Infix-Extensible Record Types for Tabular Data

Adam Paszke
apaszke@google.com
Google DeepMind
Berlin, Germany

Ningning Xie
ningningxie@google.com
Google DeepMind
Toronto, Canada

Abstract

We present a novel row-polymorphic record calculus, supporting a unique combination of features: scoped labels, first-class labels and rows, and record concatenation. Our work is motivated by the similarity of record types and data table (or data frame) schemas, commonly used in data processing tasks. After presenting our record calculus, we demonstrate its applicability to data frame manipulation by showing that it can be used to successfully assign types to the functions listed in the Brown Benchmark for Tabular Types. Our typing discipline is remarkably lightweight, compared to calculi that require reasoning about type-level constraints when manipulating record types, making it a viable candidate for practical use.

CCS Concepts: • **Software and its engineering → Data types and structures**; *Formal language definitions*; *Domain specific languages.*

Keywords: data frame, table, schema, row polymorphism

ACM Reference Format:
Adam Paszke and Ningning Xie. 2023. Infix-Extensible Record Types for Tabular Data. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe '23), September 4, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3609027.3609406

1 Introduction

Quantitative research has taken over most scientific fields and business analytics, making computerized analysis of tabular data a commonplace task. Most of the programs used to perform data analysis present a graphical user interface, making them very accessible to a wide audience. Yet, the convenience they offer often comes with constraints. And, with the growing accessibility of programming education, advanced users often turn to popular (and usually dynamically typed) programming languages (R, Python, Julia) to perform their more advanced data processing.

TyDe '23, September 4, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0299-0/23/09.
https://doi.org/10.1145/3609027.3609406

With greater power comes greater responsibility. Mistakes in data analysis can be costly, and the flexibility of programmable tools also makes it easier to make mistakes, especially for non-expert programmers. The risks are only inflated by many libraries inheriting arguably unintuitive practices, such as the implicit use of ternary logic to model missing data that is so prevalent, e.g., in SQL. To increase their trustworthiness, data science workloads can be extended with forms of formal verification such as type systems. Still, we believe that practical verification tools should remain lightweight, to make sure that the added safety and clarity are not outweighed by lesser accessibility caused by the need to appease a picky verifier or compiler.

In this work we present one potential component of such lightweight verification. We build on the well-studied type theory of row-polymorphic record types. Record types are a natural way to model the *schema* of a heterogeneous data table, that is, the set of (named) columns annotated with a homogeneous type for every column. Row polymorphism keeps typed programs from being overly sensitive to table schema changes, such that the verification is robust against, e.g., the addition of new data sources as new columns.

While seemingly a good fit, most work focusing on row-polymorphic records assumes a fairly advanced type system, such as one supporting qualified type theories. In turn, most functions dealing with records require a fairly ceremonious typing discipline. And, even if the constraints can be inferred by the compiler, we argue that something of value is lost: Types increase correctness, yes, but perhaps even more important, they provide a high-level view of the data transformations performed by a function. The long rows of qualifications necessary for a function type to be well-defined in a conventional row-polymorphic type system create unnecessary syntactic noise and hamper understanding.

As a solution to this problem, we propose a novel row-polymorphic record calculus with a unique combination of features: (1) *scoped labels* [10], where unlike in many row type systems, duplicate labels are allowed and retained in our system; (2) *first-class labels* [9], where labels are first-class values that can be passed in and returned by functions; (3) an extension to *first-class rows*, where a row can be considered a group of labels; and (4) *record concatenation* [5], where two records can be merged into a single record. Except for first-class rows, some of the individual features have been studied in existing literature, but not in ways compatible with each other. For example, Harper and Pierce [5] can concatenate

two records only if they do not share common fields, as labels cannot be duplicated; while Leijen [10] alone cannot express polymorphic record concatenation. We discuss related work in more detail in §5. As we will see (§4), with this set of features we can express table types fairly well.

The rest of the paper is structured as follows:

- We begin our presentation in §2, by informally demonstrating a sequence of record calculus extensions that leads to our proposed system.
- In §3 we present $\lambda^{\langle\rangle}$, a novel row-polymorphic record calculus. We present the static and the operational semantics, and prove that $\lambda^{\langle\rangle}$ is type-safe. Moreover, we discuss type inference and present a sound unification algorithm.
- In §4 we evaluate our proposal on the Brown Benchmark for Tabular Types (B2T2) [11] to better display the low-noise characteristics and satisfactory expressiveness of our type system.
- We discuss related work in §5 and conclude in §6.

2 Multi-Tailed Records, Informally

Product types are a standard tool in modern programming languages, allowing convenient grouping of related values in a single entity. Perhaps the simplest and most common version is the tuple type, where product fields (and their respective projections) are indexed by integers. But, when tuples grow large or long-lived, it becomes difficult to keep track of the positional indices. To resolve that problem many languages allow product types with *named fields* (`structs` in C(++), `namedtuple` in Python, ...), allowing for much cleaner and less error-prone access to fields. Products with named fields are exactly what we refer to as *records*.

For example, imagine a program for video manipulation. It is important to be able to store video metadata such as the frame resolution and the number of frames. It could be stored in a tuple of type $(\mathsf{Int}, \mathsf{Int}, \mathsf{Int})$, but its usage would be error-prone, due to "integer blindness" (does height come before width or after?). We can model the data as a record type instead, with clear semantics:

$$\{\mathsf{length} : \mathsf{Int}, \mathsf{height} : \mathsf{Int}, \mathsf{width} : \mathsf{Int}\}.$$

To construct a record value, one replaces field types for concrete expressions. Projections can then be performed by supplying a field name to the $(.)$ operator:

$$\{\mathsf{length} = 240, \mathsf{height} = 480, \mathsf{width} = 640\}.\mathsf{height} == 480$$

A record type can be constructed from a sequence of labeled types called a *row*. Rows, in turn, are constructed inductively, starting from an empty row $\langle\rangle$, that can be left-extended by labeled fields $\langle l : \tau \mid \langle \ldots \rangle\rangle$, where we write τ for types. For clarity of presentation, we flatten the repeated extensions and abbreviate $\langle l_1 : \tau_1 \mid \langle l_2 : \tau_2 \mid \langle\rangle\rangle\rangle$ as $\langle l_1 : \tau_1, l_2 : \tau_2\rangle$ (and similarly for record types). We use angle brackets $\langle\cdot\rangle$ to notate row types (which have kind Row) and curly braces $\{\cdot\}$ to notate record types (which have kind $\star$).

In many record calculi, it is common to assume that fields are unordered, and we make that assumption as well. That means, that we treat the rows (and record types constructed from them) $\langle l_1 : \mathsf{Int}, l_2 : \mathsf{String}\rangle$ and $\langle l_2 : \mathsf{String}, l_1 : \mathsf{Int}\rangle$ as equivalent.

2.1 Tail-Extensible Records

A very common extension to the row calculus that our work builds upon, is to allow arbitrary row extensions in tail positions [4]. It extends the row language to allow type variables with kind Row, including in the tail position of the row constructor $\langle\cdot \mid \cdot\rangle$, as in $\langle l : \tau \mid \rho\rangle$. Extensible rows are interesting, because they make it possible to simulate structural subtyping of rows. For example, one could write a function computing the area of a rectangular object and have it apply to records with arbitrary additional fields.

$$\begin{aligned}&\mathsf{area} :: \forall(\rho :: \mathsf{Row}).\ \{\mathsf{height} : \mathsf{Int}, \mathsf{width} : \mathsf{Int} \mid \rho\} \to \mathsf{Int}\\ &\mathsf{area}\ x = x.\mathsf{height} * x.\mathsf{width}\end{aligned}$$

$$\begin{aligned}&\mathsf{area}\ \langle \mathsf{height} = 10, \mathsf{width} = 20, \mathsf{color} = \mathsf{Red}\rangle :: \mathsf{Int}\\ &\mathsf{area}\ \langle \mathsf{height} = 10, \mathsf{width} = 20, \mathsf{texture} = \mathsf{Smooth}\rangle :: \mathsf{Int}\end{aligned}$$

2.2 Scoped Labels

Since we said records are unordered, what happens if the same label appears in a record more than once? Most published and used record type systems (such as in [4, 5]) forbid this. Unfortunately, they must then turn to heavy-weight machinery like *qualified types* [7] to enforce label uniqueness in the face of polymorphism.

We address this by adopting *scoped labels* [10]. Instead of forbidding duplicate labels, we give them a semantics: When a label is repeated, the first field *shadows* all the other fields with the same label. For example:

$$\{l = 2, l = \text{“}\mathsf{asdf}\text{”}\}.l :: \mathsf{Int}$$

To retain access to the second field, we also adopt a record restriction operation, which removes a specified field from a record:

$$\{l_1 = 1, l_2 = 2\} \backslash l_1 == \{l_2 = 2\}.$$

Restriction respects shadowing, so we can access the second field with a shared name by first restricting away the first:

$$(\{l = 2, l = \text{“}\mathsf{asdf}\text{”}\} \backslash l).l :: \mathsf{String}$$

It is important to note that while we still treat rows that permute fields with *distinct* labels as equivalent, permutations of fields with the *same* label is not allowed, because we are now relying on their order to disambiguate them.

2.3 First-Class Labels

We also add first-class labels [9] to our calculus. To do so, we add another kind Label and a type constructor $(\!|\cdot|\!) :: \mathsf{Label} \to \star$ to our system. The only way to construct a value of type $(\!|\mathsf{height}|\!)$ is through the *label literal* expression:

$$\mathsf{height} :: (\!|\mathsf{height}|\!).$$

Thanks to first-class labels, the (.) operator (as well as record restriction and extension) can be assigned a type:

$$(.) :: \forall(l : \text{Label})(\alpha : \star)(\rho : \text{Row}).\ \{l : \alpha \mid \rho\} \to (\!|l|\!) \to \alpha.$$

First-class labels interact with scoped labels in an interesting and, to the best of our knowledge, previously undescribed way. Consider the following expression:

$$f :: \forall(l : \text{Label}).\ (\!|l|\!) \to \{l : \text{String}, \text{foo} : \text{Int}\} \to ?$$
$$f\ l\ x = x.\text{foo} \qquad \text{// rejected}$$

Since labels are no longer guaranteed to be unique, nothing prevents the label variable l from being instantiated as foo later in the program. As such, there is no way to statically decide upon the return type of function f and our calculus rejects the program. Access to the integer field is still possible, but it requires explicitly removing the field l:

$$f :: \forall(l : \text{Label}).\ (\!|l|\!) \to \{l : \text{String}, \text{foo} : \text{Int}\} \to \text{Int}$$
$$f\ l\ x = (x \backslash l).\text{foo}$$

Note that the above issue would not arise if the record had type $\{\text{foo} : \text{Int}, l : \text{String}\}$, as no instantiation of l could shadow the literal field foo. For this reason, we restrict the row-equivalence relation such that a field labeled by a variable can never be swapped with another label (either constant or variable). This means that two initially unequal types can become equal after instantiation of label variables.[1] In contrast, [9] does not support scoped labels, and the type of f must be rewritten with a *lacks* predicate ensuring that l cannot be instantiated with foo.

In our system, we take the idea of first-class labels further and support *first-class rows*, written as $\langle r \rangle$, where $r : \text{Row}$. Essentially, rows can be considered as a group of labels. First-class rows can be passed in or returned by functions. As a contrived example, we can define a specialization g of the field access operator

$$g ::\forall(l : \text{Label})(\alpha : \star)(\rho : \text{Row}).$$
$$\langle \rho \rangle \to \{l : \alpha \mid \rho\} \to (\!|l|\!) \to \alpha$$

Then g applied to a row ρ can be used to project out the l label only for records with the given common postfix ρ.

First-class rows are more interesting when the system is extended with record concatenation, which we explain next.

2.4 Infix-Extensible Records

A final piece of our record calculus is *record concatenation* [5, 16, 20], in the form of a row concatenation operator:

$$\langle \cdot \mid \cdot \rangle :: \forall(r_1 : \text{Row})(r_2 : \text{Row}).\ \langle r_1 \rangle \to \langle r_2 \rangle \to \langle r_1 \mid r_2 \rangle$$

In this calculus, rows can be extended anywhere, i.e. they are now infix-extensible and not just tail-extensible, making

$$\{r_1 \mid l : \text{Int} \mid r_2 \mid \text{foo} : \text{String}\}$$

a valid record type, with row variables r_1 and r_2, a label variable l, and a label literal foo.

