Abstracting Extensible Data Types

Or, Rows by Any Other Name

J. GARRETT MORRIS, The University of Kansas, USA
JAMES MCKINNA, The University of Edinburgh, UK

We present a novel typed language for extensible data types, generalizing and abstracting existing systems of row types and row polymorphism. Extensible data types are a powerful addition to traditional functional programming languages, capturing ideas from OOP-like record extension and polymorphism to modular compositional interpreters. We introduce row theories, a monoidal generalization of row types, giving a general account of record concatenation and projection (dually, variant injection and branching). We realize them via qualified types, abstracting the interpretation of records and variants over different row theories. Our approach naturally types terms untypable in other systems of extensible data types, while maintaining strong metatheoretic properties, such as coherence and principal types. Evidence for type qualifiers has computational content, determining the implementation of record and variant operations; we demonstrate this in giving a modular translation from our calculus, instantiated with various row theories, to polymorphic λ-calculus.

CCS Concepts: • **Theory of computation → Type theory**; • **Software and its engineering → Data types and structures**;

Additional Key Words and Phrases: Extensible data types, row types, row polymorphism, qualified types

ACM Reference Format:
J. Garrett Morris and James McKinna. 2019. Abstracting Extensible Data Types: *Or,* Rows by Any Other Name. *Proc. ACM Program. Lang.* 3, POPL, Article 12 (January 2019), 28 pages. https://doi.org/10.1145/3290325

1 INTRODUCTION

The goal of extensible data types is type-safe modular software development. We want to be able to define large software systems in terms of independent components. For example, in a compiler for a rich source language, we might want to define individual desugaring passes independently of the remainder of the syntax tree. However, the typing of these components should guarantee the typing of their composition: inconsistent assumptions about the underlying AST should be reflected as type errors, not run-time failures.

Row types provide one approach to typing extensible data types. Originally introduced to model inheritance [Wand 1987], row types combine structural typing for records and variants with parametric polymorphism. They achieve similar expressiveness to structural subtyping, while maintaining a purely parametric approach to typing (and correspondingly simplifying type inference). Row types have been the subject of much theoretical effort, and have been used to provide polymorphic variants, and types for objects, in the programming language OCaml [Garrigue 1998]. More recently, they have been applied to provide extensibility in effect type systems [Lindley and

Authors' addresses: J. Garrett Morris, Information and Telecommunication Technology Center, The University of Kansas, 2335 Irving Hill Rd, Lawrence, KS, USA, garrett@ittc.ku.edu; James McKinna, Laboratory for Foundations of Computer Science, The University of Edinburgh, 10 Crichton Street, Edinburgh, UK, James.McKinna@ed.ac.uk.

2475-1421/2019/1-ART12
https://doi.org/10.1145/3290325

Cheney 2012], to capture algebraic effects and handlers [Hillerström and Lindley 2016; Leijen 2014, 2017], and to express extensible choice and branching in session types [Lindley and Morris 2017].

Despite these successes, there are still several open problems in row types. First, there are several differing notions of rows, particularly in the conditions on *row extension* (i.e., adding individual new entries to rows). This impedes adoption of row types in other languages, as different applications apparently require different notions of rows. Second, while there are several competing approaches to extension, most existing systems do not support *concatenation* (i.e., combining multiple records) at all. There are several reasons to desire record concatenation. It naturally captures multiple inheritance-like patterns, in which the final behavior of a system arises from combining multiple independent components; the dual pattern arises in modular interpreters [Liang et al. 1995] (i.e. the expression problem [Wadler 1998]). Existing approaches to typing record concatenation depend on expressive type systems, such as intersection types [Wand 1991] or dependent types [Chlipala 2010], and require programmer-specified annotations not required in other row typed systems.

We propose a new approach to row types for extensible data types, based on two key ideas. The first is a *monoidal* account of rows and row extension. This supports record concatenation (and, dually, branching on variants) naturally and directly in our type system. The second is using qualified types to connect record and variant types to rows and row operations. This avoids expressiveness limitations in purely syntactic accounts of row types, while abstracting the meaning of records and variants over different monoidal row theories. We instantiate our approach in a simple functional language with Hindley-Milner polymorphism, which we call Rose. As an example of Rose, and of our approach in general, consider the following term (originally proposed by Wand):

$$\lambda mn.(m \star n).\mathtt{x}$$

This term defines a function on two records, which concatenates those records and projects the x field from the concatenation. The problem in typing this term is that we need to require that there be a x field in at least one of the input records (otherwise, the projection must fail), but requiring that it appear in either m or n individually overconstrains the use of the term (and overdetermines its meaning). The type for this term in Rose is

$$\forall t\, z_1 z_2.(\mathtt{x} \triangleright t) \circledcirc (z_1 \odot z_2) \Rightarrow \Pi z_1 \to \Pi z_2 \to t.$$

We interpret this type as follows: a function taking a record with fields given by z_1 (i.e., Πz_1) and a record with fields given by z_2 to a value of type t, such that the concatenation of z_1 and z_2 (i.e., $z_1 \odot z_2$) contains an entry for field x with type t. Rose maintains the strong metatheoretic properties of other Hindley-Milner languages, including principal types and decidable type inference.

One consequence of our use of qualified types is that, subject to some basic constraints, our type and row system can evolve independently. This means that we can use Rose with any of the different notions of rows in the literature (we demonstrate this for two systems, the simple rows of Wand [1987] and Rémy [1989] and the scoped rows of Berthomieu and le Moniès de Sagazan [1995] and Leijen [2005]), or even with multiple models of rows simultaneously (such as to support *both* extensible data types *and* algebraic effects). Alternatively, we could extend Rose to incorporate other language features, such as extensible effects or session-typed communication, without needing to alter our interpretation of rows.

In summary, this paper contributes:

- The design of the Rose language, highlighting the role of our predicate system in supporting expressive record concatenation and variant branching (§2);
- The introduction of *row theories*, an abstract characterization of row systems based on partial monoids, relating the syntax of rows to their interpretations and generalizing Rose across a variety of systems of records and variants (§3); and,

- A formalization of the Rose type system, and notions of translation both among row theories in Rose and from Rose to the polymorphic λ-calculus, giving computational content to proofs of Rose predicates in terms of the implementations of ground record and variant operations (§4).

We conclude by discussing related work (§5) and two areas of future work, focusing on the connections between our approach and those of prisms and lenses (§6) and extensible effects and effect handlers (§7).

2 PROGRAMMING IN ROSE

This section gives an intuitive overview of programming with extensible data types in Rose. Later in the paper, we will give a formal account of our type system and semantics (§4), and put our work in the context of other systems of row types and extensible data types (§5).

2.1 Extensible Records

We begin with a simple question: what do we mean by the field `x`? The answer, of course, depends upon context. If we are discussing points on a plane, then we use `x` to denote the first component of pairs of floating-point values. If we are instead considering points in space, `x` denotes the first component of triples of floating-point values. If we consider pixels on a screen, then `x` is likely to denote an integer value rather than a floating-point value, and the source object might contain color information as well as coordinates. In each of these cases, we have a well-defined (but differing) view of what it means to select the `x` field from a value. Now suppose that we abstract over selecting the `x` field (using a term such as $\lambda r.\ r.\mathtt{x}$). What type should we give this term? Clearly, we do not want to limit ourselves to only one kind of input value, or to only one kind of result type. Instead, we expect the type of this function to accept the variety of possible objects that have a field `x`, and to produce a result appropriate to the type of its input.

In Rose, we use *row types* to capture this kind of variance in types. (Row types, initially introduced by Wand [1987], have been the subject of significant research attention. We give a brief account of their history later in the paper.) We can think of a row ζ as a type-level association of field names ℓ to types τ—although we will generalize this idea shortly (§3)—and think of records ($\Pi\zeta$) and variants ($\Sigma\zeta$) as being types constructed from rows. For example, the type of two-dimensional points would be expressed $\Pi(\mathtt{x} \triangleright \mathtt{Double}, \mathtt{y} \triangleright \mathtt{Double})$; $(\mathtt{x} \triangleright \mathtt{Double}, \mathtt{y} \triangleright \mathtt{Double})$ is a row, while Π is a type constructor which, when applied to row, denotes records with those fields. Similarly, pixels would be described by the type $\Pi(\mathtt{x} \triangleright \mathtt{Int}, \mathtt{y} \triangleright \mathtt{Int}, \mathtt{r} \triangleright \mathtt{Byte}, \mathtt{g} \triangleright \mathtt{Byte}, \mathtt{b} \triangleright \mathtt{Byte})$.

We identify two key relations on rows, *containment* and *combination*. Intuitively, the containment relation $\zeta_1 \circledS \zeta_2$ holds if the fields of row ζ_1 are a subset of the fields of row ζ_2; for example, we would expect that $(\mathtt{x} \triangleright \mathtt{Double}) \circledS (\mathtt{x} \triangleright \mathtt{Double}, \mathtt{y} \triangleright \mathtt{Double})$, but not that $(\mathtt{x} \triangleright \mathtt{Double}) \circledS (\mathtt{x} \triangleright \mathtt{Int}, \mathtt{y} \triangleright \mathtt{Int}, \mathtt{r} \triangleright \mathtt{Byte}, \mathtt{g} \triangleright \mathtt{Byte}, \mathtt{b} \triangleright \mathtt{Byte})$. We can use the containment relation to express the type of selection abstractly: the term $\lambda r.\ r.\mathtt{x}$ has type

$$\forall t\ z.(\mathtt{x} \triangleright t) \circledS z \Rightarrow \Pi z \to t.$$

The containment relation here *qualifies* the polymorphism in the type; we read this type as "for all t and z *such that the row* $(\mathtt{x} \triangleright t)$ *is contained in* z, a function from a z-record to a t". Following the meaning of the containment relation, $\lambda r.\ r.\mathtt{x}$ would project a floating-point value from a point, but would project an integer from a pixel. In Rose, we will both instantiate type variables and discharge the corresponding predicates automatically, using an extended version of Hindley-Milner type inference.

The combination relation is used to describe the construction of rows; intuitively, $\zeta_1 \odot \zeta_2$ denotes a row that contains all the fields in ζ_1 and ζ_2. We use combination to generalize record construction.

$$\frac{P \mid \Gamma \vdash M : \Pi\zeta_1 \quad P \Rightarrow \zeta_2 \lhd \zeta_1}{P \mid \Gamma \vdash \mathsf{prj}\, M : \Pi\zeta_2} \qquad \frac{P \mid \Gamma \vdash M : \Pi\zeta_1 \quad P \mid \Gamma \vdash N : \Pi\zeta_2 \quad P \Rightarrow \zeta_1 \odot \zeta_2 \sim \zeta_3}{P \mid \Gamma \vdash M \star N : \Pi\zeta_3}$$

$$\frac{\{\ell_1 \triangleright \tau_1, \ldots, \ell_{\mathtt{m}} \triangleright \tau_m\} \subseteq \{\ell'_1 \triangleright \tau'_1, \ldots, \ell'_{\mathtt{n}} \triangleright \tau'_n\}}{P \Rightarrow (\ell_1 \triangleright \tau_2, \ldots, \ell_{\mathtt{m}} \triangleright \tau_m) \lhd (\ell'_1 \triangleright \tau'_1, \ldots, \ell'_{\mathtt{n}} \triangleright \tau'_n)}$$

$$\frac{\{\ell_1 \triangleright \tau_1, \ldots, \ell_{\mathtt{k}} \triangleright \tau_k\} \uplus \{\ell_{\mathtt{k}+1} \triangleright \tau_{k+1}, \ldots, \ell_{\mathtt{m}} \triangleright \tau_m\} = \{\ell'_1 \triangleright \tau'_1, \ldots, \ell'_{\mathtt{m}} \triangleright \tau'_m\}}{P \Rightarrow (\ell_1 \triangleright \tau_1, \ldots, \ell_{\mathtt{k}} \triangleright \tau_k) \odot (\ell_{\mathtt{k}+1} \triangleright \tau_{k+1}, \ldots, \ell_{\mathtt{m}} \triangleright \tau_m) \sim (\ell'_1 \triangleright \tau'_1, \ldots, \ell'_{\mathtt{m}} \triangleright \tau'_m)}$$

Fig. 1. Excerpted typing and entailment rules for record and row operations in Rose; context Γ tracks typing assumptions for term variables while context P tracks predicate assumptions on type variables.

For example, the term $\lambda v r.(\mathtt{x} = v \mid r)$ extends record r by adding field x with value v; it has type

$$\forall t\, z.t \to \Pi z \to \Pi((\mathtt{x} \triangleright t) \odot z).$$

That is to say, it consumes a record of type Πz, whatever row z happens to be, and produces a record containing all of z and an x-labeled field of type t. There is a significant additional source of complexity to record extension, however. Suppose that record r already contains an x-labeled field. In that case, should we treat extension with a new x field as a type error, as overwriting the existing field, or attempt to preserve both fields? Each of these solutions appears in existing systems of extensible records. Regardless of which approach we pick, how should it be reflected in the typing of $\lambda v r.(\mathtt{x} = v \mid r)$?

We capture these possibilities by treating combination as a three-place predicate rather than as a binary type constructor. The relation $\zeta_1 \odot \zeta_2 \sim \zeta_3$ holds if ζ_3 is a row capturing the combination of rows ζ_1 and ζ_2. The type $\forall t\, z.t \to \Pi z \to \Pi((\mathtt{x} \triangleright t) \odot z)$ is shorthand for the qualified type

$$\forall t\, z_1 z_2.(\mathtt{x} \triangleright t) \odot z_1 \sim z_2 \Rightarrow t \to \Pi z_1 \to \Pi z_2;$$

that is, given a type t and two rows z_1 and z_2 *such that the combination of* $(\mathtt{x} \triangleright t)$ *and* z_1 *is* z_2, the term consumes a t value and a Πz_1 record and produces a Πz_2 record. (While for any given z_1 there may be multiple z_2 such that this predicate is satisfiable, we insist that they be equivalent up to permutation of the fields, and so indistinguishable in the language. This assures that there is no ambiguity resulting from the choice of z_2.) Now consider our three distinct interpretations of record extension.

- If we interpret extension as overwriting existing fields, then the constraint $(\mathtt{x} \triangleright t) \odot z_1 \sim z_2$ is always satisfiable, but some of the fields in z_1 (i.e., fields with label x) may not appear in z_2.
- If we interpret extension as only being valid for new field labels, then we would consider predicates like $(\mathtt{x} \triangleright \mathtt{Int}) \odot (\mathtt{x} \triangleright \mathtt{Double}, \mathtt{y} \triangleright \mathtt{Double}) \sim \zeta$ unsatisfiable for any choice of ζ.
- Finally, if we interpret combination as preserving both the original and new fields, then predicate $(\mathtt{x} \triangleright t) \odot z_1 \sim z_2$ is again always satisfiable, but where z_2 now maintains the contents of both of its arguments (i.e., both old and new mappings for field x).

For the remainder of this section, we will assume that attempting to replace existing fields is a type error—an approach we term "simple rows"—and return to other possibilities in the next section. The primitives and predicates introduced so far are summarized in Figure 1.

Having both the combination and containment predicates may seem superfluous: surely $\zeta_1 \lhd \zeta_3$ holds in exactly those cases where there is a ζ_2 such that $\zeta_1 \odot \zeta_2 \sim \zeta_3$. While this characterizes the satisfiability of $\zeta_1 \lhd \zeta_3$, having distinct predicates more closely parallels the term structure, and so,

as we will see when we discuss the semantics of Rose (§4), captures the translation of Rose into languages without row types.

