana` below (changes highlighted).

$$(\text{T-ana}_3)\ \frac{\Gamma \vdash \rho : \mathsf{R}^\kappa \quad \Gamma \vdash \phi : \kappa \to \star \quad \Gamma \vdash M : \forall l : \mathsf{L}, u : \kappa, y : \mathsf{R}^\kappa.\, (l \triangleright u) \odot y \sim \rho \Rightarrow \lfloor l \rfloor \to \phi\, u \to \tau}{\Gamma \vdash \mathsf{ana}_\phi\, M : \Sigma(\phi\, \rho) \to \tau}$$

[6] *Strength* here refers, as usual, to implication—and indeed, combination implies containment in both ROSE and Rω.

We have strengthened the body of ana by replacing (shaded) the weaker containment predicate, $(l \triangleright u) \lesssim \rho$, with the stronger combination predicate, $(l \triangleright u) \odot y \sim \rho$. The term eq_Σ has the evidence it needs now to type. Along nearly dual lines, we can similarly construct a record comparator.

$$\forall z : \mathsf{R}^\star.\Pi(\mathtt{eq}\, z) \to \Pi z \to \Pi z \to \mathtt{Bool}$$

But to inhabit this term we need one additional primitive for label-generic folding. Describing this primitive does not convey much new, and is thus omitted; see §3.4 of the paper [15] for a full definition.

This section has demonstrated label-generic programming in Rω; we next discuss some additional features that have been omitted.

6.5 Other Novelties of Rω

We have described two novelties of Rω—that is, higher-ordered rows and the combinators described above. To conclude this section, we summarize some omitted contributions of the work cited.

- Like ROSE, Rω is parametric over row theories. This is done slightly differently—we require a more involved definition of *row theory*—but is implemented similarly by parameterizing the syntax and typing rules of Rω. Hubers and Morris [15] give example implementations of the simple and scoped row theories, among others.
- Rω, like System Fω, is an impredicative system. The authors give a universe stratification of Rω so that it may denote into a predicative system.
- The authors denote Rω into the *index calculus*, which is effectively a DSL interpreted in Agda. This is a shallow embedding. Importantly, and by consequence of the totality of Agda, we show Rω to be type safe—that is, if a term statically types at type τ then its denotation will inhabit the denotation of τ. As the semantics is denotational, not operational, we defer to a definition of type safety as defined by Milner [31] rather than the typical progress and preservation lemmas that accompany many operational semantics.

7 Conclusion

7.1 Other row type systems

We have described the evolution of row type systems from Wand [51] and Rémy [43] to Hubers and Morris [15]. Along the way, we saw a handful of other row type theories and summarized their (lasting) contributions. We, of course, have omitted some other styles and interesting features. Most notable is perhaps Chlipala's [6] web programming language, Ur, which is (like Rω) based on System Fω, and supports type-level operations over rows and row and record concatenation with first-class labels. However, it does not support extensible variants. Other novel applications of rows include: Makholm and Wells's [28] system for first-class mixins; Hillerström and Lindley's system for extensible effects [14]; Lindley and Cheney's [25] type system for effect polymorphism; and Lindley and Morris's [26] type system for extensible session types. There are also some omitted novel language features—notably, there is a fairly rich body of literature on *dependent records* [27, 41], in which the types of record fields are successively dependent upon the terms of fields defined before them.

7.2 Other approaches

The expression problem, with respect to datatype extensiblity, is by now quite saturated with solutions [56]—row types not even chief among them. The dual direction, structural typing of objects, is even more well-trodden ground. We conclude with a summary of alternative proposed solutions to these problems.

Subtyping. Subtyping is the most widely adopted techniques for building type systems with records and variants [3, 16, 46, 36, 58, 38]. Mitchell [32] first introduced a subtyping relation as denoting implicit coercions from one type to another: given types A and B, the subtyping relation $A \leq B$ states that we may coerce from type A to B. It is natural to consider then coercions from records with more fields to less fields (projection) and from variants with less fields to more fields (injection). Observe that this relationship is monotone for variants and antitone for records. For variants v_i and records r_i,

$$v_1 \leq v_2 := \mathtt{dom}(v_1) \subseteq \mathtt{dom}(v_2)$$
$$r_1 \leq r_2 := \mathtt{dom}(r_2) \subseteq \mathtt{dom}(r_1)$$