As before, types that differ by an exchange of a row variable with a neighboring field or row are also considered non-equivalent, since shadowing of fields can depend on variable instantiation.

To complement field-level operations, we extend our language with a row projection ($_\bullet$) operator:

$$(_\bullet) :: \forall(r_1 : \text{Row})(r_2 : \text{Row}).\ \{r_1 \mid r_2\} \to \langle r_1 \rangle \to \{r_1\}$$

Similarly, we can provide a row restriction operator, which subsumes the previously described field restriction.

As we demonstrate in §4, the flexibility to extend and break up record rows in multiple places is crucial to ensure our calculus can type important table manipulation functions such as joins and aggregations. Since variable rows and labels cannot be exchanged, in polymorphic contexts the record types behave more akin to tuples of records, with points of concatenation delineating the tuple components. However, once the record types are fully instantiated (e.g. at the top level of the program), field reordering becomes possible again (up to shadowing preservation). This places additional burden on the implementers of polymorphic table manipulation functions, but the *users* of those functions, which we expect are a much larger group, can usually remain unaware of the restrictions.

3 A Calculus with Infix-extensible Records

In this section we formally describe our proposed row polymorphic calculus $\lambda^{\langle\rangle}$.

3.1 Types and Contexts

The syntax of kinds, types, and contexts is defined in Fig. 1.

Kinds distinguish types. A kind κ is either a type kind $\star$, a function $\kappa_1 \to \kappa_2$, a label kind Label, or a row kind Row.

A polymorphic type σ is a list of universal quantifiers followed by a monotype τ, where each quantified type variable a is annotated with its kind κ. Monotypes τ include type variables a, base types[2], functions $\to$, applications $\tau_1\ \tau_2$, records $\{\rho\}$, rows $\langle\rho\rangle$, and labels $(\!|\ell|\!)$.

Row types ρ include type variables a, empty rows (Empty), singleton rows $(\ell : \tau)$ and record concatenation $(\rho_1 \mid \rho_2)$. Label types ℓ include type variables a and label literals l_c.

A context Γ maps each term variable to its type, and each type variable to its kind. Lastly, to avoid presentation clutter, we use syntactic sugar outlined in Fig. 1. That is, we never explicitly write Empty, and we often write commas (,) for a series of row concatenations.

The kinding rules are presented in Fig. 2. We omit a detailed exposition, as most rules are entirely standard.

[1] This should not be surprising for any polymorphic types. For example, while in $\forall a\ b.\ a \to b \to a$, a and b are considered different, it is possible that a and b are later instantiated with the same type and are then equal.

[2] We present the formal syntax with Int being the only base type, but we use other common types such as Bool or String in examples throughout the section.

$$
\begin{array}{rrcl}
\text{kind} & \kappa & ::= & \star \mid \kappa_1 \to \kappa_2 \mid \mathsf{Label} \mid \mathsf{Row} \\
\text{poly type} & \sigma & ::= & \forall a : \kappa.\ \sigma \mid \tau \\
\text{type} & \tau & ::= & a \mid \mathsf{Int} \mid \to \mid \tau_1\ \tau_2 \mid \{\rho\} \mid \langle \rho \rangle \mid (\!|\ell|\!) \\
\text{row} & \rho & ::= & a \mid \mathsf{Empty} \mid \ell : \tau \mid (\rho_1 \mid \rho_2) \\
\text{label} & \ell & ::= & a \mid l_c \\
\text{context} & \Gamma & ::= & \bullet \mid \Gamma, a : \kappa \mid \Gamma, x : \tau \\
\multicolumn{4}{c}{\text{syntactic sugar}} \\
\multicolumn{4}{c}{\{\} \triangleq \{\mathsf{Empty}\} \qquad \langle\rangle \triangleq \langle \mathsf{Empty} \rangle} \\
\multicolumn{4}{c}{\ell_1 : \tau_1, \rho \triangleq \ell_1 : \tau_1 \mid \rho \qquad \ell_1 : \tau_1, \ldots, \ell_n : \tau_n \triangleq \ell_1 : \tau_1 \mid (\ldots \mid (\ell_n : \tau_n))}
\end{array}
$$

Figure 1. Syntax of types and contexts

$$
\frac{a : \kappa \in \Gamma}{\Gamma \vdash a : \kappa} \qquad \frac{}{\Gamma \vdash \mathsf{Int} : \star} \qquad \frac{}{\Gamma \vdash \to : \star \to \star \to \star}
$$

$$
\frac{\Gamma \vdash \tau_1 : \kappa_1 \to \kappa_2 \qquad \Gamma \vdash \tau_2 : \kappa_1}{\Gamma \vdash \tau_1\ \tau_2 : \kappa_2} \qquad \frac{\Gamma \vdash \rho : \mathsf{Row}}{\Gamma \vdash \{\rho\} : \star}
$$

$$
\frac{\Gamma \vdash \rho : \mathsf{Row}}{\Gamma \vdash \langle \rho \rangle : \star} \qquad \frac{\Gamma \vdash \ell : \mathsf{Label}}{\Gamma \vdash (\!|\ell|\!) : \star} \qquad \frac{}{\Gamma \vdash l_c : \mathsf{Label}}
$$

$$
\frac{}{\Gamma \vdash \mathsf{Empty} : \mathsf{Row}} \qquad \frac{\Gamma \vdash \rho_1 : \mathsf{Row} \qquad \Gamma \vdash \rho_2 : \mathsf{Row}}{\Gamma \vdash (\rho_1 \mid \rho_2) : \mathsf{Row}}
$$

$$
\frac{\Gamma \vdash \ell : \mathsf{Label} \qquad \Gamma \vdash \tau : \star}{\Gamma \vdash (\ell : \tau) : \mathsf{Row}}
$$

Figure 2. Kinding

$$
\frac{}{\rho \approx \rho}\ \textsc{refl} \qquad \frac{\rho_1 \approx \rho_2}{\rho_2 \approx \rho_1}\ \textsc{symm} \qquad \frac{\rho_1 \approx \rho_2 \qquad \rho_2 \approx \rho_3}{\rho_1 \approx \rho_3}\ \textsc{trans}
$$

$$
\frac{\tau_1 \approx \tau_2}{l_c : \tau_1 \approx l_c : \tau_2}\ \textsc{singleton} \qquad \frac{\rho_1 \approx \rho_2 \qquad \rho_1' \approx \rho_2'}{\rho_1 \mid \rho_1' \approx \rho_2 \mid \rho_2'}\ \textsc{concat}
$$

$$
\frac{l_{c_1} \neq l_{c_2}}{l_{c_1} : \tau_1 \mid l_{c_2} : \tau_2 \approx l_{c_2} : \tau_2 \mid l_{c_1} : \tau_1}\ \textsc{comm}
$$

$$
\frac{}{(\rho_1 \mid \rho_2) \mid \rho_3 \approx \rho_1 \mid (\rho_2 \mid \rho_3)}\ \textsc{assoc}
$$

$$
\frac{}{(\ell : \tau \mid \mathsf{Empty}) \approx \ell : \tau}\ \textsc{empR}
$$

$$
\frac{}{(\mathsf{Empty} \mid \ell : \tau) \approx \ell : \tau}\ \textsc{empL}
$$

Figure 3. Row equivalence

$$
\begin{array}{rrcl}
\text{expr} & e & ::= & x \mid \lambda x.\ e \mid e_1\ e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \\
& & \mid & \{\} \mid \{e_1 = e_2\} \mid \{e_1 \mid e_2\} \mid l_c \mid e_1.e_2 \\
& & \mid & \langle\rangle \mid \langle e \rangle \mid \langle e_1 \mid e_2 \rangle \mid e_1 \bullet e_2 \mid e_1 \backslash e_2 \\
\multicolumn{4}{c}{\text{syntactic sugar}} \\
\multicolumn{4}{c}{e \backslash l_c \triangleq e \backslash \langle l_c \rangle \qquad \langle l_c, e \rangle \triangleq \langle \langle l_c \rangle \mid e \rangle} \\
\multicolumn{4}{c}{\langle l_{c_1} \ldots, l_{c_n} \rangle \triangleq \langle \langle l_{c_1} \rangle \mid \langle \ldots \mid \langle l_{c_n} \rangle \rangle \rangle} \\
\multicolumn{4}{c}{\{e_1 = e_1' \ldots, e_n = e_n'\} \triangleq \{\{e_1 = e_1'\} \mid \{\ldots \mid \{e_n = e_n'\}\}\}}
\end{array}
$$

Figure 4. Syntax of expressions

$$
\frac{x : \sigma \in \Gamma \qquad \Gamma \vdash \sigma \sqsubseteq \tau}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \equiv \tau_2}{\Gamma \vdash e : \tau_2}\ \textsc{Eq}
$$

$$
\frac{\Gamma \vdash \tau_1 : \star \qquad \Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}\ \textsc{Lam} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \textsc{App}
$$

$$
\frac{\overline{\alpha_i} \notin \mathsf{ftv}(\Gamma) \qquad \begin{array}{c}\Gamma \vdash e_1 : \tau_1 \\ \Gamma, x : \forall \overline{\alpha_i : \kappa_i}.\ \tau_1 \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}\ \textsc{Let}
$$

$$
\frac{}{\Gamma \vdash \{\} : \{\}}\ \textsc{Empty} \qquad \frac{\Gamma \vdash e_1 : (\!|\ell|\!) \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \{e_1 = e_2\} : \{\ell : \tau\}}\ \textsc{Rcd}
$$

$$
\frac{\Gamma \vdash e_1 : \{\rho_1\} \qquad \Gamma \vdash e_2 : \{\rho_2\}}{\Gamma \vdash \{e_1 \mid e_2\} : \{\rho_1 \mid \rho_2\}}\ \textsc{Concat}
$$

$$
\frac{}{\Gamma \vdash l_c : (\!|l_c|\!)}\ \textsc{Lab} \qquad \frac{\Gamma \vdash e_1 : \{\ell : \tau \mid \rho\} \qquad \Gamma \vdash e_2 : (\!|\ell|\!)}{\Gamma \vdash e_1.e_2 : \tau}\ \textsc{Prj}
$$

$$
\frac{}{\Gamma \vdash \langle\rangle : \langle\rangle}\ \textsc{EmptyR} \qquad \frac{\Gamma \vdash e : (\!|\ell|\!) \qquad \Gamma \vdash \tau : \star}{\Gamma \vdash \langle e \rangle : \langle \ell : \tau \rangle}\ \textsc{Row}
$$

$$
\frac{\Gamma \vdash e_1 : \langle \rho_1 \rangle \qquad \Gamma \vdash e_2 : \langle \rho_2 \rangle}{\Gamma \vdash \langle e_1 \mid e_2 \rangle : \langle \rho_1 \mid \rho_2 \rangle}\ \textsc{ConcatR}
$$

$$
\frac{\Gamma \vdash e_1 : \{\rho_1 \mid \rho_2\} \qquad \Gamma \vdash e_2 : \langle \rho_1 \rangle}{\Gamma \vdash e_1 \bullet e_2 : \{\rho_1\}}\ \textsc{PrjR} \qquad \frac{\Gamma \vdash e_1 : \{\rho_1 \mid \rho_2\} \qquad \Gamma \vdash e_2 : \langle \rho_1 \rangle}{\Gamma \vdash e_1 \backslash e_2 : \{\rho_2\}}\ \textsc{Del}
$$

Figure 5. Typing rules

Row Equivalence. We formalize the row equivalence relation $\rho_1 \equiv \rho_2$ in Fig. 3. We overload the $\equiv$ operator such that $\tau_1 \equiv \tau_2$ for types that are equivalent up to row equivalence and otherwise syntactically equal.