2.2 Records by Concatenation

So far, we have only discussed record operations one field at a time—selecting individual fields from records, or extending records with individual fields. We can generalize these ideas to operate on (sub-)records instead of individual fields. First, we introduce a general projection operator, prj, which computes an arbitrary substructure of its (record) argument. For example, suppose that r is a pixel—that is, it has type $\Pi(\mathtt{x} \triangleright \mathtt{Int}, \mathtt{y} \triangleright \mathtt{Int}, \mathtt{r} \triangleright \mathtt{Byte}, \mathtt{g} \triangleright \mathtt{Byte}, \mathtt{b} \triangleright \mathtt{Byte})$. Depending on context, prj r could denote either the coordinates of r, of type $\Pi(\mathtt{x} \triangleright \mathtt{Int}, \mathtt{y} \triangleright \mathtt{Int})$, or the color of r, of type $\Pi(\mathtt{r} \triangleright \mathtt{Byte}, \mathtt{b} \triangleright \mathtt{Byte}, \mathtt{g} \triangleright \mathtt{Byte})$, or indeed any of the individual fields of r. The prj operation captures the full expressiveness of the containment relation: $\lambda r.\mathsf{prj}\ r$ has type

$$\forall z_1 z_2 . z_2 \Subset z_1 \Rightarrow \Pi z_1 \rightarrow \Pi z_2 .$$

The behavior of this term (indeed of any use of prj) is determined by the evidence for $z_2 \Subset z_1$. That evidence, in turn, is determined by the instantiations of type variables z_1 and z_2. The behavior of prj is thus determined by the (types in) the context in which it is used; this is what allows us to use such an apparently general operator without introducing ambiguity in the meaning of programs.

Similarly, we generalize record extension to record concatenation: if M_1 and M_2 are two records, then $M_1 \star M_2$ denotes the concatenation of those two records. This term captures the full expressiveness of the concatenation relation: $\lambda mn.m \star n$ has type

$$\forall z_1 z_2 z_3 . z_1 \odot z_2 \sim z_3 \Rightarrow \Pi z_1 \rightarrow \Pi z_2 \rightarrow \Pi z_3 .$$

As in the single-field case, we capture the limitations on the structure of rows by expressing combination as a relation: we can only instantiate z_1 and z_2 to rows that have a well-defined combination. (In the remainder of the paper, we will generally write combination as if it were an infix type constructor:

$$\forall z_1 z_2 . \Pi z_1 \rightarrow \Pi z_2 \rightarrow \Pi(z_1 \odot z_2).$$

This should always be interpreted in terms of the $\_\odot\_\sim\_$ relation, and the partiality that it provides; it may not be the case that $z_1 \odot z_2$ is well-defined for arbitrary choices of z_1 and z_2.

We introduce syntax for singleton records: for any label ℓ and term M of type τ, the term $\ell \triangleright M$ has type $\Pi(\ell \triangleright \tau)$; similarly, for any term N of type $\Pi(\ell \triangleright \tau)$, the term N/ℓ has type τ. Now, we can define the field-at-a-time operations from the last section in terms of our more general operations. Abstraction over field selection, $\lambda r.\ r.\mathtt{x}$, is implemented by $\lambda r.\mathsf{prj}\ r/\mathtt{x}$. For prj $r/\mathtt{x}$ to be well-typed, we must have that prj r has type $\Pi(\mathtt{x} \triangleright \tau)$ for some type τ; in turn, from the type of prj, we must have that r has some record type $\Pi\zeta$ such that the constraint $(\mathtt{x} \triangleright \tau) \Subset \zeta$ is satisfiable. (This demonstrates how the behavior of prj is determined by context: the context $-/\mathtt{x}$ forces the result of prj to be the $\Pi(\mathtt{x} \triangleright \tau)$ singleton record.) Generalizing, we get

$$\forall t\ z . (\mathtt{x} \triangleright t) \Subset z \Rightarrow \Pi z \rightarrow t,$$

which is the type we expected for x-selection. The record extension term $\lambda vr.(\mathtt{x} = v \mid r)$ is expressed as $\lambda vr.(\mathtt{x} \triangleright v) \star r$. Its typing is more straightforward: if v is of type τ, and r is of type $\Pi\zeta$, then $(\mathtt{x} \triangleright v) \star r$ is of type $\Pi((\mathtt{x} \triangleright \tau) \odot \zeta)$; generalizing, we get

$$\forall t\ z . t \rightarrow \Pi z \rightarrow \Pi((\mathtt{x} \triangleright t) \odot z).$$

Update and extension. Requiring that $\zeta_1 \odot \zeta_2$ is defined only when ζ_1 and ζ_2 have disjoint label sets simplifies reasoning about the resulting type system. For example, it allows us to assume that $\odot$ is commutative (and so that $\lhd$ corresponds to subset) without worrying about whether $(\mathtt{x} \triangleright \mathtt{Int}) \odot (\mathtt{x} \triangleright \mathtt{Double})$ should be $(\mathtt{x} \triangleright \mathtt{Int})$ or $(\mathtt{x} \triangleright \mathtt{Double})$. However, it can make defining record update (as opposed to extension) more complicated. Some existing systems [Gaster and Jones 1996; Leijen 2005] introduce a separate primitive, *restriction*, that remove fields from records. Update can then be defined as the composition of restriction and extension. Rémy [1989] introduces *presence* flags, denoting the presence or absence of fields, and types polymorphic in presence flags. This allows him to define a single operation that can serve as either extension or update. We can express the combined extension and update operation in our system using only the primitives already introduced, without needing to introduce presence flags or new forms of polymorphism. To do so, instead of our earlier interpretation, we implement the term $\lambda vr.(\mathtt{x} = v \mid r)$ by

$$\lambda vr.(\mathtt{x} \triangleright v) \star \mathsf{prj}\ r$$

(where the change is the application of projection to r). As before, if v has type τ and $\mathsf{prj}\ r$ has type $\Pi\zeta_2$, then the result will be of type $\Pi((\mathtt{x} \triangleright \tau) \odot \zeta_2)$. Now, however, rather than $\Pi\zeta_2$ being the type of the input record, it only needs to be contained in the full type of input record $\Pi\zeta_1$; generalizing over τ, ζ_1, and ζ_2, we get

$$\forall t\, z_1 z_2 . z_2 \lhd z_1 \Rrightarrow t \to \Pi z_1 \to \Pi((\mathtt{x} \triangleright t) \odot z_2).$$

This raises two questions. First, is the term general enough? That is, does it express both record extension and record update? Second, does the use of the projection operator (in a generic context) leave the behavior of the term too unconstrained?

To answer the first question, we consider both possibilities. Suppose that the input value is type τ, the input record is of type $\Pi\zeta_1$, and $\mathtt{x}$ does not appear in ζ_1. Then, we can pick z_2 to also be ζ_1; the constraint $\zeta_1 \lhd \zeta_1$ is trivially satisfiable, and so we get the expected extension operator of type $\tau \to \Pi\zeta_1 \to \Pi((\mathtt{x} \triangleright t) \odot \zeta_1)$ (where, by assumption, the result of $\odot$ is defined). Alternatively, suppose that $\mathtt{x}$ does appear in ζ_1; then, there must exist a row ζ_2 and type τ_2 such that $(\mathtt{x} \triangleright \tau_2) \odot \zeta_2 \sim \zeta_1$. In this case, we can pick z_2 to be ζ_2; then, we get the expected record update operator, of type $\tau \to \Pi((\mathtt{x} \triangleright \tau_2) \odot \zeta_2) \to \Pi((\mathtt{x} \triangleright \tau) \odot \zeta_2)$. To answer the second question, observe that, even though the result of the projection is used inside the extension function, it is still determined by a type parameter z_2 determined by the caller (or the calling context). So, there is no possibility for the projection operator to discard fields that might be required by the surrounding code.

Nevertheless, we can give a more precise type to the record update function, further constraining its possible behaviors:

$$\forall t u\, z_1 z_2 . z_2 \lhd (\mathtt{x} \triangleright u) \Rrightarrow t \to \Pi(z_2 \odot z_1) \to \Pi((\mathtt{x} \triangleright t) \odot z_1)$$

Again, we consider the two use cases, assuming in each that the input value is of type τ and input record is of type $\Pi\zeta$. If the input record, of type $\Pi(z_2 \odot z_1)$, does not contain an $\mathtt{x}$-labeled field, then z_2 must be the empty row. (Otherwise, since $z_2 \lhd (\mathtt{x} \triangleright u)$, there would have to be an $\mathtt{x}$-labeled field in the input record.) So, z_1 must be ζ, and we get the expected extension function, with type

$$\tau \to \Pi\zeta \to \Pi((\mathtt{x} \triangleright \tau) \odot \zeta).$$

Type variable u can be freely instantiated: while it is not constrained by the remainder of the type, it also plays no role in the behavior of the term. Alternatively, suppose the input record does contain an $\mathtt{x}$-labeled field $\mathtt{x} \triangleright v$. Then z_2 must be instantiated to the row $(\mathtt{x} \triangleright v)$, and so u must be instantiated to v. (If not, then z_1 must contain $\mathtt{x} \triangleright v$, and so $(\mathtt{x} \triangleright \tau) \odot z_1$ is not defined.) This gives the expected

update function, with type

$$\tau \to \Pi((\mathtt{x} \triangleright v) \odot \zeta) \to \Pi((\mathtt{x} \triangleright \tau) \odot \zeta).$$

Note that all we have done is to constrain the use of the extension function; the two possibilities allowed by the more elaborate type were also included in the simpler type. Therefore, while the more precise type is a useful demonstration of the expressiveness of the Rose type system, we prefer the simpler (and more general) alternative.

Default values. We can make a similar use of concatenation and projection to implement default values for fields. For example, suppose that we wanted pixels to be black by default, but not to have any default position information. We can define a function to add these default values to an existing record: $\lambda r.r \star \mathrm{prj}\,(\mathtt{r} \triangleright 0, \mathtt{g} \triangleright 0, \mathtt{b} \triangleright 0)$. Letting $\zeta_C = (\mathtt{r} \triangleright \mathtt{Byte}, \mathtt{g} \triangleright \mathtt{Byte}, \mathtt{b} \triangleright \mathtt{Byte})$, this function has type

$$\forall z_1 z_2 . z_2 \lessdot \zeta_C \Rightarrow \Pi z_1 \to \Pi(z_1 \odot z_2).$$

The projection ensures that, if r contains any color values, then those values are not replaced by the corresponding defaults. However, it does not guarantee that the result is a pixel. This could be either because the input record does not contain x and y fields, or because the projection operation removed colors from the defaults that are also not present in r. Either happening would be a consequence of the context (i.e., the instantiation of type variables z_1 and z_2), not of some capricious choice of the default value function. Nevertheless, we can give a more precise type that guarantees the result is a pixel. Letting $\zeta_P = (\mathtt{x} \triangleright \mathtt{Int}, \mathtt{y} \triangleright \mathtt{Int}, \mathtt{r} \triangleright \mathtt{Byte}, \mathtt{g} \triangleright \mathtt{Byte}, \mathtt{b} \triangleright \mathtt{Byte})$, we can give the same term the following type:

$$\forall z_1 z_2 z_3 . (z_2 \lessdot \zeta_C, z_1 \odot z_2 \sim z_3, \zeta_P \lessdot z_3) \Rightarrow \Pi z_1 \to \Pi z_3$$

Here we guarantee not only that the added fields are a subset of the color fields, but also that the resulting record includes at least all the pixel fields. (Of course, because our goal is extensibility, we do not want to overconstrain the result to contain *only* the pixel fields.) Thus, for example, if the input record does not contain an r field, the projection operator must preserve the r field in the default values. Observe the central role played by the predicates: in the result type, z_3 is constrained both to be the combination of z_1 and z_2 and by the requirement that it contain the pixel fields ζ_P.

Wand's problem. Record concatenation introduces typing challenges not present in systems that include only single-field record extension. Wand [1991] illustrates these difficulties with the term $\lambda mn.(m \star n).\mathtt{x}$. This function concatenates two records m and n, and projects field x from the result. (Wand's motivation for considering this term was multiple inheritance: records m and n represent the method implementations arising from two superclasses, while the result of the term is the implementation of method x for their subclass.) For this term to be well-typed, we should require that either m or n contain the x field, but we should not restrict x to necessarily appearing in either argument. This type in inexpressible in most existing type systems for extensible records. (The notable exceptions are Wand's [1991] system with intersection types, and Chlipala's [2010] dependently typed record calculus. We will contrast our approach with theirs later in the paper.) On the other hand, its principal type is quite naturally expressed in Rose:

$$\forall t\, z_1 z_2 . (\mathtt{x} \triangleright t) \lessdot (z_1 \odot z_2) \Rightarrow \Pi z_1 \to \Pi z_2 \to t.$$

That is: for any type t and rows z_1 and z_2 such that $\mathtt{x} \triangleright t$ appears in the combination of z_1 and z_2, a function from Πz_1 and Πz_2 records to a t value. The use of constraints is essential to expressing this type: the combination $z_1 \odot z_2$ does not appear in either the argument types or the result type of the function, but is central to understanding the typing of the function as a whole.

$$\frac{P \mid \Gamma \vdash M : \Sigma\zeta_1 \quad P \Rrightarrow \zeta_1 \lessdot \zeta_2}{P \mid \Gamma \vdash \mathsf{inj}\, M : \Sigma\zeta_2} \qquad \frac{P \mid \Gamma \vdash M : \Sigma\zeta_1 \to \tau \quad P \mid \Gamma \vdash N : \Sigma\zeta_2 \to \tau \quad P \Rrightarrow \zeta_1 \odot \zeta_2 \sim \zeta_3}{P \mid \Gamma \vdash M \bigtriangledown N : \Sigma\zeta_3 \to \tau}$$

Fig. 2. Excerpted typing rules for variant operations in Rose; entailment is given by the rules in Figure 1.

2.3 Extensible Variants

We turn to the dual of extensible records, extensible variants. Here, our motivating question might be what we mean by a data constructor `And`: if we are describing formulae in either classical or substructural logic, then we mean logical conjunction (albeit built out of different sorts of formulae); if we are describing ML definitions, then we may mean the combiner of mutually recursive definitions. In each case, we expect the `And` construct to operate on different argument types: formulae in classical or substructural logic, or (lists of) ML definitions, respectively. Now, suppose we want to abstract over data construction (using a term such as $\lambda xy.\mathtt{And}\ x\ y$). What type do we give this term?

Our approach here is exactly dual to our approach to describing extensible products. We introduce a type constructor Σ which, applied to a row, denotes a variant with those constructors. For example, following the initial algebra approach to defining recursive data types, we might describe Boolean formulae by taking the fixed point of the functor[1]

$$F(t) = \Sigma(\mathtt{And} \triangleright t \times t, \mathtt{Or} \triangleright t \times t, \mathtt{Not} \triangleright t, \mathtt{Atom} \triangleright \mathtt{String}).$$

As in the case for records, the operations on variants correspond to the containment and combination predicates. The prj operation provided record elimination, projecting substructures from records. The dual operation, inj, provides variant introduction, injecting values from smaller variants into larger variants. The $\star$ operation provided record introduction, building larger records out of smaller records. Its dual, $\bigtriangledown$, provides variant elimination, building eliminators for larger variants out of eliminators for smaller variants. The typing rules for these constructs are given in Figure 2.