In contrast, containment is expressed on *rows*, which inhabit terms indirectly via type constructors Π and Σ. Containment is thus strictly monotone: For rows ρ_1 and ρ_2,

$$\rho_1 \leq \rho_2 := \mathtt{dom}(\rho_1) \subseteq \mathtt{dom}(\rho_2)$$

and we have instead the following explicit coercions.

$$\mathtt{prj} :: \rho_1 \leq \rho_2 \Rightarrow \Pi\rho_2 \to \Pi\rho_1$$
$$\mathtt{inj} :: \rho_1 \leq \rho_2 \Rightarrow \Sigma\rho_1 \to \Sigma\rho_2$$

It has long been known that inheritance is not subtyping [7]. Row containment thus gives a more limited (but tamed) account of inheritance.

Bounded quantification. *Bounded quantification* [4, 5, 9] addresses the interaction between parametric polymorphism and subtyping by allowing the restriction of type variables by upper bounds. For example, the type

$$\forall X.X \leq \mathtt{Int}.\tau$$

bounds instantiations of `X` to just the subtypes of `Int`. This additional expressivity comes with its own limits. Bounded subtyping in System $F_{\leq}$ (System F with bounded second-order polymorphism) is known to be undecidable [40], although there has been some recent progress on taming this interaction [57].

Constrained quantification. We describe in §5 that concatenation can prove difficult to type. One intuitive approach to ensuring the safety of concatenation is to constrain the input records to have disjoint label sets, as originally proposed by Harper and Pierce [13]. A system with constrained quantification can type concatenation as

$$\forall(X\#\emptyset).(Y\#X).X \to Y \to X \parallel Y$$

where $(Y\#X)$ asserts that X is disjoint from Y and the type $X \parallel Y$ denotes the concatenation of X and Y. Committing to such a constraint necessarily fixes your

row theory to one in which labels identify types uniquely, and so is not suitable for all theories (e.g., scoped rows). For this reason, ROSE and Rω more generally qualify types with the stipulation that concatenation simply be *well-defined* for the instantiated row theory.

intersection types & the merge operator. Subtyping and bounded quantification lift notions of containment to arbitrary types; the merge operator, commonly written as two commas $(,,)$, lifts the notion of concatenation to arbitrary types. The merge operator permits the merger of arbitrary terms into an *intersection type*. The intersection of types A an B is denoted (in these systems) as $A\&B$. *Intersection* here means that this term behaves as *both* `Int` and `Bool`, and not, as is also common [8], that it inhabits the (set theoretic) intersection of `Bool` and `Int`. The merge operator has received an influx of attention in recent years [54, 55, 45], no doubt in part due to its presence in many popular gradually typed systems, e.g., as *union types* in TypeScript [30] and *intersection types* in Scala [19]. In the case of records, merger is simply concatenation.

$$\{x : \texttt{Int}\},,\{y : \texttt{Int}\} \equiv \{x : \texttt{Int}, y : \texttt{Int}\}$$

More interesting is the merger of arbitrary types. The term below is in fact well-typed at `Int`: when `x` is of type `Bool`, the conditional will trigger and 1 is returned; when `x` is of type `Int`, then `x` is returned.

$$\lambda(x : \texttt{Int}\,\&\,\texttt{Bool}).\texttt{if } x \texttt{ then 1 else } x$$

The merge operator can be used to encode extensible data types through a paradigm called *compositionanl programming* [2, 45].

Disjoint polymorphism. Left unrestricted, the merge operator can quickly lead to non-determinism. Consider the term `if true ,, false then` ψ `else` ψ. The "unwieldly beast" [45], however, may be tamed by *disjoint polymorphism*—that is, to lift the idea of disjointedness of records to arbitrary type. Disjoint polymorphism constrains the merger of types with non-trivial intersection of terms. In the case of records, this is simply the constrained quantification of Harper and Pierce [13]. For arbitrary types, disjointedness constraints can rule out non-deterministic merges. For example, the merger `true ,, false` will not type, as the types of each term (both typed `Bool`) are not disjoint.

Aspects of both row polymorphism and bounded polymorphism can be shown to elaborated into systems with disjoint polymorphism [54].

Functorial encodings of extensible variants. Records and variants have a well known categorical semantics as polynomial functors [10]. Consequently, languages with (general) recursion that are capable of expressing functors as type operators (e.g., Haskell or OCaml) may encode variants. This approach was widely popularized by Swierstra [48], and may often be referred to simply as the "datatypes à la Carte" approach. The encoding relies on Haskell's typeclass system to, in essence, encode the containment and combination relations of ROSE, lifted to functors.