As an equivalence relation, row equivalence is reflexive (REFL), symmetric (SYMM), transitive (TRANS), and congruent with respect to type constructors (SINGLETON and CONCAT). Moreover, the concatenation operation is commutative over distinct labels (COMM), associative (ASSOC), and has Empty as its unit (EMPR,EMPL). Notably, (COMM) compares only label

$$\frac{}{\Gamma \vdash \tau \sqsubseteq \tau}\ \textsc{Eq} \qquad \frac{\Gamma \vdash \tau : \star \qquad \Gamma \vdash \sigma[a \mapsto \tau] \sqsubseteq \sigma'}{\Gamma \vdash \forall a : \star.\ \sigma \sqsubseteq \sigma'}\ \textsc{InstK}$$

$$\frac{\Gamma \vdash \ell : \mathsf{Label} \qquad a \blacktriangleright^l \sigma \vee a \notin \mathsf{ftv}(\sigma) \qquad \Gamma \vdash \sigma[a \mapsto \ell] \sqsubseteq \sigma'}{\Gamma \vdash \forall a : \mathsf{Label}.\ \sigma \sqsubseteq \sigma'}\ \textsc{InstL}$$

$$\frac{\Gamma \vdash \rho : \mathsf{Row} \qquad a \blacktriangleright^r \sigma \vee a \notin \mathsf{ftv}(\sigma) \qquad \Gamma \vdash \sigma[a \mapsto \rho] \sqsubseteq \sigma'}{\Gamma \vdash \forall a : \mathsf{Row}.\ \sigma \sqsubseteq \sigma'}\ \textsc{InstR}$$

Figure 6. Instantiation

$$\frac{a \blacktriangleright^* \sigma}{a \blacktriangleright^* \forall \beta : \kappa.\sigma} \qquad \frac{a \blacktriangleright^* \tau_1}{a \blacktriangleright^* \tau_1\ \tau_2} \qquad \frac{a \blacktriangleright^* \tau_2}{a \blacktriangleright^* \tau_1\ \tau_2}$$

$$\frac{\rho \equiv (\rho' \mid a)}{a \blacktriangleright^r \{\rho\}} \qquad \frac{\rho \equiv \langle \rho' \mid a \rangle}{a \blacktriangleright^r \langle \rho \rangle}$$

$$\frac{}{a \blacktriangleright^l (\!|a|\!)} \qquad \frac{\rho \equiv (\rho' \mid a : \tau)}{a \blacktriangleright^l \{\rho\}} \qquad \frac{\rho \equiv \langle \rho' \mid a : \tau \rangle}{a \blacktriangleright^l \langle \rho \rangle}$$

Figure 7. Tail check rules. We use $\blacktriangleright^*$ to indicate that a rule applies to both $\blacktriangleright^l$ and $\blacktriangleright^r$.

literals, but not label variables, as we cannot predict if a label variable will introduce shadowing after instantiation.

3.2 Expressions and Typing

We outline expression syntax in Fig. 4. Expressions e are variables x, function literals $\lambda x.\ e$, applications $e_1\ e_2$, let-bindings **let** $x = e_1$ **in** e_2, the empty record $\{\}$, a singleton record $\{e_1 = e_2\}$, record concatenation $\{e_1 \mid e_2\}$, a label literal l_c, label projects $e_1.e_2$, the empty row $\langle\rangle$, a singleton row $\langle e \rangle$, row concatenation $\langle e_1 \mid e_2 \rangle$, row projections $e_1 \bullet e_2$, and row deletion $e_1 \backslash e_2$. As syntactic sugar, we define label deletion $e \backslash l_c$ as deletion of a singleton row. And we often write commas for record and row concatenation.

Fig. 5 gives the typing rules, which are essentially the Hindley-Milner type system extended with records and rows. Rule (Var) type-checks a variable by first looking up its type σ in the context. Since σ is potentially polymorphic, the rule instantiates σ to τ; the instantiation rules are given in Fig. 6, which we will explain shortly.

Rule (Eq) converts between equivalent types. This rule makes the typing rules non-syntax-directed; however, it is easy to integrate the rule into each rule that requires type equivalence (e.g., in rule (App)) and thus make a syntax-directed version of the type system.

Rules (Lam), (App), and (Let) are standard. The next three rules type-check records. An empty record always has an empty record type (rule (Empty)). For a singleton record (rule (Rcd)), since the system supports first-class labels, we first get the label type $(\!|\ell|\!)$ from e_1, and then get the field type τ from e_2, and return the record type $\{\ell : \tau\}$. For record concatenation (rule (Concat)), the result type is simply a record of row concatenation.

In rule (Lab), a label literal has itself in the type; namely, $(\!|l_c|\!)$ is a singleton type. Labels can be used to project a field from a record (rule (Prj)). Here we first get the record type $\{\ell : \tau \mid \rho\}$ from e_1, and then get the record type $(\!|\ell|\!)$ from e_2, and return the field type τ. Notably, ℓ may not be the first label in a record type, but by using row equivalence (rule (Eq)) we may be able to move ℓ to be the head. Take $e : \{l_1 : \mathsf{Int}, l_2 : \mathsf{Bool}\}$ with l_1 and l_2 being constants as an example. By rule (Eq) and rule (Assoc) we can deduce $e : \{l_2 : \mathsf{Bool}, l_1 : \mathsf{Int}\}$, allowing us to conclude $e.l_2 : \mathsf{Bool}$. However, since rule (Assoc) only works on distinct label literals, if $e' : \{a : \mathsf{Int}, l_2 : \mathsf{Bool}\}$ for some label variable a (or $e' : \{a \mid l_2 : \mathsf{Bool}\}$ for some row variable a), then $e'.l_2$ is not considered well-typed (as usual, shadowing is yet unclear).

The rest of the rules concern rows. An empty row always has an empty row type (rule (EmptyR)). For a singleton row (rule (Row)), we get the label type $(\!|\ell|\!)$ from e, and we guess a field type τ, returning the row type $\langle \ell : \tau \rangle$. We could also have the programmers provide an explicit type annotation for the field. Such annotations are not necessary, however.[3] This is because rows are mainly used for projection (and deletion), and just like label projection, a label in a row always projects out the first corresponding field in a record.[4] In the row concatenation (rule (ConcatR)), the result type is simply the concatenated row type. Finally, projection with a row (rule (PrjR)) gets a subset of the original record, and deletion of a row (rule (Del)) removes a subset of the record.

Instantiation and tail check. Recall that rule (Var) uses instantiation ($\sigma \sqsubseteq \tau$) to turn a polymorphic type into a monotype. Fig. 6 presents the instantiation rules.

Rule (Eq) says that a monotype is an instantiation of itself. Rule (InstK) instantiates a type variable $\alpha : \star$ by a monotype τ. The rules for record and label variables are similar, except for the extra condition $\alpha \blacktriangleright^* \sigma \vee \alpha \notin \mathsf{ftv}(\sigma)$; that is, either α is not used in the body, or it satisfies $\alpha \blacktriangleright^* \sigma$.

We call the $\alpha \blacktriangleright^* \sigma$ condition *tail check*, whose rules are given in Fig. 7. At a high-level, tail check only allows instantiating a label (with $\blacktriangleright^l$) or a row (with $\blacktriangleright^r$) variable if it appears in a *tail position*. While this seems artificial at first, take $e : \forall\ a.\ \{a : \mathsf{Int}, l_2 : \mathsf{Bool}\}$ as an example and consider what the type is for $e.l_2$. Without the tail check, our type system would allow a derivation leading to $e.l_2 : \mathsf{Int}$ (by

[3] Nor are they useful: even if labels are type annotated, a projection like $e \bullet \langle l : \mathsf{Bool} \rangle$ where $e : \{l : \mathsf{Int}, l : \mathsf{Bool}\}$ is rejected, since again we can only access the first l label from a record, and here the types mismatch. However, we do not lose any expressive power, as we could rewrite the program as $(e \backslash l) \bullet \langle l : \mathsf{Bool} \rangle$.

[4] A repeated label in a row will get the next corresponding field.

$$
\begin{array}{rrcll}
\text{value} & v & ::= & \lambda x.\ e \mid l_c \mid \{l_{c_1} = v_1, \ldots, l_{c_n} = v_n\} \mid \langle l_{c_1}, \ldots, l_{c_n}\rangle & (n \geq 0) \\
\text{evaluation context} & E & ::= & \square \mid E\ e \mid v\ E \mid \textbf{let}\ x = E\ \textbf{in}\ e & \\
& & \mid & \{E = e\} \mid \{v = E\} \mid \{E \mid e\} \mid \{v \mid E\} \mid E.e \mid v.E & \\
& & \mid & \langle E\rangle \mid \langle E \mid e\rangle \mid \langle v \mid E\rangle \mid E_\bullet e \mid v_\bullet E \mid E\backslash e \mid v\backslash E &
\end{array}
$$

$$
\begin{array}{llcll}
(app) & (\lambda x.\ e)\ v & \longrightarrow & e[x \mapsto v] & \\
(let) & \textbf{let}\ x = v\ \textbf{in}\ e & \longrightarrow & e[x \mapsto v] & \\
(prj) & \{l_{c_1} = v_1, \ldots, l_{c_n} = v_n\}.l_{c_i} & \longrightarrow & v_i & \text{where } l_{c_1}, \ldots, l_{c_{i-1}} \neq l_{c_i} \\
(prj_{r_1}) & v_\bullet\langle\rangle & \longrightarrow & \{\} & \\
(prj_{r_2}) & v_\bullet\langle l_c, v'\rangle & \longrightarrow & \{l_c = v.l_c \mid (v\backslash l_c)_\bullet v'\} & \\
(concat) & \{\{l_{c_1} = v_1, \ldots, l_{c_i} = v_i\} \mid \{l_{c_j} = v_j, \ldots, l_{c_n} = v_n\}\} & \longrightarrow & \{l_{c_1} = v_1, \ldots, l_{c_i} = v_i, l_{c_j} = v_j, \ldots, l_{c_n} = v_n\} & \\
(concat_r) & \langle\langle l_{c_1}, \ldots, l_{c_i}\rangle \mid \langle l_{c_j}, \ldots, l_{c_n}\rangle\rangle & \longrightarrow & \langle l_{c_1}, \ldots, l_{c_i}, l_{c_j}, \ldots, l_{c_n}\rangle & \\
(del_1) & v\backslash\langle\rangle & \longrightarrow & v & \\
(del_2) & \{l_{c_1} = v_1, \ldots, l_{c_n} = v_n\}\backslash\langle l_{c_i}, v\rangle & \longrightarrow & \{l_{c_1} = v_1, \ldots l_{c_{i-1}} = v_{i-1}, l_{c_{i+1}} = v_{i+1}, l_{c_n} = v_n\}\backslash v & \\
 & & & & \text{where } l_{c_1}, \ldots, l_{c_{i-1}} \neq l_{c_i} \\
(step) & E[e_1] & \longmapsto & E[e_2] & \text{if } e_1 \longrightarrow e_2
\end{array}
$$

Figure 8. Operational semantics

instantiating $\alpha = l_2$), as well as one leading to $e.l_2 : \text{Bool}$ (for $\alpha = l_1 \neq l_2$). Hence, $\blacktriangleright^*$ will help rule out this example.

The first three rules in Fig. 7 traverse the structure of the type. Note that we only require the variable to appear in a tail position once[5], hence the two type application rules. For row variable ($\blacktriangleright^r$), we check that the variable appears at the tail of a record or a row type. For label variables ($\blacktriangleright^l$), the variable can appear in a singleton type, or at the tail of a record of a row type.

Going back to the instantiation rules in Fig. 6, we can consider the condition $\alpha \notin \text{ftv}(\sigma)$ as a relaxation of the tail check: we do not want to reject a polymorphic type where the quantified variable a is not used and thus does not appear in any tail position.[6]

3.3 Operational Semantics

Fig. 8 defines values, evaluation contexts, and the operational semantics for $\lambda^{\langle\rangle}$.

Values v include lambdas $\lambda x.\ e$, label literals l_c, and *canonicalized* records and rows; recall that commas are syntactic sugar for a series of row concatenations (Fig. 4).

An evaluation context E is essentially an expression with a hole, in which we can plug in another expression. The definition here is standard. We only remark that the evaluation context decides the evaluation order. For example, the form $v\ E$ means that for applications we evaluate the argument only when the function is already a value.

[5]A more restricted version of the rules could require all occurrences of a variable to be in tail positions, but such a system would also rule out some useful programs.

