Revisiting Row Polymorphism for Set-Theoretic Types

MICKAËL LAURENT, Charles University, Czech Republic
PIERRE DONAT-BOUILLUD, Czech Technical University, Czech Republic
FILIP KŘIKAVA, Czech Technical University, Czech Republic
JAN VITEK, Charles University, Czech Republic

Set-theoretic types support expressive record types through unions, intersections, and negations, but they lack the row polymorphism needed to type operations that propagate unknown fields across records. Prior work addresses this by allowing Boolean combinations of rows in type substitutions, which complicates the formalism and prevents the tallying algorithm from being complete. We propose an alternative: instead of enriching substitutions, we allow Boolean combinations of row variables directly within record type constructors, where the tail of a record has the same shape as any field. This design keeps substitutions simple—a row variable maps to a single row—and yields a natural extension of the subtyping and tallying algorithms. Tallying is complete for all solutions whose rows are constant over labels not mentioned in the constraints. We implement our approach in the set-theoretic type library SSTT and the type checker MLsem, providing the first implementation of a type system that combines semantic subtyping with row polymorphism. We demonstrate the expressiveness of the system by encoding several data structures from the R programming language: heterogeneous lists, variadic function arguments, and class-based dispatch.

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

Additional Key Words and Phrases: set-theoretic types, row polymorphism, semantic subtyping, tallying

1 Introduction

Dynamically-typed languages such as R, JavaScript, and Python make heavy use of records (objects, dictionaries, data frames) whose fields are created, extended, and combined at run time. Typing such operations precisely requires *row polymorphism*: the ability to abstract over the unknown fields of a record so that operations which add, remove, or update fields can propagate type information about the fields they do not mention [15, 18].

Set-theoretic types offer a natural framework for typing dynamic languages. They interpret types as sets of values and define subtyping as set containment, giving unions, intersections, and negations their standard meaning. Record types in this framework are expressive—one can state, for example, that a value is a record with a field ℓ of type `int` *or* a closed record with no fields—but they lack row polymorphism. Without it, common operations cannot be typed precisely. For instance, consider a function `add_id` that takes a record and returns a copy with a new field `id` of type `int`. The best monomorphic type, $\{\ ..\} \to \{\texttt{id} : \texttt{int}\ ..\}$ (where .. denotes an open record—it may contain any other field), loses all information about the fields already present in the input. With row polymorphism, we can write $\{\texttt{id} : \mathbb{1}? \mid \rho\} \to \{\texttt{id} : \texttt{int} \mid \rho\}$[1], which says that every field other than `id` is preserved.

Bringing row polymorphism into the set-theoretic setting raises a difficulty that does not arise in classical type systems. Polymorphic set-theoretic type systems rely on *tallying* [5], a constraint-solving algorithm that finds all type substitutions satisfying a set of subtyping constraints. When

[1]We note $\mathbb{1}$ (*any*) the top type, and $\mathbb{0}$ (*empty*) the bottom type. Moreover, $\mathbb{1}?$ is an *option type*: it denotes a field that may either be absent or have type $\mathbb{1}$.

Authors' Contact Information: Mickaël Laurent, mickael.laurent@matfyz.cuni.cz, Charles University, Prague, Czech Republic; Pierre Donat-Bouillud, pierre.donat.bouillud@fit.cvut.cz, Czech Technical University, Prague, Czech Republic; Filip Křikava, filip.krikava@fit.cvut.cz, Czech Technical University, Prague, Czech Republic; Jan Vitek, vitekj@me.com, Charles University, Prague, Czech Republic.

record types carry row variables, tallying must solve constraints such as $\{\ |\ \rho_1\} \dot{\leq} \{\ |\ \rho_2\} \wedge \{\ |\ \rho_3\}$, where ρ_1 is to be instantiated and ρ_2, ρ_3 are fixed. No single row for ρ_1 can satisfy both conjuncts simultaneously. The recent work of [7] addresses this by allowing substitutions to map row variables to Boolean combinations of rows. While expressive, this choice complicates the formalism—applying a substitution to a record constructor may produce a Boolean combination of constructors—and makes tallying incomplete: no algorithm can enumerate all solutions.

Key idea. We take a different path. Instead of enriching substitutions, we allow Boolean combinations of row variables to appear directly inside record type constructors. In our formulation, a record type $\{\ell_1 : f_1, \ \ldots, \ \ell_n : f_n \mid f\}$ maps each label to a *field type* f—an arbitrary Boolean combination of option types and row variables—and the tail f has the same shape as any field. Substitutions remain simple: a row variable maps to a single row, never to a Boolean combination of rows. This design makes it possible to type operations that cannot be expressed with a single row variable in tail-position. For instance, a function `mix` that takes two records and, for each label, non-deterministically selects the value from one argument or the other can be typed as:

$$\texttt{mix} : \{\ |\ \rho_1\} \to \{\ |\ \rho_2\} \to \{\ |\ \rho_1 \vee \rho_2\}$$

The union $\rho_1 \vee \rho_2$ in the tail captures the fact that each field of the result has the type it has in the first argument, or in the second. This behavior could not be expressed with parametric polymorphism alone: the type $\alpha \to \beta \to \alpha \vee \beta$ (with record constraints on α and β) would only allow returning one of the two input records wholesale, not a field-by-field mix.

Results. We formalize this approach and show that the subtyping and tallying algorithms extend naturally. The subtyping algorithm reduces record subtyping to subtyping over field types, which in turn reduces to subtyping over option types and row-variables. The tallying algorithm is complete for all solutions whose rows are constant over labels not mentioned in the constraints; a first version that finds only constant row solutions is presented, then we show that the general case reduces to this restricted one by a renaming transformation. We implement our approach in the set-theoretic type library SSTT [13] and extend the type checker MLsem [12] to use it. We demonstrate the expressiveness of the system by encoding several data structures from the R programming language—heterogeneous lists, variadic function arguments, and class-based dispatch—all within the existing type algebra, without any special-purpose extensions.

Contributions.

- A formalization of record types whose fields and tails are Boolean combinations of row variables and option types, together with a set-theoretic semantics that interprets records as quasi-constant functions from labels to field values (Section 3).
- An extension of the subtyping algorithm to decide subtyping over these record types, and an extension of the tallying algorithm that is complete for all solutions whose rows are constant outside a given finite set of labels (Section 4).
- Encodings of R's heterogeneous lists, variadic function arguments, and S3 class dispatch as record types with row polymorphism, demonstrating that these structures can be typed without extending the underlying type algebra (Section 5.1).
- An implementation within SSTT and MLsem, providing the first type checker that combines semantic subtyping with row polymorphism (Section 5.2).

2 Related work

Row polymorphism was introduced by Wand [18] to type record concatenation and multiple inheritance, and later refined by Rémy and Pottier [15] in the context of ML type inference. In the classical setting, a row variable stands in the tail position of a record type and abstracts over the unknown fields; substitutions replace row variables by rows, and the theory integrates smoothly with Hindley–Milner inference. Extensible records have since been studied multiple times. In a recent work, Toohey, Chen, Jamalzadeh and Xie [16] revisit row-polymorphism in the presence of *type classes*. The challenge addressed in this paper is bringing row polymorphism into a set-theoretic type algebra, where types are interpreted as sets of values and subtyping is set containment.

Benzaken, Castagna, Frisch [2, 10]. They introduce a set-theoretic type algebra with extensible records. They represent record values as quasi-constant functions from a set of labels to (optional) values: we use a similar representation in the present paper, where record values are quasi-constant functions from labels to *field values* that can feature row variables. Their record types can be of two kinds: closed record types ($\{\ell_1 : t_1, ..., \ell_n : t_n\}$) that capture records with exactly the fields $\ell_1, ..., \ell_n$, and open record types ($\{\ell_1 : t_1, ..., \ell_n : t_n \;..\}$) that capture records with at least the fields $\ell_1, ..., \ell_n$. They do not address polymorphism: the set-theoretic algebra they define does not feature type variables (and, needless to say, row variables).

Castagna [3]. Castagna extends set-theoretic types with record atoms whose labels are regular values of the language (integers, pairs, etc.). Record types can specify bindings ("fields") of the form $k \Rightarrow t$, where k can be chosen from a restricted set of types (e.g. `int`, `bool`, `string`). In this context, the tail is just a regular field $\_ \Rightarrow t$ where $\_$ is a special label that denotes all the values not captured by another field. Consequently, the tail is generalized to have the same shape as a field, but is constrained to always contain Ω (the absent marker), so that every record has a finite domain. This work does not address row polymorphism.

Castagna and Peyrot [7]. The closest work to ours. Castagna and Peyrot bring row polymorphism to set-theoretic records by extending the tail ζ of record types with a third kind: in addition to closed records (tail ϵ) and open records (tail $..$), they introduce polymorphic records whose tail is a row variable ρ. A row $\langle \ell : t, ..., \ell : t \mid \zeta \rangle$ has a similar shape as a record atom $\{\ell : t, ..., \ell : t \mid \zeta\}$. Rows can be of different kinds depending on the labels they define: a row is *complete* when it covers the set of all labels, and *incomplete* otherwise. In contrast, all our rows are complete—they always cover the set of all labels. In their formalism, the tail of a record cannot be a Boolean combination of row variables, but substitutions can map a row variable to any Boolean combination of rows (i.e. unions, intersections, and negations of rows). This makes substitutions technically more powerful than ours: applying a substitution to a record constructor $\{\ell : t, ..., \ell : t \mid \rho\}$ may produce a Boolean combination of record constructors rather than a single record. However, this additional power comes at a cost: their tallying algorithm is not complete, meaning there is no algorithm that can enumerate all solutions. In our approach, we move the complexity from the substitutions to the record constructors: field types (and tails) can be arbitrary Boolean combinations of row variables and option types, but substitutions remain simple—a row variable maps to a single row. This yields a tallying algorithm that is complete for all solutions whose rows are constant over labels not mentioned in the constraints. No implementation is given for the formalism and algorithms defined in [7], thus we cannot provide a practical comparison based on concrete type-checking examples. However, we illustrate how the two approaches differ in the theory in Section 4.4.

Approaches based on algebraic subtyping. Approaches based on algebraic subtyping also feature unions and intersections of types, but subtyping is defined through a set of syntactic rules (or equivalently, a set of algebraic properties). In a recent work, Chau and Parreaux [9] introduce Boolean-algebraic subtyping (BAS), a subtyping approach enabling backtracking-free principal type inference. They do not need row-polymorphism, as the corresponding programming patterns (e.g. field deletion and update) can already be expressed in their algebra using intersections, unions and negations: for instance, a record type t to which the field ℓ has been removed can be expressed as the union $t \sqcup \neg\{\ell : \bot\}$. However, this relies on specific properties of their algebra, such as $\top \leq \{\ell_1 : t_1\} \sqcup \{\ell_2 : t_2\}$ for any t_1, t_2, and $\ell_1 \neq \ell_2$. Their type algebra enables fast principal type inference, but it fails to capture some patterns of dynamic languages: for instance, they cannot use intersections of arrow types to express overloaded behaviors (e.g. $(\texttt{int} \to \texttt{int}) \wedge (\neg\texttt{int} \to \texttt{nil})$ for a function mapping integers to integers, and any other value to `nil`).

3 Types

3.1 Syntax and substitutions

Our definition of types follows that introduced by [10] and later extended with type variables [6]. We extend it with record atoms whose fields can feature row variables.

Definition 3.1 (Set-theoretic types). The sets $\mathcal{T}$ of *set-theoretic types*, $\mathcal{F}$ of *field types*, and $\mathcal{R}$ of rows are the sets of regular and contractive terms t, f, and R coinductively defined by the following grammar:

Types	t, s	$::=$	$b \mid \alpha \mid t \to t \mid t \times t \mid r \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1}$
Option Types	$\bar{t}, \bar{s}$	$::=$	$t \mid t?$
Field Types	f	$::=$	$\bar{t} \mid \rho \mid f \wedge f \mid f \vee f \mid \neg f$
Record Bindings	$\vec{B}$	$::=$	$\ell : f,\ \ldots,\ \ell : f$ (with all ℓ distinct)
Record Atoms	r	$::=$	$\{\vec{B} \mid f\}$
Rows	R	$::=$	$\langle \vec{B} \mid f \rangle$

where $b \in \mathcal{B}$ is a base type, $\alpha \in \mathcal{V}_{\mathsf{Ty}}$ is a type variable, and $\rho \in \mathcal{V}_{\mathsf{Row}}$ is a row variable. The notation $t_1 \setminus t_2$ is a syntactic sugar for $t_1 \wedge \neg t_2$. When writing a term, we use the following precedence (by decreasing priority): $\neg$, $\setminus$, $\wedge$, $\vee$, $\times$, $\to$. Syntactic equality between two types t_1 and t_2 is noted $t_1 = t_2$.

The set $\mathcal{B}$ of base types, the set $\mathcal{V}_{\mathsf{Ty}}$ of type variables and the set $\mathcal{V}_{\mathsf{Row}}$ of row variables are fixed. For each constant of the language, $\mathcal{B}$ contains the associated *singleton type*. It may also contain some non-singleton types like `bool` and `int`.

As they are defined coinductively, types can be infinite trees, provided that they satisfy the constraints of regularity and contractivity explained below. This yields a definition of equirecursive types that does not require explicit binders for recursion.

A term is said *regular* if it only has a finite number of distinct subterms; this constraint ensures decidability of the subtyping relation. A term is said *contractive* if every infinite branch goes through an infinite number of arrows and products ($\to$ and $\times$); this ensures that every type has a meaningful interpretation: for instance, it prevents expressing types such as the one satisfying the equation $t = \neg t$.

The type $\mathbb{0}$ is a special type that has no inhabitant, and is the subtype of all types. Conversely, the type $\mathbb{1}$ is the supertype of all types. A type variable α is a symbolic marker that can be substituted by any type t (*cf.* Definition 3.3).