The rule for $\bigtriangledown$ is more complex than the corresponding rule for $\star$, as it operates on functions $\Sigma\zeta \to \tau$ instead of directly on values $\Sigma\zeta$. By defining $\bigtriangledown$ on functions, we avoid needing new terms and types for first-class case-branches. We also get an appealing similarity to the categorical account of variants, given by (colimiting cones defining) coproducts, with premises of the elimination rule describing a cone over $\zeta_1 \odot \zeta_2$. To emphasize the connection to the category theoretic account, we define the corresponding operation for records: $f \vartriangle g$ is defined to be $\lambda x.f\ x \star g\ x$.

We reuse the syntax for singleton records to describe singleton variants: for any label ℓ and term M of type τ, the term $\ell \triangleright M$ has type $\Sigma(\ell \triangleright \tau)$, while if N has type $\Sigma(\ell \triangleright \tau)$ then N/ℓ has type τ. (That is, we identify singleton products and sums with the underlying type; we will return to this point in the next section.) Now, we can recover the expected primitive operations on variants in terms of our primitives. For example, the generic constructor term $\lambda xy.\mathtt{And}\ x\ y$ would be implemented by $\lambda xy.\mathsf{inj}\,(\mathtt{And} \triangleright (x, y))$; its type,

$$\forall t_1\, t_2\, z.(\mathtt{And} \triangleright t_1 \times t_2) \lessdot z \Rrightarrow t_1 \to t_2 \to \Sigma z,$$

[1]To avoid cluttering our presentation, we will not formalize the extension of Rose with recursive types. Any of the several approaches to encoding iso-recursive types, such as two-level types [Sheard and Pasalic 2004] or conjugate hylomorphisms [Hinze et al. 2015], could be combined with the Rose type system. Adding equi-recursive types could raise interesting usability challenges, particularly in reporting error messages. We will explore these issues further in future work, particularly the extent to which they can be addressed by a combination of programmer-specified type annotation and top-down/bi-directional type inference.

captures that its result can be a member of any variant type so long as that type contains at least the constructor `And`. Expressing elimination is more verbose, but no more complicated. For example, we could represent Haskell's `Maybe` type as the type $\Sigma(\texttt{Just} \triangleright t, \texttt{Nothing} \triangleright 1)$. Then, we would express a defaulting operation (i.e., Haskell's `fromMaybe`) with the following term:

$$\lambda d.(\lambda y.y/\texttt{Just}) \bigtriangledown (\lambda y.\text{let } () = y/\texttt{Nothing} \text{ in } d)$$

The left-hand side of $\bigtriangledown$ strips a `Just` label from its argument; it has type $\Sigma(\texttt{Just} \triangleright \tau) \to \tau$. The right-hand side strips a `Nothing` label from its argument and then returns d; if d has type τ, then this term has type $\Sigma(\texttt{Nothing} \triangleright 1) \to \tau$. Generalizing over τ, the function as a whole has type

$$\forall t.t \to \Sigma(\texttt{Just} \triangleright t, \texttt{Nothing} \triangleright 1) \to t.$$

We can generalize this term in several ways. For example, we could write a defaulting operation that works on any variant containing a `Just` label, regardless of the other labels in the variant:

$$\lambda d.(\lambda y.y/\texttt{Just}) \bigtriangledown (\lambda y.d).$$

This term has type

$$\forall t\, z.t \to \Sigma((\texttt{Just} \triangleright t) \odot z) \to t;$$

that is, it eliminates any variant type that contains at least the `Just` label, returning the default value for any input not of the form $\texttt{Just} \triangleright M$.

The $\bigtriangledown$ operator requires that its arguments eliminate disjoint sets of constructors. Following the pattern we used for default values in records, we can define an operator that selectively replaces the handling of individual branches. The term $\lambda fg.f \bigtriangledown (g \circ \text{inj})$ allows f to override the behavior of g. Its type is $\forall t\, z_1 z_2 z_3.z_2 \Subset z_3 \Rightarrow (\Sigma z_1 \to t) \to (\Sigma z_3 \to t) \to (\Sigma(z_1 \odot z_2) \to t)$, capturing that the resulting term handles all the cases handled by its first argument (z_1), and a subset (z_2) of those cases handled by its second argument (z_3) such that the combination of z_1 and z_2 is well-defined. As in the similar examples for extensible records, the subset z_2 of z_3 is picked by the caller, not by the term itself, and so cannot capriciously eliminate behavior needed in the remainder of the program.

3 GENERALIZING EXTENSIBILITY

Our examples so far have all assumed one model of rows: that rows uniquely associate labels to types, and that the uniqueness of labels is required by the combination relation. However, this is not the only model of rows: others include models in which labels need not be unique [Berthomieu and le Moniès de Sagazan 1995; Leijen 2005], or in which row fields are unlabeled [Bahr 2014; Morris 2015]. In this section, we show that Rose can capture such systems as well, by considering *non-commutative* interpretations of the combination relation $\_\odot\_\sim\_$, and corresponding refinements of the containment predicate, as well as the projection, and injection, operators, to reflect such extensions. We then generalize our approach to give both an abstract characterization of rows, called *row theories*, and a connection between the syntax of rows and their interpretations.

3.1 Scoped Records

We begin by considering scoped records [Berthomieu and le Moniès de Sagazan 1995; Leijen 2005], as used in the programming languages Koka and LCS. The key difference between scoped and simple records is in the handling of extension. With scoped records, record extension does not require that labels be unique, but also does not overwrite existing fields. Instead, new fields "shadow" existing fields; field selection returns the value most recently associated with a given label, but the earlier values can be recovered by dropping the newer fields from the record.

For an example, suppose that we were to extend the empty record with two different `x`-labeled values: `(x = 4 | (x = True | ()))`. Following the approach discussed in the prior section, this

$$\frac{}{\zeta \sim \zeta} \qquad \frac{\zeta_1 \sim \zeta_2 \quad \zeta_2 \sim \zeta_3}{\zeta_1 \sim \zeta_3} \qquad \frac{\zeta_1 \sim \zeta_2}{(\ell \triangleright \tau, \zeta_1) \sim (\ell \triangleright \tau, \zeta_2)} \qquad \frac{\ell \neq \ell' \quad \zeta_1 \sim \zeta_2}{(\ell \triangleright \tau, \ell' \triangleright \tau', \zeta_1) \sim (\ell' \triangleright \tau', \ell \triangleright \tau, \zeta_2)}$$

$$\frac{\zeta \sim (\ell_1 \triangleright \tau_1, \ldots, \ell_n \triangleright \tau_n)}{(\ell_1 \triangleright \tau_1, \ldots, \ell_i \triangleright \tau_i) \odot (\ell_{i+1} \triangleright \tau_{i+1}, \ldots, \ell_n \triangleright \tau_n) \sim \zeta} \qquad \frac{\zeta_1 \odot \zeta_2 \sim \zeta_3}{\zeta_1 \circledcirc_{\mathsf{L}} \zeta_3} \qquad \frac{\zeta_1 \odot \zeta_2 \sim \zeta_3}{\zeta_2 \circledcirc_{\mathsf{R}} \zeta_3}$$

$$\frac{P \mid \Gamma \vdash M : \Pi\zeta_1 \quad P \Rightarrow \zeta_2 \circledcirc_d \zeta_1}{P \mid \Gamma \vdash \mathsf{prj}_d\, M : \Pi\zeta_2}\,(d \in \{\mathsf{L}, \mathsf{R}\}) \qquad \frac{P \mid \Gamma \vdash M : \Sigma\zeta_1 \quad P \Rightarrow \zeta_1 \circledcirc_d \zeta_2}{P \mid \Gamma \vdash \mathsf{prj}_d\, M : \Sigma\zeta_2}\,(d \in \{\mathsf{L}, \mathsf{R}\})$$

Fig. 3. Typing and entailment rules for scoped rows, with projection operators for potentially non-commutative combination.

would either be a type error (interpreting extension strictly as $\star$) or would have type $\Pi(\mathtt{x} \triangleright \mathtt{Int})$ (interpreting extension as incorporating update). With scoped records, this term is instead given the type $\Pi(\mathtt{x} \triangleright \mathtt{Int}, \mathtt{x} \triangleright \mathtt{Bool})$; viewing the row as a list, we have that $\mathtt{x} \triangleright \mathtt{Int}$ is the newest entry in the row, while the older entry, $\mathtt{x} \triangleright \mathtt{Bool}$, is shadowed by it. Call our example record r. If we select the x field from r, $r.\mathtt{x}$, we get 4, the most recently added x-labeled value. On the other hand, if we drop an x-labeled value before selecting x, $(r - \mathtt{x}).\mathtt{x}$, then we get `True`, the older x-labeled value.

To incorporate scoped records into our framework, we have to answer two central questions. First, how are the containment and combination relations defined for scoped rows? Second, how can we derive the primitives operations on scoped records from the Rose primitives? A particular challenge here arises from the distinction between selection $r.\mathtt{x}$ and restriction $r - \mathtt{x}$. For simple rows, these can both be viewed uniformly as special cases of projection. Now, that seems less likely: there is a crucial distinction in their handling of the initial x-labeled field.

We begin by observing that we need some way to capture the ordering of fields. We do so by making row combination non-commutative: the constraints

$$(\mathtt{x} \triangleright \mathtt{Int}) \odot (\mathtt{x} \triangleright \mathtt{Bool}) \sim (\mathtt{x} \triangleright \mathtt{Int}, \mathtt{x} \triangleright \mathtt{Bool})$$
$$(\mathtt{x} \triangleright \mathtt{Bool}) \odot (\mathtt{x} \triangleright \mathtt{Int}) \sim (\mathtt{x} \triangleright \mathtt{Bool}, \mathtt{x} \triangleright \mathtt{Int})$$

are both satisfiable, but neither of

$$(\mathtt{x} \triangleright \mathtt{Int}) \odot (\mathtt{x} \triangleright \mathtt{Bool}) \sim (\mathtt{x} \triangleright \mathtt{Bool}, \mathtt{x} \triangleright \mathtt{Int})$$
$$(\mathtt{x} \triangleright \mathtt{Bool}) \odot (\mathtt{x} \triangleright \mathtt{Int}) \sim (\mathtt{x} \triangleright \mathtt{Int}, \mathtt{x} \triangleright \mathtt{Bool})$$

are satisfiable. We do not, however, impose a strict ordering on the result of combination: both of

$$(\mathtt{x} \triangleright \mathtt{Int}) \odot (\mathtt{y} \triangleright \mathtt{Bool}) \sim (\mathtt{x} \triangleright \mathtt{Int}, \mathtt{y} \triangleright \mathtt{Bool})$$
$$(\mathtt{x} \triangleright \mathtt{Int}) \odot (\mathtt{y} \triangleright \mathtt{Bool}) \sim (\mathtt{y} \triangleright \mathtt{Bool}, \mathtt{x} \triangleright \mathtt{Int})$$

are satisfiable. We also now have two different forms of containment: given rows ζ_1 and ζ_3, it can either be the case that $\zeta_1 \odot \zeta_2 \sim \zeta_3$ or the case that $\zeta_2 \odot \zeta_1 \sim \zeta_3$; unlike in the previous section, these are *not* equivalent. We write the first case as $\zeta_1 \circledcirc_{\mathsf{L}} \zeta_3$ and the second as $\zeta_1 \circledcirc_{\mathsf{R}} \zeta_3$. Correspondingly, we will require *distinct* left and right projection and injection operators. These rules are summarized in Figure 3. This is a straightforward generalization of the operators for simple rows: since $\odot$ is commutative for simple rows, we have $\mathsf{prj}_{\mathsf{L}} = \mathsf{prj}_{\mathsf{R}}$, $\mathsf{inj}_{\mathsf{L}} = \mathsf{inj}_{\mathsf{R}}$, and $\zeta_1 \circledcirc_{\mathsf{L}} \zeta_2$ iff $\zeta_1 \circledcirc_{\mathsf{R}} \zeta_2$.

Next, we have to define the basic operations on scoped records (extension, selection, and restriction) in terms of the Rose primitives (prj_d and $\star$). As we would expect, extension is defined in terms of $\star$: the term $(\mathtt{x} = v \mid r)$ is implemented by $(\mathtt{x} \triangleright v) \star r$. The typing of extension is also

unchanged from the previous section: $\lambda vr.(\mathtt{x} \triangleright v) \star r$ still has type $\forall t\, z.t \to \Pi z \to \Pi((\mathtt{x} \triangleright t) \odot z)$. However, unlike in the previous section, the combination constraint imposes no restriction on the fields already present in z; for any row ζ_1, there is a row ζ_2 such that $(\mathtt{x} \triangleright t) \odot \zeta_1 \sim \zeta_2$. We define new operations for field selection and restriction in terms of prj_d, constraining their types to guarantee the desired behavior:

$$\begin{array}{ll} \mathsf{take}_\ell : \forall t\, z.(\ell \triangleright t) \circledcirc_{\mathsf{L}} z \Rightarrow \Pi z \to t & \mathsf{drop}_\ell : \forall t\, z.\Pi((\ell \triangleright t) \odot z) \to \Pi z \\ \mathsf{take}_\ell\, r = \mathsf{prj}_{\mathsf{L}}\, r/\ell & \mathsf{drop}_\ell\, r = \mathsf{prj}_{\mathsf{R}}\, r \end{array}$$

Selection $r.\mathtt{x}$ is interpreted as $\mathsf{take}_{\mathtt{x}} r$ and restriction $r - \mathtt{x}$ is interpreted as $\mathsf{drop}_{\mathtt{x}} r$. An update operation can be defined in terms of restriction and extension: $\lambda vr.(\mathtt{x} = v \mid r - \mathtt{x})$. However, note that it is not meaningful to define a combined extension/update operation for scoped rows as it was for simple rows: extension is still defined even if there is already an $\mathtt{x}$ field in the record.

We might hope to generalize these operations to arbitrary rows, not just single fields. For record extension and selection, this is natural. The same is not true of restriction, however; there, because the eliminated label ℓ does not appear in the result of the term, the ℓ annotation is essential to determining its behavior. This also explains why the definition of drop_ℓ required a more complicated type signature than take.

3.2 Rows Abstracted

We have seen two different notions of rows (simple and scoped rows), how they give rise to two different notions of record and variants, and how both notions can be expressed in Rose by adjusting the interpretation of the containment and combination predicates. These are far from the only formulations of rows, or their only applications: other formulations of rows include unlabeled rows (both unique and overwriting) captured by various encodings of extensible data types [Bahr 2014; Kiselyov et al. 2004; Morris 2015; Oliveira et al. 2015], while other applications include the use of rows to describe choice in session types [Lindley and Morris 2017] and in effect types [Hillerström and Lindley 2016; Leijen 2014; Lindley and Cheney 2012]. This raises several questions:

- Could these other notions of rows also be expressed using containment and combination?
- Could Rose's approach to record and variants be extended to incorporate other applications of row types?
- Most importantly, is there a general methodology to expressing row types that avoids repeatedly redeveloping the interpretation of the containment and combination predicates?

We address the final question by separating the syntactic presentation of rows and row types (which we will call a *row theory*) from the underlying model (or algebra) of rows. Simple and scoped rows can both be expressed as theories of rows, with intuitive algebras building on the idea of mappings from labels to types. In turn, the Rose type system is parametric over a row theory, guaranteeing its generality. Finally, in giving a semantics for Rose, we will show how mappings between row theories can be uniquely extended to mappings between terms of Rose.