[6]One may also relax the system and delay tail-checks. For example, take $f :: \forall\ a\ b.\text{Int} \to \langle a : \text{Int}, b\rangle \to \text{Int}$. We could consider $f\ 1$ to be well-typed, even though a does not pass the tail check, because f's type is equivalent to $f :: \text{Int} \to \forall\ a\ b.\langle a : \text{Int}, b\rangle \to \text{Int}$. Such a design can be considered as the reverse of *deep skolemisation* [8].

Rules (*app*) and (*let*) are standard. Rule (*prj*) projects a field l_{c_i} out of a record. The semantics searches for the first field corresponding to the label, expressed using the side condition $l_{c_1}, \ldots, l_{c_{i-1}} \neq l_{c_i}$. Projecting a row is defined inductively. In the base case (prj_{r1}), projecting an empty row returns the empty record. If the row is not empty (prj_{r2}), then it must be $\langle l_c, v'\rangle$, in which case we first project l_c, and then build the result by recursively projecting v'. Notably, $v\backslash l_c$ only removes the first appearance of l_c in v, making sure that shadowed labels can be projected too, if requested in v.

Rule (*concat*) canonicalizes a record by flattening it. Similarly, ($concat_r$) canonicalizes a row. Row deletion is defined in an inductive way similar to row projection. In the base case (del_1), deleting an empty row returns the original record. If the row is not empty (del_2), then it must be $\langle l_{c_i}, v\rangle$, in which case we first delete l_{c_i} by searching for the first field for l_{c_i} using the side condition $l_{c_1}, \ldots, l_{c_{i-1}} \neq l_{c_i}$, and recursively delete v.

Finally, (*step*) evaluates inside an evaluation context. Note that we write $\longmapsto$ here, instead of $\longrightarrow$ used in previous rules.

Type safety. With the operational semantics, we can now formally show that $\lambda^{\langle\rangle}$ is type safe. All proofs can be found in the appendix.

Theorem 3.1 (Preservation). *If $\Gamma \vdash e : \sigma$, and $e \longmapsto e'$, then $\Gamma \vdash e' : \sigma$.*

Theorem 3.2 (Progress). *If $\bullet \vdash e : \sigma$, then either e is a value, or there exists e' such that $e \longmapsto e'$.*

3.4 Type Inference

A type inference algorithm is often implemented as a two-stage process. In the first stage, the algorithm traverses the input program and generates type constraints according to a

unification variables	α, β, γ		
constraint	C	$::=$	$\top \mid \tau_1 \sim \tau_2 \mid \rho_1 \sim \rho_2 \mid \ell_1 \sim \ell_2 \mid C_1 \wedge C_2$
unification type	τ	$::=$	$\alpha \mid \mathsf{Int} \mid \rightarrow \mid \tau_1\ \tau_2 \mid \{\rho\} \mid \langle\rho\rangle \mid (\!\mid \ell \mid\!)$
substitution context	Δ	$::=$	$\bullet \mid \Delta, \alpha : \kappa \mid \Delta, \alpha : \kappa = \tau \mid \Delta, \alpha : \kappa = \rho \mid \Delta, \alpha : \kappa = \ell$

Figure 9. Unification types

$\boxed{\Delta_1 \vdash C_1 \Longrightarrow \Delta_2 \vdash C_2}$ $\quad (\Delta \vdash C_1 \Longrightarrow C_2 \triangleq \Delta \vdash C_1 \Longrightarrow \Delta \vdash C_2)$ $\quad (C_1 \Longrightarrow C_2 \triangleq \Delta \vdash C_1 \Longrightarrow \Delta \vdash C_2)$

(*conj*)	$\Delta_1 \vdash C_1 \wedge C_2$	$\Longrightarrow$	$\Delta_2 \vdash C_1' \wedge C_2$	if $\Delta_1 \vdash C_1 \Longrightarrow \Delta_2 \vdash C_1'$
(*top₁*)	$\top \wedge C$	$\Longrightarrow$	C	
(*top₂*)	$C \wedge \top$	$\Longrightarrow$	C	
(*Trefl*)	$\tau \sim \tau$	$\Longrightarrow$	$\top$	
(*Tapp*)	$\tau_1\ \tau_2 \sim \tau_3\ \tau_4$	$\Longrightarrow$	$\tau_1 \sim \tau_3 \wedge \tau_2 \sim \tau_4$	
(*Trcd*)	$\{\rho_1\} \sim \{\rho_2\}$	$\Longrightarrow$	$\rho_1 \sim \rho_2$	
(*Trow*)	$\langle\rho_1\rangle \sim \langle\rho_2\rangle$	$\Longrightarrow$	$\rho_1 \sim \rho_2$	
(*Tlab*)	$(\!\mid \ell_1 \mid\!) \sim (\!\mid \ell_2 \mid\!)$	$\Longrightarrow$	$\ell_1 \sim \ell_2$	
(*Tswap*)	$\tau \sim \alpha$	$\Longrightarrow$	$\alpha \sim \tau$	if $\tau \neq \beta$ for any β
(*Tsolved*)	$\Delta \vdash \alpha \sim \tau$	$\Longrightarrow$	$\tau' \sim \tau$	if $\alpha : \kappa = \tau' \in \Delta$
(*Tsolve*)	$\Delta \vdash \alpha \sim \tau$	$\Longrightarrow$	$\Delta \circ \alpha : \kappa = [\Delta]\tau \vdash \top$	if $\alpha : \kappa \in \Delta \wedge \Delta \vdash \tau : \kappa \wedge \alpha \notin \mathsf{ftv}([\Delta]\tau)$
(*Rrefl*)	$\rho \sim \rho$	$\Longrightarrow$	$\top$	
(*Rsingleton*)	$\ell_1 : \tau_1 \sim \ell_2 : \tau_2$	$\Longrightarrow$	$\ell_1 \sim \ell_2 \wedge \tau_1 \sim \tau_2$	
(*Rfield*)	$\Delta_1 \vdash l_c : \tau, \rho_1 \sim \rho_2$	$\Longrightarrow$	$\Delta_2 \vdash \tau \sim \tau' \wedge \rho_1 \sim \rho_2'$	if $\Delta_1 \vdash [\Delta_1]\rho_2 \overset{l_c}{\hookrightarrow} l_c : \tau', \rho_2' \dashv \Delta_2 \wedge [\Delta_2]\rho_1 = [\Delta_1]\rho_1$
(*Rswap*)	$\rho \sim \alpha$	$\Longrightarrow$	$\alpha \sim \rho$	if $\rho \neq \beta$ for any β
(*Rsolved*)	$\Delta \vdash \alpha \sim \rho$	$\Longrightarrow$	$\rho' \sim \rho$	if $\alpha : \kappa = \rho' \in \Delta$
(*Rsolve*)	$\Delta \vdash \alpha \sim \rho$	$\Longrightarrow$	$\Delta \circ \alpha : \kappa = [\Delta]\rho \vdash \top$	if $\alpha : \kappa \in \Delta \wedge \Delta \vdash \rho : \kappa \wedge \alpha \notin \mathsf{ftv}([\Delta]\rho)$
(*Lrefl*)	$\ell \sim \ell$	$\Longrightarrow$	$\top$	
(*Lswap*)	$\ell \sim \alpha$	$\Longrightarrow$	$\alpha \sim \ell$	if $\ell \neq \beta$ for any β
(*Lsolved*)	$\Delta \vdash \alpha \sim \ell$	$\Longrightarrow$	$\ell' \sim \ell$	if $\alpha : \kappa = \ell' \in \Delta$
(*Lsolve*)	$\Delta \vdash \alpha \sim \ell$	$\Longrightarrow$	$\Delta \circ \alpha : \kappa = [\Delta]\ell \vdash \top$	if $\alpha : \kappa \in \Delta \wedge \Delta \vdash \ell : \kappa \wedge \alpha \notin \mathsf{ftv}([\Delta]\ell)$
(*LUhead*)	$\Delta \vdash l_c : \tau, \rho$	$\overset{l_c}{\hookrightarrow}$	$l_c : \tau, \rho \dashv \Delta$	
(*LUrec*)	$\Delta_1 \vdash l_{c'} : \tau, \rho$	$\overset{l_c}{\hookrightarrow}$	$l_c : \tau, l_{c'} : \tau', \rho' \dashv \Delta_2$	if $l_{c'} \neq l_c \wedge \Delta_1 \vdash \rho \overset{l_c}{\hookrightarrow} l_c : \tau, \rho' \dashv \Delta_2$
(*LUtail*)	$\Delta \vdash \alpha$	$\overset{l_c}{\hookrightarrow}$	$l_c : \beta, \gamma \dashv \Delta \circ \alpha : \mathsf{Row} = (l_c : \beta, \gamma), \beta : \star, \gamma : \mathsf{Row}$	if $\alpha : \mathsf{Row} \in \Delta$ where β, γ fresh

Figure 10. Unification

set of syntax-directed typing rules.[7] In the second stage, the algorithm uses a specific set of constraint solving rules to resolve the generated constraints. As our system essentially extends the Hindley-Milner system with records and rows, most of the constraint generation and solving is standard. Therefore, in this section, we focus on the unique part of our system, which is the unification algorithm involving record and row types, and we refer the interested reader to [15] for general constraint-based type inference in ML-style calculi.

Fig. 9 defines the constraints and types used during unification. First, we use α, β, and γ for unification variables. We focus on equality constraints C, which include the truth $\top$, equality constraints over types, rows, and labels, and constraint conjunction $C_1 \wedge C_2$. Finally, a unification type τ[8] includes unification variables α instead of type variables a. To make the presentation clearer, we assume all types are canonicalized (e.g. all record concatenations are commuted to be left-associative and labels and variables are ordered lexicographically). Since the type system features a kind system,

[7] For our system, the type equivalence relation in (*Eq*) will be integrated into type equity constraints, and the rest of the type system is syntax-directed.

[8] We always make it clear from the context whether a type τ refers to a declarative type or a unification type.

we use substitution contexts Δ to keep track of unification variables' kinds and their solutions. We write $[\Delta]\tau$ for substituting unification variables in τ by their solutions in Δ, if any solutions exist. The substitution context is idempotent, meaning $[\Delta]([\Delta]\tau) = [\Delta]\tau$. For an unsolved $\alpha : \kappa \in \Delta$, we write $\Delta \circ \alpha : \kappa = \tau$ for solving α with τ, which will also substitute α with its solution in all other items in Δ (and canonicalize after substitution), ensuring that the resulting context remains idempotent.

The unification rules are listed in Fig. 10. The judgment $\Delta_1 \vdash C_1 \Longrightarrow \Delta_2 \vdash C_2$ reads: under the solution context Δ_1, the constraint C_1 reduces to constraint C_2, updating the solution context to Δ_2. Most rules are self-explanatory. Rule (*conj*) is defined modulo constraint associativity, commutativity and truth being a neutral element ((*top*$_1$) and (*top*$_2$)).

T-rules unify types. A reflexive constraint reduces to truth (*Trefl*). The next four rules decompose constraints over type structures into constraints over subcomponents. The last three T-rules concern unification variables. Rule (*Tswap*) puts the unification variables on the left, so that it can be solved by the next two rules. A solved unification variable is replaced by its solution (*Tsolved*). Rule (*Tsolve*) solves a unification variable by $[\Delta]\tau$, after checking that the solution is of the right kind, and that α does not occur in $[\Delta]\tau$.

Similarly, the R-rules and the L-rules unify rows and labels respectively. The most notable rule is (*Rfield*). In this rule, we are unifying a row $l_c : \tau, \rho_1$ with another row ρ_2. We first find the corresponding l_c field in ρ_2, by using the lookup rules $\overset{l_c}{\hookrightarrow}$, which decomposes ρ_2 into $l_c : \tau', \rho_2'$. We can then continue by unifying the subcomponents.

The lookup rules are defined at the end of the figure. The judgment $\Delta_1 \vdash \rho_1 \overset{l_c}{\hookrightarrow} l_c : \tau, \rho_2 \dashv \Delta_2$ reads: under the substitution context Δ_1, we look for field l_c in ρ_1, and find that ρ_1 can be represented as $l_c : \tau, \rho_2$, updating the substitution context to Δ_2. Rule (*LUhead*) finds the label successfully and returns the context unchanged. Rule (*LUrec*) discovers that the head of the row is not l_c, and thus recursively looks up in the tail of the row. Rule (*LUtail*) is when we encounter a polymorphic row tail. In this case, we know that α must be of shape $l_c : \beta, \gamma$ with fresh β and γ.