The $\times$ constructor is used to type pairs of our language. Intuitively, it corresponds to the Cartesian product of two types. In particular, the product $\mathbb{1} \times \mathbb{1}$ is the supertype of all pairs: any well-typed pair can be typed with $\mathbb{1} \times \mathbb{1}$.

The $\to$ constructor is used to type functions (i.e., λ-abstractions). Intuitively, a λ-abstraction has type $t_1 \to t_2$ if and only if it accepts as argument a value of type t_1, in which case either it yields a value of type t_2 or it diverges. The type $\mathbb{0} \to \mathbb{1}$ is the supertype of all functions.

Finally, a record atom r is composed of some explicit bindings $\vec{B}$ followed by a tail. The bindings describe a finite set of labels, while the tail describes all remaining labels: intuitively, the atom

$\{\ |\ f\}$ is equivalent to $\{\ell_1 : f,\ \ell_2 : f,\ \dots\}$ where $\{\ell_1, \ell_2, \dots\}$ is the set of all labels $\mathcal{L}$. A label is associated with a *field type* f, which consists in a Boolean combination of option types and row variables. An option type $\bar{t}$ is either a type t (the field *must be present* and hold a value of type t) or an optional type $t?$ (the field is either *absent* or it holds a value of type t). A row variable ρ is a symbolic marker that can be substituted by any row R (*cf.* Definition 3.3). Rows R have a similar shape as record atoms (though we represent them under angle brackets $\langle.\rangle$), but they do not have the same use: record atoms are types while rows are mappings from labels to field types to be used in type substitutions.

Note that the record type $\neg\{\ |\ \rho\}$ is not the same as $\{\ |\ \neg\rho\}$: intuitively, the former captures the set of all records that are disjoint with the row captured by ρ on at least one field, while the latter captures the set of all records that are disjoint with the row captured by ρ on every field. Intuitively, the latter is equivalent to $\{\ell_1 : \neg\rho,\ \ell_2 : \neg\rho,\ \dots\}$ where $\{\ell_1, \ell_2, \dots\}$ is the set of all labels $\mathcal{L}$.

Definition 3.2 (Labels, tail, projection). Let $R = \langle\vec{B}\ |\ f\rangle$ be a row and $r = \{\vec{B}\ |\ f\}$ be a record atom. We define the labels of r, written $\mathsf{labels}(r)$ (resp. the labels of R, written $\mathsf{labels}(R)$) and the tail of r, written $\mathsf{tl}(r)$ (resp. the tail of R, written $\mathsf{tl}(R)$), as well as the projection of r on a label ℓ, written $r(\ell)$ (resp. the projection of R on a label ℓ, written $R(\ell)$) as follows:

$$\begin{array}{ccccc} \mathsf{labels}(R) & = & \mathsf{labels}(r) & = & \{\ell \mid (\ell, f) \in \vec{B}\} \\ \mathsf{tl}(R) & = & \mathsf{tl}(r) & = & f \end{array} \qquad R(\ell) = r(\ell) = \begin{cases} f_\ell & \text{if } (\ell : f_\ell) \in \vec{B} \\ f & \text{otherwise} \end{cases}$$

Definition 3.3 (Type substitution). A *type substitution* is a function $\phi : \mathcal{V}_{\mathsf{Ty}} \cup \mathcal{V}_{\mathsf{Row}} \to \mathcal{T} \cup \mathcal{R}$ mapping type variables to types and row variables to rows, and which is the identity everywhere except for a finite set of type variables and row variables called its domain and denoted by $\mathsf{dom}(\phi)$. Formally, $\mathsf{dom}(\phi) = \{\alpha \in \mathcal{V}_{\mathsf{Ty}} \mid \phi(\alpha) \neq \alpha\} \cup \{\rho \in \mathcal{V}_{\mathsf{Row}} \mid \phi(\rho) \neq \langle\ |\ \rho\rangle\}$. We define $\mathsf{labels}(\phi)$ as the set of explicit labels in the rows of ϕ; formally, $\mathsf{labels}(\phi) = \bigcup_{\rho \in \mathsf{dom}(\phi)} \mathsf{labels}(\phi(\rho))$. We note $\{(\rho_i \rightsquigarrow R_i)_{i \in I}, (\alpha_j \rightsquigarrow t_j)_{j \in J}\}$ the substitution mapping each ρ_i to R_i, each α_j to t_j, and that is the identity everywhere else. The set of all type substitutions is written $\mathcal{S}$.

Definition 3.4 (Application of a type substitution on a type). The result of the application of a type substitution ϕ on a type t, noted $t\phi$, as well as the application of a type substitution ϕ on a field type f in tail-position or ℓ-position, noted $(t\phi)_{\mathsf{tl}}$ or $(t\phi)_\ell$, are coinductively defined by these equations:

$$\begin{array}{ccc} \alpha\phi = \phi(\alpha) & \mathbb{0}\phi = \mathbb{0} & \mathbb{1}\phi = \mathbb{1} \\ (\neg t)\phi = \neg(t\phi) & (t_1 \vee t_2)\phi = (t_1\phi) \vee (t_2\phi) & (t_1 \wedge t_2)\phi = (t_1\phi) \wedge (t_2\phi) \\ b\phi = b & (t_1 \to t_2)\phi = (t_1\phi) \to (t_2\phi) & (t_1 \times t_2)\phi = (t_1\phi) \times (t_2\phi) \end{array}$$

$$r\phi = \{(\ell : (r(\ell)\phi)_\ell)_{\ell \in \mathsf{labels}(r)},\ (\ell : (\mathsf{tl}(r)\phi)_\ell)_{\ell \in \mathsf{labels}(\phi) \setminus \mathsf{labels}(r)} \mid (\mathsf{tl}(r)\phi)_{\mathsf{tl}}\}$$

$$\begin{array}{ccc} (\rho\phi)_\ell = \phi(\rho)(\ell) & (t\phi)_\ell = t\phi & ((t?)\phi)_\ell = (t\phi)? \\ ((\neg f)\phi)_\ell = \neg(f\phi)_\ell & ((f_1 \wedge f_2)\phi)_\ell = (f_1\phi)_\ell \wedge (f_2\phi)_\ell & ((f_1 \vee f_2)\phi)_\ell = (f_1\phi)_\ell \vee (f_2\phi)_\ell \\ (\rho\phi)_{\mathsf{tl}} = \mathsf{tl}(\phi(\rho)) & (t\phi)_{\mathsf{tl}} = t\phi & ((t?)\phi)_{\mathsf{tl}} = (t\phi)? \\ ((\neg f)\phi)_{\mathsf{tl}} = \neg(f\phi)_{\mathsf{tl}} & ((f_1 \wedge f_2)\phi)_{\mathsf{tl}} = (f_1\phi)_{\mathsf{tl}} \wedge (f_2\phi)_{\mathsf{tl}} & ((f_1 \vee f_2)\phi)_{\mathsf{tl}} = (f_1\phi)_{\mathsf{tl}} \vee (f_2\phi)_{\mathsf{tl}} \end{array}$$

As an example, consider the following application of a type substitution:

$$\{\ell_1 : \mathtt{int} \mid \rho_1 \wedge \rho_2\}\phi \quad \text{where} \quad \phi = \{\rho_1 \rightsquigarrow \langle\ell_1 : t_1',\ \ell_2 : t_2' \mid \rho_1'\rangle\}$$

Unsurprisingly, the result of this application is a record atom whose field type for ℓ_1 is $(\mathtt{int})\phi$ (which is `int`). As for the field ℓ_2, it is not explicit in the initial record atom, but as the tail defines the field type associated with every label except the explicit ones, we know that it has type $\rho_1 \wedge \rho_2$ before the substitution. Row variables that are in a field position (i.e. not in the tail) should be interpreted as the projection on that field of the row it captures. Thus, our field type for ℓ_2 should be understood as $(\rho_1.\ell_2) \wedge (\rho_2.\ell_2)$. Applying ϕ to this field type yields $t_2' \wedge (\rho_2.\ell_2)$, or more concisely, $t_2' \wedge \rho_2$. Pursuing this reasoning, the result of the application is as follows:

$$\begin{aligned}
&\mathbf{\{}\ell_1 : \mathtt{int} \mid \rho_1 \wedge \rho_2\mathbf{\}}\{\rho_1 \rightsquigarrow \langle \ell_1 : t_1',\ \ell_2 : t_2' \mid \rho_1' \rangle\} \\
={}& \mathbf{\{}\ell_1 : \mathtt{int},\ \ell_2 : \rho_1 \wedge \rho_2 \mid \rho_1 \wedge \rho_2\mathbf{\}}\{\rho_1 \rightsquigarrow \langle \ell_1 : t_1',\ \ell_2 : t_2' \mid \rho_1' \rangle\} && \text{(expliciting field } \ell_2\text{)} \\
={}& \mathbf{\{}\ell_1 : \mathtt{int},\ \ell_2 : (\rho_1.\ell_2) \wedge (\rho_2.\ell_2) \mid \rho_1 \wedge \rho_2\mathbf{\}}\{\rho_1 \rightsquigarrow \langle \ell_1 : t_1',\ \ell_2 : t_2' \mid \rho_1' \rangle\} && \text{(expliciting projections)} \\
={}& \mathbf{\{}\ell_1 : \mathtt{int},\ \ell_2 : t_2' \wedge (\rho_2.\ell_2) \mid \rho_1' \wedge \rho_2\mathbf{\}} && \text{(applying substitution)} \\
={}& \mathbf{\{}\ell_1 : \mathtt{int},\ \ell_2 : t_2' \wedge \rho_2 \mid \rho_1' \wedge \rho_2\mathbf{\}} && \text{(implicit projections)}
\end{aligned}$$

Similarly, we can define the application of a type substitution on a row:

Definition 3.5 (Application of a type substitution on a row). The result of the application of a type substitution ϕ on a row R, noted $R\phi$, is defined as follows:

$$R\phi = \langle (\ell : (R(\ell)\phi)_\ell)_{\ell \in \mathsf{labels}(R)},\ (\ell : (\mathsf{tl}(R)\phi)_\ell)_{\ell \in \mathsf{labels}(\phi) \setminus \mathsf{labels}(R)} \mid (\mathsf{tl}(R)\phi)_{\mathsf{tl}} \rangle$$

While the formal definition of the application of a type substitution on a record atom (or a row) may seem complicated, it only aims to satisfy this simple property:

Proposition 3.6. *The record atom $r\phi$ is such that for every label ℓ, we have $(r\phi)(\ell) = (r(\ell)\phi)_\ell$. Similarly, the row $R\phi$ is such that, for every label ℓ, we have $(R\phi)(\ell) = (R(\ell)\phi)_\ell$.*

We can now define the composition of two type substitutions:

Definition 3.7 (Type substitution composition). Given two type substitutions ϕ_1 and ϕ_2, their composition $\phi_2 \circ \phi_1$ is the substitution defined as follows:

$$\forall \alpha \in \mathcal{V}_{\mathsf{Ty}}.\ (\phi_2 \circ \phi_1)(\alpha) = \begin{cases} (\phi_1(\alpha))\phi_2 & \text{if } \alpha \in \mathsf{dom}(\phi_1) \\ \phi_2(\alpha) & \text{if } \alpha \in \mathsf{dom}(s_2) \setminus \mathsf{dom}(s_1) \\ \alpha & \text{otherwise} \end{cases}$$

$$\forall \rho \in \mathcal{V}_{\mathsf{Row}}.\ (\phi_2 \circ \phi_1)(\rho) = \begin{cases} (\phi_1(\rho))\phi_2 & \text{if } \rho \in \mathsf{dom}(\phi_1) \\ \phi_2(\rho) & \text{if } \rho \in \mathsf{dom}(s_2) \setminus \mathsf{dom}(s_1) \\ \langle \mid \rho \rangle & \text{otherwise} \end{cases}$$

Up to this point, types are only syntactic terms, on which we define syntactic operations (in particular, the application of a type substitution). In the next section, we define a *semantic interpretation* for types, characterize our syntactic type substitution in terms of this semantic interpretation, and use it to define a notion of subtyping.

3.2 Set-theoretic interpretation

The main idea of set-theoretic types is to interpret each type as a set of values of our language, and then use this interpretation to define subtyping as set containment. Intuitively, each type is associated with the set of values having this type: for instance, the base type `true` may be interpreted as the singleton containing the constant `true`, while the type `bool` = `true` ∨ `false` is interpreted as the set {`true`, `false`}.

However, this idea becomes subtler when dealing with arrow types. Although an arrow type intuitively corresponds to a function (i.e., a λ-abstraction), interpreting an arrow type as a set of λ-abstractions is problematic as it yields a circular reasoning: determining if a λ-abstraction is in the interpretation of a type requires to define a type system, which in turns needs the subtyping relation that we are trying to build. In order to break this circularity, the interpretation of types is not defined over values of our language but over a domain $\mathcal{D}$ defined below. Note that this does not necessarily invalidate the "types as set of values" intuition, as it is discussed in [4, Section 2.7].

In addition, we need to define an interpretation for all types, and not only ground ones (the interpretation domain $\mathcal{D}$ should account for type variables). A simple model was proposed by [11]. In this section, we extend this model with an interpretation for our record types. For that, we take the same approach as Alain Frisch [10], considering record values to be quasi-constant functions from labels to values: they are constant on all labels but a finite number of them.

Definition 3.8 (Quasi-constant functions). Let X, Y denote two sets. A function $F : X \to Y$ is quasi-constant if there exists $y \in Y$, called the *default value* of F, denoted by $\mathsf{def}(F)$, such that the set $\mathsf{dom}(F) = \{x \in X \mid F(x) \neq y\}$ is finite. We use $X \rightarrow\!\!\!\triangleright\, Y$ to denote quasi-constant functions from X to Y.