We begin by defining the syntactic presentations of rows.

Definition 1. A *row theory* is a 3-tuple $\langle \mathcal{R}, \sim, \Rightarrow \rangle$, as follows.

- $\mathcal{R}$ is a set of syntactic rows (that is, of well-formed ground row type expressions). We impose no further restriction on $\mathcal{R}$ at this level of abstraction: it may or may not require labels, may or may not impose uniqueness restrictions on those labels, and so forth. We let ζ range over elements of $\mathcal{R}$.
- Relation $\sim$ is an equivalence relation on $\mathcal{R}$; it accounts for the identification of syntactically distinct rows in most row type systems.

- Relation $\Rightarrow$ is an entailment relation on row predicates ($\zeta_1 \lessdot_d \zeta_2$ and $\zeta_1 \odot \zeta_2 \sim \zeta_3$), invariant with respect to $\sim$, satisfying monotonicity ($P \Rightarrow \psi$ if $\psi \in P$) and transitivity $P, Q \Rightarrow \phi$ if $P \Rightarrow \psi$ and $Q, \psi \Rightarrow \phi$).

Of course, in a practical language based on Rose, the entailment relation may also be used to express other constraints, such as type classes. Our intention here is to give a minimal specification sufficient to capture row typing. Each of the notions of rows presented so far is a row theory. We define the simple row theory by $\langle \mathcal{R}_{\text{simp}}, \sim_{\text{simp}}, \Rightarrow_{\text{simp}} \rangle$, where $\mathcal{R}_{\text{simp}}$ is the set of uniquely-labeled sequences of types, $\sim_{\text{simp}}$ identifies sequences up to permutation, and $\Rightarrow_{\text{simp}}$ is as given in Figure 1. The theory of scoped rows is given by $\langle \mathcal{R}_{\text{scop}}, \sim_{\text{scop}}, \Rightarrow_{\text{scop}} \rangle$, where $\mathcal{R}_{\text{scop}}$ is the set of arbitrarily-labeled sequences of types, and $\sim_{\text{scop}}$ and $\Rightarrow_{\text{scop}}$ are as given in Figure 3. We will see other theories of rows in the remainder of the paper.

To capture the intuitive meaning of a row theory, we introduce algebraic structures corresponding to rows, and their modeling relationship to row theories.

Definition 2. A *row algebra* is any partial monoid $\langle \mathcal{M}, \cdot, \epsilon \rangle$; that is, $\cdot$ is a partial binary function $\mathcal{M} \times \mathcal{M} \rightharpoonup \mathcal{M}$ such that: $m \cdot \epsilon = m = \epsilon \cdot m$; and, if $m_1 \cdot (m_2 \cdot m_3)$ is defined, then it is equal to $(m_1 \cdot m_2) \cdot m_3$. Let $f : \mathcal{R} \to \mathcal{M}$; we write $f \models \zeta_1 \odot \zeta_2 \sim \zeta_3$ if $f(\zeta_1) \cdot f(\zeta_2) = f(\zeta_3)$, extended to $f \models \zeta_1 \lessdot_d \zeta_2$ and $f \models P$ in the obvious way. We say that such an f is a *model* of $\langle \mathcal{R}, \sim, \Rightarrow \rangle$ in $\langle \mathcal{M}, \cdot, \epsilon \rangle$ iff:

- For $\zeta_1, \zeta_2 \in \mathcal{R}$, if $\zeta_1 \sim \zeta_2$, then $f(\zeta_1) = f(\zeta_2)$;
- If $P \Rightarrow \psi$, then for each ground substitution θ on $fv(P, \psi)$, $f \models \theta\, P$ implies $f \models \theta\, \psi$; and,
- There is some $\zeta_0 \in \mathcal{R}$ such that $f(\zeta_0) = \epsilon$.

We say that $\langle \mathcal{M}, \cdot, \epsilon \rangle$ is an algebra for (or model of) $\langle \mathcal{R}, \sim, \Rightarrow \rangle$ if there is such a function $f : \mathcal{R} \to \mathcal{M}$.

Obviously, any row theory gives rise to a term algebra in an unsurprising way. Other models of row theories are more informative.

Example 3 (Simple rows). Consider partial functions f, g from labels (drawn from $\mathcal{L}$) to types. We define a partial union $f \sqcup g$ by

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

if $dom(f)$ and $dom(g)$ are disjoint, and let $f \sqcup g$ be undefined otherwise. Now, we have that $\langle \mathcal{L} \rightharpoonup \mathcal{T}, \sqcup, \emptyset \rangle$ is an algebra for the simple row theory $\langle \mathcal{R}_{\text{simp}}, \sim_{\text{simp}}, \Rightarrow_{\text{simp}} \rangle$. This is also the algebra of rows implemented by a number of existing row type systems, including those of Rémy [1989], Harper and Pierce [1991], Gaster and Jones [1996], and Chlipala [2010].[2]

Example 4 (Scoped rows). We write $\mathcal{T}^\star$ for sequences of types and $v^\frown w$ for the concatenation of sequences v and w. We define a pointwise concatenation operator $\oplus$ on functions $f, g \in \mathcal{L} \to \mathcal{T}^\star$ by $(f \oplus g)(\ell) = f(\ell)^\frown g(\ell)$, and write f_0 for the function that maps all labels to the empty sequence. Now, we have that $\langle \mathcal{L} \to \mathcal{T}^\star, \oplus, f_0 \rangle$ is an algebra for $\langle \mathcal{R}_{\text{scop}}, \sim_{\text{scop}}, \Rightarrow_{\text{scop}} \rangle$. This algebra clarifies the intention of scoped rows: each scoped row should be viewed as a map from *all* labels to sequences of types; the syntax leaves out those labels mapped to empty sequences. We have $\Rightarrow_{\text{scop}} \zeta_1 \odot \zeta_2 \sim \zeta_3$ exactly when ζ_1 contains (possibly empty) pointwise initial sequences of ζ_3, and ζ_2 contains the corresponding pointwise final sequences.

[2] A formal, if lengthy, demonstration of this point would require building similar maps from the syntax of rows present in each language, and their representations of row operations, to this algebra.

Example 5 (Unlabeled rows). Bahr [2014] and Morris [2015] give encodings of unlabeled row theories, focusing on their use in defining extensible variants. Each provides equivalents to our inj and $\triangledown$ operators, but require some amount of additional type annotation as a consequence of their respective encoding techniques. Each has $\langle \mathscr{P}(\mathcal{T}), \uplus, \emptyset \rangle$ as an algebra, where the partial disjoint union operator $\uplus$ is defined by $T \uplus U = T \cup U$ if T and U are disjoint, and is undefined otherwise.

3.3 Row Homomorphisms

Having an abstract characterization of row theories, we can now consider their relative expressiveness. A natural idea is to consider structure-preserving maps between row theories. Such maps capture the expressiveness of row theories themselves: if one theory can be mapped into another, then programs in the first theory can be mapped to programs in the second in an entirely uniform manner, changing types but never terms. We formalize this intuition as follows.

Definition 6. A function $h : \mathcal{R}_1 \to \mathcal{R}_2$, extended to predicates in the obvious fashion, is a *row theory homomorphism* (or simply: *homomorphism*) from $\langle \mathcal{R}, \sim, \Rightarrow \rangle$ to $\langle \mathcal{R}', \sim', \Rightarrow' \rangle$ if $\zeta_1 \sim \zeta_2$ implies that $h(\zeta_1) \sim' h(\zeta_2)$ and $P \Rightarrow \psi$ implies that $h(P) \Rightarrow' h(\psi)$.

Row algebras are related by partial monoid homomorphisms. Homomorphisms of row theories and partial monoids are related in an intuitive way. Given that f is a model of $\langle \mathcal{R}, \sim, \Rightarrow \rangle$ in $\langle \mathcal{M}, \cdot, \epsilon \rangle$, and g is a model of $\langle \mathcal{R}', \sim', \Rightarrow' \rangle$ in $\langle \mathcal{M}', \cdot', \epsilon' \rangle$, if there is a row theory homomorphism from $\mathcal{R}$ to $\mathcal{R}'$, then there is a corresponding partial monoid homomorphism from $\mathcal{M}$ to $\mathcal{M}'$. Under the same assumptions, if there is a partial monoid homomorphism j from $\mathcal{M}$ to $\mathcal{M}'$, and for every $\zeta \in \mathcal{R}$, there is a $\zeta' \in \mathcal{R}'$ such that $j(f(\zeta)) = g(\zeta')$, there is a row theory homomorphism from $\mathcal{R}$ to $\mathcal{R}'$.

We can use row homomorphisms to capture the relationships among the row theories we have discussed so far. Intuitively, the scoped row theory encompasses the simple row theory, and so we expect that there is a homomorphism from simple to scoped rows. The homomorphism from $\mathcal{R}_{\mathsf{simp}}$ to $\mathcal{R}_{\mathsf{scop}}$ is given by the inclusion map, and the preservation of $\sim_{\mathsf{simp}}$ and $\Rightarrow_{\mathsf{simp}}$ is immediate. The corresponding homomorphism from $\langle \mathcal{L} \rightharpoonup \mathcal{T}, \uplus, \emptyset \rangle$ to $\langle \mathcal{L} \to \mathcal{T}^\star, \sqcup, f_0 \rangle$ is given by mapping defined cases to singleton sequences and undefined cases to the empty sequence (i.e., the standard mapping from `Maybe` to `List`). Similarly, we can see that there are row theory homomorphisms from the simple to unlabeled row theories and, assuming some suitable way to invent labels from types, a row theory homomorphism from unlabeled to simple rows. This captures our intuition that the labels in rows, while central to the *pragmatics* of records and variants, are not essential to their *semantics*.

The row theories that we have considered so far all provide some additional abstraction compared to simple products and coproducts. To describe the implementation of these theories in our formal development, it will be useful to capture products and coproducts themselves as a row theory.

Example 7 (Trivial rows). The trivial theory of rows is given by $\langle \mathcal{T}^\star, =, \Rightarrow_{\mathsf{triv}} \rangle$, where the atomic axioms of $\Rightarrow_{\mathsf{triv}}$ are of the form $\vdash \zeta_1 \odot \zeta_2 \sim \zeta_1 \frown \zeta_2$.

We will use its term algebra as our model of the trivial row theory. This theory identifies products and coproducts up to associativity, but not commutativity; each indexed projection is obtained by the composition $\mathsf{prj}_{\mathsf{L}} \circ \mathsf{prj}_{\mathsf{R}}$ (or vice versa), for a suitable choice of the intermediate result type, and similarly for injection.

It is easy to see that there is a row theory homomorphism from trivial to scoped rows: simply give the same label to each entry in the trivial row. It is more interesting, however, to consider homomorphisms between simple or scoped rows into trivial rows. Any such homomorphism would give uniform implementations of records and variants in terms of products and coproducts. Unfortunately, *no* such homomorphism can exist. We can see that there cannot be a row homomorphism

Variables	$x, v \in \mathcal{V}$	Directions	$d ::= \mathsf{L} \mid \mathsf{R}$
Rows	$\zeta, \xi \in \mathcal{R}$	Labels	$\ell \in \mathcal{L}$
Predicates	$\psi, \phi ::= \zeta \circledcirc_d \zeta \mid \zeta \odot \zeta \sim \zeta$	Qualified types	$\rho ::= \tau \mid \psi \Rightarrow \rho$
Types	$\tau ::= t \mid \tau \to \tau \mid \Pi\zeta \mid \Sigma\zeta \mid \ell \triangleright \tau$	Schemes	$\sigma ::= \rho \mid \forall t.\sigma \mid \forall z.\sigma$
Terms	$M, N ::= x \mid \lambda x.M \mid M\,N \mid \mathsf{let}\ x :: \sigma = M\ \mathsf{in}\ N$ $\mid \ell \triangleright M \mid M/\ell \mid \mathsf{prj}_d\, M \mid M \star N \mid \mathsf{inj}_d\, M \mid M \triangledown N$		
Environments	$\Gamma ::= \varepsilon \mid \Gamma, x : \sigma$	Contexts	$P ::= \varepsilon \mid P, v : \psi$

Fig. 4. Syntax of Rose($\mathcal{R}, \sim, \Rightarrow$)

from simple to trivial rows. Consider these results for simple rows:

$$\Rightarrow_{\mathsf{simp}} (\mathtt{x} \triangleright \mathtt{Int}) \circledcirc_{\mathsf{L}} (\mathtt{x} \triangleright \mathtt{Int}, \mathtt{y} \triangleright \mathtt{Bool}) \qquad \Rightarrow_{\mathsf{simp}} (\mathtt{y} \triangleright \mathtt{Bool}) \circledcirc_{\mathsf{L}} (\mathtt{x} \triangleright \mathtt{Int}, \mathtt{y} \triangleright \mathtt{Bool})$$

It is easy to show that, if $\Rightarrow_{\mathsf{triv}} \zeta_1 \circledcirc_{\mathsf{L}} \zeta_2$, $\Rightarrow_{\mathsf{triv}} \zeta_1' \circledcirc_{\mathsf{L}} \zeta_2$, and ζ_1 and ζ_1' are of the same length, then $\zeta_1 = \zeta_1'$. Consequently, we cannot hope to have a uniform mapping of simple to trivial rows that preserves $\Rightarrow_{\mathsf{simp}}$; since scoped rows are more expressive, we similarly cannot have a homomorphism from scoped to trivial rows. Neither can we find a homomorphism from trivial rows to simple rows. If we could, it would have to provide a mapping from each trivial row to a simple row, preserving combination of trivial rows. Consider the following entailments:

$$\Rightarrow_{\mathsf{triv}} (\mathtt{Int}) \odot (\mathtt{Bool}) \sim (\mathtt{Int}, \mathtt{Bool}) \qquad \Rightarrow_{\mathsf{triv}} (\mathtt{Bool}) \odot (\mathtt{Int}) \sim (\mathtt{Bool}, \mathtt{Int})$$
$$\Rightarrow_{\mathsf{triv}} (\mathtt{Int}) \circledcirc_{\mathsf{L}} (\mathtt{Int}, \mathtt{Bool}) \qquad \not\Rightarrow_{\mathsf{triv}} (\mathtt{Int}) \circledcirc_{\mathsf{L}} (\mathtt{Bool}, \mathtt{Int})$$

The crux of the problem is attempting to encode the non-commutative $\odot$ of scoped rows in terms of the commutative $\odot$ of simple rows. While we can imagine encodings of scoped rows that capture the order of labels, we cannot do so in a uniform way. In the examples, we must somehow be able to translate the row $(\mathtt{x} \triangleright \mathtt{Bool})$ differently depending on the combinations in which it later occurs.

However, all is not lost. In the following section (§4.6), we will show how we can give (local) translations from terms of Rose with simple rows to terms of Rose with trivial rows. This shows that, even for row theories that are incomparable directly, Rose gives us a framework to compare their expressiveness and relate their implementations.

4 THE ROSE TYPE SYSTEM

This section gives a formal specification of the type system and semantics of Rose. The syntax and typing of Rose are parameterized by a row theory $\langle \mathcal{R}, \sim, \Rightarrow \rangle$, giving the interpretation of the predicates. We give Rose's semantics by giving a (typing-directed) translation from Rose, instantiated with the trivial row theory, to a small extension of System F. We show some simple formal results on Rose, including the coherence of our interpretations. Finally, we show how other row theories can be translated to the trivial row theory; by composition, we thus obtain semantics for Rose instantiated with simple or scoped rows. Our goal in giving this translation is to make the abstract aspects of Rose concrete, particularly the roles played by the containment and combination predicates, and to show that Rose does not require any foundational extension to our understanding of computation in functional languages; we leave questions of efficiency to future work.