Returning to (*Rfield*), note that when the lookup rule returns, we have an extra side condition $[\Delta_2]\rho = [\Delta_1]\rho$.[9] The condition prevents us from unifying two rows with a shared tail but different prefixes. For example, consider unifying $(l_1 : \mathsf{Int}, \alpha) \sim (l_2 : \mathsf{Int}, \alpha)$ (with $l_1 \neq l_2$). By (*LUtail*), we will generate $\alpha = l_1 : \beta, \gamma$ (with fresh β, γ) and rewrite the right-hand-side to $(l_1 : \beta, l_2 : \mathsf{Int}, \gamma)$. Without the side condition, by (*Rfield*) the original constraint is decomposed to $\mathsf{Int} \sim \beta$ (with solution $\beta = \mathsf{Int}$) and $\alpha \sim l_2 : \mathsf{Int}, \gamma$. The latter constraint turns into $(l_1 : \mathsf{Int}, \gamma) \sim (l_2 : \mathsf{Int}, \gamma)$ by (*Tsolved*). Now we are back to where we started, modulo renaming of unification variables, and thus unification will loop forever. The side condition ensures that this kind of situation will never happen, by checking that the newly solved unification variable in Δ_2 does not occur in $[\Delta_1]\rho_1$.

Unification Soundness. We prove that our unification is sound. First, we define the notion of a solution context.

Definition 3.3 (Solution Context)**.** A solution context solves a constraint, written as $\Delta \models C$, if for any $\tau_1 \sim \tau_2 \in C$, we have $[\Delta]\tau_1 \equiv [\Delta]\tau_2$. Similarly for $\rho_1 \sim \rho_2$, and $\ell_1 \sim \ell_2 \in C$.

We prove that unification finds a solution context.

Theorem 3.4 (Unification Soundness)**.** *If* $\Delta_1 \vdash C \Longrightarrow^* \Delta_2 \vdash \top$*, then* $\Delta_2 \models C$.

Most General Unifier. Since our system supports first-class labels and rows, unification may get stuck (thus is *incomplete*) when there is no unique most general unifier for the constraint. This has already been discussed in Leijen [9], so we only present a short example. For

$$(\alpha : \mathsf{Int}, \beta) \sim (l_1 : \mathsf{Int}, l_2 : \mathsf{Int}),$$

there are two unifiers that are incompatible with each other:

$$\alpha = l_1, \beta = l_2 : \mathsf{Int} \qquad \text{or} \qquad \alpha = l_2, \beta = l_1 : \mathsf{Int}.$$

Fortunately, we can prove that when our unification succeeds, it always returns the most general unifier (in which case the type inference returns the principal type). To prove this, we first define the notion of context extension.

Definition 3.5 (Context Extension)**.** A solution context extends to another solution context, written as $\Delta_1 \rightsquigarrow \Delta_2$, if for any τ, we have $[\Delta_2]\tau \equiv [\Delta_2]([\Delta_1]\tau)$.

The definition effectively captures the semantics of the context extension notion defined in [3]. Intuitively, context extension expresses a form of information increase: if $\Delta_1 \rightsquigarrow \Delta_2$, then Δ_2 preserves all equivalence relations derivable from Δ_1[10], and may additionally include more variable solutions. We can prove that unification extends the context.

Lemma 3.6 (Unification extends context)**.** *If* $\Delta_1 \vdash C_1 \Longrightarrow \Delta_2 \vdash C_2$*, then* $\Delta_1 \rightsquigarrow \Delta_2$.

Now we define the notion of a most general unifier:

Theorem 3.7 (Unification Produces the Most General Unifier)**.** *If* $\Delta_1 \vdash C \Longrightarrow^* \Delta_2 \vdash \top$*, then for any* Δ *such that* $\Delta_1 \rightsquigarrow \Delta$ *and* $\Delta \models C$*, there exists* Δ' *such that* $\Delta_2 \rightsquigarrow \Delta'$ *and* $\Delta \rightsquigarrow \Delta'$.

The lemma states that, for a unification problem C starting with the substitution context Δ_1, the unification result Δ_2 is the most general solution, as for any other possible solution Δ, we can establish some relation between Δ_2 and Δ. The reader might expect that $\Delta_2 \rightsquigarrow \Delta$ which, however, is not

[9] A more efficient way of implementing this condition is to calculate the tails of ρ_1 and pass them into the lookup rules, so (*LUtail*) checks that α is not in the tails.

[10] That is, if $[\Delta_1]\tau_1 \equiv [\Delta_1]\tau_2$, then $[\Delta_2]\tau_1 \equiv [\Delta_2]\tau_2$, as $[\Delta_2]\tau_1 = [\Delta_2]([\Delta_1]\tau_1) = [\Delta_2]([\Delta_1]\tau_2) = [\Delta_2]\tau_2$.

true in general: during unification, we may allocate extra fresh unification variables (such as in (*LUtail*)) to help solve the constraint, which the solution context Δ may not contain. Therefore, instead of $\Delta_2 \rightsquigarrow \Delta$, we can find another solution context Δ' that both Δ_2 and Δ extend to. This effectively ensures that Δ_2 is the most general unifier: suppose Δ_2 is not the most general unifier, then there exists at least one unification variable that does not have to be solved or could be solved differently, say $\alpha = \mathsf{Int} \in \Delta_2$. Then we can make the solution context Δ contain a different solution $\alpha = \mathsf{String}$, and now it is impossible to find a Δ' that both Δ_2 and Δ extend to. Thus Δ_2 must be the most general unifier.

Design Variants. While not having most general unifier in certain cases is unsatisfactory, we do not find it a significant limitation in practice[11], largely because of the tail check (Fig. 7) imposed in the system. Back to the problematic example above:

$$(\alpha : \mathsf{Int}, \beta) \sim (l_1 : \mathsf{Int}, l_2 : \mathsf{Int})$$

One may find it a bit strange to have α in the unification constraint, since α does not appear in a tail position. Indeed, consider the type of the following function:

$f :: \forall (a : \mathsf{Label})\ (b : \mathsf{Row}).\{a : \mathsf{Int}, b\} \to (\!|a|\!) \to \mathsf{Int}$
$f\ r\ l = r.l$

In this case, both a and b can be instantiated (to α and β respectively) as they both satisfy the tail check. Applying f to a record of type $\{l_1 : \mathsf{Int}, l_2 : \mathsf{Int}\}$ will give us the above equality constraint. However, note that the reason a passes the tail check is because there is a singleton label $(\!|a|\!)$ which is the second argument to the function. Therefore, once the function is applied to a second argument, we will know exactly how to solve α, and thus β. This is often the case in practice — unification could get resolved once more information is available.[12] Indeed, this is true for all examples in the evaluation (§4).

As a design variant, we could follow Leijen [9] to have the unification derive a set of most general unifiers, making unification complete. That is, a unifier is not *the* most general one in the usual sense, but a most general one up to permutation of row fields. For example for the problematic example, both of the two incompatible unifiers will be returned. However, as discussed in [9], such an algorithm is exponential in the number of polymorphic labels.

[11] We also remark this is not uncommon in practical systems, including e.g., the type inference [18] and the kind inference algorithms [23] in GHC.

[12] The tail check is not always helpful. For example, consider $f :: \forall a\ b.\{a : \mathsf{Int}, b\} \to ((\!|a|\!) \to (\!|a|\!)) \to \mathsf{Int}$. In this case, a and b still pass the tail check, but the second argument to f could be a polymorphic identity function that contains no useful information about a. Still, this is a contrived scenario that we expect to be rare in practice. It is also possible to design a stricter tail check to make sure each tail position is useful.

4 Evaluation

To motivate the development of this new flavor of row types, we evaluate their capability, both in terms of the functions that can be typed, and in terms of errors that can be caught statically, over the Brown Benchmark for Tabular Types (B2T2) [11]. Furthermore, an important part of the evaluation, that we will stress for the last time now, is that the type theory is arguably *simple*, as it is based purely on unification and does not require the use of qualified or dependent types.

Throughout this section, we will use the usual facilities found in most functional programming languages, such as sum and product types. We adopt Haskell's naming convention and write Maybe for an optional type.

Since our presentation here focuses on the type system features and not a concrete implementation, we do not report on implementation details such as error messages, but focus on the typeability. Similarly, we do not present example implementations of most functions, as they are either fairly trivial (see hcat) or would not resemble a real implementation (joins might require hash tables for efficiency).

4.1 What is a Table?

First, we represent a table as a homogeneous array of records, whose type we denote as $[\{r\}]$, with r being the row representing the table schema. This satisfies the benchmark requirements of tables being a typed, *rectangular collection of cells*. Moreover, since labels are first-class, column names are, as requested, *string-like first-class values*.

While we consider a table as a homogeneous array (of records), the array type constructor is not our focus here. In particular, we assume that the array type does not put any limitations on the element type, and any valid record type can be supplied (as is the case in many array languages such as [6, 14]). Thus, we only use parts of the B2T2 benchmark that deal with column manipulation for evaluation. Functions that focus purely on modifying the rows of a data table (subsampling, etc.) are mostly capabilities of the array type and are unaffected by the row type[13].

Our main deviation from the benchmark is in the relaxed structure of the schema. Firstly, columns are unordered. We consider this to be a fairly unimportant property of tables, since most data processing does not actually rely on the ordering of columns. Secondly, column names are not unique. This *could* lead to potential errors by accidentally introducing shadowing between columns. But, we consider the usability upside of not having to enforce distinctness at the type level to balance the potential risks. After all, programmers already have to deal with the risk of accidental shadowing in pretty much all common programming languages. And, similar to

[13] The list of omitted functions includes: emptyTable, addRows, vcat, values, nrows, getRow, selectRows, head, distinct, find, tfilter, update, select, selectMany. We additionally omit tsort, sortByColumns, orderBy, count and bin, as all of them can be implemented by composing column projections/insertions (that we discuss) with row rearrangements.

variable shadowing in typed languages, it only goes unnoticed when the shadowing does not change the type of the variable (or changes it in a compatible way, e.g. in systems with subtyping).

4.2 Example Tables

B2T2 provides a collection of example tables. Since neither the array type nor the record type constrain the column types, all example tables can be successfully represented and typed. We omit their definitions, but provide full type signatures.

```
students :: [{name : String, age : Int, favColor : String}]
studentsMissing :: [
  {name : String, age : Int, favColor : Maybe String}]
gradebookSeq :: [
  {name : String, age : Int, quizzes : [Int], final : Int}]
gradebookTable :: [
  {name : String, age : Int, final : Int,
   quizzes : [{quiz : Int, grade : Int}]}
]
```

4.3 Expressible Table Manipulation Functions

In this section, we list the data manipulation functions proposed in the benchmark that our system can successfully assign types to. We group them into subcategories that have similar behaviors and share similar challenges.

Most functions are presented with the same signatures as in the benchmark. When we modify the signature, we attach a subscript to the function name, e.g., $\mathsf{almostOk}_{\mathsf{modified}}$. We discuss all the functions in the benchmark, here or in §4.5.

4.3.1 Column Manipulation Functions.

We start with functions that manipulate the columns of a data table independently over its rows. These are implementable as fmap/zipWith of a record-manipulating function.