Quasi-constant functions have a finite representation:

Definition 3.9 (Quasi-constant function terms). We use the finite term $\{\!\!\{x_1 = y_1, ..., x_n = y_n, \_ = y\}\!\!\}$, where all the x_i are distinct and all the y_i are different from y, to denote the quasi-constant function F defined by $\forall i \in [1..n].\ F(x_i) = y_i$ and $\forall x \in X \setminus \{x_1, ..., x_n\}.\ F(x) = y$. We have a one-to-one correspondance between such terms and quasi-constant functions.

We can now define our interpretation domain:

Definition 3.10 (Interpretation domain [11]). The *interpretation domain* $\mathcal{D}$ (respectively *field interpretation domain* $\mathcal{D}_\mathsf{f}$) is the set of finite terms d (respectively δ) produced inductively by the following grammar

$$\begin{array}{l} v ::= c \mid (d, d) \mid \{(d, \partial), \ldots, (d, \partial)\} \mid \{\!\!\{\ell = \delta, \ldots, \ell = \delta, \_ = \delta\}\!\!\} \\ d ::= v^L \qquad\qquad \partial ::= d \mid \Omega \qquad\qquad \delta ::= \partial^F \end{array}$$

where c ranges over the set $\mathcal{C}$ of constants, L ranges over finite sets of type variables, F ranges over finite sets of row variables, and where Ω is such that $\Omega \notin \mathcal{C}$.

The elements of $\mathcal{D}$ correspond, intuitively, to denotations of the values of our language, labeled by finite sets of type variables. In particular, in a higher-order language, the values include functions which, in this model, are represented by sets of finite relations of the form $\{(d_1, \partial_1), \ldots, (d_n, \partial_n)\}^L$, where Ω (which is not in $\mathcal{C}$) can appear in second components to signify that the function fails (i.e., evaluation is stuck) on the corresponding input. This is implemented by using in the second projection the meta-variable ∂ which ranges over $\mathcal{D}_\Omega = \mathcal{D} \cup \{\Omega\}$ (we reserve d to range over $\mathcal{D}$, thus excluding Ω). This constant Ω is used to ensure that $\mathbb{1} \to \mathbb{1}$ is not a supertype of all function types: if we used d instead of ∂, then every well-typed function could be subsumed to $\mathbb{1} \to \mathbb{1}$ and, therefore, every application could be given the type $\mathbb{1}$, independently of its argument as long as this argument is typeable.

As explained before, record types are represented by sets of quasi-constant functions from labels to elements of the field interpretation domain $\mathcal{D}_\mathsf{f}$. Elements δ of $\mathcal{D}_\mathsf{f}$ are labeled by finite sets of row variables. Their possible values include Ω, but its meaning is different from before: in the context of a field, Ω denotes an absent value.

Finally, the sets of type variables that label the elements of the domain are used to interpret type variables: we interpret a type variable α by the set of all elements that are labeled by α, that is the set $\{v^L \in \mathcal{D} \mid \alpha \in L\}$. Likewise, we interpret a row variable ρ by the set of all field elements that are labeled by ρ, that is the set $\{\partial^F \in \mathcal{D}_\mathsf{f} \mid \rho \in F\}$.

This interpretation of variables is the default one, for variables that are not in the process of being substituted ; in particular it will be used to define subtyping. However, in order to be able to characterize the result of type substitutions, we define a more general type interpretation, parametrized by an assignment η that maps type variables and row variables to their interpretation.

Definition 3.11 (Assignment). An assignment η is a function $\mathcal{V}_{\mathsf{Ty}} \cup \mathcal{V}_{\mathsf{Row}} \to \mathcal{P}(\mathcal{D}) \cup (\mathcal{L} \rightharpoonup \mathcal{P}(\mathcal{D}_{\mathsf{f}}))$ mapping type variables to subsets of $\mathcal{D}$ and row variables to quasi-constant functions from labels to subsets of $\mathcal{D}_{\mathsf{f}}$. We note $\mathcal{H}$ the set of assignments.

One additional difficulty comes from the fact that types can be infinite (they are defined coinductively), thus preventing from giving a direct inductive definition for the type interpretation $[\![t]\!]^\eta$ of t. Instead, we exploit the fact that terms from our interpretation domain $\mathcal{D}$ are finite, and we define a predicate $(\partial : t)^\eta$ ("the element ∂ belongs to the type t") and a predicate $(\delta : f)^\eta_\ell$ ("the field element δ under the label ℓ belongs to the field type f") by structural induction on the pair (∂, t) or (δ, f) ordered lexicographically. Notice that the contractivity condition of Definition 3.1 ensures that the binary relation $\triangleright \subseteq \mathcal{T} \times \mathcal{T}$ defined by $t_1 \vee t_2 \triangleright t_i$, $t_1 \wedge t_2 \triangleright t_i$, $\neg t \triangleright t$ is well-founded. This gives an induction principle[2] on $\mathcal{T}$ that we use combined with structural induction on $\mathcal{D}$ to give the following definition.

Definition 3.12 (Interpretation predicate). Let η be an assignment. We define two binary predicates $(\partial : t)^\eta$ and $(\delta : f)^\eta_\ell$, where $\partial \in \mathcal{D}_\Omega$, $t \in \mathcal{T}$, $\delta \in \mathcal{D}_{\mathsf{f}}$ and $f \in \mathcal{F}$, by structural induction on the pair (∂, t) and (δ, f) ordered lexicographically. The predicate is defined as follows:

$$
\begin{array}{lrcl}
\boxed{\textbf{Types:}} & (d : \mathbb{1})^\eta & = & \text{true} \\
& (d : \alpha)^\eta & = & d \in \eta(\alpha) \\
& (c^L : b)^\eta & = & c \in \mathbb{B}(b) \\
& ((d_1, d_2)^L : t_1 \times t_2)^\eta & = & (d_1 : t_1)^\eta \text{ and } (d_2 : t_2)^\eta \\
& (\{(d_1, \partial_1), \ldots, (d_n, \partial_n)\}^L : t_1 \to t_2)^\eta & = & \forall i \in [1..n].\ \text{if } (d_i : t_1)^\eta \text{ then } (\partial_i : t_2)^\eta \\
& (\{\!\!\{\ell_1 = \delta_1, \ldots, \ell_n = \delta_n, \_ = \delta\}\!\!\}^L : r)^\eta & = & \forall i \in [1..n].\ (\delta_i : r(\ell_i))^\eta_{\ell_i} \\
& & & \text{and } \ \forall \ell \in \mathcal{L} \setminus \{\ell_1, \ldots, \ell_n\}.\ (\delta : r(\ell))^\eta_\ell \\
& (d : t_1 \wedge t_2)^\eta & = & (d : t_1)^\eta \text{ and } (d : t_2)^\eta \\
& (d : t_1 \vee t_2)^\eta & = & (d : t_1)^\eta \text{ or } (d : t_2)^\eta \\
& (d : \neg t)^\eta & = & \text{not } (d : t)^\eta \\
& (\partial : t)^\eta & = & \text{false} \qquad\qquad \text{otherwise} \\
\boxed{\textbf{Fields:}} & (\delta : \rho)^\eta_\ell & = & \delta \in \eta(\rho)(\ell) \\
& (d^F : t)^\eta_\ell = (d^F : t?)^\eta_\ell & = & (d : t)^\eta \\
& (\Omega^F : t?)^\eta_\ell & = & \text{true} \\
& (\delta : f_1 \wedge f_2)^\eta_\ell & = & (\delta : f_1)^\eta_\ell \text{ and } (\delta : f_2)^\eta_\ell \\
& (\delta : f_1 \vee f_2)^\eta_\ell & = & (\delta : f_1)^\eta_\ell \text{ or } (\delta : f_2)^\eta_\ell \\
& (\delta : \neg f)^\eta_\ell & = & \text{not } (\delta : f)^\eta_\ell \\
& (\delta : f)^\eta_\ell & = & \text{false} \qquad\qquad \text{otherwise}
\end{array}
$$

where $\mathbb{B}$ denotes the function that assigns to each base type the set of constants of that type.

[2] In a nutshell, we can do proofs and give definitions by induction on the structure of unions and negations—and, thus, intersections—but arrows, products, and base types are the base cases for the induction.

Definition 3.13. We define the *set-theoretic interpretation* $[\![t]\!]^\eta$ of a type t, and the *set-theoretic interpretation* $[\![f]\!]^\eta_\ell$ of a field type f, as follows:

$$[\![t]\!]^\eta = \{d \in \mathcal{D} \mid (d : t)^\eta\}$$
$$[\![f]\!]^\eta_\ell = \{\delta \in \mathcal{D}_\mathsf{f} \mid (\delta : f)^\eta_\ell\}$$

We now define the *identity assignment* $\eta_\circ$: when it is not in the process of being substituted, a type variable α is interpreted as the set of values $\{v^L \in \mathcal{D} \mid \alpha \in L\}$, and a row variable ρ is interpreted as the set of values $\{\partial^F \in \mathcal{D}_\mathsf{f} \mid \rho \in F\}$.

Definition 3.14 (Identity assignment). We define the identity assignment $\eta_\circ$ as follows:

$$\forall \alpha \in \mathcal{V}_{\mathsf{Ty}}.\ \eta_\circ(\alpha) = \{v^L \in \mathcal{D} \mid \alpha \in L\}$$
$$\forall \rho \in \mathcal{V}_{\mathsf{Row}}.\ \forall \ell \in \mathcal{L}.\ \eta_\circ(\rho)(\ell) = \{\partial^F \in \mathcal{D}_\mathsf{f} \mid \rho \in F\}$$

As the definition of $\eta_\circ(\rho)(\ell)$ does not depend on ℓ, we have the following trivial property:

Proposition 3.15. *For any f, ℓ and ℓ', we have $[\![f]\!]^{\eta_\circ}_\ell = [\![f]\!]^{\eta_\circ}_{\ell'}$.*

When no assignment is specified, the set-theoretic interpretation defaults to using $\eta_\circ$:

Definition 3.16 (Set-theoretic interpretation of types). We define the *set-theoretic interpretation* $[\![.]\!] : \mathcal{T} \to \mathcal{P}(\mathcal{D})$ as $[\![t]\!] = [\![t]\!]^{\eta_\circ}$, and $[\![.]\!] : \mathcal{F} \to \mathcal{P}(\mathcal{D}_\mathsf{f})$ as $[\![f]\!] = [\![f]\!]^{\eta_\circ}_\ell$, where ℓ can be any label (according to Proposition 3.15, the interpretation is the same whichever label ℓ we choose).

Finally, we can associate to any type substitution its interpretation as an assignment:

Definition 3.17 (Interpretation of type substitutions). The *set-theoretic interpretation* $[\![\phi]\!]$ of a type substitution ϕ is the assignment η defined as follows:

$$\forall \alpha \in \mathcal{V}_{\mathsf{Ty}}.\ \eta(\alpha) = [\![\phi(\alpha)]\!]$$
$$\forall \rho \in \mathcal{V}_{\mathsf{Row}}.\ \forall \ell \in \mathcal{L}.\ \eta(\rho)(\ell) = [\![\phi(\rho)(\ell)]\!]$$

This allows us characterizing the result of the application of a type substitution on a type:

Proposition 3.18. *For any type t and type substitution ϕ, $[\![t\phi]\!] = [\![t]\!]^{[\![\phi]\!]}$.*

3.3 Semantic subtyping

Now that we have a set-theoretic interpretation of types, we can define a subtyping preorder and its associated equivalence relation as follows.

Definition 3.19 (Subtyping relation). We define the *subtyping* relation $\leq$ and the *subtyping equivalence* relation $\simeq$ as follows:

$$t_1 \leq t_2 \overset{\text{def}}{\iff} [\![t_1]\!] \subseteq [\![t_2]\!] \quad \text{and} \quad t_1 \simeq t_2 \overset{\text{def}}{\iff} [\![t_1]\!] = [\![t_2]\!]$$

This subtyping relation is decidable and is sometimes referred to as *semantic subtyping* [10], as it is not defined on the syntax of the types but on their semantic interpretation.

With this set-theoretic definition of subtyping, usual properties of sets are inherited, for instance:

$$\begin{array}{ccr}
t_1 \vee t_2 \simeq t_2 \vee t_1 & t_1 \wedge t_2 \simeq t_2 \wedge t_1 & \text{(commutativity)} \\
t \vee t \simeq t & t \wedge t \simeq t & \text{(idempotence)} \\
\neg(\neg t) \simeq t & & \text{(double complement)} \\
t \vee (s_1 \wedge s_2) \simeq (t \vee s_1) \wedge (t \vee s_2) & t \wedge (s_1 \vee s_2) \simeq (t \wedge s_1) \vee (t \wedge s_2) & \text{(distributivity)} \\
\neg(t_1 \vee t_2) \simeq \neg t_1 \wedge \neg t_2 & \neg(t_1 \wedge t_2) \simeq \neg t_1 \vee \neg t_2 & \text{(De Morgan's law)}
\end{array}$$

We can also define a semantic equivalence relation over type substitutions.

Definition 3.20 (Type substitutions equivalence). We define the equivalence relation $\simeq$ over type substitutions as follows:

$$\phi_1 \simeq \phi_2 \overset{\text{def}}{\iff} [\![\phi_1]\!] = [\![\phi_2]\!]$$

An important property of our interpretation is that subtyping is preserved by type substitutions:

Proposition 3.21 (Preservation of subtyping by type substitutions).

$$\forall t_1, t_2, \phi.\ t_1 \leq t_2 \Rightarrow t_1\phi \leq t_2\phi$$

This is a direct consequence of Proposition 3.18, combined with the following property:

Proposition 3.22. *For any type t, $[\![t]\!] = \varnothing \Rightarrow \forall \eta.\ [\![t]\!]^\eta = \varnothing$.*

Our definition of subtyping as set containment cannot be used to decide subtyping as it involves infinite sets. In the next section, we present algorithms to decide subtyping and to solve *tallying* instances, that is, to find all substitutions that make a given set of subtyping constraints hold.