4.1 Rose Syntax

We write Rose($\mathcal{R}, \sim, \Rightarrow$) to denote the core Rose system instantiated with the row theory $\langle \mathcal{R}, \sim, \Rightarrow \rangle$; we will omit the instantiating row theory when it is apparent from context. Figure 4 gives the syntax

$$\begin{array}{lrcl} \text{Types} & A, B & ::= & a \mid A \to B \mid \forall a.A \mid \otimes\{A_i\} \mid \oplus\{A_i\} \\ \text{Terms} & E, F & ::= & x \mid \lambda x : A.E \mid F\,E \mid \Lambda a.E \mid E[A] \\ & & & \mid (E_1, \ldots, E_n) \mid \pi_i\,E \mid \iota_i\,E \mid \mathsf{case}\,E\,\{F_1, \ldots, F_n\} \end{array}$$

Fig. 5. $\mathrm{F}^{\otimes\oplus}$ term and type syntax.

of Rose($\mathcal{R}, \sim, \Rightarrow$); it should mostly be familiar from our earlier examples. In addition to the standard terms for abstraction, application, and polymorphic definition, we have terms for record projection ($\mathsf{prj}_d\,M$) and concatenation ($M \star N$) and variant injection ($\mathsf{inj}_d\,M$) and branching ($M \triangledown N$). We also include terms to label ($\ell \triangleright M$) and unlabel (M/ℓ) terms; these capture introduction and elimination of singleton records and variants. As is standard for Hindley-Milner style polymorphism, the type language is stratified into types (or monotypes) τ and type schemes (or polytypes) σ, with the latter extended by *row polymorphic* variables z. Types include type variables, functions, records, variants, and labeled types. Records and variants are built out of rows. The syntax of rows themselves we leave as a parameter of the system; in practice, syntactic formation rules for rows could be captured with kinds. The predicates are exactly the containment and combination predicates discussed so far. We do not assume any other uses of qualified types, such as to support type classes; however, nothing in Rose would prevent its combination with other uses of predicates. Finally, the formation of qualified types and type schemes is standard.

4.2 Target Language

The syntax for the target language of our semantics is given in Figure 5. It is a version of System F, extended with arbitrary width product and sum types. (While we could encode these types in unextended System F, doing so would only serve to complicate the presentation.) Types include functions, polymorphism, products ($\otimes\{A_i\}$) and coproducts ($\oplus\{A_i\}$). In addition to the usual terms, we include terms for product introduction ($E_1, \ldots, E_n$) and elimination ($\pi_i\,E$) and coproduct introduction ($\iota_i\,E$) and elimination ($\mathsf{case}\,E\,\{F_1, \ldots, F_n\}$). In each of these forms, i and n refer to concrete natural numbers; we do not assume any form of quantification over or dynamic use of indices. The typing rules are for $\mathrm{F}^{\otimes\oplus}$ are given in Figure 6. Environment Δ tracks type variables in scope, and the judgment $\Delta \vdash A$ type assures that all free type variables mentioned in A are included in Δ. We do not assume terms or types specific to encoding extensible data types, such as generic extension or projection operations, or labels appearing in types or terms.

4.3 Interpreting Terms and Types

Figure 7 gives the type system for Rose($\mathcal{R}, \sim, \Rightarrow$), and the translation from Rose to $\mathrm{F}^{\otimes\oplus}$. The figure has two parts. First we give the typing rules for Rose, and a generic translation from Rose typings into $\mathrm{F}^{\otimes\oplus}$ terms. This translation is parametric over two aspects of the translation. We assume a translation $(-)^\bullet$ from Rose types to $\mathrm{F}^{\otimes\oplus}$ types; the mechanics of this translation will depend, of course, upon the particular structure of rows and their interpretation in $\mathrm{F}^{\otimes\oplus}$. We also rely on an augmented entailment judgment $P \Rightarrow F : \psi$, denoting that F is a $\mathrm{F}^{\otimes\oplus}$ term giving evidence for the predicate ψ. Second, we extend the equivalence relation $\zeta_1 \sim \zeta_2$ on rows to an equivalence $\vdash \tau_1 \approx \tau_2$ on types. The remainder of this section discusses the translation in detail; the following sections discuss the translation of types and interpretation of entailment for different row theories.

Functional terms and types. Following Jones [1993; 1994], our typing derivations take the form $P \mid \Gamma \vdash M \rightsquigarrow E : \sigma$, where P is the predicate environment, Γ is the type environment, M is the Rose term with type σ, and E is its $\mathrm{F}^{\otimes\oplus}$ translation. The predicate environment P maps evidence

$$\frac{a \in \Delta}{\Delta \vdash a \text{ type}} \qquad \frac{\Delta \vdash A \text{ type} \quad \Delta \vdash B \text{ type}}{\Delta \vdash A \to B \text{ type}} \qquad \frac{\Delta, a \vdash A \text{ type}}{\Delta \vdash \forall a.A \text{ type}} \qquad \frac{\{\Delta \vdash A_i \text{ type}\}_i}{\Delta \vdash \circledast\{A_i\} \text{ type}} \, (\circledast \in \{\oplus, \otimes\})$$

$$(\textsc{var})\, \frac{(x : A) \in \Gamma}{\Delta; \Gamma \vdash x : A} \qquad (\forall \text{I})\, \frac{\Delta, a; \Gamma \vdash E : A}{\Delta; \Gamma \vdash \Lambda a.E : \forall a.A} \, (a \notin \Delta) \qquad (\forall \text{E})\, \frac{\Delta; \Gamma \vdash E : \forall a.A \quad \Delta \vdash B \text{ type}}{\Delta; \Gamma \vdash E[B] : A\{B/a\}}$$

$$(\to \text{I})\, \frac{\Delta \vdash A \text{ type} \quad \Delta; \Gamma, x : A \vdash E : B}{\Delta; \Gamma \vdash \lambda x : A.E : A \to B} \qquad (\to \text{E})\, \frac{\Delta; \Gamma \vdash F : A \to B \quad \Delta; \Gamma \vdash E : A}{\Delta; \Gamma \vdash F\,E : B}$$

$$(\otimes \text{I})\, \frac{\{\Delta; \Gamma \vdash E_i : A_i\}_i}{\Delta; \Gamma \vdash (E_1, \ldots, E_n) : \otimes\{A_i\}} \qquad (\otimes \text{E})\, \frac{\Delta; \Gamma \vdash E : \otimes\{A_j\}}{\Delta; \Gamma \vdash \pi_i\, E : A_i} \, (i \in \{j\})$$

$$(\oplus \text{I})\, \frac{\Delta; \Gamma \vdash E : A_i}{\Delta; \Gamma \vdash \iota_i\, E : \oplus\{A_j\}} \, (i \in \{j\}) \qquad (\oplus \text{E})\, \frac{\Delta; \Gamma \vdash E : \oplus\{A_i\} \quad \{\Delta; \Gamma \vdash F_i : A_i \to B\}_i}{\Delta; \Gamma \vdash \text{case}\, E\, \{F_1, \ldots, F_n\} : B}$$

Fig. 6. $\text{F}^{\otimes\oplus}$ typing rules.

$$(\textsc{var})\, \frac{(x : \sigma) \in \Gamma}{P \mid \Gamma \vdash x \rightsquigarrow x : \sigma} \qquad (\textsc{let})\, \frac{P \mid \Gamma \vdash M \rightsquigarrow E : \sigma \quad P \mid \Gamma, x : \sigma \vdash N \rightsquigarrow F : \tau}{P \mid \Gamma \vdash \text{let } x :: \sigma = M \text{ in } N \rightsquigarrow (\lambda x : (\sigma)^\bullet.F)\, E : \tau}$$

$$(\to \text{I})\, \frac{P \mid \Gamma, x : \tau \vdash M \rightsquigarrow E : \upsilon}{P \mid \Gamma \vdash \lambda x.M \rightsquigarrow \lambda x : (\tau)^\bullet.E : \tau \to \upsilon} \qquad (\to \text{E})\, \frac{P \mid \Gamma \vdash M \rightsquigarrow F : \tau \to \upsilon \quad P \mid \Gamma \vdash N \rightsquigarrow E : \tau}{P \mid \Gamma \vdash M\,N \rightsquigarrow F\,E : \upsilon}$$

$$(\Rightarrow \text{I})\, \frac{P, \nu : \psi \mid \Gamma \vdash M \rightsquigarrow E : \rho}{P \mid \Gamma \vdash M \rightsquigarrow \lambda \nu : (\psi)^\bullet.E : \psi \Rightarrow \rho} \qquad (\Rightarrow \text{E})\, \frac{P \mid \Gamma \vdash M \rightsquigarrow F : \psi \Rightarrow \rho \quad P \Rightarrow E : \psi}{P \mid \Gamma \vdash M \rightsquigarrow F\,E : \rho}$$

$$(\forall \text{I})\, \frac{P \mid \Gamma \vdash M \rightsquigarrow E : \sigma}{P \mid \Gamma \vdash M \rightsquigarrow \Lambda t.E : \forall t.\sigma} \, (t \notin fv(P, \Gamma)) \qquad (\forall \text{E})\, \frac{P \mid \Gamma \vdash M \rightsquigarrow E : \forall t.\sigma}{P \mid \Gamma \vdash M \rightsquigarrow E[(\tau)^\bullet] : \sigma\{\tau/t\}}$$

$$(\triangleright \text{I})\, \frac{P \mid \Gamma \vdash M \rightsquigarrow E : \tau}{P \mid \Gamma \vdash \ell \triangleright M \rightsquigarrow E : \ell \triangleright \tau} \qquad (\triangleright \text{E})\, \frac{P \mid \Gamma \vdash M \rightsquigarrow E : \ell \triangleright \tau}{P \mid \Gamma \vdash M/\ell \rightsquigarrow E : \tau} \qquad (\textsc{sim})\, \frac{P \mid \Gamma \vdash M \rightsquigarrow E : \tau \quad \vdash \tau \approx \upsilon}{P \mid \Gamma \vdash M \rightsquigarrow E : \upsilon}$$

$$(\Pi \text{I})\, \frac{P \mid \Gamma \vdash M_1 \rightsquigarrow E_1 : \Pi\zeta_1 \quad P \mid \Gamma \vdash M_2 \rightsquigarrow E_2 : \Pi\zeta_2 \quad P \Rightarrow F : \zeta_1 \odot \zeta_2 \sim \zeta_3}{P \mid \Gamma \vdash M_1 \star M_2 \rightsquigarrow F_\star\, E_1\, E_2 : \Pi\zeta_3}$$

$$(\Pi \text{E}_d)\, \frac{P \mid \Gamma \vdash M \rightsquigarrow E : \Pi\zeta_2 \quad P \Rightarrow F : \zeta_1 \circledcirc_d \zeta_2}{P \mid \Gamma \vdash \text{prj}_d\, M \rightsquigarrow F_{\text{prj}_d}\, E : \Pi\zeta_1} \qquad (\Sigma \text{I}_d)\, \frac{P \mid \Gamma \vdash M \rightsquigarrow E : \Sigma\zeta_1 \quad P \Rightarrow F : \zeta_1 \circledcirc_d \zeta_2}{P \mid \Gamma \vdash \text{inj}_d\, M \rightsquigarrow F_{\text{inj}_d}\, E : \Sigma\zeta_2}$$

$$(\Sigma \text{E})\, \frac{P \mid \Gamma \vdash M_1 \rightsquigarrow E_1 : \Sigma\zeta_1 \to \tau \quad P \mid \Gamma \vdash M_2 \rightsquigarrow E_2 : \Sigma\zeta_2 \to \tau \quad P \Rightarrow F : \zeta_1 \odot \zeta_2 \sim \zeta_3}{P \mid \Gamma \vdash M_1 \bigtriangledown M_2 \rightsquigarrow F_\bigtriangledown\, [(\tau)^\bullet]\, E_1\, E_2 : \Sigma\zeta_3 \to \tau}$$

$$\frac{}{\vdash \Pi(\tau) \approx \tau} \qquad \frac{}{\vdash \Sigma(\tau) \approx \tau} \qquad \frac{\zeta_1 \sim \zeta_2}{\vdash \Xi\zeta_1 \approx \Xi\zeta_2} \, (\Xi \in \{\Pi, \Sigma\}) \qquad \frac{}{\vdash \tau \approx \tau} \qquad \frac{\vdash \tau_1 \approx \tau_2}{\vdash \tau_2 \approx \tau_1} \qquad \frac{\vdash \tau_1 \approx \tau_2 \quad \vdash \tau_2 \approx \tau_3}{\vdash \tau_1 \approx \tau_3}$$

Fig. 7. Rose typing rules and translation to $\text{F}^{\otimes\oplus}$.

variables ν to predicates ψ; it should be viewed as constraining all the type variables appearing in the remainder of the judgment. (Although we distinguish evidence variables ν from term variables x for clarity, they do not constitute different syntactic classes in the translation.) The translation of variables (VAR), quantification ($\forall$I), and instantiation ($\forall$E) are all unsurprising. ROSE distinguishes between polymorphic (LET) and monomorphic ($\rightarrow$I) binding; as a corresponding distinction does not exist in $\mathrm{F}^{\otimes\oplus}$, the images of their translations are identical. Qualified type introduction ($\Rightarrow$I) corresponds to abstraction over evidence, while their elimination ($\Rightarrow$E) requires the provision of that evidence. This is one of the points of interface between the type system and the predicate system. The entailment judgment $P \Rightarrow E : \psi$ denotes that, if the predicates in P hold (with evidence assigned to the corresponding evidence variables), then E is evidence that ψ holds as well. The rules for this judgment determine the interpretation of the predicates, and so are left abstract.

Labels and equality. As the meaning of labels is captured solely by the predicates, labeled types and terms are translated identically to their unlabeled equivalents ($\triangleright\, I, \triangleright\, E$). There are two sources of type equality in ROSE. One arises from equivalence of rows; for example, we expect that the types $\Pi(\mathtt{x} \triangleright \mathtt{Int}, \mathtt{y} \triangleright \mathtt{Bool})$ and $\Pi(\mathtt{y} \triangleright \mathtt{Bool}, \mathtt{x} \triangleright \mathtt{Int})$ are equivalent. The other arises from our interpretation of singleton records and variants as equivalent to their underlying type. These are captured in the equivalence relationship $\vdash \tau_1 \approx \tau_2$. Equivalence is defined only for types; it can be generalized to qualified types and type schemes using pairs of the corresponding introduction and elimination rules. We require that equivalence be preserved in the interpretation (SIM): if E is the translation of a term of type τ_1, and $\vdash \tau_1 \approx \tau_2$, then E must also be the translation of that term at type τ_2.