Our type system can successfully assign type signatures to almost all the functions in this category. However, due to columns being unordered, the overloads of getColumn and selectColumns that use integers or boolean masks as selectors are inexpressible.

```
hcat :: ∀r₁ r₂. [{r₁}] → [{r₂}] → [{r₁ | r₂}]
getValue :: ∀r l a. {l : a | r} → ⦇l⦈ → a
getColumn :: ∀r l a. [{l : a | r}] → ⦇l⦈ → [a]
selectColumns :: ∀r₁ r₂. [{r₁ | r₂}] → ⟨r₁⟩ → [{r1}]
dropColumn :: ∀r l a. [{l : a | r}] → ⦇l⦈ → [{r}]
dropColumns :: ∀r₁ r₂. [{r₁ | r₂}] → ⟨r₁⟩ → [{r₂}]
buildColumn
  :: ∀r l a. [{r}] → ⦇l⦈ → ({r} → a) → [{l : a | r}]
transformColumn
  :: ∀r l a. [{l : a | r}] → ⦇l⦈ → (a → b) → [{l : b | r}]
```

One interesting case is the addColumn function. Because columns are not distinct, it can be implemented in two ways: one that adds the column at the front, and one that adds it in the back. Of course, both are equivalent if the column name does not appear in the table, but they do differ in shadowing semantics when it does.

```
addColumn_front
  :: ∀r l a. [{r}] → ⦇l⦈ → [a] → [{l : a | r}]
addColumn_back
  :: ∀r l a. [{r}] → ⦇l⦈ → [a] → [{r | l : a}]
```

Another interesting case is the renameColumns function. In the benchmark specification, it renames a set of columns, but in our type system it can only rename one column at a time (or any fixed statically known number). We add a $_{\mathsf{column}}$ subscript to denote functions that work on a single column only. Luckily, the general variant can be recovered by composing multiple single-column applications. For renameColumns, the single-column version has clearer semantics too, since otherwise applying the name substitution in parallel or in sequence could lead to different results if the domain and range of the substitution overlap.

```
renameColumns_column
  :: ∀r l₁ l₂ a. [{l₁ : a | r}] → ⦇l₁⦈ → ⦇l₂⦈ → [{l₂ : a | r}]
```

4.3.2 Undefined Values.

We handle undefined values using explicit Maybe types, thus giving correct types to all functions that manipulate optional values. Since the benchmark specification assumes that every single column of a table can contain missing values, the dropna function removes all rows in which any column does not have a value specified. Since in our system columns are non-optional by default, we present $\mathsf{dropna}_{\mathsf{column}}$, which operates on and eliminates the optional from a single column.

```
completeCases
  :: ∀r l a. [{l : Maybe a | r}] → ⦇l⦈ → [Bool]
dropna_column
  :: ∀r l a. [{l : Maybe a | r}] → ⦇l⦈ → [{l : a | r}]
fillna
  :: ∀r l a. [{l : Maybe a | r}] → ⦇l⦈ → a → [{l : a | r}]
```

4.3.3 Aggregations.

Aggregations are some of the most important functions in data analysis, since they allow the modeller to synthesize a fixed size answer from a large body of data. Here, we analyze the aggregations listed in the B2T2 benchmark.

The pivotTable function partitions rows into groups and summarizes each group with an aggregation function. We present its one-column version, as well as another one that replaces a sequence of per-column reducers with a reduction function from an array of records to a single record. This should be as capable as the original pivotTable.

$$\begin{aligned}
&\mathsf{groupBy}\\
&\quad :: \forall r\ r_2\ k\ v.\ [\{r\}] \to (\{r\} \to k) \to (\{r\} \to v) \to\\
&\qquad\qquad (k \to [v] \to \{r_2\}) \to [\{r2\}]\\
&\mathsf{groupByRetentive}\\
&\quad :: \forall r\ l\ a.\ [\{l : a \mid r\}] \to (\!|l|\!) \to\\
&\qquad\qquad [\{\mathsf{key} : a, \mathsf{groups} : [\{l : a \mid r\}]\}]\\
&\mathsf{groupBySubtractive}\\
&\quad :: \forall r\ l\ a.\ [\{l : a \mid r\}] \to (\!|l|\!) \to\\
&\qquad\qquad [\{\mathsf{key} : a, \mathsf{groups} : [\{r\}]\}]\\
&\mathsf{pivotTable}_{\mathsf{column}}\\
&\quad :: \forall r\ l_1\ l_2\ k\ a\ b.\ [\{k \mid l_1 : a \mid r\}] \to \langle k\rangle \to (\!|l_1|\!) \to\\
&\qquad\qquad (\!|l_2|\!) \to ([a] \to b) \to [\{k \mid l_2 : b\}]\\
&\mathsf{pivotTable}_{\mathsf{record}}\\
&\quad :: \forall r_1\ r_2\ k.\ [\{k \mid r_1\}] \to \langle k\rangle \to ([\{r_1\}] \to \{r_2\}) \to\\
&\qquad\qquad [\{k \mid r_2\}]
\end{aligned}$$

4.3.4 Joins. Another important class of table manipulation functions are joins, which collate the related data points from multiple tables into a single one. As before, our system can successfully assign types to most join-related functions, with the most interesting case being leftJoin. As specified in the benchmark, it assumes that all columns are nullable. Since this is not an assumption that is possible to express in our system, we suggest $\mathsf{leftJoin}_{\mathsf{maybe}}$ that adds the whole record of columns of the second table as an optional value, instead of concatenating both record types. Then, handling of missing cases in the resulting table can be handled explicitly, and once done the non-optional record type can be concatenated into the left table columns.

$$\begin{aligned}
&\mathsf{crossJoin} :: \forall r_1\ r_2.\ [\{r_1\}] \to [\{r_2\}] \to [\{r_1 \mid r_2\}]\\
&\mathsf{leftJoin}_{\mathsf{maybe}}\\
&\quad :: \forall r_1\ r_2\ k.\ [\{k \mid r_1\}] \to [\{k \mid r_2\}] \to \langle k\rangle \to\\
&\qquad\qquad [\{k \mid r_1 \mid \mathsf{joined} : \mathsf{Maybe}\ \{r2\}\}]\\
&\mathsf{groupJoin}\\
&\quad :: \forall r_1\ r_2\ r_3\ k.\ [\{r_1\}] \to [\{r_2\}] \to\\
&\qquad\qquad (\{r_1\} \to k) \to (\{r_2\} \to k) \to\\
&\qquad\qquad (\{r_1\} \to [\{r_2\}] \to \{r_3\}) \to [\{r_3\}]\\
&\mathsf{join}\\
&\quad :: \forall r_1\ r_2\ r_3\ k.\ [\{r_1\}] \to [\{r_2\}] \to\\
&\qquad\qquad (\{r_1\} \to k) \to (\{r_2\} \to k) \to\\
&\qquad\qquad (\{r_1\} \to \{r_2\} \to \{r_3\}) \to [\{r_3\}]
\end{aligned}$$

4.3.5 Flattening. $\mathsf{flatten}_{\mathsf{column}}$ flattens a single column at a time. As in the case of renameColumns flattening a number of columns simultaneously is equivalent to flattening them one by one, so little expressiveness is lost.

$$\mathsf{flatten}_{\mathsf{column}} :: \forall r\ l\ a.\ [\{l : [a] \mid r\}] \to (\!|l|\!) \to [\{l : a \mid r\}]$$

4.4 Static Checks and Errors Caught

Our system requires the specification of the table schema statically and uses it to verify the construction of a table, ruling out all of the malformed tables listed in the benchmark. Additionally, our type system is capable of verifying that columns are accessed correctly (missing names are rejected) and used according to the types listed in their schema. Both of those are sufficient to catch pretty much all of the example errors listed in the benchmark[14].

Most of the invariants used to annotate benchmark functions are proven statically, except for the array length equality checks, which we leave out of scope. Verification of those depends on the ability to model array length at the type level rather than on the record calculus, and is possible in many array languages (for example in [6, 14]).

We stress the value of types for illuminating the intermediate steps in the data transformations, for example, when columns are added, how column types change, etc. Often just by looking at the type signatures of the table interface, one can easily deduce the semantics of each function (i.e., *theorems for free* [19]). This is especially powerful when used in conjunction with proper development environments that, e.g., allow the programmer to query types on hover over identifiers.

4.5 Negative Results

Here, we present an analysis complementary to the previous sections, by listing all the benchmark functions that we have found to be difficult to assign a type or implementation to. We treat this list as a set of open problems for applying record types to data science programming, and hope to resolve them in the future, and for now only sketch potential solutions if known to us.

4.5.1 Table Pivots. The two functions we have found to be the most difficult to represent in our system are pivotLonger and pivotWider. They are each other's inverse, and convert between the standard "wide" table format to a "long" format, where a number of same-typed columns are replaced by two columns instead: one for the column name, the other for its value.

There are a number of issues that make those two functions difficult to represent. pivotLonger (the "wide" to "long" conversion) requires that all flattened columns are of the same type, which cannot be enforced on row variables in the current system. It additionally requires extra run-time reflection facilities to be able to extract column names (discussed further in §4.5.3). Finally, the order in which the new rows are to be produced is unclear, since table columns are unordered in our system. In the other direction, the main two difficulties with pivotWider are that it needs to construct records from unstructured column names and allow missing values in any of the flattened columns, since nothing guarantees that all their values are present in the long table.

[14] The list of errors caught is as follows: missingSchema, missingRow, missingCell, swappedColumns, schemaTooShort, schemaTooLong, midFinal, blackAndWhite, pieCount, brownGetAcne, favoriteColor, brownJellybeans

4.5.2 Typeclasses for Records. Many of the benchmark functions depend on records supporting a number of basic operations, such as equality checking (unique, leftJoin, ...) or having a hash function.

In many functional languages, such requirements are encoded as typeclass constraints. Try as we might, we have not been able to find a satisfactory way to integrate row-polymorphic records with existing typeclass systems. The main difficulty we encountered is that while record types are usually defined inductively, a single record type allows multiple equivalent inductive definitions. This means that typeclass constraint resolution for records is actually a non-deterministic procedure. There are typeclasses for which the order of derivation wouldn't change the eventual result. For example, when checking record equality (Eq in Haskell), it does not matter in which order we reduce the per-field equality decisions, because the $(\wedge, \mathsf{True})$ monoid is commutative. But, it is not straightforward for a compiler to prove this, and therefore be confident in typeclass coherence.

One can extract insight from the previous paragraph: we likely want to exploit the programmer's knowledge about certain algebraic structures being commutative monoids to ensure consistency over record type equivalence. But, we have not (yet) found a satisfactory way to express that in a real language, and leave this as future work.

4.5.3 Schema Reflection. Since our system lacks support for run-time reflection over table (or record) schemata, it is difficult to implement functions such as header. Another similar example is ncols, though support for typeclasses for records could help.

While out of scope for the B2T2 benchmark, this also makes record I/O difficult. It is convenient to order columns for printing, but without a more extensive constraint system there is no way to take a list of labels that are known to correspond to fields of a record/table.

5 Related Work

Record calculi. Following Rémy [17], we categorize record calculi into two groups based on how how record concatenation is supported: the *strict* group does not allow duplicate labels, while the *free* group does. In the strict group, record concatenation is *symmetric* (i.e., $\{r_1 \mid r_2\} \equiv \{r_2 \mid r_1\}$) [2, 5] where the type system must check that r_1 and r_2 are disjoint, often in a form of bounded quantification or qualified types. In the free group, record concatenation has been given different semantics: it can be *asymmetric* [1, 20] where concatenation overwrites a field if it is already present, or *scoped* (following [10]) where concatenation shadows existing labels, or *recursive* [13, 22] where common fields will recursively concatenate their values. Morris and McKinna [12] give a general account of record concatenation abstracting the interpretation of records, realized via qualified types. As we have seen, $\lambda^{\langle\rangle}$ supports scoped record concatenation, with a lightweight type system compared to most existing calculi, since $\lambda^{\langle\rangle}$ does not depend on qualified types or bounded polymorphism. However, it also means that certain programs with qualified type constraints cannot be expressed in $\lambda^{\langle\rangle}$. In particular, we can relatively easily express *positive* information, but not forms of *negative* information. That is, we could write r as $\{l : a \mid \rho\}$ to express the constraint "a record r has a label l". However, there is no direct way to say "a record r lacks a label l". As such, $\lambda^{\langle\rangle}$ alone cannot enforce distinct labels, or the "extend or overwrite" operator [20] that extends a record with a label, or overwrites its field if the label already exists in the record.

First-class labels in [9] are realized via singleton types, while they can also be naturally supported in record calculi based on dependent types such as in [2]. Following [9], $\lambda^{\langle\rangle}$ supports first-class labels using singleton types, and further generalizes the system to include first-class rows.

Tabular types. Tables are a widely used format for storing data. Lu et al. [11] proposed the B2T2 benchmark, whose purpose is to serve as "a focal point for research on type systems for tabular programming". Using record types to express tabular types is not new. Indeed, all record calculi discussed above can express some form of tabular types. Our evaluation (§4) shows that while relatively lightweight, $\lambda^{\langle\rangle}$ can express a significant number of functions in the benchmark using its unique combination of features. Recently, the dependently typed language Idris has been used to evaluate dependently-typed tables on B2T2 [21]. With full dependent types, their system is effectively more expressive, but arguably also more complex.