4 Implementation

Our formal definition of subtyping cannot be used directly to decide whether two types are in subtyping relation, as it would require manipulating infinite sets. Instead, we proceed in three steps: (i) first, the subtyping problem $t_1 \leq t_2$ is reduced to checking emptiness of a type $t = t_1 \setminus t_2$; (ii) second, this type t is expressed in disjunctive normal form (Section 4.1), and (iii) we use an inductive relation to decide emptiness of this DNF (Section 4.2).

4.1 Disjunctive Normal Form

A convenient way to decide the emptiness of a type is to put it in *disjunctive normal form* (DNF).

Definition 4.1 (Disjunctive Normal Form, [8]). Given a type t, its DNF is a syntactic term, equivalent to t, of the following form:

$$\mathsf{DNF}(t) = \begin{array}{ll} & \bigvee \bigwedge_i \alpha_i^{\mathrm{b}} \wedge \bigwedge_j \neg\beta_j^{\mathrm{b}} \wedge \bigwedge_k b_k \wedge \bigwedge_l \neg b_l \\ \vee & \bigvee \bigwedge_i \alpha_i^{\mathrm{p}} \wedge \bigwedge_j \neg\beta_j^{\mathrm{p}} \wedge \bigwedge_k (t_k^1 \times t_k^2) \wedge \bigwedge_l \neg(s_l^1 \times s_l^2) \\ \vee & \bigvee \bigwedge_i \alpha_i^{\mathrm{a}} \wedge \bigwedge_j \neg\beta_j^{\mathrm{a}} \wedge \bigwedge_k (t_k^1 \to t_k^2) \wedge \bigwedge_l \neg(s_l^1 \to s_l^2) \\ \vee & \bigvee \bigwedge_i \alpha_i^{\mathrm{r}} \wedge \bigwedge_j \neg\beta_j^{\mathrm{r}} \wedge \bigwedge_k r_k \wedge \bigwedge_l \neg r'_l \end{array}$$

where each summand satisfies $\forall i.\ \forall j.\ \alpha_i^{\mathrm{b}} \neq \beta_j^{\mathrm{b}}$, $\forall i.\ \forall j.\ \alpha_i^{\mathrm{p}} \neq \beta_j^{\mathrm{p}}$, $\forall i.\ \forall j.\ \alpha_i^{\mathrm{a}} \neq \beta_j^{\mathrm{a}}$, $\forall i.\ \forall j.\ \alpha_i^{\mathrm{r}} \neq \beta_j^{\mathrm{r}}$.

In this formula, each row is itself a DNF of positive and negative type variables and positive and negative instances of a particular type constructor: a basic type, a product, an arrow, or a record. The constraints on type variables ensures no summand is trivially empty: a conjunction $\alpha \wedge \neg\alpha \wedge \ldots$ is necessarily empty and can thus be eliminated. It is easy to compute the DNF of a type using the distributivity property and the De Morgan's law. The subtyping algorithm can then iterate through the elements of this DNF to test the emptiness of the type.

Similarly, the DNF of a field type f is a syntactic term, equivalent to f, of the following form:

$$\mathsf{DNF}(f) = \bigvee \bigwedge_i \rho_i \wedge \bigwedge_j \neg\rho'_j \wedge \bigwedge_k \bar{t}_k \wedge \bigwedge_l \neg\bar{t}'_l$$

where each summand satisfies $\forall i.\ \forall j.\ \rho_i \neq \rho'_j$.

4.2 Semantic subtyping

To test the emptiness of a type in DNF form, we just have to test the emptiness of every summand. In the paper, we only focus on the summands involving record constructors: the other cases can be found in [8]. Consider the following conjunction of literals:

$$\bigwedge_i \alpha_i^{\mathrm{r}} \ \wedge\ \bigwedge_j \neg\beta_j^{\mathrm{r}} \ \wedge\ \bigwedge_k r_k \ \wedge\ \bigwedge_l \neg r'_l$$

According to [8], deciding emptiness of this conjunction is equivalent to deciding emptiness of $\bigwedge_k r_k \ \wedge\ \bigwedge_l \neg r'_l$: type variables can simply be ignored (provided that the same type variable does not appear in both positive and negative positions, which is ensured by our definition of DNF). In turn, deciding the emptiness of $\bigwedge_k r_k \ \wedge\ \bigwedge_l \neg r'_l$ is equivalent to deciding the subtyping relation $\bigwedge_k r_k \leq \bigvee_l r'_l$. In this section, we give an inductive relation that reduces this subtyping instance into several subtyping instances that are Boolean combinations of the types of the fields of the atoms $\{r_k\}_k$ and $\{r'_l\}_l$. A subtyping algorithm can be derived from this inductive relation. Note that, as types are defined coinductively (and thus can be infinite), the subtyping algorithm must maintain a cache of the visited types in order to ensure termination. The regularity constraint over types ensures only a finite number of types will be visited. A pseudo-code implementation of the subtyping algorithm for pairs and arrows can be found in [1].

4.2.1 Subtyping of field types. First, we give an inductive relation for deciding the subtyping over field types. As for types, checking subtyping between two field types $f_1 \leq f_2$ is equivalent to checking the emptiness of the field type $f = f_1 \setminus f_2$. We thus consider the DNF of f:

$$\mathsf{DNF}(f) = \bigvee \bigwedge_i \rho_i \ \wedge\ \bigwedge_j \neg\rho'_j \ \wedge\ \bigwedge_k \bar{t}_k \ \wedge\ \bigwedge_l \neg\bar{t}'_l$$

The emptiness of each summand of the DNF can be checked independently. Row variables can be ignored when checking the emptiness of a summand:

Proposition 4.2 (Field type subtyping).

$$\bigwedge_{i\in I} \rho_i \ \wedge\ \bigwedge_{j\in J} \neg\rho'_j \ \wedge\ \bigwedge_{k\in K} \bar{t}_k \ \wedge\ \bigwedge_{l\in L} \neg\bar{t}'_l \simeq \mathbb{0} \ \textit{(where } \forall i.\ \forall j.\ \rho_i \neq \rho'_j\textit{)} \quad \Leftrightarrow \quad \bigwedge_{k\in K} \bar{t}_k \leq \bigvee_{l\in L} \bar{t}'_l$$

For subtyping of option types, we use the following property:

Proposition 4.3 (Option type subtyping). *For any option types* $\{\bar{t}_p\}_{p\in P}$ *and* $\{\bar{t}'_n\}_{n\in N}$, *we have:*

$$\bigwedge_{p\in P} \bar{t}_p \leq \bigvee_{n\in N} \bar{t}'_n \quad \Leftrightarrow \quad (\forall p \in P.\ \mathrm{opt}(\bar{t}_p) \Rightarrow \exists n \in N.\mathrm{opt}(\bar{t}'_n)) \ \textit{and} \ \bigwedge_{p\in P} \mathrm{get}(\bar{t}_p) \leq \bigvee_{n\in N} \mathrm{get}(\bar{t}'_n)$$

where $\forall t.\ \mathrm{opt}(t?) = \mathrm{true}, \mathrm{opt}(t) = \mathrm{false}$, *and* $\mathrm{get}(t) = \mathrm{get}(t?) = t$.

4.2.2 Subtyping of records. We now focus on deciding subtyping for record types. We recall that, in our interpretation domain, record values are quasi-constant functions $\mathcal{L} \rightharpoonup \mathcal{D}_{\mathrm{f}}$ from labels to field values. First, we introduce a notation to help us express the set of quasi-constant functions captured by a record type.

Definition 4.4. Let $(X_\ell)_{\ell\in\mathcal{L}}$ be a family of subsets of $\mathcal{D}_{\mathrm{f}}$ indexed by $\mathcal{L}$. The notation $\prod^{\triangleright}_{\ell\in\mathcal{L}} X_\ell$ denotes the subset of $\mathcal{L} \rightharpoonup \mathcal{D}_{\mathrm{f}}$ formed by all quasi-constant functions F such that $\forall \ell \in \mathcal{L}.\ F(\ell) \in X_\ell$.

In particular, we have:

$$[\![r]\!] = \prod^{\triangleright}_{\ell\in\mathcal{L}} [\![r(\ell)]\!]$$

Deciding the subtyping instance $\bigwedge_{p\in P} r_p \leq \bigvee_{n\in N} r_n$ is equivalent to deciding the set containment instance $(\bigcap_{p\in P} \overset{\triangleright}{\Pi}_{\ell\in\mathcal{L}}[\![r_p(\ell)]\!]) \subseteq (\bigcup_{n\in N} \overset{\triangleright}{\Pi}_{\ell\in\mathcal{L}}[\![r_n(\ell)]\!])$. This can be decided inductively by using the following property, established by Alain Frisch in his PhD dissertation [10].

Proposition 4.5 (Record containment). *Let $(X_p)_{p\in P}$ and $(X_n)_{n\in N}$ be two families of elements of $\mathcal{L} \mathrel{-\!\!\triangleright} \mathcal{P}(\mathcal{D}_f)$. Let $L = \bigcup_{i\in P\cup N} \mathsf{dom}(X_i)$. Then, $(\bigcap_{p\in P} \overset{\triangleright}{\Pi}_{\ell\in\mathcal{L}} X_p(\ell)) \subseteq (\bigcup_{n\in N} \overset{\triangleright}{\Pi}_{\ell\in\mathcal{L}} X_n(\ell))$ if and only if either $\bigcap_{p\in P} \mathsf{def}(X_p) = \varnothing$, or for every map $\iota : N \to L \cup \{\_\}$*

$$\left(\exists \ell \in L.\left(\bigcap_{p\in P} X_p(\ell) \subseteq \bigcup_{n\in N|\iota(n)=\ell} X_n(\ell)\right)\right) \textit{ or } \left(\exists n_\circ \in N.(\iota(n_\circ) = \_) \textit{ and } \left(\bigcap_{p\in P} \mathsf{def}(X_p) \leq \mathsf{def}(X_{n_\circ})\right)\right)$$

This gives us the following inductive relation for deciding subtyping of records:

Proposition 4.6 (Record subtyping). *For any record atoms $\{r_p\}_{p\in P}$ and $\{r_n\}_{n\in N}$, we have:*

$$\bigwedge_{p\in P} r_p \leq \bigvee_{n\in N} r_n \quad\Leftrightarrow\quad \bigwedge_{p\in P} \mathsf{tl}(r_p) \leq \mathbb{0} \textit{ or } \forall\iota : N \to L \cup \{\_\}.$$

$$\left(\exists \ell \in L.\ (\bigwedge_{p\in P} r_p(\ell) \leq \bigvee_{n\in N|\iota(n)=\ell} r_n(\ell))\right) \textit{ or } \left(\exists n_\circ \in N.(\iota(n_\circ) = \_) \textit{ and } \left(\bigwedge_{p\in P} \mathsf{tl}(r_p) \leq \mathsf{tl}(r_{n_\circ})\right)\right)$$

where $L = \bigcup_{i\in P\cup N} \mathsf{labels}(r_i)$.

Coupled with the inductive relation for the subtyping of field types, we obtain an inductive relation that reduces a subtyping instance between record atoms into several subtyping instances between Boolean combinations of strict subtypes of these record atoms. Combined with the usual inductive relations for other type constructors, as well as a caching mechanism to ensure termination in the case of recursive types, this results in a subtyping algorithm that is sound and complete.

4.3 Tallying

In a polymorphic type system, subtyping alone is not enough to type applications. Indeed, we may need to instantiate the type variables in the type of the function or argument. For this reason, Castagna, Nguyen, Xu and Abate introduced the tallying problem [5], which can be seen as a unification problem but with subtyping constraints instead of syntactic equality.

Definition 4.7 (Tallying, [5]). Let $S = \{(s_1, t_1), \ldots, (s_n, t_n)\}$ be a finite set of pairs of types and Δ a set of type variables. The solution of the tallying problem for S and Δ is the following set of type substitutions:

$$\{\sigma \in \mathcal{S} \mid \mathsf{dom}(\sigma) \cap \Delta = \varnothing \ \wedge\ \forall (s,t) \in S.\ s\sigma \leq t\sigma\}$$

The set S contains the subtyping constraints, and the set Δ is the set of *monomorphic* variables (i.e. the type variables that cannot be substituted). An important property of tallying is that there exists a sound and complete algorithm: it computes a finite set of substitutions which exactly characterizes all the possible solutions.

4.3.1 Base algorithm. In this section, we briefly explain how the tallying algorithm works for a set-theoretic type algebra without row variables. The tallying algorithm has been first described in [5], and revisited in [1] to which the reader may refer for more details. We fix a set Δ of monomorphic type variables that should not be substituted. We also fix a total order $\preceq$ over type variables.

As for the subtyping algorithm, satisfying a subtyping constraint $t_1 \leq t_2$ is equivalent to making the type $t = t_1 \setminus t_2$ empty. At its heart, the tallying algorithm recursively traverses the type t and generates constraint sets on the variables of t which, when satisfied, make the type empty. These constraint sets are defined as follows:

Definition 4.8 (Normalized constraint set). A normalized constraint set C is a set of triples $\{(s_i, \alpha_i, t_i)\}_{i\in I}$ where all variables α_i are distinct. Intuitively, it represents constraints $s_i \leq \alpha_i \leq t_i$. The set $\{\alpha_i\}_{i\in I}$ is called the *domain* of C and is written $\mathsf{dom}(C)$. The empty constraint set is noted $\top$. Lastly, we define the constraint associated with a variable α in a constraint set C by:

$$C(\alpha) \ = \ (s,t) \ \text{ if } (s,\alpha,t) \in C \qquad\qquad C(\alpha) \ = \ (\mathbb{0}, \mathbb{1}) \ \text{ otherwise}$$

Definition 4.9 (Intersection of two normalized constraint sets). Let C_1 and C_2 be two normalized constraint sets. We define the *intersection* $C_1 \sqcap C_2$ as the constraint set $\{(s_1 \vee s_2, \alpha, t_1 \wedge t_2) \mid \alpha \in \mathsf{dom}(C_1) \cup \mathsf{dom}(C_2),\ (s_1, t_1) = C_1(\alpha),\ (s_2, t_2) = C_2(\alpha)\}$.