Predicates and evidence. The meaning of record and variant terms is determined by the interpretation of the predicates. We specify the form that we expect this interpretation to take. Each of our predicates appears in two contexts: once in the treatment of records, and once in the treatment of variants. Correspondingly, the evidence for each of these predicates must support two operations: the evidence for containment supports record projection and variant injection, while the evidence for combination supports record concatenation and variant branching. We capture these requirements as follows:

$$\begin{array}{ll} \text{If } F : \zeta_1 \circledcirc_d \zeta_2 : & \text{If } F : \zeta_1 \odot \zeta_2 \sim \zeta_3 : \\ \quad F_{\mathsf{prj}_d} : (\Pi\zeta_2)^\bullet \rightarrow (\Pi\zeta_1)^\bullet & \quad F_\star : (\Pi\zeta_1)^\bullet \rightarrow (\Pi\zeta_2)^\bullet \rightarrow (\Pi\zeta_3)^\bullet \\ \quad F_{\mathsf{inj}_d} : (\Sigma\zeta_1)^\bullet \rightarrow (\Sigma\zeta_2)^\bullet & \quad F_\triangledown : \forall a.((\Sigma\zeta_1)^\bullet \rightarrow a) \rightarrow ((\Sigma\zeta_2)^\bullet \rightarrow a) \rightarrow (\Sigma\zeta_3)^\bullet \rightarrow a \end{array}$$

The interpretation of combination for variants is complicated by the need to refer to the result type, but otherwise these should be unsurprising. We place no other restrictions on the structure or implementation of evidence.

Records and variants. Finally, we have the rules for product introduction (ΠI) and elimination (ΠE) and variant introduction (ΣI) and elimination (ΣE). These should be largely unsurprising—each operation is interpreted in terms of the relevant evidence. The rule for variant elimination is slightly complicated by having to refer to the result type. Given that this type appears in the types of M and M', however, this introduces no potential for ambiguity in the typing or translation. One observation at this point is that we could have treated the record and variant operations purely as constants, rather than with distinct typing rules; following that approach, evidence would only have been computed in ($\Rightarrow$E), and the interpretation of that evidence would have been determined by the constants. In our view, while that approach would have required fewer typing rules, it would have camouflaged the central contribution of the type system.

$$(t)^\bullet = t \qquad (\tau \to v)^\bullet = (\tau)^\bullet \to (v)^\bullet \qquad (\psi \Rightarrow \rho)^\bullet = (\psi)^\bullet \to (\rho)^\bullet \qquad (\forall t.\sigma)^\bullet = \forall t.(\sigma)^\bullet$$

$$(\ell \triangleright \tau)^\bullet = (\tau)^\bullet \qquad (\Pi(\tau_1, \ldots, \tau_n))^\bullet = \otimes\{(\tau_i)^\bullet : 1 \le i \le n\} \qquad (\Sigma(\tau_1, \ldots, \tau_n))^\bullet = \oplus\{(\tau_i)^\bullet : 1 \le i \le n\}$$

$$(\zeta_1 \otimes \zeta_1)^\bullet = \otimes \begin{Bmatrix} (\Pi\zeta_2)^\bullet \to (\Pi\zeta_1)^\bullet \\ (\Sigma\zeta_1)^\bullet \to (\Sigma\zeta_2)^\bullet \end{Bmatrix}$$

$$(\zeta_1 \odot \zeta_2 \sim \zeta_3)^\bullet = \otimes \begin{Bmatrix} (\Pi\zeta_1)^\bullet \to (\Pi\zeta_2)^\bullet \to (\Pi\zeta_3)^\bullet, \forall a.((\Sigma\zeta_1)^\bullet \to a) \to ((\Sigma\zeta_2)^\bullet \to a) \to (\Sigma\zeta_3)^\bullet \to a \\ (\zeta_1 \otimes \zeta_3)^\bullet, (\zeta_2 \otimes \zeta_3)^\bullet \end{Bmatrix}$$

$$\pi_{i:j}\, E = \begin{cases} E & \text{if } j = 1 \\ \pi_i\, E & \text{otherwise} \end{cases} \qquad \iota_{i:j}\, E = \begin{cases} E & \text{if } j = 1 \\ \iota_i\, E & \text{otherwise} \end{cases}$$

Let $\zeta_1 = (\tau_1, \ldots, \tau_j)$, and $\zeta_2 = (\tau'_1, \ldots, \tau'_k)$.

$$\mathsf{prj}_{\mathsf{L}}^{\zeta_1,\zeta_2} = \lambda x : (\Pi\zeta_2)^\bullet.\pi_{1:k}\, x, \text{ if } j = 1; \quad \lambda x : (\Pi\zeta_2)^\bullet.(\pi_{i:k}\, x)_{1 \le i \le j}, \text{ otherwise.}$$

$$\mathsf{prj}_{\mathsf{R}}^{\zeta_1,\zeta_2} = \lambda x : (\Pi\zeta_2)^\bullet.\pi_{k:k}\, x, \text{ if } j = 1; \quad \lambda x : (\Pi\zeta_2)^\bullet.(\pi_{i:k}\, x)_{k-j \le i \le k}, \text{ otherwise.}$$

$$\mathsf{inj}_{\mathsf{L}}^{\zeta_1,\zeta_2} = \lambda x : (\Sigma\zeta_1)^\bullet.\iota_{1:k}\, x, \text{ if } j = 1; \quad \lambda x : (\Sigma\zeta_1)^\bullet.\mathsf{case}\, x\, \{\lambda y : (\tau_i)^\bullet.\iota_{i:k}\, y\}_{1 \le i \le j}, \text{ otherwise.}$$

$$\mathsf{inj}_{\mathsf{R}}^{\zeta_1,\zeta_2} = \lambda x : (\Sigma\zeta_1)^\bullet.\iota_{k:k}\, x, \text{ if } j = 1; \quad \lambda x : (\Sigma\zeta_1)^\bullet.\mathsf{case}\, x\, \{\lambda y : (\tau_i)^\bullet.\iota_{i:k}\, y\}_{k-j \le i \le k}, \text{ otherwise.}$$

$$\mathsf{concat}^{\zeta_1,\zeta_2} = \lambda x : (\Pi\zeta_1)^\bullet.\lambda y : (\Pi\zeta_2)^\bullet.(\pi_{1:j}\, x, \ldots, \pi_{j:j}\, x, \pi_{1:k}\, y, \ldots, \pi_{k:k}\, y)$$

$$\mathsf{branch}^{\zeta_1,\zeta_2} = \Lambda a.\lambda f : (\Sigma\zeta_1)^\bullet \to a.\lambda g : (\Sigma\zeta_2)^\bullet \to a.\lambda z : (\Sigma(\zeta_1 \frown \zeta_2))^\bullet.$$
$$\mathsf{case}\, z \begin{Bmatrix} \lambda x : (\tau_i)^\bullet.f\,(\iota_{i:j}\, x) & \text{if } i \le j \\ \lambda y : (\tau'_{i-j})^\bullet.g\,(\iota_{i-j:k}\, y) & \text{otherwise} \end{Bmatrix}_{1 \le i \le j+k}$$

$$\frac{v : \psi \in P}{P \Rightarrow v : \psi} \qquad \frac{P \Rightarrow F : \zeta_1 \odot \zeta_2 \sim \zeta_3}{P \Rightarrow \pi_3\, F : \zeta_1 \otimes_{\mathsf{L}} \zeta_2} \qquad \frac{P \Rightarrow F : \zeta_1 \odot \zeta_2 \sim \zeta_3}{P \Rightarrow \pi_4\, F : \zeta_2 \otimes_{\mathsf{R}} \zeta_3}$$

$$\frac{\zeta_1 \frown \zeta_2 = \zeta_3}{P \Rightarrow (\mathsf{concat}^{\zeta_1,\zeta_2}, \mathsf{branch}^{\zeta_1,\zeta_2}, (\mathsf{prj}_{\mathsf{L}}^{\zeta_1,\zeta_3}, \mathsf{inj}_{\mathsf{L}}^{\zeta_1,\zeta_3}), (\mathsf{prj}_{\mathsf{R}}^{\zeta_2,\zeta_3}, \mathsf{inj}_{\mathsf{R}}^{\zeta_2,\zeta_3})) : \zeta_1 \odot \zeta_2 \sim \zeta_3}$$

$$F_\star = \pi_1\, F \qquad F_\triangledown = \pi_2\, F \qquad F_{\mathsf{prj}_d} = \pi_1\, F \qquad F_{\mathsf{inj}_d} = \pi_2\, F$$

Fig. 8. Interpretation of trivial rows in $\mathrm{F}^{\otimes\oplus}$

4.4 Entailment and Evidence for Trivial Rows

Figure 8 shows the entailment relation and evidence for the trivial row theory. The top of the figure gives the translation on types; in the center we give metatheoretic definitions of the terms used to construct evidence in the entailment relation, and at the bottom we give the entailment relation and define the projections from evidence to implementations of the record and variant operations.

Types. The translation on types $(-)^\bullet$ broadly interprets each Rose type as the parallel $\mathrm{F}^{\otimes\oplus}$ construct: functions are interpreted as functions, polymorphism as polymorphism, records and variants as products and coproducts respectively. There are, however, several subtleties. First, we interpret qualified types $\psi \Rightarrow \rho$ as functions from the evidence for ψ to ρ. This means our translation on types depends on a interpretation of predicates. In our translation, we interpret predicates by the interpretations of the corresponding record and variant operations; with a more expressive target language, we could give more concise descriptions of evidence. Second, we have

no representation of labels in the target language, so labeled types $\ell \triangleright \tau$ are interpreted identically to their unlabeled versions. This emphasizes that the interpretation of the labels will be captured entirely through the interpretation of predicates.

Evidence. The trivial row theory is a close match for the n-ary products and coproducts in $\mathrm{F}^{\otimes\oplus}$. The only complexities in the construction of evidence arise from the identification of singleton products and variants with their underlying types. The projection and injection operators $\pi_{i:j}$ and $\iota_{i:j}$ abstract much of this handling. The constructions $\mathsf{prj}_d^{\zeta_1,\zeta_2}$, $\mathsf{inj}_d^{\zeta_1,\zeta_2}$, $\mathsf{concat}^{\zeta_1,\zeta_2}$, and $\mathsf{branch}^{\zeta_1,\zeta_2}$ each denote families of $\mathrm{F}^{\otimes\oplus}$ terms, one for each ground instantiation of ζ_1 and ζ_2. The terms $\mathsf{prj}_{\mathsf{L}}^{\zeta_1,\zeta_2}$ and $\mathsf{prj}_{\mathsf{R}}^{\zeta_1,\zeta_2}$ extract initial and final subsequences of their argument. Dually, the terms $\mathsf{inj}_{\mathsf{L}}^{\zeta_1,\zeta_2}$ and $\mathsf{inj}_{\mathsf{R}}^{\zeta_1,\zeta_2}$ inject values from sub-coproducts into larger coproducts. In each case, the only complexity arises from the possibility that the ζ_i may be singleton rows. The term $\mathsf{concat}^{\zeta_1,\zeta_2}$ concatenates two products. The term $\mathsf{branch}^{\zeta_1,\zeta_2}$ combines eliminators for coproducts; other than the need to abstract over the result type, this term is straightforward.

Entailment. The entailment relation is pleasingly direct. The containment relations $\circledcirc_{\mathsf{L}}$ and $\circledcirc_{\mathsf{R}}$ are proved in terms of the corresponding combination predicate; while we could have given the containment relations directly, this approach limits the need to specify semantically unnecessary containment predicates in types. The evidence for the combination predicate contains both the implementations of the operations corresponding to combination (record concatenation and variant branching), but also contains the (tuples of) evidence arising from both corresponding containment relations. The evidence for the containment relations can then be projected directly from the evidence for the combination. Finally, we interpret the operations by projecting the implementations from the evidence.

Lemma 8. *Suppose that* $P \Rightarrow F : \psi$, *and let* $\Delta = fv(P, \psi)$. *Let* $(P)^\bullet$ *be the* $\mathrm{F}^{\otimes\oplus}$ *typing environment* $\{v : (\psi)^\bullet \mid (v : \psi) \in P\}$. *Then:*

(1) $\Delta; (P)^\bullet \vdash F : (\psi)^\bullet$;

(2) If $\psi = \zeta_1 \circledcirc_d \zeta_2$, *then* $\Delta; (P)^\bullet \vdash F_{\mathsf{prj}_d} : (\Pi\zeta_2)^\bullet \to (\Pi\zeta_1)^\bullet$ *and* $\Delta; (P)^\bullet \vdash F_{\mathsf{inj}_d} : (\Sigma\zeta_1)^\bullet \to (\Sigma\zeta_2)^\bullet$; *and,*

(3) If $\psi = \zeta_1 \odot \zeta_2 \sim \zeta_3$, *then* $\Delta; (P)^\bullet \vdash F_\star : (\Pi\zeta_1)^\bullet \to (\Pi\zeta_2)^\bullet \to (\Pi(\zeta_1 \frown \zeta_2))^\bullet$ *and* $\Delta; (P)^\bullet \vdash F_\triangledown : \forall a.((\Sigma\zeta_1)^\bullet \to a) \to ((\Sigma\zeta_1)^\bullet \to a) \to (\Sigma(\zeta_1 \frown \zeta_2))^\bullet \to a$.

Each is shown by a combination of induction on the structure of the entailment derivation and routine type checking for the evidence constructions.

4.5 Properties of Typing and Translation

We establish several formal properties validating our type system and semantics. We begin a preliminary definition. In a typing judgment $P \mid \Gamma \vdash M \rightsquigarrow E : \sigma$, type variables in σ are constrained both by the predicates in P and by any predicates in σ. We reflect this using constrained type schemes.

Definition 9. A *constrained type scheme* $(P \mid \sigma)$ pairs a type scheme σ with a set of predicates P constraining its free type variables. We can interpret constrained type schemes in $\mathrm{F}^{\otimes\oplus}$: if $P = v_1 : \psi_1, \ldots, v_n : \psi_n$, then $(P \mid \sigma)^\bullet = (\psi_1)^\bullet \to \cdots \to (\psi_n)^\bullet \to (\sigma)^\bullet$.

We then introduce a generality relation $C : \sigma_1 \sqsupseteq \sigma_2$ on type schemes, shown in Figure 9.[3] Intuitively, we have $\sigma_1 \sqsupseteq \sigma_2$ if every ground instance of σ_2 is also a ground instance of σ_1. If that

[3] Mitchell and Jones call this relation "containment"; we have chosen different terminology to avoid confusion with the relation on rows.

$$\frac{\vdash \tau_1 \approx \tau_2}{\mathsf{id} : \sigma \sqsupseteq \sigma} \qquad \frac{C_1 : \sigma_1 \sqsupseteq \sigma_2 \quad C_2 : \sigma_2 \sqsupseteq \sigma_3}{C_2 \circ C_1 : \sigma_1 \sqsupseteq \sigma_3} \qquad \frac{C : \sigma_1[\tau/t] \sqsupseteq \sigma_2}{\lambda x : \forall a.(\sigma_1)^\bullet.C\,(x[(\tau)^\bullet]) : \forall t.\sigma_1 \sqsupseteq \sigma_2}$$

$$\frac{C : \sigma_1[b/a] \sqsupseteq \sigma_2 \quad b \notin fv(\sigma_1)}{\lambda x : \forall a.(\sigma_1)^\bullet.\Lambda b.C\,(x[b]) : \forall a.\sigma_1 \sqsupseteq \forall b.\sigma_2} \qquad \frac{C : \sigma_1 \sqsupseteq \sigma_2}{\lambda x : \forall a.\forall b.(\sigma_1)^\bullet.\Lambda b.\Lambda a.C\,(x[a][b]) : \forall a.\forall b.\sigma_1 \sqsupseteq \forall b.\forall a.\sigma_2}$$

$$\frac{\{\vdash v_1 : \phi_1, \ldots, v_m : \phi_m \Rightarrow E_i : \psi_i\}_i \quad C : \rho_1 \sqsupseteq \rho_2}{\lambda x.\lambda v_1 : (\phi_1)^\bullet.\ldots.\lambda v_m : (\phi_m)^\bullet.C\,(x\,E_1\ \ldots\ E_n) : (\psi_1 \Rightarrow \cdots \Rightarrow \psi_n \Rightarrow \rho_1) \sqsupseteq (\phi_1 \Rightarrow \cdots \Rightarrow \phi_m \Rightarrow \rho_2)}$$

Fig. 9. Ordering type schemes.

is the case, it should also be the case that we can transform an $\mathrm{F}^{\otimes\oplus}$ term in the interpretation of σ_1 into one in the interpretation of σ_2. The term C provides this transformation. This judgment is extended to constrained type schemes $C : (P_1 \mid \sigma_1) \sqsupseteq (P_2 \mid \sigma_2)$ in the obvious manner, following the rule for qualified types. The following is lemma is proved by a simple induction on $\sqsupseteq$ derivations.