6 Conclusion

We have described a novel record calculus, that brings many previously described calculi together in a coherent way. The resulting type system is remarkably lightweight, as it requires no support for qualified types and only slightly extends well-established type checking and inference procedures. Finally, we show that the new record calculus can be used to successfully analyze the transformations of table schema in programs manipulating tabular data, as measured by the B2T2 benchmark.

While we do not provide a concrete implementation for our system yet, we have explored integrating it into Dex [14]. Beyond that, we hope that this document can also serve as an inspiration for other programming language designers wanting to improve data science applications.

References

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

[2] Adam Chlipala. 2010. Ur: Statically-Typed Metaprogramming with Type-Level Record Computation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 122–133. https://doi.org/10.1145/1806596.1806612

[3] Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 429–442. https://doi.org/10.1145/2500365.2500582

[4] Benedict R. Gaster and Mark P. Jones. 1996. *A Polymorphic Type System for Extensible Records and Variants*. Technical Report NOTTCS-TR-96-3. Department of Computer Science, University of Nottingham.

[5] Robert Harper and Benjamin Pierce. 1991. A Record Calculus Based on Symmetric Concatenation. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '91)*. Association for Computing Machinery, New York, NY, USA, 131–142. https://doi.org/10.1145/99583.99603

[6] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-Programming with Nested Parallelism and in-Place Array Updates. *SIGPLAN Not.* 52, 6 (jun 2017), 556–571. https://doi.org/10.1145/3140587.3062354

[7] Mark P Jones. 2003. *Qualified types: theory and practice*. Cambridge University Press.

[8] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of functional programming* 17, 1 (2007), 1–82. https://doi.org/10.1017/S0956796806006034

[9] Daan Leijen. 2004. *First-class labels for extensible rows*. Technical Report UU-CS-2004-051. Institute of Information and Computing Sciences, Utrecht University.

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

[11] Kuang-Chen Lu, Ben Greenman, and Shriram Krishnamurthi. 2022. Types for Tables: A Language Design Benchmark. *Art Sci. Eng. Program.* 6, 2 (2022), 8. https://doi.org/10.22152/programming-journal.org/2022/6/8

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

[13] Atsushi Ohori and Peter Buneman. 1988. Type Inference in a Database Programming Language. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming (LFP '88)*. Association for Computing Machinery, New York, NY, USA, 174–183. https://doi.org/10.1145/62678.62700

[14] Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. 2021. Getting to the Point: Index Sets and Parallelism-Preserving Autodiff for Pointful Array Programming. *Proc. ACM Program. Lang.* 5, ICFP, Article 88 (aug 2021), 29 pages. https://doi.org/10.1145/3473593

[15] François Pottier and Didier Rémy. 2005. The essence of ML type inference. In *Advanced Topics in Types and Programming Languages*. The MIT Press.

[16] Didier Rémy. 1992. Typing Record Concatenation for Free. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)*. Association for Computing Machinery, New York, NY, USA, 166–176. https://doi.org/10.1145/143165.143202

[17] Didier Rémy. 1994. *Type Inference for Records in Natural Extension of ML*. MIT Press, Cambridge, MA, USA, 67–95.

[18] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn (X) Modular type inference with local assumptions. *Journal of functional programming* 21, 4-5 (2011), 333–412. https://doi.org/10.1017/S0956796811000098

[19] Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*. Association for Computing Machinery, New York, NY, USA, 347–359. https://doi.org/10.1145/99370.99404

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

[21] Robert Wright, Michel Steuwer, and Ohad Kammar. 2022. Idris2-Table: evaluating dependently-typed tables with the Brown Benchmark for Table Types (Extended Abstract). TyDe 2022.

[22] Ningning Xie, Bruno C. d. S. Oliveira, Xuan Bi, and Tom Schrijvers. 2020. Row and Bounded Polymorphism via Disjoint Polymorphism. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs))*, Robert Hirschfeld and Tobias Pape (Eds.), Vol. 166. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 27:1–27:30. https://doi.org/10.4230/LIPIcs.ECOOP.2020.27

[23] Ningning Xie, Richard A. Eisenberg, and Bruno C. d. S. Oliveira. 2019. Kind Inference for Datatypes. *Proc. ACM Program. Lang.* 4, POPL, Article 53 (dec 2019), 28 pages. https://doi.org/10.1145/3371121

A Type Safety

Theorem 3.1 (Preservation). *If* $\Gamma \vdash e : \sigma$, *and* $e \longmapsto e'$, *then* $\Gamma \vdash e' : \sigma$.

Proof. By induction on the evaluation rule. Most rules are straightforward. We discuss below the rules specific to rows.

- rule (*prj*). Since the rule checks that we get the first field for l_{c_i}, it's guaranteed that we will get τ_i when the type of $\{l_{c_1} = v_1, \ldots, l_{c_n} = v_n\}$ is $\{l_{c_i} : \tau_i \mid \rho\}$.
- rule (*concat*). We can derive that the type $\{l_{c_1} : \tau_1, \ldots, l_{c_i} : \tau_i \mid l_{c_j} : \tau_j, \ldots, l_{c_n} : \tau_n\} \equiv \{l_{c_1} : \tau_1, \ldots, l_{c_i} : \tau_i, l_{c_j} : \tau_j, \ldots, l_{c_n} : \tau_n\}$ following Assoc.
- rule ($concat_r$). We can derive that the type $\langle l_{c_1} : \tau_1, \ldots, l_{c_i} : \tau_i \mid l_{c_j} : \tau_j, \ldots, l_{c_n} : \tau_n\rangle \equiv \langle l_{c_1} : \tau_1, \ldots, l_{c_i} : \tau_i, l_{c_j} : \tau_j, \ldots, l_{c_n} : \tau_n\rangle$ following Assoc.
- rule (del_1). We know that $\{\rho\} = \{\mathsf{Empty} \mid \rho\}$ by EMPL. So the goal follows.
- rule (del_2). In this case we know that $\{l_{c_1} = v_1, \ldots, l_{c_n} = v_n\}$ has type $\{l_{c_1} : \tau_1, \ldots, l_{c_n} : \tau_n\} \equiv \{l_{c_i} : \tau \mid \rho \mid \rho'\}$, and $\langle l_{c_i} : \tau \mid v\rangle$ has type $\langle l_{c_i} : \tau \mid \rho\rangle$ where $v : \langle\rho\rangle$. Thus $\{l_{c_1} = v_1, \ldots, l_{c_n} = v_n\} \backslash \langle l_{c_i} : \tau_i \mid v\rangle$ has type $\{\rho'\}$.
  Since the rule checks that we remove the first field for l_{c_i}, we know that $\{l_{c_1} = v_1, \ldots, l_{c_{i-1}} = v_{i-1}, l_{c_{i+1}} = v_{i+1}, \ldots, l_{c_n} = v_n\}$ is of type $\{l_{c_1} : \tau_1, \ldots, l_{c_{i-1}} : \tau_{i-1}, l_{c_{i+1}} : \tau_{i+1}, \ldots l_{c_n} : \tau_n\}$, namely $\{\rho \mid \rho'\}$.
  By the typing rule, we know that $\{l_{c_1} : \tau_1, \ldots, l_{c_{i-1}} : \tau_{i-1}, l_{c_{i+1}} : \tau_{i+1}, \ldots l_{c_n} : \tau_n\} \backslash v$ has type $\{\rho'\}$.

□

Theorem 3.2 (Progress). *If $\bullet \vdash e : \sigma$, then either e is a value, or there exists e' such that $e \longmapsto e'$.*

Proof. We first generalize the theorem statement, letting the context be non-empty as long as it only consists of type variables: if $\Gamma \vdash e : \sigma$, where Γ has only type variables, then either e is a value, or there exists e' such that $e \longmapsto e'$.

Now we proceed by induction on the typing rule. Most cases are straightforward. The purpose of the above generalization step to make the case Gen go through, and it won't affect any other cases.

Below we discuss the rules specific to rows.

- Empty. $\{\}$ is a value.
- Rcd. $\{e_1 = e_2\}$. If e_1 is not a value, then by I.H., $e_1 \longmapsto e_3$, namely $e_1 = E[e_1']$, and $e_1' \Longrightarrow e_3'$ and $E[e_1'] \longmapsto E[e_3']$. Therefore $(\{E = e_2\})[e_1'] \longmapsto (\{E = e_2\})[e_3']$.
  If e_1 is a value, then by typing it must be $e_1 = l_c$.
  If e_2 is not a value, then e_2 reduces.
  If e_2 is also a value v, then $\{l = v\}$ is also a value.
- Concat for $\{e_1 \mid e_2\}$. The case is similar to the above, except that when e_1 and e_2 are values, because of their types we know that $e_1 = \{l_{c_1} = v_1, \ldots, l_{c_i} = v_i\}$, and $e_2 = \{l_{c_j} = v_j, \ldots, l_{c_n} = v_n\}$, so by (*concat*) we have $\{e_1 \mid e_2\} \longmapsto \{l_{c_1} = v_1, \ldots, l_{c_i} = v_i, l_{c_j} = v_j, \ldots, l_{c_n} = v_n\}$.
- Lab. l_c is a value.
- Prj for $e_1.e_2$. The reasoning is similar to the case for Rcd. When both e_1 and e_2 are values, then it must be $e_1 = \{l_{c_1} = v_1, \ldots, l_{c_n} = v_n\}$ and $e_2 = l_{c_i}$ for some $1 \le i \le n$. So we reduce following (*prj*).
- EmptyR. $\langle\rangle$ is a value.
- Row for $\langle e : \tau \rangle$. If e reduces, then we can reason similarly to the case for Rcd. If e is a value, then it must be l_c, and $\langle l_c : \tau \rangle$ is a value.
- ConcatR. Similar to the case for Concat, except for the evaluation step we use rule ($concat_r$).
- PrjR. Simlar to the case for Prj except for the evaluation step we use the rule (prj_{r1}) or (prj_{r2}) according to whether the row is empty.
- Del for $e_1 \backslash e_2$. Again the reasoning is similar to case Concat. When both e_1 and e_2 are values, it must be that $e_1 = \{l_{c_1} = v_1, \ldots l_{c_n} = v_n\}$, and e_2 is a row value.
  If $e_2 = \langle\rangle$, then following (del_1) we have $e_1 \backslash e_2 \longmapsto e_1$.
  If $e_2 = \langle\langle l_{c_i} : \tau_i \rangle \mid v\rangle$, then $e_1 \backslash e_2$ reduces following (del_2).
- Inst. The goal follows from I.H..
- Gen. The goal follows from I.H.. □

B Unification Soundness

B.1 Context Extension

Lemma B.1 (Context Extension Transitivity). *If $\Delta_1 \rightsquigarrow \Delta_2$, and $\Delta_2 \rightsquigarrow \Delta_3$, then $\Delta_1 \rightsquigarrow \Delta_3$.*

Proof. We have:

$$\begin{aligned}
[\Delta_3]\tau &\equiv [\Delta_3]([\Delta_2]\tau) && (\Delta_2 \rightsquigarrow \Delta_3)\\
&\equiv [\Delta_3]([\Delta_2]([\Delta_1]\tau)) && (\Delta_1 \rightsquigarrow \Delta_2)\\
&\equiv [\Delta_3]([\Delta_1]\tau)) && (\Delta_2 \rightsquigarrow \Delta_3)
\end{aligned}$$

□

Lemma 3.6 (Unification extends context). *If $\Delta_1 \vdash C_1 \Longrightarrow \Delta_2 \vdash C_2$, then $\Delta_1 \rightsquigarrow \Delta_2$.*

Proof. By induction on $\Longrightarrow$. Most cases follow directly from I.H.. The two interesting cases are:

- (*Tsolve*). In this case we have Δ_2 being $\Delta_1 \circ \alpha : \kappa = [\Delta_1]\tau$, and our goal is to prove $[\Delta_2]\tau' \equiv [\Delta_2]([\Delta_1]\tau')$.
  We can do a case-analysis on τ', and the only interesting case is when $\tau' = \alpha$. Since we know $\alpha : \kappa \in \Delta_1$, we know that $[\Delta_1]\alpha = \alpha$. Therefore $[\Delta_2]([\Delta_1]\alpha) = [\Delta_2]\alpha$.
- (*Rfield*). For this one we can prove a similar lemma for lookup rules; namely, if $\Delta_1 \vdash \rho \overset{l_c}{\hookrightarrow} \rho_2 \dashv \Delta_2$, then $\Delta_1 \rightsquigarrow \Delta_2$.
  We do induction on $\overset{l_c}{\hookrightarrow}$, and the only interesting case is (*LUtail*), which is similar to (*Tsolve*) above. □

B.2 Solution Context

Lemma B.2 (Context Extension Preserves Solution Context). *If $\Delta_1 \models C$, and $\Delta_1 \rightsquigarrow \Delta_2$, then $\Delta_2 \models C$.*

Proof. For any $\tau_1 \sim \tau_2 \in C$, we have:

$$\begin{aligned}
[\Delta_2]\tau_1 &\equiv [\Delta_2]([\Delta_1]\tau_1) && (\Delta_1 \rightsquigarrow \Delta_2)\\
&\equiv [\Delta_2]([\Delta_1]\tau_2) && (\Delta_1 \models C)\\
&\equiv [\Delta_2]\tau_2 && (\Delta_1 \rightsquigarrow \Delta_2)
\end{aligned}$$

Similarly for rows and labels in C. □

Lemma B.3 (Constraint Solving Preserves Solution Context). *If $\Delta_1 \vdash C_1 \Longrightarrow \Delta_2 \vdash C_2$ and $\Delta_2 \rightsquigarrow \Delta$, then:*

1. *if $\Delta \models C_2$ then $\Delta \models C_1$; and*
2. *if $\Delta \models C_1$ then $\Delta \models C_2$.*

Proof. By Lemma 3.6, we can infer that $\Delta_1 \rightsquigarrow \Delta_2$. So by Lemma B.1, we also have $\Delta_1 \rightsquigarrow \Delta$.