Definition 4.10 (Union and intersection of two sets of constraint sets). Given two sets of normalized constraint sets $\mathscr{C}_1$ and $\mathscr{C}_2$, we define their intersection and union as:

$$\begin{array}{rcll} \mathscr{C}_1 \sqcap \mathscr{C}_2 & = & \{C_1 \sqcap C_2 \mid C_1 \in \mathscr{C}_1, C_2 \in \mathscr{C}_2\} & \text{(intersection)} \\ \mathscr{C}_1 \sqcup \mathscr{C}_2 & = & \mathscr{C}_1 \cup \mathscr{C}_2 & \text{(union)} \end{array}$$

Note that additional care can be given to eliminate redundant constraints from a set and to eliminate trivially unsatisfiable constraint sets, but this is out of the scope of this paper. The reader may refer to [1] for more details.

The tallying algorithm is composed of three main steps:

Normalize Transforms a subtyping constraint $t_1 \leq t_2$ (initially, those given as input) into a set of normalized constraint sets $\mathscr{C}$ involving the type variables in t_1 and t_2.

Propagate Each normalized constraint set is then processed through a function which enriches them with new induced constraints: for every constraint (s, α, t) produced by `normalize`, the induced subtyping relation $s \leq t$ may require new constraints (that are computed by calling `normalize` again). These new constraints are added by `propagate` to form a final set of constraint sets $\mathscr{C}'$.

Solve As a final step, each constraint set in $\mathscr{C}'$ is turned into a set of recursive type equations and solved, yielding a substitution. Formally, a constraint $s_1 \leq \alpha \leq s_2$ is turned into an equation $\alpha = (s_1 \vee \alpha') \wedge s_2$ for a fresh α' ; then a type t solution of this equation is built, and we define $t' = t\{\alpha' \rightsquigarrow \alpha\}$; we then apply the substitution $\{\alpha \rightsquigarrow t'\}$ to the remaining constraints and call `solve` recursively on them, yielding a solution ϕ to which we add the binding $\{\alpha \rightsquigarrow t'\phi\}$.

In this paper, we focus on the `normalize` step, as this is the step that is most impacted by the addition of polymorphic records.

The `normalize` function takes a type t as input, and returns a set $\mathscr{C}$ of normalized constraint sets, each $C \in \mathscr{C}$ defining some constraints over the type variables in t that are sufficient to make t empty. To normalize a constraint $t_1 \leq t_2$, we can thus just call `normalize` on the type $t_1 \setminus t_2$. Let us consider the following type t consisting in a conjunction of literals (for concision, we only consider record components):

$$t = \bigwedge_{p\in P'} \alpha_p \wedge \bigwedge_{n\in N'} \neg\alpha_n \wedge \bigwedge_{p\in P} r_p \wedge \bigwedge_{n\in N} \neg r_n$$

If at least one of the type variables $\{\alpha_p\}_{p\in P'}$ is not in our set Δ of monomorphic type variables, we extract the minimal one according to $\preceq$, $\alpha_{p'}$, and generate the following constraint set that makes the type t empty:

$$C = \{(\mathbb{0}, \alpha_{p'}, \neg(\bigwedge_{p\in P'\setminus\{p'\}} \alpha_p \wedge \bigwedge_{n\in N'} \neg\alpha_n \wedge \bigwedge_{p\in P} r_p \wedge \bigwedge_{n\in N} \neg r_n))\}$$

Otherwise, if at least one of the type variables $\{\alpha_n\}_{n\in N'}$ is not in our set Δ of monomorphic type variables, we extract the minimal one according to $\preceq$, $\alpha_{n'}$, and generate the following constraint set that makes the type t empty:

$$C = \{(\neg(\bigwedge_{p\in P'} \alpha_p \wedge \bigwedge_{n\in N'\setminus\{n'\}} \neg\alpha_n \wedge \bigwedge_{p\in P} r_p \wedge \bigwedge_{n\in N} \neg r_n), \alpha_{n'}, \mathbb{1})\}$$

Otherwise, if all top-level type variables of t are in Δ, we generate the constraints recursively by using the record subtyping relation (Proposition 4.6):

$$\mathscr{C} = \mathtt{normalize}\left(\bigwedge_{p\in P} \mathsf{tl}(r_p)\right) \sqcup \bigsqcap_{\iota:N\to L\cup\{\_\}} \left(\left(\bigsqcup_{\ell\in L} \mathtt{normalize}\left(\bigwedge_{p\in P} r_p(\ell) \wedge \bigwedge_{n\in N \text{ s.t. } \iota(n)=\ell} \neg r_n(\ell)\right)\right) \sqcup \left(\bigsqcup_{n_\circ\in N \text{ s.t. } \iota(n_\circ)=\_} \mathtt{normalize}\left(\bigwedge_{p\in P} \mathsf{tl}(r_p) \setminus \mathsf{tl}(r_{n_\circ})\right)\right)\right)$$

where $L = \bigcup_{i\in P\cup N} \mathsf{labels}(r_i)$.

For now, the tallying algorithm is presented in a context without row polymorphism, so we assume the field types involved in the relation above do not contain row variables. Normalizing field types with row-variables requires extending the tallying algorithm: this is the purpose of the next section. Also note that, similarly to the subtyping algorithm, the `normalize` function must maintain a cache of the types already visited in order to ensure termination. The reader may refer to [1] for a pseudo-code implementation.

4.3.2 Constant rows extension. We now focus on extending the tallying algorithm to make it able to generate solutions that substitute row variables. A row variable should be substituted by a row $\langle \vec{B} \mid f\rangle$. However, in this section, we will focus on finding solutions of a specific form: all the rows we will generate will be of the form $\langle\ \mid f\rangle$. In other words, we only look for solutions that involve rows that are constant functions from $\mathcal{L}$ to $\mathcal{P}(\mathcal{D}_\mathsf{f})$. We will see in the next section (Section 4.3.3) that a more general tallying problem can be reduced to this limited version.

The set Δ of monomorphic type variables is extended to also contain monomorphic row variables ($\Delta \subseteq \mathcal{V}_{\mathsf{Ty}} \cup \mathcal{V}_{\mathsf{Row}}$). We also consider a total order $\preceq$ over row variables. In order to find solutions that can substitute row variables, we extend our notion of constraint: in addition to constraints over type variables (t_1, α, t_2), we also consider constraints over field variables (f_1, ρ, f_2). Note that a row variable (ρ) is bounded by a field types (f_1 and f_2) and not rows: this is possible without loss of generality because we are only looking for solutions involving constant rows of the form $\langle\ \mid f\rangle$. Intuitively, a constraint (f_1, ρ, f_2) should be understood as $\langle\ \mid f_1\rangle \leq \rho \leq \langle\ \mid f_2\rangle$.

Type constraint sets	V	::=	$\{(t,\alpha,t), ..., (t,\alpha,t)\}$
Field constraint sets	F	::=	$\{(f,\rho,f), ..., (f,\rho,f)\}$
Mixed constraint sets	C	::=	(V, F)
Sets of constraint sets	$\mathscr{C}$	::=	$\{C, ..., C\}$

For a field constraint set $\{(f_i, \rho_i, f_i')\}_{i\in I}$, the set $\{\rho_i\}_{i\in I}$ is called the *domain* of F and is written $\mathsf{dom}(F)$. We define the constraint associated with a row variable ρ in a field constraint set F by:

$$F(\rho) = (f_1, f_2) \text{ if } (f_1, \rho, f_2) \in F \qquad\qquad F(\rho) = (\mathbb{0}, \mathbb{1}?) \text{ otherwise}$$

The operation $\sqcap$ on field constraint sets is defined as $F_1 \sqcap F_2 = \{(f_1 \vee f_2, \rho, f_1' \wedge f_2') \mid \rho \in \mathsf{dom}(F_1) \cup \mathsf{dom}(F_2),\ (f_1, f_1') = F_1(\rho),\ (f_2, f_2') = F_2(\rho)\}$. It is extended component-wise on mixed constraint sets.

The `normalize` and `solve` functions need to be modified to account for this new kind of constraints. For `solve`, it is straightforward: a constraint (f_1, ρ, f_2) in the constraint set is turned into an equation $\rho = (f_1 \vee \rho') \wedge f_2$ for a fresh ρ' ; then a field type f solution of this equation is built, and we define $f' = f\{\rho' \rightsquigarrow \rho\}$; we then apply the substitution $\{\rho \rightsquigarrow \langle \mid f' \rangle\}$ to the remaining constraints and call `solve` recursively on them, yielding a solution ϕ to which we add the binding $\{\rho \rightsquigarrow t'\phi\}$.

The function `normalize` is extended to handle field types involving row variables in the same way as it treats type variables in types. Let us consider the following field type f consisting in a conjunction of field literals:

$$f = \bigwedge_{p \in P'} \rho_p \wedge \bigwedge_{n \in N'} \neg\rho_n \wedge \bigwedge_{p \in P} \bar{t}_p \wedge \bigwedge_{n \in N} \neg\bar{t}_n$$

If at least one of the row variables $\{\rho_p\}_{p \in P'}$ is not in our set Δ of monomorphic variables, we extract the minimal one according to $\preceq$, $\rho_{p'}$, and generate the following constraint set that makes the field type f empty:

$$C = \{(\mathbb{0}, \rho_{p'}, \neg(\bigwedge_{p \in P' \setminus \{p'\}} \rho_p \wedge \bigwedge_{n \in N'} \neg\rho_n \wedge \bigwedge_{p \in P} \bar{t}_p \wedge \bigwedge_{n \in N} \neg\bar{t}_n))\}$$

Otherwise, if at least one of the row variables $\{\rho_n\}_{n \in N'}$ is not in our set Δ of monomorphic variables, we extract the minimal one according to $\preceq$, $\rho_{n'}$, and generate the following constraint set that makes the field type f empty:

$$C = \{(\neg(\bigwedge_{p \in P'} \rho_p \wedge \bigwedge_{n \in N' \setminus \{n'\}} \neg\rho_n \wedge \bigwedge_{p \in P} \bar{t}_p \wedge \bigwedge_{n \in N} \neg\bar{t}_n), \rho_{n'}, \mathbb{1}?)\}$$

Otherwise, if all top-level row variables of f are in Δ, then we just call `normalize` recursively on the following option type:

$$\mathscr{C} = \texttt{normalize}(\bigwedge_{p \in P} \bar{t}_p \wedge \bigwedge_{n \in N} \neg\bar{t}_n)$$

This extension of the `normalize` function, together with the straightforward extensions of the `propagate` and `solve` functions, define a new tallying algorithm that supports row variables and computes solutions involving constant rows (i.e. solutions in $\mathcal{S}_\varnothing$).

Soundness and completeness. For a set of constraints S and a set of monomorphic variables Δ, we note $\mathsf{tally}(S, \Delta, \varnothing)$ the result of this constant-row tallying algorithm. The third parameter $\varnothing$ indicates that we are only searching amongst a restricted set of solutions: for a set of labels L, $\mathsf{tally}(S, \Delta, L)$ only computes solutions in $\mathcal{S}_L$, defined as follows:

Definition 4.11 (Quasi-constant substitutions for a set of labels). Let L be a finite set of labels. The set of all quasi-constant substitutions for L, noted $\mathcal{S}_L$, is the following set:

$$\mathcal{S}_L = \{\phi \in \mathcal{S} \mid \mathsf{labels}(\phi) \subseteq L\}$$

In our case, having $\varnothing$ as the third parameter of $\mathsf{tally}(S, \Delta, \varnothing)$ means that we only solve for solutions involving constant rows. The soundness and completeness of this tallying algorithm can be expressed as follows:

Theorem 4.12 (Soundness).

$$\forall \phi \in \mathsf{tally}(S, \Delta, \varnothing).\quad \phi \in \mathcal{S}_\varnothing \quad \textit{and} \quad \mathsf{dom}(\phi) \cap \Delta = \varnothing \quad \textit{and} \quad \forall (s, t) \in S.\ s\phi \leq t\phi$$

THEOREM 4.13 (COMPLETENESS).

$$\forall \phi \in \mathcal{S}_\varnothing.\quad (\mathsf{dom}(\phi) \cap \Delta = \varnothing \quad \textit{and} \quad \forall (s,t) \in S.\ s\phi \leq t\phi) \\ \Rightarrow\ (\exists \phi' \in \mathsf{tally}(S, \Delta, \varnothing).\ \exists \phi'' \in \mathcal{S}_\varnothing.\ \phi \simeq \phi'' \circ \phi')$$

4.3.3 Quasi-constant rows extension. Let Δ be a set of monomorphic variables and S a set of constraints. The tallying extension of the previous section only finds row substitutions of the form $\{\rho \rightsquigarrow \langle\ \mid f\rangle\}$, instead of the more general form $\{\rho \rightsquigarrow \langle \vec{B} \mid f\rangle\}$. In this section, we use this limited tallying algorithm to solve a more general problem: given a finite set of labels $L = \{\ell_1, ..., \ell_n\}$, finding all substitutions of the form $\{\rho \rightsquigarrow \langle \vec{B} \mid f\rangle\}$ where $\mathsf{labels}(\langle \vec{B} \mid f\rangle) \subseteq L$. We can choose, for instance, L to be the set of all labels appearing in the initial constraints S.