While particularly clever, this embedding has both technical and ergonomic limitations. In practice, GHC's typeclass resolution does not reason about polynomial functors with respect to their monoidal structure—that is, functors which are equivalent modulo associativity and commutativity are not considered equal. So, we cannot conclude during typeclass resolution that the functor $F :+: G$ is contained by $G :+: F$, despite the two being isomorphic and containment being reflexive. Nor can we conclude that $(F :+: G) :+: H$ is contained in $F :+: (G :+: H)$! Attempts have been made around this, e.g., by way of closed type families and instance chains [33], however the work cited concludes that functorial encodings may be too elaborate to ever be ergonomically viable. If anything, the work shows that any functorial encoding of extensible variants must reason modulo algebraic structure, which (this author argues) is best done in the type theory itself.

References

[1] *The Glasgow Haskell Compiler*. URL HTTPS://WWW.HASKELL.ORG/GHC/.

[2] Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. Distributive disjoint polymorphism for compositional programming. In Luís Caires, editor, *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 381–409. Springer, 2019. ISBN 978-3-030-17183-4. doi: 10.1007/978-3-030-17184-1_14. URL HTTPS://DOI.ORG/10.1007/978-3-030-17184-1_14.

[3] Luca Cardelli. A semantics of multiple inheritance. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984, Proceedings*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer, 1984. doi: 10.1007/3-540-13346-1_2.

[4] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, 1985. doi: 10.1145/6041.6042. URL HTTPS://DOI.ORG/10.1145/6041.6042.

[5] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Inf. Comput.*, 109(1/2):4–56, 1994. doi: 10.1006/inco.1994.1013. URL HTTPS://DOI.ORG/10.1006/INCO.1994.1013.

[6] Adam Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 122–133. ACM, 2010. doi: 10.1145/1806596.1806612.

[7] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In Frances E. Allen, editor, *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 125–135. ACM Press, 1990. ISBN 0-89791-343-4. doi: 10.1145/96709.96721. URL HTTPS://DOI.ORG/10.1145/96709.96721.

[8] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Math. Log. Q.*, 27(2-6):45–58, 1981.

doi: 10.1002/malq.19810270205. URL https://doi.org/10.1002/malq.19810270205.

[9] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in $f_{<=}$. *Math. Struct. Comput. Sci.*, 2(1): 55–91, 1992. doi: 10.1017/S0960129500001134. URL https://doi.org/10.1017/S0960129500001134.

[10] Benedict R. Gaster. *Records, variants and qualified types*. PhD thesis, University of Nottingham, UK, 1998. URL https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.262959.

[11] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, University of Nottingham, 1996.

[12] Joseph A. Goguen, James W. Thatcher, Eric G. Wagner, and Jesse B. Wright. Initial algebra semantics and continuous algebras. *J. ACM*, 24(1):68–95, 1977. doi: 10.1145/321992.321997. URL https://doi.org/10.1145/321992.321997.

[13] Robert Harper and Benjamin C. Pierce. A record calculus based on symmetric concatenation. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*, pages 131–142, 1991. doi: 10.1145/99583.99603. URL https://doi.org/10.1145/99583.99603.

[14] Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016*, pages 15–27, 2016. doi: 10.1145/2976022.2976033. URL https://doi.org/10.1145/2976022.2976033.

[15] Alex Hubers and J. Garrett Morris. Generic programming with extensible data types; or, making ad hoc extensible data types less ad hoc. ICFP 23, 2023.

[16] Lalita Jategaonkar and John C. Mitchell. ML with extended pattern matching and subtypes. In Jérôme Chailloux, editor, *Proceedings of the 1988 ACM Conference on LISP and Functional Programming, LFP 1988, Snowbird, Utah, USA, July 25-27, 1988*, pages 198–211. ACM, 1988. doi: 10.1145/62678.62702. URL https://doi.org/10.1145/62678.62702.

[17] Mark P. Jones. A theory of qualified types. In Bernd K. Bruckner, editor,

Proceedings of the 4th European symposium on programming, volume 582 of *ESOP'92*. Springer-Verlag, Rennes, France, 1992.

[18] Mark P. Jones. Simplifying and improving qualified types. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, FPCA '95, pages 160–169, La Jolla, California, USA, 1995. ACM.