Lemma 10. *If $C : (P_1 \mid \sigma_1) \sqsupseteq (P_2 \mid \sigma_2)$, then $\vdash C : (\sigma_1)^\bullet \to (\sigma_2)^\bullet$.*

We now turn to the main results of this section.

Theorem 11 (Principality). *If $P \mid \Gamma \vdash M \rightsquigarrow E : \sigma$, then there is some $(P_0 \mid \sigma_0)$ such that $P_0 \mid \Gamma \vdash M \rightsquigarrow E : \sigma_0$ and for all $(P \mid \sigma)$ such that $P \mid \Gamma \vdash M \rightsquigarrow E : \sigma$, $(P_0 \mid \sigma_0) \sqsupseteq (P \mid \sigma)$.*

We are able to draw almost entirely on existing work on principality for qualified types systems [Jones 1994]. We construct a syntax-directed variant of the type system, adapt Algorithm M [Lee and Yi 1998] to our type system, and then show soundness and completeness relationships between each. The only complexities arise from the use of (sim) to wrap and unwrap singleton products and sums; however, we can push these to the elimination rules in the syntax-directed type system, and incorporate them into unification for type inference.

Theorem 12 (Soundness). *If $P \mid \Gamma \vdash M \rightsquigarrow E : \sigma$, then $\Delta; (\Gamma)^\bullet \vdash E : (\sigma)^\bullet$, where $\Delta = fv(P, \Gamma, \sigma)$.*

The proof is by induction on typing derivations in Rose. Because Rose terms are implicitly quantified, we build the $\mathrm{F}^{\otimes\oplus}$ type variable environment Δ from the free type variables in the Rose typing derivation, in an entirely standard way. The only interesting cases of the induction are those that manipulate evidence. The soundness of those cases follows from the entailment judgment producing well-typed evidence (Lemma 8).

Rose terms may have multiple typing derivations, and so have translations to differing $\mathrm{F}^{\otimes\oplus}$ terms; this is typical for qualified type systems, and is known as the *coherence problem*. We show that, assuming a syntactically verifiable condition on source terms, the interpretations of these type derivations are all interconvertible, and so the meaning of terms is well-defined. Our approach follows that of Jones [1993; 1994], in turn built on the work of Mitchell [1988], adapted to the specifics of the Rose type system.

Unfortunately, it is simply not the case that all Rose terms, in all row theories, have coherent meanings. For example, consider the term $\mathsf{prj}_\mathsf{L} \circ \mathsf{prj}_\mathsf{R}$ in either the trivial or scoped row theory. Depending on the type assigned to the invocation of prj_R, this term can select any subset of its input record. This ambiguity is reflected in the term's principal type.

$$\vdash \mathsf{prj}_\mathsf{L} \circ \mathsf{prj}_\mathsf{R} : \forall z_1 z_2 z_3.(z_3 \circledcirc_\mathsf{L} z_2, z_2 \circledcirc_\mathsf{R} z_1) \Rightarrow \Pi z_1 \to \Pi z_3$$

The type variable z_2 appears in the qualifiers, but nowhere in the type of the term, so its instantiation can be chosen arbitrarily when constructing typings of this term; differing choices for its

instantiation give rise to different interpretations of the term. However, it is not the case that any term whose typing includes type variables that do not appear in the type gives rise to incoherence. Recall the (fully-desugared) type we gave to Wand's example (§2.2):

$$\lambda mn.(\mathrm{prj}\,(m \star n))/\ell : \forall t\, z_1 z_2 z_3.(z_1 \odot z_2 \sim z_3, (\ell \triangleright t) \lessdot z_3) \Rightarrow \Pi z_1 \to \Pi z_2 \to t$$

Type variable z_3 appears in the qualifiers, but not in the type itself. However, this does not introduce incoherence: z_3 is determined (up to $\sim$) by z_1 and z_2, which do appear in the type. To capture this notion, we introduce the idea of the closure of a set of type variables T over a set of predicates Ψ.

Definition 13. We define T^+_Ψ as the smallest set U such that

- $T \subseteq U$; and,
- if $\zeta_1 \odot \zeta_2 \sim \zeta_3 \in \Psi$, and $fv(\zeta_1, \zeta_2) \subseteq U$, then $fv(\zeta_3) \subseteq U$.

This approach was originally introduced by Jones [2000] to account for functional dependencies in type class predicates. We can then define notions of coherent type schemes and coherent terms.

Definition 14. A type scheme $\forall T.\Psi \Rightarrow \tau$ is *coherent* if $T \subseteq fv(\tau)^+_\Psi$. A term M is coherent if its principal type scheme is coherent.

Finally, we can state our coherence result for the translation.

Theorem 15 (Coherence). *If M is coherent, $P_1 \mid \Gamma \vdash M \rightsquigarrow E_1 : \sigma_1$, $P_2 \mid \Gamma \vdash M \rightsquigarrow E_2 : \sigma_2$ and $C : (P_1 \mid \sigma_1) \sqsupseteq (P_2 \mid \sigma_2)$, then $C\,E_1 =_{\beta\eta} E_2$.*

The proof is routine, following standard approaches to showing coherence for qualified type systems [Jones 1993; Morris 2014].

4.6 Translations Between Row Theories

We have both a generic account of the interpretation of Rose$(\mathcal{R}, \sim, \Rightarrow)$ and a specific account of the interpretation of the trivial theory $\langle \mathcal{R}_{\mathsf{triv}}, \sim_{\mathsf{triv}}, \Rightarrow_{\mathsf{triv}} \rangle$ in $\mathrm{F}^{\otimes\oplus}$. Now, we extend our approach to give an account of the simple and scoped theories of rows, in a modular and reusable fashion.

4.6.1 Translations, generally. We might hope that row theory homomorphisms, the intuitive structure-preserving maps for row theories, would be sufficient to extend our semantics from trivial to simple or scoped rows. Unfortunately, the trivial row theory is not expressive enough to capture the other row theories (§3.3); this means that a translation from simple or scoped rows to trivial rows must induce some transformation on terms, not only on types. The problem is that there are equivalent rows in the source theory (such as $(\mathtt{x} \triangleright \mathtt{Int}, \mathtt{y} \triangleright \mathtt{Bool})$) that cannot be translated to a single row in the target theory (neither of the trivial rows $(\mathtt{Int}, \mathtt{Bool})$ nor $(\mathtt{Bool}, \mathtt{Int})$ satisfies all the predicates on the original simple row). However, there are *families* of rows in the target theory that do satisfy all of the predicates, and are all interconvertible! For example, if we are allowed to switch between the trivial rows $(\mathtt{Int}, \mathtt{Bool})$ and $(\mathtt{Bool}, \mathtt{Int})$ (in this case, how to do so is clear), then we can model all of the predicates on the original simple row.

Definition 16. Let $\langle \mathcal{R}, \sim, \Rightarrow \rangle$ and $\langle \mathcal{R}', \sim', \Rightarrow' \rangle$ be two theories of rows. Let $\eta : \mathcal{R} \to \mathcal{R}'$, and extend η homomorphically from rows to predicates, types, and contexts. We say η is a *row theory translation* (or simply: *translation*) from $\langle \mathcal{R}, \sim, \Rightarrow \rangle$ to $\langle \mathcal{R}', \sim', \Rightarrow' \rangle$ if:

(1) If $\zeta_1 \sim \zeta_2$, then there are Rose$(\mathcal{R}', \sim', \Rightarrow')$ terms T_1, T_2 such that $T_1 : \Pi\eta(\zeta_1) \to \Pi\eta(\zeta_2)$ and $T_2 : \Sigma\eta(\zeta_1) \to \Sigma\eta(\zeta_2)$.
(2) If $P \Rightarrow \zeta_1 \odot \zeta_2 \sim \zeta_3$, then there is some $\zeta_3' \sim \zeta_3$ such that $\eta(P) \Rightarrow' \eta(\zeta_1) \odot \eta(\zeta_2) \sim \eta(\zeta_3')$.

Such a translation on row theories gives rise to a corresponding translation on terms of Rose instantiated by those theories.

$$\frac{}{\langle \mathsf{id}, \mathsf{id} \rangle : \zeta \sim \zeta} \qquad \frac{\langle f_1, g_1 \rangle : \zeta_1 \sim \zeta_2 \quad \langle f_2, g_2 \rangle : \zeta_2 \sim \zeta_3}{\langle f_2 \circ f_1, g_2 \circ g_1 \rangle : \zeta_1 \sim \zeta_3} \qquad \frac{\langle f, g \rangle : \zeta_1 \sim \zeta_2}{\langle \mathsf{prj_L} \vartriangle f \circ \mathsf{prj_R}, \mathsf{inj_L} \triangledown \mathsf{inj_R} \circ g \rangle : \ell \triangleright \tau, \zeta_1 \sim \ell \triangleright \tau, \zeta_2}$$

$$\frac{\langle f, g \rangle : \zeta_1 \sim \zeta_2 \quad \ell \neq \ell'}{\langle (\mathsf{prj_R} \vartriangle \mathsf{prj_L}) \circ \mathsf{prj_L} \vartriangle f \circ \mathsf{prj_R}, \mathsf{inj_L} \circ (\mathsf{inj_R} \triangledown \mathsf{inj_L}) \triangledown \mathsf{inj_R} \circ g \rangle : \ell \triangleright \tau, \ell' \triangleright \tau' \zeta_1 \sim \ell' \triangleright \tau', \ell \triangleright \tau, \zeta_2}$$

Fig. 10. Equivalence of simple and scoped rows, with witnessing terms in the trivial row theory. The shaded condition holds trivially for simple rows.

Theorem 17. *If η is a row theory translation from $\langle \mathcal{R}, \sim, \Rightarrow \rangle$ to $\langle \mathcal{R}', \sim', \Rightarrow' \rangle$, then it extends to a translation $\eta^\bullet$ from terms of Rose$(\mathcal{R}, \sim, \Rightarrow)$ to terms of Rose$(\mathcal{R}', \sim', \Rightarrow')$ such that if $P \mid \Gamma \vdash M : \sigma$, then $\eta(P) \mid \eta(\Gamma) \vdash \eta^\bullet(M) : \eta(\sigma)$.*

The proof is by induction on the derivation of $P \mid \Gamma \vdash M : \sigma$. The translation introduces mediating terms, as are guaranteed by Definition 16 condition 1, at record and variant operations and applications of (sim), and is otherwise homomorphic.

4.6.2 Translations, specifically. To use Theorem 17 to give semantics for simple or scoped rows, we need to exhibit row theory translations from these theories to the trivial theory. We can do so in an entirely regular fashion. We map rows in the source theory to syntactically identical rows in the target theory; since the trivial theory is less expressive than either simple or scoped rows, however, we have some work to do to show we can preserve the operations from the source theories.

Proposition 18. *Let $\langle \mathcal{R}, \sim, \Rightarrow \rangle$ be the simple (resp. scoped) theory of rows. Then $\eta_{\mathsf{I}} : \mathcal{R} \to \mathcal{R}_{\mathsf{triv}}; \eta_{\mathsf{I}} : (\ell_{\mathtt{i}} \triangleright \tau_i)_i \mapsto (\ell_{\mathtt{i}} \triangleright \tau_i)_i$ is a translation from the simple (resp. scoped) row theory to the trivial row theory.*

The key step is showing that equivalences in the source theories can be witnessed in the trivial theory. Figure 10 reiterates the equivalence relation for both simple and scoped rows, extending it with witness information (recall that $\vartriangle$ is the dual operator to $\triangledown$). The side condition that $\ell \neq \ell'$ holds trivially for simple rows, as simple rows prohibit repetitions of labels within a row.

Lemma 19. *If $\langle f, g \rangle : \zeta_1 \sim \zeta_2$ (in either the simple or scoped row theory), then $\vdash f : \Pi \eta_{\mathsf{I}}(\zeta_1) \to \Pi \eta_{\mathsf{I}}(\zeta_2)$ and $\vdash g : \Sigma \eta_{\mathsf{I}}(\zeta_1) \to \Sigma \eta_{\mathsf{I}}(\zeta_2)$.*

Finally, we have to show that satisfiable predicates from the source theory can be mapped to satisfiable predicates in the trivial theory. This amounts to observing that, for any non-trivial derivation $P \Rightarrow \zeta_1 \odot \zeta_2 \sim \zeta_3$, we have that $\zeta_3 \sim \zeta_1 \frown \zeta_2$. This is a simple induction on the length of ζ_3.

4.6.3 Fusion and efficiency. The translations in the previous section demonstrate the utility of the row theory abstraction, while showing that, for the row theories we have considered, Rose programs can still be interpreted in System F. The resulting implementations may not be particularly efficient; in particular, the translations to the trivial row theory introduce regular, and apparently unnecessary, reshuffling of product and sum values. We observe that most of these reshuffling steps, however, could be combined by a simple analysis of the final $\mathrm{F}^{\otimes\oplus}$ terms. There are still significant questions, both about how to best represent extensible records and variants, and how to take advantage of contextual information (perhaps captured in typing) to optimize representation choices. We hope to return to these questions in future work.

5 RELATED WORK

There is a significant literature on extensible data types in general, and on row type systems in particular. Here we highlight that work which is closest to ours.

Establishing row types. Wand [1987] originally introduced row types, motivated by modeling inheritance in object-oriented languages. He initially proposed treating rows as arbitrary partial functions from labels to types; updates were always allowed, and overwrote fields if they were present. Wand also observed that the dual of his record calculus gave a system of extensible variants. Unfortunately, unification for this system proved problematic. Rémy [1989] introduced an alternate approach to row typing, in which rows mapped labels to *fields*, rather than types. Fields could either indicate that a label was present, at a given type, or that a field was absent. In this system, unlike Wand's, record extension was limited to cases in which fields were not already present. Rémy introduced a restriction operator, which "forgets" fields already present in a record, making it possible to define record update in terms of restriction followed by extension. However, he further proposed polymorphism in fields; by making record extension polymorphic in the new label's field, it could serve as both extension and update.