The idea is to run our previous tallying algorithm on an instance where row variables are renamed depending on the label they capture. For that, we apply the following substitution to the initial constraints:

$$\phi = \{\rho \rightsquigarrow \langle \ell_1 : \rho_1,\ \ldots,\ \ell_n : \rho_n \mid \rho\rangle\}_{\rho \in V}$$

where V is the set of all row variables that appear in the initial constraints S but not in Δ, and where all the $\{\rho_i\}_{\rho\in V, i\in 1..n}$ are fresh row variables. This yields a new set of constraints $S' = \{(s\phi, t\phi) \mid (s,t) \in S\}$. Each solution of that new tallying instance can then be transposed to the initial problem, yielding this set of solutions for the initial problem:

$$\mathsf{tally}(S, \Delta, L) = \{(\phi'' \circ \phi' \circ \phi)|_{V \cup \mathcal{V}_{\mathsf{Ty}}} \mid \phi' \in \mathsf{tally}(S', \Delta, \varnothing)\}$$

where $\phi'' = \{\rho_i \rightsquigarrow \langle\ \mid \rho\rangle\}_{\rho \in V, i \in 1..n}$ and $\phi|_{V\cup\mathcal{V}_{\mathsf{Ty}}}$ denotes the restriction of ϕ to the domain $V \cup \mathcal{V}_{\mathsf{Ty}}$.

For instance, let us consider the following tallying instance:

$$S = \{(\{\ \mid \rho\},\ \{\ell_1 : \mathtt{int},\ \ell_2 : \mathtt{bool} \mid \mathbb{1}?\})\}$$

We consider the set of labels $L = \{\ell_1, \ell_2\}$ and the associated substitution $\phi = \{\rho \rightsquigarrow \langle \ell_1 : \rho_{\ell_1},\ \ell_2 : \rho_{\ell_2} \mid \rho\rangle\}$. Our set of constraints becomes:

$$S' = S\phi = \{(\{\ell_1 : \rho_{\ell_1},\ \ell_2 : \rho_{\ell_2} \mid \rho\},\ \{\ell_1 : \mathtt{int},\ \ell_2 : \mathtt{bool} \mid \mathbb{1}?\})\}$$

Now, we call the tallying algorithm, which finds the following set of substitutions:

$$\begin{aligned} \{\phi_1, \phi_2, \phi_3, \phi_4\} \text{ where } \phi_1 &= \{\rho_{\ell_1} \rightsquigarrow \langle\ \mid \rho_{\ell_1} \wedge \mathtt{int}\rangle\ ;\ \rho_{\ell_2} \rightsquigarrow \langle\ \mid \rho_{\ell_2} \wedge \mathtt{bool}\rangle\} \\ \phi_2 &= \{\rho_{\ell_1} \rightsquigarrow \langle\ \mid \mathbb{0}\rangle\} \\ \phi_3 &= \{\rho_{\ell_2} \rightsquigarrow \langle\ \mid \mathbb{0}\rangle\} \\ \phi_4 &= \{\rho \rightsquigarrow \langle\ \mid \mathbb{0}\rangle\} \end{aligned}$$

Finally, these solutions are transposed to the initial problem as follows:

$$\begin{aligned} \{\phi_1, \phi_2, \phi_3, \phi_4\} \text{ where } \phi_1 &= \{\rho \rightsquigarrow \langle \ell_1 : \rho \wedge \mathtt{int},\ \ell_2 : \rho \wedge \mathtt{bool} \mid \rho\rangle\} \\ \phi_2 &= \{\rho \rightsquigarrow \langle \ell_1 : \mathbb{0} \mid \rho\rangle\} \\ \phi_3 &= \{\rho \rightsquigarrow \langle \ell_2 : \mathbb{0} \mid \rho\rangle\} \\ \phi_4 &= \{\rho \rightsquigarrow \langle \ell_1 : \rho,\ \ell_2 : \rho \mid \mathbb{0}\rangle\} \end{aligned}$$

Soundness and completeness. This method finds all the solutions whose rows have the form $\langle \vec{B} \mid f\rangle$ such that $\mathsf{labels}(\langle \vec{B} \mid f\rangle) \subseteq L$ (i.e. rows that are constant over $\mathcal{L} \setminus L$). Actually, for some tallying instances such as the one above, it is not possible to capture the set of all solutions (with no restriction over the shape of the rows) with only finitely-many type substitutions. For instance, in our previous example, for any $\ell \in \mathcal{L} \setminus \{\ell_1, \ell_2\}$, the following substitution is a solution that is not captured by our set of solutions:

$$\phi_\ell = \{\rho \rightsquigarrow \langle \ell : \mathbb{0} \mid \rho\rangle\}$$

There is no type substitution that is solution of our tallying instance and that subsumes all the ϕ_ℓ: for that, we would need a type substitution that maps ρ to *any row that have at least one* $\mathbb{0}$ *field*, which is not expressible within our formalism.

The soundness and completeness of this extended tallying algorithm can be expressed as follows:

Theorem 4.14 (Soundness).

$$\forall\phi \in \mathsf{tally}(S,\Delta,L).\quad \phi \in \mathcal{S}_L \quad \textit{and} \quad \mathsf{dom}(\phi)\cap\Delta=\varnothing \quad \textit{and} \quad \forall(s,t)\in S.\ s\phi \le t\phi$$

Theorem 4.15 (Completeness).

$$\begin{aligned}\forall\phi \in \mathcal{S}_L.\quad &(\mathsf{dom}(\phi)\cap\Delta=\varnothing \quad \textit{and} \quad \forall(s,t)\in S.\ s\phi \le t\phi)\\ &\Rightarrow\ (\exists\phi' \in \mathsf{tally}(S,\Delta,L).\ \exists\phi'' \in \mathcal{S}_L.\ \phi \simeq \phi''\circ\phi')\end{aligned}$$

Proofs of these two theorems (and those of the previous sections) can be found in Appendix A.

4.4 Comparison with Castagna and Peyrot [7]

We now compare our approach with the closest prior work [7] in detail. The two systems make dual design choices: we allow Boolean combinations of row variables inside record constructors but keep substitutions simple, while [7] keeps record constructors simple (either a single row variable in the tail, or $\mathbf{..}$ for open records, or ϵ for closed records) but allows substitutions to map row variables to Boolean combinations of rows. We argue that neither system strictly subsumes the other at the type level, but that our choice is simpler to formalize and leads to better algorithmic properties. The main differences are summarized in Table 1.

Expressivity of substitutions. In our system, applying a substitution to a record atom always yields a single record atom: if r is a record atom and ϕ a substitution, then $r\phi$ is again a record atom. This is because each row variable in r is replaced by a single row. In [7], applying a substitution may produce a Boolean combination of record atoms. For instance, if ϕ maps ρ to $R_1 \vee R_2$ where R_1 and R_2 have different explicit labels, then $\{\ |\ \rho\}\phi$ produces $\{\ |\ R_1\}\vee\{\ |\ R_2\}$[3], a union of two record atoms with different shapes. This allows expressing solutions to tallying instances that are not expressible within our formalism. For instance, consider a function `foo` of type $\{\ell_1 : \mathtt{int}\ |\ \rho\} \to \{\ell_1 : \mathtt{int}\ |\ \rho\}$ (for instance, a function that increments the field ℓ_1, leaving other fields unchanged) applied to an argument of type $\{\ell_1 : 42,\ \ell_2 : 42,\ \ell_3 : 42\} \vee \{\ell_1 : 73,\ \ell_2 : 73,\ \ell_3 : 73\}$. In [7], a solution to the underlying tallying instance is the substitution

$$\{\rho \rightsquigarrow \langle \ell_2 : 42,\ \ell_3 : 42\ |\ \rho\rangle \vee \langle \ell_2 : 73,\ \ell_3 : 73\ |\ \rho\rangle\}$$

which yields for this application the type $\{\ell_1 : \mathtt{int},\ \ell_2 : 42,\ \ell_3 : 42\} \vee \{\ell_1 : \mathtt{int},\ \ell_2 : 73,\ \ell_3 : 73\}$. Within our formalism, however, the type of the argument must be unified with a single row, such as $\{\rho \rightsquigarrow \langle \ell_2 : 42 \vee 73,\ \ell_3 : 42 \vee 73\ |\ \rho\rangle\}$, yielding the type $\{\ell_1 : \mathtt{int},\ \ell_2 : 42\vee 73,\ \ell_3 : 42 \vee 73\}$ for the application (which is strictly less precise). Note that this limitation is not specific to polymorphic records: the same issue may occur with type variables occuring inside a type constructor (e.g. arrows, products); it is known as the *expansion problem* or the *application problem* [5, Section 3.2.3].

Per-field set operations. Conversely, our system can express a record type $\{\ |\ \rho_1 \vee \rho_2\}$, which has a per-field semantics: for each label, the field value may come from ρ_1 or from ρ_2, independently of the other labels. The type $\{\ |\ \rho_1\} \vee \{\ |\ \rho_2\}$, the closest expression available in [7], is strictly more restrictive: it requires that *all* fields agree on which row they come from. The `mix` example in the introduction (Section 1) exploits this distinction: it can be typed $\{\ |\ \rho_1\} \to \{\ |\ \rho_2\} \to \{\ |\ \rho_1 \vee \rho_2\}$ within our formalism, while in [7] it cannot be typed polymorphically but can only be given a monomorphic type such as $\{\ \mathbf{..}\} \to \{\ \mathbf{..}\} \to \{\ \mathbf{..}\}$.

[3]Here, the term $\{\ |\ R\}$ is just a notation that denotes the record atom containing the same bindings as the row R.

Tallying completeness. Both systems face limitations in tallying, but of different kinds. In [7], the tallying algorithm is incomplete: there exist tallying instances for which no finite set of substitutions can represent all solutions, and the algorithm may miss solutions entirely. In our system, the tallying algorithm is complete for all solutions in $\mathcal{S}_L$, i.e., solutions whose rows are constant over labels not in L (where L is the set of labels appearing in the constraints). The solutions outside $\mathcal{S}_L$ that we miss are those involving rows with explicit bindings on labels that do not appear anywhere in the constraints. We conjecture that missing such solutions is not an issue in practice during type inference, since we do not expect a type inference to manipulate types that introduce labels that do not appear in the program being typed.

	Castagna–Peyrot [7]	**This paper**
Record tail	single row variable (ρ, ϵ, or **..**)	Boolean combination of row variables and option types
Substitutions	$\rho \rightsquigarrow$ Boolean combination of rows	$\rho \rightsquigarrow$ single row
Limitations	impossible to express per-field union (*cf.* `mix`)	a union of records is collapsed into a single atom when unified with a row variable (*cf.* `foo`)
Tallying	incomplete	complete for $\mathcal{S}_L$

Table 1. Comparison with Castagna and Peyrot [7].

5 Applications

We show a practical application of our approach to row polymorphism in the context of R, a popular programming language used for statistical computing and data science. It has been designed for interactive use. As such, it contains features that make it ergonomic to use from a REPL, but that are difficult to type with a traditional type system [17]. In this section we first detail how we encode these features, followed by an overview of the implementation, which extends both the set-theoretic type library and the type-checker.

5.1 Encoding of R data structures

R features several built-in data structures that may carry labels and hold a variable number of elements: function parameters and arguments, lists and attributes. We show that it is possible to type these data structures without modifying the underlying set-theoretic type algebra. Concretely, we look at function parameters and arguments, lists and classes.

5.1.1 Function parameters and arguments. Functions in R can have optional parameters. When not provided, a parameter receives either a default value specified in the function definition or the special `missing` value. Parameters can be variadic, in which case extra arguments are captured by the ellipsis parameter (`...`). Unlike in many other languages, the ellipsis parameter in R may appear anywhere in the parameter list and not necessarily at the end.

For example, the function definition

```
foo <- function(a, ..., b) { }
```

expects at least two arguments bound to parameters `a` and `b` (which must be named explicitly since it follows the ellipsis), along with any number of additional arguments captured by the ellipsis. Arguments passed to a function call can be either positional or named. From the type perspective, we treat all arguments as named; positional arguments are assigned a label representing their position ($p_1, p_2, \ldots$). For instance, the call

```
foo(1, 2, 3, b=42)
```

has three positional arguments and one named argument `b`. In the callee, `a` is bound to 1, `b` to 42, and the ellipsis captures a list containing 2 and 3.

We encode arguments as records. That yields the following encoding, where $\mathtt{A}(.)$ is a *tag* type constructor [14] used to identify the encoding and make it disjoint from plain records:

$$\llbracket \mathtt{@}(t_1,\ t_2?,\ \ell : t) \rrbracket = \mathtt{A}(\{p_1 : \llbracket t_1 \rrbracket,\ p_2 : \llbracket t_2 \rrbracket?,\ \ell : \llbracket t \rrbracket \mid \mathbb{0}?\}) \qquad \text{(closed record type)}$$
$$\llbracket \mathtt{@}(t_1,\ t_2?,\ \ell : t\ ..) \rrbracket = \mathtt{A}(\{p_1 : \llbracket t_1 \rrbracket,\ p_2 : \llbracket t_2 \rrbracket?,\ \ell : \llbracket t \rrbracket \mid \mathbb{1}?\}) \qquad \text{(open record type)}$$
$$\llbracket \mathtt{@}(t_1,\ t_2?,\ \ell : t \mid \rho) \rrbracket = \mathtt{A}(\{p_1 : \llbracket t_1 \rrbracket,\ p_2 : \llbracket t_2 \rrbracket?,\ \ell : \llbracket t \rrbracket \mid \rho\}) \qquad \text{(polymorphic record type)}$$

To demonstrate the expressivity of this encoding, let's consider `lapply`:

```
lapply <- function(X, FUN, ...)
```

an R function equivalent to `map` in functional programming languages. It applies `FUN` to every element of `X`, forwarding any additional arguments captured by the ellipsis to each function call. For example, the call:

```
lapply(list(1:10, c(1, NA)), mean, na.rm = TRUE)
```

applies `mean` to each of the two list elements, passing the additional argument `na.rm` to each call:

```
mean(1:10, na.rm = TRUE); mean(c(1, NA), na.rm = TRUE)
```

The result is a list of two elements `(5.5, 1)`. The first is the mean of the integers from 1 to 10, and the second is the mean of the vector `c(1, NA)` with `NA` values removed. Using our encoding of function arguments, `lapply` can be assigned the following type:

$$\mathtt{lapply} : \mathtt{@}(\mathtt{list}(\alpha), (\mathtt{@}(\alpha \mid \rho) \to \beta) \mid \rho) \to \mathtt{list}(\beta)$$

It takes an argument record with two positional parameters and an arbitrary tail ρ. The first positional parameter is a list[4] of elements of type α. The second positional parameter is a function that accepts an argument record with one positional parameter of type α and the *same* tail ρ—meaning any extra named arguments captured by the ellipsis in the outer call are forwarded verbatim to `FUN`. The return type of `FUN` is β, and `lapply` returns a list of β values.