[19] LAMP/EPFL. *Scala: Intersection Types*, 2023. URL HTTPS://DOCS.SCALA-LANG.ORG/SCALA3/REFERENCE/NEW-TYPES/INTERSECTION-TYPES.HTML.

[20] Daan Leijen. First-class labels for extensible rows. Technical Report UU-CS-2004-51, Dept. of Computer Science, Universiteit Utrecht, December 2004. URL HTTPS://WWW.MICROSOFT.COM/EN-US/RESEARCH/PUBLICATION/FIRST-CLASS-LABELS-FOR-EXTENSIBLE-ROWS/.

[21] Daan Leijen. Extensible records with scoped labels. In *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005*, pages 179–194, 2005.

[22] Daan Leijen. Koka: Programming with row polymorphic effect types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*, pages 100–126, 2014. doi: 10.4204/EPTCS.153.8. URL HTTPS://DOI.ORG/10.4204/EPTCS.153.8.

[23] Daan Leijen. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 486–499, 2017. doi: 10.1145/3009837.3009872. URL HTTPS://DOI.ORG/10.1145/3009837.3009872.

[24] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52 (7):107–115, 2009. doi: 10.1145/1538788.1538814. URL HTTPS://DOI.ORG/10.1145/1538788.1538814.

[25] Sam Lindley and James Cheney. Row-based effect types for database integration. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Languages Design and Implementation, TLDI 2012, Philadelphia, PA, USA, Saturday, January 28, 2012*, pages 91–102, 2012. doi: 10.1145/2103786.2103798. URL HTTPS://DOI.ORG/10.1145/2103786.2103798.

[26] Sam Lindley and J. Garrett Morris. Lightweight functional session types. In

Simon Gay and antònio Ravara, editors, *Behavioural Types: From Theory to Tools*, chapter 12. River Publishers, 2017. doi: 10.13052/rp-9788793519817. URL HTTP://DX.DOI.ORG/10.13052/RP-9788793519817.

[27] Zhaohui Luo. Dependent record types revisited. *Proceedings of the 1st Workshop on Modules and Libraries for Proof Assistants*, Aug 2009. doi: 10.1145/1735813.1735819. URL HTTP://DX.DOI.ORG/10.1145/1735813.1735819.

[28] Henning Makholm and J. B. Wells. Type inference, principal typings, and let-polymorphism for first-class mixin modules. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 156–167, 2005. doi: 10.1145/1086365.1086386. URL HTTPS://DOI.ORG/10.1145/1086365.1086386.

[29] Robert C. Martin. *Clean Code - a Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009. ISBN 978-0-13-235088-4. URL HTTP://VIG.PEARSONED.COM/STORE/PRODUCT/1,1207,STORE-12521_ISBN-0132350882,00.HTML.

[30] Microsoft. *TypeScript: Documentation – Everyday Types*, 2023. URL HTTPS://WWW.TYPESCRIPTLANG.ORG/DOCS/HANDBOOK/2/EVERYDAY-TYPES.HTML#UNION-TYPES.

[31] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, pages 348–375, 1978.

[32] John C. Mitchell. Coercion and type inference. In Ken Kennedy, Mary S. Van Deusen, and Larry Landweber, editors, *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah, USA, January 1984*, pages 175–185. ACM Press, 1984. doi: 10.1145/800017.800529. URL HTTPS://DOI.ORG/10.1145/800017.800529.

[33] J. Garrett Morris. Variations on variants. In Ben Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 71–81, Vancouver, BC, 2015. ACM.

[34] J. Garrett Morris and James McKinna. Abstracting extensible data types: or, rows by any other name. *Proc. ACM Program. Lang.*, 3(POPL):12:1–12:28, 2019. doi: 10.1145/3290325. URL HTTPS://WWW.YOUTUBE.COM/WATCH?V=5RDFYB2UDKA.

[35] OCaml. *Object types*. Institut National de Recherche en Informatique et en

Automatique. URL https://v2.ocaml.org/manual/types.html#sss:typexpr-obj.

[36] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Trans. Program. Lang. Syst.*, 17(6):844–895, 1995. doi: 10.1145/218570.218572. URL https://doi.org/10.1145/218570.218572.

[37] Oracle. *Multiple Inheritance of State. The Java Tutorials.* Oracle. URL https://docs.oracle.com/javase/tutorial/java/IandI/multipleinheritance.html.