(1) Since $\Delta \models C_2$, we know that for all $\tau_1 \sim \tau_2 \in C_2$, we have $[\Delta]\tau_1 \equiv [\Delta]\tau_2$.

By induction on $\Longrightarrow$. Most cases follow directly. We discuss the only interesting cases below.

- (*conj*). We have $\Delta \models C_1' \wedge C_2$. By I.H., we have $\Delta \models C_1$. Threfore $\Delta \models C_1 \wedge C_2$.
- (*Tapp*). We have $\Delta \models \tau_1 \sim \tau_3 \wedge \tau_2 \sim \tau_4$. Therefore $[\Delta]\tau_1 \equiv [\Delta]\tau_3$, and $[\Delta]\tau_2 \equiv [\Delta]\tau_4$. Thus $[\Delta](\tau_1\ \tau_2) \equiv [\Delta](\tau_3\ \tau_4)$. All other congruent rules are similar.
- (*Tsolved*). We have $[\Delta]\tau' \equiv [\Delta]\tau$. Therefore $[\Delta]\alpha \equiv [\Delta]([\Delta_1]\alpha) = [\Delta]\tau' \equiv [\Delta]\tau$. All other solved rules are similar.

- (*Tsolve*). We have $[\Delta]\alpha \equiv [\Delta]([\Delta_2]\alpha) = [\Delta]([\Delta_1]\tau) \equiv [\Delta]\tau$. All other solve rules are similar.
- (*Rfield*). For this case we need to first prove that: if $\Delta_1 \vdash \rho_1 \overset{l_c}{\hookrightarrow} \rho_2 \dashv \Delta_2$, then $[\Delta_2]\rho_2 = [\Delta_2]\rho_1$. This follows directly by induction on $\overset{l_c}{\hookrightarrow}$.
  Thus,
  $[\Delta](l_c : \tau, \rho_1) = l_c : [\Delta]\tau, [\Delta]\rho_1$
  $\equiv l_c : [\Delta]\tau', [\Delta]\rho_2' = [\Delta](l_c : \tau', \rho_2')$
  $= [\Delta]([\Delta_2](l_c : \tau', \rho_2')) = [\Delta]([\Delta_2]\rho_2)$
  $\equiv [\Delta]\rho_2$

(2) Since $\Delta \models C_1$, we know that for all $\tau_1 \sim \tau_2 \in C_1$, we have $[\Delta]\tau_1 \equiv [\Delta]\tau_2$.

By induction on $\Longrightarrow$. Most cases follow directly. We discuss the only interesting cases below.

- (*conj*). We have $\Delta \models C_1 \wedge C_2$. By I.H., we have $\Delta \models C_1'$. Therefore $\Delta \models C_1' \wedge C_2$.
- (*Tapp*). We have $\Delta \models \tau_1\ \tau_2 \sim \tau_3\ \tau_4$ Therefore $[\Delta](\tau_1\ \tau_2) \equiv [\Delta](\tau_3\ \tau_4)$. Thus $[\Delta]\tau_1 \equiv [\Delta]\tau_2$. and $[\Delta]\tau_2 \equiv [\Delta]\tau_4$. All other congruent rules are similar.
- (*Tsolved*). We have $[\Delta]\alpha \equiv [\Delta]\tau$. Therefore $[\Delta]\tau' \equiv [\Delta]([\Delta_1]\tau') = [\Delta]([\Delta_1]\alpha) = [\Delta](\alpha) \equiv [\Delta]\tau$. All other solved rules are similar.
- (*Tsolve*). $\Delta \models \top$ trivially. All other solve rules are similar.
- (*Rfield*). We have proven in the previous (*Rfield*) case that: if $\Delta_1 \vdash \rho_1 \overset{l_c}{\hookrightarrow} \rho_2 \dashv \Delta_2$, then $[\Delta_2]\rho_2 \equiv [\Delta_2]\rho_1$.
  Thus,
  $l_c : [\Delta]\tau, [\Delta]\rho_1$
  $= [\Delta](l_c : \tau, \rho_1) \equiv [\Delta]\rho_2$
  $\equiv [\Delta]([\Delta_2]\rho_2)$
  $\equiv [\Delta]([\Delta_2](l_c : \tau', \rho_2'))$
  $= l_c : [\Delta]([\Delta_2]\tau'), [\Delta]([\Delta_2]\rho_2')$
  $= l_c : [\Delta]\tau', [\Delta]\rho_2'$

  Thus $[\Delta]\tau \equiv [\Delta]\tau'$ and $[\Delta]\rho_1 \equiv [\Delta]\rho_2'$. □

B.3 Unification Soundness

Theorem 3.4 (Unification Soundness). *If $\Delta_1 \vdash C \Longrightarrow^* \Delta_2 \vdash \top$, then $\Delta_2 \models C$.*

Proof. We know $\Delta_2 \models \top$ trivially. By Lemma B.3, we deduce that $\Delta_2 \models C$. □

C Most General Unifier

Lemma C.1 (Constraint Solving Preserves the Most General Unifier). *If $\Delta_1 \rightsquigarrow \Delta$ such that $\Delta \models C_1$, and $\Delta_1 \vdash C_1 \Longrightarrow \Delta_2 \vdash C_2$, then there exists Δ' such that $\Delta_2 \rightsquigarrow \Delta'$ and $\Delta \rightsquigarrow \Delta'$.*

Proof. By induction on the derivation of $\Delta_1 \vdash C_1 \Longrightarrow \Delta_2 \vdash C_2$. Most cases follow directly. Below we discuss a few interesting cases.

- (*Tsolve*). We are given $\Delta_1 \rightsquigarrow \Delta$ and $\Delta \models \alpha \sim \tau$.
  We know $\Delta_2 = (\Delta_1 \circ \alpha : \kappa = [\Delta_1]\tau)$, and we want to prove that there exists Δ' such that $\Delta \rightsquigarrow \Delta'$ and $\Delta_2 \rightsquigarrow \Delta'$.
  Let Δ' be Δ. Now we need to prove $[\Delta]\tau' \equiv [\Delta]([\Delta_2]\tau')$ for all τ' (similarly for ρ and ℓ).
  We do induction on τ'. Most cases follow by I.H.. The only interesting case is when $\tau' = \alpha$.
  $[\Delta]\alpha$
  $\equiv [\Delta]\tau$ $(\Delta \models \alpha \sim \tau)$
  $\equiv [\Delta]([\Delta_1]\tau)$ $(\Delta_1 \rightsquigarrow \Delta)$
  $= [\Delta]([\Delta_2]\alpha)$ $(\Delta_2 = (\Delta_1 \circ \alpha : \kappa = [\Delta_1]\tau))$
  Therefore, $\Delta_2 \rightsquigarrow \Delta$.
- (*Rfield*). If $\Delta_1 = \Delta_2$, then the goal follows directly.
  However, if $\Delta_1 \neq \Delta_2$, then it must be $\Delta_2 = \Delta_1 \circ \alpha : \mathrm{Row} = (l_c : \beta, \gamma), \beta : \star, \gamma : \mathrm{Row}$.
  We first prove that (*): if $\Delta_1 \rightsquigarrow \Delta$, and $\Delta \models \rho_1 \sim \rho_2$ and $\Delta_1 \vdash \rho_1 \overset{l_c}{\hookrightarrow} \rho_2 \dashv \Delta_2$, then $\Delta_2 \rightsquigarrow \Delta$.
  We do induction on $\overset{l_c}{\hookrightarrow}$ and most cases follow directly.
  The case for (*LUtail*) is simiar to (*Tsolve*) above.
  Let Δ' be $\Delta, \beta : \star = [\Delta]\tau, \gamma : \mathrm{Row} = [\Delta]\rho_1$. Therefore $\Delta \rightsquigarrow \Delta'$. Therefore by Lemma B.1, $\Delta_1 \rightsquigarrow \Delta'$.
  Now we show that
  $[\Delta']([\Delta_1]\rho_2)$
  $\equiv [\Delta']([\Delta]([\Delta_1]\rho_2))$ $(\Delta \rightsquigarrow \Delta')$
  $\equiv [\Delta']([\Delta]\rho_2)$ $(\Delta_1 \rightsquigarrow \Delta)$
  $\equiv [\Delta']([\Delta](l_c : \tau, \rho_1))$ $(\Delta \models l_c : \tau, \rho_1 \sim \rho_2)$
  $\equiv [\Delta'](l_c : \tau, \rho_1)$ $(\Delta \rightsquigarrow \Delta')$
  $= l_c : [\Delta']\tau, [\Delta']\rho_1$
  $= l_c : [\Delta]\tau, [\Delta]\rho_1$ $(\beta, \gamma$ are fresh$)$
  $= l_c : [\Delta']\beta, [\Delta']\gamma$ (definition of Δ')
  $= [\Delta'](l_c : \beta, \gamma)$
  Therefore $\Delta' \models [\Delta_1]\rho_2 \sim (l_c : \beta, \gamma)$.
  Therefore by the lemma (*) we proved above we have $\Delta_2 \rightsquigarrow \Delta'$. □

Theorem 3.7 (Unification Produces the Most General Unifier). *If $\Delta_1 \vdash C \Longrightarrow^* \Delta_2 \vdash \top$, then for any Δ such that $\Delta_1 \rightsquigarrow \Delta$ and $\Delta \models C$, there exists Δ' such that $\Delta_2 \rightsquigarrow \Delta'$ and $\Delta \rightsquigarrow \Delta'$.*

Proof. We do induction on $\Longrightarrow^*$.

- The base case is $\Delta_1 \vdash C \Longrightarrow \Delta_1 \vdash C$. Let Δ' be Δ and we are done.
- The inductive case is $\Delta_1 \vdash C \Longrightarrow C' \dashv \Delta_1'$ and $\Delta_1' \vdash C' \Longrightarrow^* \Delta_2 \vdash \top$.
  We are given $\Delta_1 \rightsquigarrow \Delta$, and $\Delta \models C$. By Lemma C.1, we know that there exists Δ' such that $\Delta_1' \rightsquigarrow \Delta'$, and $\Delta \rightsquigarrow \Delta'$.
  By Lemma B.3, we have $\Delta \models C'$. By Lemma B.2, we have $\Delta' \models C$.
  By I.H., there exists Δ'' such that $\Delta' \rightsquigarrow \Delta''$ and $\Delta_2 \rightsquigarrow \Delta''$.
  By Lemma B.1, we have $\Delta \rightsquigarrow \Delta''$.
  Therefore Δ'' is what we need. □

Received 2023-06-01; accepted 2023-06-29