5.1.2 Lists. Lists are generic containers that can hold heterogeneous data. They serve a dual role in R: they can be used as records, where elements are named and accessed by label, or as indexed sequences. They are also used to represent data frames, a rectangular data structure in which each list element represents a column. R provides a rich set of operators for projecting data from lists. For example, the `$` operator accesses named elements (`rec$id`), while the `[[.]]` operator accesses elements either by name (`rec[["id"]]`) or by index (`rec[[1]]`). The index can be any expression that evaluates to an integer, a character,[5] or a logical vector. Listing 1 shows a sequence of R list operations and their corresponding types using our encoding.

We represent lists as records using a similar encoding as for arguments, but using braces, $\{.\}$, instead of parentheses, $\mathtt{@}(.)$. Fields indexed by integers do not receive explicit bindings. They are captured by the tail of the record. To infer the types shown in Listing 1, we use the following type

[4]For simplification, we assume it must be a list, in practice it can be any object that can be converted to a list.
[5]In R, a string is called *character*...

```
# 1. Create a list with a single named element.
xs <- list(a=1)     # xs : {a : 1}
# 2. Add a new named element to the list.
xs$b <- 2           # xs : {a : 1, b : 2}
# 3. Create a second list with a single named element.
ys <- list(c=3)     # ys : {c : 3}
# 4. Create a new list by appending ys after xs
zs <- append(xs, ys) # zs : {a : 1, b : 2, c : 3}
# 5. Access the second element by index.
n <- zs[[2]]        # n  : 1 v 2 v 3
# 6. Update the first element by index.
zs[[1]] <- n        # zs : {a : 1 v 2 v 3, b : 1 v 2 v 3, c : 1 v 2 v 3}
```

Listing 1. Example of list operations in R

definitions for the built-in list operations:

$$\begin{aligned}(\texttt{\$<-})_{\mathsf{b}} &: @(\{\mathsf{b} : \mathbb{1}? \mid \rho\}, \alpha) \to \{\mathsf{b} : \alpha \mid \rho\} \\ \mathsf{append} &: @(\{ \mid \rho_1 \land \mathbb{0}? \lor \rho_1' \land \mathbb{1}\}, \{ \mid \rho_2 \land \mathbb{0}? \lor \rho_2' \land \mathbb{1}\}) \to \{ \mid (\rho_1 \land \rho_2) \lor (\rho_1' \lor \rho_2')\} \\ \texttt{[[.]]} &: @(\{ \mid \alpha?\}, \texttt{int}) \to \alpha \\ \texttt{[[.]]<-} &: @(\{ \mid \rho\}, \texttt{int}, \alpha) \to \{ \mid \rho \lor \alpha\}\end{aligned}$$

The functions $(\texttt{\$<-})_{\mathsf{b}}$ and `[[.]]<-` are typed as if they return a new list. They can be combined with a variable assignment primitive to reproduce the behavior of the R program above. For append, every named field appearing in either argument will appear in the result. For named fields that appear in both arguments, the result can be seen as non-deterministically selecting the value from one or the other. Focusing on the type of append, the fields of the first argument are captured by the row variables ρ_1 and ρ_1': the former only captures the *absence* of a field while the latter only captures the type of the associated value *assuming it is present*. The same is done for the second argument. The result is a list such that each field has type $(\rho_1 \land \rho_2) \lor (\rho_1' \lor \rho_2')$: it is absent if and only if it is absent in both arguments $(\rho_1 \land \rho_2)$, and the type of the associated value, when non-absent, is the union of the types it has in both arguments $(\rho_1' \lor \rho_2')$.

We type the function `[[.]]` as if it selects any value present in the list. We cannot be more precise in general as (*i*) the key we lookup may be given by an expression that cannot be statically resolved to a constant, and (*ii*) even if the key is an integer constant, this does not help since we do not track element positions. For the same reasons, the type of the function `[[.]]<-` unions every field with the type of the assigned element.

5.1.3 Classes. There are multiple object systems in R; here we focus on the most commonly used one, S3. Any value that is not a symbol or `NULL` or `NA` can have attributes attached. If one of these attributes is named `class`, S3 generics will dynamically dispatch to methods based on the attribute's value. The value is a character vector that can contain one or more class names.

Let $\langle \mathsf{c}_1, \ldots, \mathsf{c}_n \rangle$ denote the set of values that have the classes $\mathsf{c}_1, \ldots, \mathsf{c}_n$ attached. This type must satisfy the following subtyping properties:

$$\langle \mathsf{c}_1, \ldots, \mathsf{c}_n \rangle \leq \langle \mathsf{c}_1', \ldots, \mathsf{c}_m' \rangle \iff \{\mathsf{c}_1, \ldots, \mathsf{c}_n\} = \{\mathsf{c}_1', \ldots, \mathsf{c}_m'\} \qquad \text{(invariance)}$$

$$\langle \mathsf{c}_1, \ldots, \mathsf{c}_n \rangle \land \langle \mathsf{c}_1', \ldots, \mathsf{c}_m' \rangle \simeq \mathbb{0} \iff \{\mathsf{c}_1, \ldots, \mathsf{c}_n\} \neq \{\mathsf{c}_1', \ldots, \mathsf{c}_m'\} \qquad \text{(disjointness)}$$

The encoding must also capture the behavior of functions that add or remove a class from one of their arguments. In R, a function may take an object and modify it by stacking a new class on top in order to change its behavior for downstream functions.

```
xs <- data.frame(id=sample(c("a","b"), 5, replace=T), val=rnorm(5))
#   id        val
# 1  b -1.2143181
# ...
# 5  a -0.1625002
class(xs) # "data.frame"
ys <- group_by(xs, id)
class(ys) # "grouped_df" "data.frame"
# ...
zs <- ungroup(ys)
class(zs) # "data.frame"
```

Listing 2. Example of class-based state changes in R

Listing 2 shows an example of such state changes, used extensively in R code that uses the `dplyr` [19] package (one of the most popular packages in the R ecosystem). It starts with a two column data frame `xs`. The call `group_by(xs, id)` does not modify the underlying data, instead it prepends the class `grouped_df` to the `xs` class attribute (as well as adding the grouping variable into a new object attribute). This changes the behavior of downstream operations, which then act group-wise. Prepending `grouped_df` is important since functions that understand this class dispatch to specialized methods, while others can still fall back to normal data-frame methods. Conversely, `ungroup` removes the grouping class, restoring the usual data-frame behavior. This add/remove-class idiom is common in R and is precisely what our encoding must capture.

The `group_by` and `ungroup` functions may be given the following types (we ignore the shape of values and only consider their classes here):

$$\texttt{group\_by} : \texttt{@}(\texttt{<} \mid \rho\texttt{>}) \to \texttt{<grouped\_df} \mid \rho\texttt{>}$$
$$\texttt{ungroup} : \texttt{@}(\texttt{<grouped\_df} \mid \rho\texttt{>}) \to \texttt{<} \mid \rho\texttt{>}$$

The encoding we propose for the <.> type constructor is as follows. Intuitively, we represent a set of classes by its characteristic function (i.e. for a set of classes C, the function f such that $f(\mathtt{c}) = 1$ if $\mathtt{c} \in C$, and $f(\mathtt{c}) = 0$ otherwise). This characteristic function is in turn represented by a record:

$$\llbracket \texttt{<}\mathtt{c}_1, \mathtt{c}_2\texttt{>} \rrbracket = \mathtt{C}(\{\mathtt{c}_1 : \mathtt{true},\ \mathtt{c}_2 : \mathtt{true} \mid \mathtt{false}\})$$
$$\llbracket \texttt{<}\mathtt{c}_1, \mathtt{c}_2 \ldots\texttt{>} \rrbracket = \mathtt{C}(\{\mathtt{c}_1 : \mathtt{true},\ \mathtt{c}_2 : \mathtt{true} \mid \mathtt{bool}\})$$
$$\llbracket \texttt{<}\mathtt{c}_1, \mathtt{c}_2 \mid \rho\texttt{>} \rrbracket = \mathtt{C}(\{\mathtt{c}_1 : \mathtt{true},\ \mathtt{c}_2 : \mathtt{true} \mid \mathtt{bool} \wedge \rho\})$$

This encoding does satisfy the invariance and disjointness properties we want, and reuses the row-polymorphism of records to implement polymorphism for classes.

5.2 Implementation

We implemented our approach in two stages. First, we extended the SSTT library [13] with native support for row-polymorphic records and the corresponding operations. We then integrated these extensions into the MLsem [12] type-checker and used the resulting prototype to validate the encodings presented above on representative examples from R.

Extension of the set-theoretic type library SSTT. Our row-polymorphic records have been implemented within the set-theoretic type library SSTT [13]. This extension features:

- A definition for row type variables, record atoms, and rows,
- An updated definition of set-theoretic types that features record atoms,
- An updated definition of substitutions that can now map row variables to rows,

```
> [ { ;; `R } <= { l1: int ; l2: bool ;; any? } ];;
[ `R: { l1 : `R ; l2 : `R ;; empty } ]
[ `R: { l1 : empty ;; `R } ]
[ `R: { l1 : `R & int ; l2 : `R & bool ;; `R } ]
[ `R: { l2 : empty ;; `R } ]
```

Listing 3. Example session in the SSTT REPL typing the tallying instance from Section 4.3.3

- The implementation of the application of a type substitution on a type (Section 3),
- The extension of the subtyping and tallying algorithms (Section 4),
- An extension of the type pretty-printer and parser.

Our implementation can be tested using the REPL of SSTT, available with this paper as a software artifact (*cf.* Listing 3). Row variables are preceded by a backtick (`` ` ``), type variables are preceded by a single quote (`'`), and `;;` separates the bindings from the tail of a record. Type unions and intersections are denoted `|` and `&` respectively.

Extension of the type-checker MLsem. In order to evaluate our approach to row polymorphism, we extended the set-theoretic type checker MLsem [12] to use our extended version of the SSTT library. Adding support for row-polymorphism was then straightforward, and only consisted in:

- Adding to the language record constructors, as well as field update and deletion operators. We type them as regular functions, whose signature depends on the label ℓ that is being updated or removed:

$$\texttt{update}_\ell : \ \mathbf{\{}\ell : \mathbb{1}? \mid \rho\mathbf{\}} \to \alpha \to \mathbf{\{}\ell : \alpha \mid \rho\mathbf{\}} \qquad \texttt{remove}_\ell : \ \mathbf{\{}\ell : \mathbb{1}? \mid \rho\mathbf{\}} \to \mathbf{\{}\ell : \mathbb{0}? \mid \rho\mathbf{\}}$$

- Tracking the row variables in the environment to differentiate bound variables from free variables (this is needed to determine the parameter Δ of the tallying algorithm).
- Updating the type simplification heuristics to get rid of redundant row variables that may appear after solving tallying constraints. For instance, a row variable that only appears in covariant positions (resp. contravariant positions) can be substituted by $\mathbb{1}?$ (resp. $\mathbb{0}$).

Demonstration. We include below a few snippets written in the surface language of MLsem (whose syntax is close to OCaml's) illustrating the encodings defined in Section 5.1. The keyword `val` is used to specify the signature of a variable without giving it a definition. The keyword `let` is used to define a variable, whose type is inferred (types inferred are written as comments). The type `[t*]` denotes a list whose elements are of type `t`.

```
(* Encoding of R arguments: typing lapply (Section 5.1.1) *)
val mean: {p1: [(int|Na)*] ; na_rm: true} | {p1: [int*] ; na_rm: false?}
        -> [int*] (* Na allowed if and only if na_rm=true *)
val lapply: {p1:['a*]; p2: {p1:'a; p2:empty? ;; `r} -> 'b ;; `r} -> ['b*]
let test_lapply = (* Type inferred: [ [int*]* ] *)
  lapply { p1=[[1;2;3;4;5;6;7;8;9;10];[1;Na]] ; p2=mean ; na_rm=true }
```

The definition `test_lapply` calls the function `lapply` with three parameters: (i) a first positional parameter defining a list of two elements whose type—a list of lists of `int|Na`—is captured by the type variable `'a` in the signature of `lapply`, (ii) a second positional parameter that indicates the function we want to apply on each element, and (iii) a named parameter labeled `na_rm` of type `true`, captured by the row variable `` `r `` in the signature of `lapply`. To type-check the application, the type of the second argument, `mean`, must be "unified" (using tallying) with ``{p1:'a; p2:empty? ;; `r} -> 'b``, which is done by selecting the branch `{p1: [(int|Na)*] ; na_rm: true} -> [int*]` of `mean`.

```
(* Encoding of R lists as records (Section 5.1.2) *)
let test_r_lists =
  let mut xs = { a=1 } in   (* xs <- list(a=1) *)
  xs := set_b xs 2 ;        (* xs$b <- 2 *)
  let mut ys = { c=3 } in   (* ys <- list(c=3) *)
  let mut zs = append xs ys in (* zs <- append(xs, ys) *)
  let mut n = get zs 2 in   (* n <- zs[[2]] *)
  zs := set zs 1 n ;        (* zs[[1]] <- n *)
  zs (* Type inferred: { b:(1..3) ; a:(1..3) ; c:(1..3) ;; (1..3)? } *)
```

The definition `test_r_lists` exactly replicates the operations of Section 5.1.2 (we use similar signatures for `set_b`, `set`, `get` and `append`). Note that the mutation operations, `set` and `set_b`, are encoded with an assignment. This mimics the semantics of R which implements value independence: a list that may be shared is never mutated in place; instead the list is duplicated, and the copy is mutated and reassigned to the variable on the left-hand-side of the mutation operator.

```
(* Encoding of R classes (Section 5.1.3) *)
val data_frame : () -> { data_frame:true ;; false }
val group_by : { ;; bool & `c } -> string -> { grouped_df:true ;; bool & `c }
val ungroup : { grouped_df:true ;; bool & `c } -> { grouped_df:false ;; bool & `c }
let test_classes =
  let xs = data_frame () in    (* xs <- data.frame(...) *)
  let ys = group_by xs "id" in (* ys <- group_by(xs, id) *)
  let zs = ungroup ys in       (* zs <- ungroup(ys) *)
  zs (* Type inferred: { data_frame : true ;; false } *)
```

The definition `test_classes` replicates the operations of Section 5.1.3. The type inferred corresponds to the encoding of `<data_frame>`.