[38] Jens Palsberg and Tian Zhao. Efficient type inference for record concatenation and subtyping. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 125–136. IEEE Computer Society, 2002. doi: 10.1109/LICS.2002.1029822.

[39] John Peterson and Mark Jones. Implementing type classes. *ACM SIGPLAN Notices*, 28(6):227–236, Jun 1993. ISSN 1558-1160. doi: 10.1145/173262.155112. URL http://dx.doi.org/10.1145/173262.155112.

[40] Benjamin C. Pierce. Bounded quantification is undecidable. *Inf. Comput.*, 112 (1):131–165, 1994. doi: 10.1006/inco.1994.1055. URL https://doi.org/10.1006/inco.1994.1055.

[41] Robert Pollack. Dependently typed records in type theory. *Formal Aspects Comput.*, 13(3-5):386–402, 2002. doi: 10.1007/s001650200018.

[42] Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. In Dan R. Ghica, editor, *The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015, Nijmegen, The Netherlands, June 22-25, 2015*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 19–35. Elsevier, 2015. doi: 10.1016/j.entcs.2015.12.003. URL https://doi.org/10.1016/j.entcs.2015.12.003.

[43] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 77–88. ACM Press, 1989.

[44] J. C. Reynolds. User-defined types and procedural data as complementary approaches to data abstraction. In S. A. Schuman, editor, *IFIP Working Group 2.1 on Algol, INRIA*, 1975.

[45] Nick Rioux, Xuejing Huang, Bruno C. d. S. Oliveira, and Steve Zdancewic. A bowtie for a beast: Overloading, eta expansion, and extensible data types

in f$\bowtie$. *Proc. ACM Program. Lang.*, 7(POPL):515–543, 2023. doi: 10.1145/3571211. URL HTTPS://DOI.ORG/10.1145/3571211.

[46] Ryan Stansifer. Type inference with subtypes. In Jeanne Ferrante and Peter Mager, editors, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 88–97. ACM Press, 1988. doi: 10.1145/73560.73568. URL HTTPS://DOI.ORG/10.1145/73560.73568.

[47] Martin Sulzmann. Designing record systems. Technical Report YALEU/DCS/RR-1128, Yale University, 1997.

[48] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, July 2008. ISSN 0956-7968.

[49] Philip Wadler. The expression problem, 1998. URL HTTPS://HOMEPAGES.INF.ED.AC.UK/WADLER/PAPERS/EXPRESSION/EXPRESSION.TXT.

[50] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76, 1989. doi: 10.1145/75277.75283. URL HTTPS://DOI.ORG/10.1145/75277.75283.

[51] Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987*, pages 37–44. IEEE Computer Society, 1987.

[52] Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988* Wand [51], page 132. doi: 10.1109/LICS.1988.5111. URL HTTPS://DOI.ORG/10.1109/LICS.1988.5111.

[53] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Inf. Comput.*, 93(1):1–15, 1991.

[54] Ningning Xie, Bruno C. d. S. Oliveira, Xuan Bi, and Tom Schrijvers. Row and bounded polymorphism via disjoint polymorphism. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPIcs*, pages 27:1–27:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi: 10.4230/LIPIcs.ECOOP.2020.27.

[55] Xu Xue, Bruno C. d. S. Oliveira, and Ningning Xie. Applicative intersection types. In *Programming Languages and Systems - 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings*, pages 155–174, 2022. doi: 10.1007/978-3-031-21037-2_8. URL HTTPS://DOI.ORG/10.1007/978-3-031-21037-2_8.

[56] Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. Technical report, EPFL Lausanne, 2012. URL HTTP://LAMPWWW.EPFL.CH/~ODERSKY/PAPERS/EXPRESSIONPROBLEM.HTML.

[57] Litao Zhou, Yaoda Zhou, and Bruno C. d. S. Oliveira. Recursive subtyping for all. *Proceedings of the ACM on Programming Languages*, 7(POPL): 1396–1425, Jan 2023. ISSN 2475-1421. doi: 10.1145/3571241. URL HTTP://DX.DOI.ORG/10.1145/3571241.

[58] Jan Zwanenburg. A type system for record concatenation and subtyping. HTTP://CITESEERX.IST.PSU.EDU/VIEWDOC/DOWNLOAD;JSESSIONID=9AA475365678088BC1146F881F7D1007?DOI=10.1.1.36.4147&REP=REP1&TYPE=PDF, 1997.