Modern row types. Pottier and Rémy [2005] give an updated presention of Rémy's original row system. Among the innovations of their presentation is that they lift arbitrary type operators to row operators; for example, if ζ_1 and ζ_2 are rows, then they also support the row $\zeta_1 \to \zeta_2$, in which each label ℓ is mapped to $\zeta_1(\ell) \to \zeta_2(\ell)$. This allows them to express some programs inexpressible in our system, such as one that applies a record of functions to a record of arguments. Unlike the other systems we have discussed, they restrict record and variant types to *complete* rows, in which all labels are mapped to some type. They still, however, limit record extension to avoid overwriting fields, and so support incomplete rows in the building of complete rows. Blume et al. [2006] describe a calculus that supports extensible records and first-class cases. Unlike other approaches to extensible variants, they distinguish case-branch functions from arbitrary functions, and restrict their branching operator to combining case-branch function. This allows them to give a more efficient implementation of extensible variants, by translating case blocks into (extensible) records of functions.

Rows and qualified types. The systems of Rémy [1989] and Pottier and Rémy [2005] use kinds to track the labels present in (or absent from) a given record, to assure that record types themselves are well formed. Gaster and Jones [1996] proposed using qualified types instead. They introduce a predicate $\zeta \setminus \ell$, pronounced "ζ lacks ℓ", which is only satisfied if row ζ does not include label ℓ. Otherwise, their operations are similar to other systems of extensible data types: they include record extension (limited to cases where the new field is not already present), restriction, and the dual operations for variants. The use of "lacks" constraints also guides the implementation of records: the presence of a constraint $\zeta \setminus \ell$ indicates that at some point, ζ will be extended with label ℓ. The evidence for such a constraint is the offset into a $\Pi\zeta$ record at which field ℓ should be stored.

Rows and concatenation. There are several existing accounts of rows and record concatenation. Wand [1991] first discussed the typing problems that arise with record concatenation. He described an approach to typing concatenation that used row types as in Rémy [1989], but concluded multiple typing derivations for each possible term. (These are essentially intersection types, although he did not express them as intersections.) While expressive, we believe that this approach introduces obstacles to understanding types and the corresponding terms. Harper and Pierce [1991] give an extension of System F with record concatenation, in which record concatenation does not replace fields. They assure this by introducing a form of constrained type abstraction $\forall a\#\vec{\zeta}.A$, which requires that instantiations of type variable a be with record types disjoint from any of those in $\vec{\zeta}$. Otherwise, they support the standard record selection, restriction, and construction operations. (In particular, while their concatenation operation behaves identically to ours, their primitives project only single fields, not subrecords.) However, their system cannot express Wand's problem: while it

can express the concatenation, it lacks a mechanism to require that a field is in the result of the concatenation without specifying in which record it originates.

Featherweight Ur [Chlipala 2010] incorporates record concatenation and row typing in a dependently-typed programming language; in addition to standard row types, Featherweight Ur includes types witnessing disjointness of rows and type-level maps on rows. Wand's problem can be expressed in Featherweight Ur (in contrast to our approach, the programmer must provide an explicit witness of row disjointness, although they can do so in an entirely routine fashion). We do not claim that Rose provides more expressive row types than Featherweight Ur. Rather, we view our work as demonstrating that expressive record concatenation can be expressed in type systems with only a fraction of the overall expressiveness (and corresponding complexity) of systems like Ur.

Encoding extensible data types. There have been a number of systems encoding extensible data types using other type system features, notably type classes and type families in Haskell. Kiselyov et al. [2004] demonstrate how to encode extensible records in Haskell; their approach initially corresponds to one with *unlabeled* rows, in which projection is based solely on the types of record fields, but they also give an encoding of labeled fields. Predictably, they rely on a number of extensions to the Haskell class system, most prominently functional dependencies. Swierstra [2008] develops an encoding of extensible variants in Haskell, using two-level types to capture extensible recursive types. While his injection functions are similar to ours, he does not provide a general approach to branching, instead expressing each branching with its own type class. Bahr [2014] and Morris [2015] give independent encodings of extensible variants, with similar but subtly different expressions of branching. Both systems are essentially equivalent to unlabeled simple rows (Morris also presented a system like scoped rows, but, apparently unaware of the work on scoped rows, discarded it as undesirable). Oliveira et al. [2015] give an encoding of extensible data types using type-level lists of types, an approach quite close to rows. However, in doing so, they lose precision in their typing of case blocks.

Other applications of rows. We have considered rows solely in the context of extensible data types. Row types have been used to capture similar ideas of extensibility in several other contexts. Lindley and Cheney [2012] use row types to provide effect polymorphism, while Hillerström and Lindley [2016]; Leijen [2014]; Lindley et al. [2017] use row types to provide accounts of algebraic effects and handlers. Lindley and Cheney [2012] use a row type system based on that of Rémy; Hillerström and Lindley [2016] extend this approach to also encompass handlers for algebraic effects. Leijen uses scoped rows to account for algebraic effects and handlers; the use of scoping in the rows corresponds to the nesting of handlers in source code, where there may be, for example, multiple nested catch blocks all handling (different levels or types of) exceptions. Lindley and Morris [2017] use row types to account for extensible choice in session types, a type-based abstraction of communication protocols.

Makholm and Wells [2005] introduce a row-based approach to typing mixin modules and their composition, called Martini. In this approach, modules are typed by pairs of rows, one that captures the definitions they provide and another that captures their dependencies on their environment. Composition of modules is only allowed when the two modules have disjoint output sets, and the modules may (mutually) satisfy each other's dependencies. Martini can be captured by Rose, by observing that its module types (that is, its pairs of rows) themselves form a partial monoid. Following this characterization, Martini's various module projection and restriction operations, and their types, can be encoded using the record operations of Rose. This application also highlights Rose's unique benefit of capturing multiple row systems within a single language: a practical

instantiation of Rose might use simple rows to capture extensible data types, while using Martini like rows to capture module structure.

Subsumption and intersection types. Records and variants, in general, have some similarities to intersection and union types, particularly in light of type systems that allow arbitrary or almost-arbitrary intersections [Dunfield 2012; Oliveira et al. 2016]. There are two important differences between our approach and those based on intersections. First, we consider a view parametric over the interpretation of rows, and thus the meaning of records and variants; in contrast, work on intersections and unions focuses on a particular interpretation of intersections and unions. Second, and more importantly, our approach to projection (and injection) is not equivalent to subtyping. This is demonstrated in our discussion of default values (§2.2), where the presence of projection only on the right of the concatenation is essential to the meaning of the term.

6 FUTURE WORK: LENSES

Not for nothing have we presented $\star$ and $\triangledown$ for Π-rows (records, §2.2) and Σ-rows (variants, §2.3) as offering an *abstract* presentation of product and coproduct types, subject to the ability to project/inject at will along a chosen coordinate axis given by a field label/constructor, or more generally along any (sub-)row type. Elsewhere, the rich literature on (asymmetric, state-based) *lenses* [Foster et al. 2007, for example], has similarly given a "coordinate-free" account, at least for product types. An (asymmetric) lens (get, put) from source type S to view type V consists of (partial) functions $get : S \to V$, constructing a view from a given source value, and $put : V \times S \to S$, responsible for updating a source value with changes made to the view. Without further belaboring the details, a very-well-behaved asymmetric lens from S to V exhibits an isomorphism between S and $V \times C$, for some type C, called the "constant complement", which stores the information in S not visible in the view V [Hofmann et al. 2011]. In this light, we can see that the record operations in Rose exactly define a set of very-well-behaved lenses. For any rows $\zeta_1 \odot \zeta_2 \sim \zeta_3$, we have that $(\lambda z.\mathrm{prj}\ z, \lambda(x, z).x \star \mathrm{prj}\ z)$ is a lens from $\Pi\zeta_3$ to $\Pi\zeta_1$, with constant complement $\Pi\zeta_2$ (or, by different instantiations of the type variables, from $\Pi\zeta_3$ to $\Pi\zeta_2$ with constant complement $\Pi\zeta_1$). This is true regardless of the theory of rows in which the predicates are interpreted.

Dual to lenses are *prisms* [Kmett 2018]: a prism from S to V consists of functions $(match, build)$ where $match : S \to \texttt{Maybe}\ V$ performs a case match, returning `Nothing` if the S value does not match the pattern captured by the prism, and $build : V \to S$ constructs an S value matching the prism from V. As with lenses, we see that the variant operations in Rose define prisms. For any rows $\zeta_1 \odot \zeta_2 \sim \zeta_3$, we have that $((\lambda y.y/\texttt{Just}) \triangledown (\lambda y.\texttt{Nothing}), \lambda y.\mathrm{inj}\ y)$ is a prism from $\Sigma\zeta_3$ to $\Sigma\zeta_1$. However, prisms discard a crucial piece of information captured in Rose: in the case that *match* returns `Nothing`, we have lost any knowledge of the remaining cases. That is to say, the idea of the constant complement, essential to characterizing the behavior of very-well-behaved lenses, which is captured implicitly by Rose constraints and used to check branching, is missing from prisms.

Just as McKinna [2016] draws attention to the *logical* meaning of lens complement, we exploit Curry-Howard to give *computational* content to logical constraints on rows. In future work, we hope to further explore the connections between lenses and prisms and our type-theoretic, monoidal view of extensible data types. In particular, we plan to compare our work to the profunctor-based generalization of lenses and prisms given by Pickering et al. [2017] and Boisseau and Gibbons [2018].

7 FUTURE WORK: EFFECTS AND EFFECT HANDLERS

We have considered applications of row types to extensible records and variants. Row types have also recently been applied to type systems for (algebraic) effects and effect handlers [Hillerström

and Lindley 2016; Leijen 2014; Lindley et al. 2017]. As future work, we plan to study how our view of rows and row theories can be applied to effect typing. Our goal is to leverage the expressiveness of higher-order functional programming to express effects and handlers in terms of the existing primitives (records and variants), rather than endlessly extending our language with new primitive types and new primitive operations. Different views of effects and handlers imply different accounts of recursive data types, and so take advantage of Rose's ability to capture multiple row systems in the same language.

To do so, we expect to develop several features not present in Rose. Our approach exposes the duality between records and variants in the type system but does not reflect it directly in terms. A general account of effectful computations (essentially, variants over the possible operations) and effect handlers (essentially, records of implementations of operations) will require terms that witness the duality between record construction and variant branching. (A similar approach is used by Blume et al. [2006], but in their implementation of variants rather than being materialized in the language.) Many existing accounts of extensible effects also depend on transformations on the effect labels in a term, such as the row-level maps introduced by Pottier and Rémy [2005] and Chlipala [2010]. We intend a fuller study of first-class labels and of how label transformations and their properties can be characterized, consistent with our separation of labeling from the core record and variant operations

ACKNOWLEDGEMENTS

We are grateful to the reviewers for their help and encouragement in improving the presentation of this work. The second author gratefully acknowledges the support of LFCS, University of Edinburgh, and the U.S. Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number FA8655-13-1-3006 (PI: James Cheney). The U.S. Government, the University of Edinburgh, and the University of Kansas are authorised to reproduce and distribute reprints for their purposes notwithstanding any copyright notation thereon.

REFERENCES

P. Bahr. Composing and decomposing data types: a closed type families implementation of data types à la carte. In J. P. Magalhães and T. Rompf, editors, *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*, pages 71–82. ACM, 2014.

B. Berthomieu and C. le Moniès de Sagazan. A calculus of tagged types, with applications to process languages. In *Workshop on types for program analysis*, Aarhus, 1995.

M. Blume, U. A. Acar, and W. Chae. Extensible programming with first-class cases. In J. H. Reppy and J. L. Lawall, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, pages 239–250. ACM, 2006.

G. Boisseau and J. Gibbons. What you needa know about Yoneda: profunctor optics and the Yoneda Lemma (Functional Pearl). *PACMPL*, 2(ICFP):84:1–84:27, 2018.

A. Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In B. G. Zorn and A. 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.

J. Dunfield. Elaborating intersection and union types. In P. Thiemann and R. B. Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 17–28. ACM, 2012.

J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17, 2007.

J. Garrigue. Programming with polymorphic variants. In *ML Workshop*. ACM, 1998.

B. R. Gaster and M. P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, University of Nottingham, 1996.

R. Harper and B. Pierce. A record calculus based on symmetric concatenation. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 131–142. ACM, 1991.

D. Hillerström and S. Lindley. Liberating effects with rows and handlers. In *TyDe@ICFP*, pages 15–27. ACM, 2016.

R. Hinze, N. Wu, and J. Gibbons. Conjugate hylomorphisms - or: The mother of all structured recursion schemes. In S. K. Rajamani and D. Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 527–538. ACM, 2015.

M. Hofmann, B. C. Pierce, and D. Wagner. Symmetric lenses. In T. Ball and M. Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 371–384. ACM, 2011.

M. P. Jones. Coherence for qualified types. Technical Report YALEU/DCS/RR-989, Yale University, 1993.

M. P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.

M. P. Jones. Type classes with functional dependencies. In G. Smolka, editor, *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244. Springer, 2000.

O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In H. Nilsson, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004, Snowbird, UT, USA, September 22-22, 2004*, pages 96–107. ACM, 2004.

E. Kmett. The `lens` package, version 4.16, 2018.

O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4): 707–723, 1998.

D. 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.

D. Leijen. Koka: Programming with row polymorphic effect types. In P. Levy and N. Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014.*, volume 153 of *EPTCS*, pages 100–126, 2014.

D. Leijen. Type directed compilation of row-typed algebraic effects. In G. Castagna and A. D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 486–499. ACM, 2017.

S. Liang, P. Hudak, and M. P. Jones. Monad transformers and modular interpreters. In R. K. Cytron and P. Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 333–343. ACM Press, 1995.

S. Lindley and J. Cheney. Row-based effect types for database integration. In B. C. Pierce, editor, *Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, Saturday, January 28, 2012*, pages 91–102. ACM, 2012.

S. Lindley and J. G. Morris. Lightweight functional session types. In S. Gay and A. Ravara, editors, *Behavioural Types: from Theory to Tools*. River Publishers, 2017.

S. Lindley, C. McBride, and C. McLaughlin. Do be do be do. In G. Castagna and A. D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 500–514. ACM, 2017.

H. Makholm and J. B. Wells. Type inference, principal typings, and let-polymorphism for first-class mixin modules. In O. Danvy and B. C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 156–167. ACM, 2005.

J. McKinna. Complements witness consistency. In *Bx@ETAPS*, volume 1571 of *CEUR Workshop Proceedings*, pages 90–94. CEUR-WS.org, 2016.

J. C. Mitchell. Polymorphic type inference and containment. *Inf. Comput.*, 76(2-3):211–249, Feb. 1988.

J. G. Morris. A simple semantics for Haskell overloading. In W. Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 107–118. ACM, 2014.

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

B. Oliveira, S. Mu, and S. You. Modular reifiable matching: a list-of-functors approach to two-level types. In *Haskell*, pages 82–93. ACM, 2015.

B. Oliveira, Z. Shi, and J. Alpuim. Disjoint intersection types. In J. Garrigue, G. Keller, and E. Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 364–377. ACM, 2016.

M. Pickering, J. Gibbons, and N. Wu. Profunctor Optics: Modular Data Accessors. *The Art, Science, and Engineering of Programming*, 1(2), 2017. Article 7.

F. Pottier and D. Rémy. The essence of ML type inference. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. The MIT Press, 2005.

D. 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.
T. Sheard and E. Pasalic. Two-level types and parameterized modules. *J. Funct. Program.*, 14(5):547–587, 2004.
W. Swierstra. Data types à la carte. *J. Funct. Program.*, 18(04):423–436, 2008.
P. Wadler. The expression problem. http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt, 1998.
M. 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.
M. Wand. Type inference for record concatenation and multiple inheritance. *Inf. Comput.*, 93(1):1–15, 1991.