6 Conclusion

We presented a new approach to row polymorphism for set-theoretic types. By allowing Boolean combinations of row variables inside record type constructors rather than in type substitutions, we obtain a system that is simple to formalize and implement, yet expressive enough to type operations that combine fields from multiple records in non-trivial ways. Our extension of the tallying algorithm is complete for all solutions whose rows are constant over labels not mentioned in the constraints, a property that the alternative approach of enriching substitutions with Boolean combinations of rows does not enjoy.

We implemented our approach in the set-theoretic type library SSTT and the type checker MLsem. The integration required only modest changes to the existing type checker: record constructors and operations are typed as regular polymorphic applications. Our encodings of R data structures—heterogeneous lists, variadic function arguments, and S3 class dispatch—demonstrate that a single mechanism, row-polymorphic records, can capture the typing behavior of diverse language features without special-purpose extensions to the type algebra.

Several directions remain open. Our tallying completeness result is restricted to solutions whose rows are constant outside a finite set of labels; whether a finite representation exists for the full set of solutions is an open question. Finally, applying this approach to a full-scale type checker for R or another dynamic language would test its practical limits and could motivate further language-specific extensions, such as support for mutable fields.

Data Availability Statement

All the definitions and proofs that we omitted from the main text are available in the appendices of the extended version. Our implementation of the algorithms presented in this paper, as well as the examples of Section 5, is available in the supplementary material and will be submitted to artifact evaluation in case of acceptance.

References

[1] Anonymized. 2026. Anonymized article (under review). (2026).

[2] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. 2003. CDuce: an XML-centric general-purpose language. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming* (Uppsala, Sweden) *(ICFP '03)*. Association for Computing Machinery, New York, NY, USA, 51–63. doi:10.1145/944705.944711

[3] Giuseppe Castagna. 2023. Typing Records, Maps, and Structs. *Proc. ACM Program. Lang.* 7, ICFP, Article 196 (Aug. 2023), 44 pages. doi:10.1145/3607838

[4] Giuseppe Castagna and Alain Frisch. 2005. A gentle introduction to semantic subtyping. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Lisbon, Portugal) *(PPDP '05)*. Association for Computing Machinery, New York, NY, USA, 198–208. doi:10.1145/1069774.1069793

[5] Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. 2015. Polymorphic functions with set-theoretic types. Part 2: local type inference and type reconstruction. In *Proceedings of the 42nd Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '15)*. 289–302. doi:10.1145/2676726.2676991

[6] Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. 2014. Polymorphic Functions with Set-Theoretic Types. Part 1: Syntax, Semantics, and Evaluation. In *Proceedings of the 41st Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '14)*. 5–17. doi:10.1145/2676726.2676991

[7] Giuseppe Castagna and Loïc Peyrot. 2025. Polymorphic Records for Dynamic Languages. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 132 (April 2025), 28 pages. doi:10.1145/3720497

[8] Giuseppe Castagna and Zhiwu Xu. 2011. Set-theoretic foundation of parametric polymorphism and subtyping. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 94–106. doi:10.1145/2034773.2034788

[9] Chun Yin Chau and Lionel Parreaux. 2026. The Simple Essence of Boolean-Algebraic Subtyping: Semantic Soundness for Algebraic Union, Intersection, Negation, and Equi-recursive Types. *Proc. ACM Program. Lang.* 10, POPL, Article 47 (Jan. 2026), 30 pages. doi:10.1145/3776689

[10] Alain Frisch. 2004. *Theory, conception and realisation of a programming language adapted to XML*. Ph. D. Dissertation. Université Paris Diderot. https://www.irif.fr/_media/users/gduboc/these-frisch-eng.pdf

[11] Nils Gesbert, Pierre Genevès, and Nabil Layaïda. 2015. A logical approach to deciding semantic subtyping. *ACM Transactions on Programming Languages and Systems* 38, 1 (2015), 3. doi:10.1145/2812805

[12] Mickaël Laurent. 2026. *MLsem: Type-checker for Dynamic Languages*. doi:10.5281/zenodo.18724039

[13] Mickaël Laurent and Kim Nguyen. 2026. *SSTT: Simple Set-Theoretic Types library*. doi:10.5281/zenodo.18723936

[14] Mickaël Laurent and Jan Vitek. 2026. Type Inference for Functional and Imperative Dynamic Languages. (2026). https://mlaurent.ovh/publications/stt_implem.pdf

[15] Francois Pottier and Didier Rémy. 2005. *The Essence of ML Type Inference*. 389–489.

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

[17] Alexi Turcotte, Aviral Goel, Filip Křikava, and Jan Vitek. 2020. Designing types for R, empirically. OOPSLA (2020). https://doi.org/10.1145/3428249

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

[19] Hadley Wickham, Romain François, Lionel Henry, and Kirill Müller. [n. d.]. *dplyr: A Grammar of Data Manipulation (R package)*. https://dplyr.tidyverse.org

A Full formalism with proofs

This appendix regroups the definitions and formal results of the paper, including proofs.

A.1 Syntax

This section includes the proofs relative to Section 3.1.

Definition A.1 (Set-theoretic types). The sets $\mathcal{T}$ of *set-theoretic types*, $\mathcal{F}$ of *field types*, and $\mathcal{R}$ of rows are the sets of regular and contractive terms t, f, and R coinductively defined by the following grammar:

$$\begin{array}{lrcl}
\textbf{Types} & t, s & ::= & b \mid \alpha \mid t \to t \mid t \times t \mid r \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \\
\textbf{Option Types} & \bar{t}, \bar{s} & ::= & t \mid t? \\
\textbf{Field Types} & f & ::= & \bar{t} \mid \rho \mid f \wedge f \mid f \vee f \mid \neg f \\
\textbf{Record Bindings} & \vec{B} & ::= & \ell : f,\ \ldots,\ \ell : f \qquad \text{(with all } \ell \text{ distinct)} \\
\textbf{Record Atoms} & r & ::= & \{\vec{B} \mid f\} \\
\textbf{Rows} & R & ::= & \langle \vec{B} \mid f \rangle
\end{array}$$

where $b \in \mathcal{B}$ is a base type, $\alpha \in \mathcal{V}_{\mathsf{Ty}}$ is a type variable, and $\rho \in \mathcal{V}_{\mathsf{Row}}$ is a row variable. The notation $t_1 \setminus t_2$ is a syntactic sugar for $t_1 \wedge \neg t_2$. When writing a term, we use the following precedence (by decreasing priority): $\neg$, $\setminus$, $\wedge$, $\vee$, $\times$, $\to$. Syntactic equality between two types t_1 and t_2 is noted $t_1 = t_2$.

Definition A.2 (Labels, tail, projection). Let $R = \langle \vec{B} \mid f \rangle$ be a row and $r = \{\vec{B} \mid f\}$ be a record atom. We define the labels of r, written $\mathsf{labels}(r)$ (resp. the labels of R, written $\mathsf{labels}(R)$) and the tail of r, written $\mathsf{tl}(r)$ (resp. the tail of R, written $\mathsf{tl}(R)$), as well as the projection of r on a label ℓ, written $r(\ell)$ (resp. the projection of R on a label ℓ, written $R(\ell)$) as follows:

$$\begin{array}{ccccc}
\mathsf{labels}(R) & = & \mathsf{labels}(r) & = & \{\ell \mid (\ell, f) \in \vec{B}\} \\
\mathsf{tl}(R) & = & \mathsf{tl}(r) & = & f
\end{array}
\qquad
R(\ell) = r(\ell) = \begin{cases} f_\ell & \text{if } (\ell : f_\ell) \in \vec{B} \\ f & \text{otherwise} \end{cases}$$

Definition A.3 (Type substitution). A *type substitution* is a function $\phi : \mathcal{V}_{\mathsf{Ty}} \cup \mathcal{V}_{\mathsf{Row}} \to \mathcal{T} \cup \mathcal{R}$ mapping type variables to types and row variables to rows, and which is the identity everywhere except for a finite set of type variables and row variables called its domain and denoted by $\mathsf{dom}(\phi)$. Formally, $\mathsf{dom}(\phi) = \{\alpha \in \mathcal{V}_{\mathsf{Ty}} \mid \phi(\alpha) \neq \alpha\} \cup \{\rho \in \mathcal{V}_{\mathsf{Row}} \mid \phi(\rho) \neq \langle\ \mid \rho \rangle\}$. We define $\mathsf{labels}(\phi)$ as the set of explicit labels in the rows of ϕ; formally, $\mathsf{labels}(\phi) = \bigcup_{\rho \in \mathsf{dom}(\phi)} \mathsf{labels}(\phi(\rho))$. We note $\{(\rho_i \rightsquigarrow R_i)_{i \in I}, (\alpha_j \rightsquigarrow t_j)_{j \in J}\}$ the substitution mapping each ρ_i to R_i, each α_j to t_j, and that is the identity everywhere else. The set of all type substitutions is written $\mathcal{S}$.

Definition A.4 (Application of a type substitution on a type). The result of the application of a type substitution ϕ on a type t, noted $t\phi$, as well as the application of a type substitution ϕ on a field type f in tail-position or ℓ-position, noted $(t\phi)_{\mathsf{tl}}$ or $(t\phi)_\ell$, are coinductively defined by these equations:

$$\begin{array}{ccc}
\alpha\phi = \phi(\alpha) & \mathbb{0}\phi = \mathbb{0} & \mathbb{1}\phi = \mathbb{1} \\
(\neg t)\phi = \neg(t\phi) & (t_1 \vee t_2)\phi = (t_1\phi) \vee (t_2\phi) & (t_1 \wedge t_2)\phi = (t_1\phi) \wedge (t_2\phi) \\
b\phi = b & (t_1 \to t_2)\phi = (t_1\phi) \to (t_2\phi) & (t_1 \times t_2)\phi = (t_1\phi) \times (t_2\phi) \\
\multicolumn{3}{l}{r\phi = \{(\ell : (r(\ell)\phi)_\ell)_{\ell \in \mathsf{labels}(r)},\ (\ell : (\mathsf{tl}(r)\phi)_\ell)_{\ell \in \mathsf{labels}(\phi)\setminus\mathsf{labels}(r)} \mid (\mathsf{tl}(r)\phi)_{\mathsf{tl}}\}} \\
(\rho\phi)_\ell = \phi(\rho)(\ell) & (t\phi)_\ell = t\phi & ((t?)\phi)_\ell = (t\phi)? \\
((\neg f)\phi)_\ell = \neg(f\phi)_\ell & ((f_1 \wedge f_2)\phi)_\ell = (f_1\phi)_\ell \wedge (f_2\phi)_\ell & ((f_1 \vee f_2)\phi)_\ell = (f_1\phi)_\ell \vee (f_2\phi)_\ell \\
(\rho\phi)_{\mathsf{tl}} = \mathsf{tl}(\phi(\rho)) & (t\phi)_{\mathsf{tl}} = t\phi & ((t?)\phi)_{\mathsf{tl}} = (t\phi)? \\
((\neg f)\phi)_{\mathsf{tl}} = \neg(f\phi)_{\mathsf{tl}} & ((f_1 \wedge f_2)\phi)_{\mathsf{tl}} = (f_1\phi)_{\mathsf{tl}} \wedge (f_2\phi)_{\mathsf{tl}} & ((f_1 \vee f_2)\phi)_{\mathsf{tl}} = (f_1\phi)_{\mathsf{tl}} \vee (f_2\phi)_{\mathsf{tl}}
\end{array}$$

Definition A.5 (Application of a type substitution on a row). The result of the application of a type substitution ϕ on a row R, noted $R\phi$, is defined as follows:

$$R\phi = \langle(\ell : (R(\ell)\phi)_\ell)_{\ell \in \mathsf{labels}(R)},\ (\ell : (\mathsf{tl}(R)\phi)_\ell)_{\ell \in \mathsf{labels}(\phi)\setminus\mathsf{labels}(R)} \mid (\mathsf{tl}(R)\phi)_{\mathsf{tl}}\rangle$$

Lemma A.6. *Let f be a field type, ϕ be a type substitution, and $\ell \in \mathcal{L}$ be a label. We have $\ell \notin \mathsf{labels}(\phi) \Rightarrow (f\phi)_{\mathsf{tl}} = (f\phi)_\ell$.*

Proof. This claim is proved by structural induction on the field type f: for a row variable ρ, we have $(\rho\phi)_\ell = \phi(\rho)(\ell)$ and $(\rho\phi)_{\mathsf{tl}} = \mathsf{tl}(\phi(\rho))$; since $\ell \notin \mathsf{labels}(\phi)$, the label ℓ is not explicit in any $\phi(\rho')$ for $\rho' \in \mathsf{dom}(\phi)$, so in particular $\phi(\rho)(\ell) = \mathsf{tl}(\phi(\rho))$, giving $(\rho\phi)_\ell = (\rho\phi)_{\mathsf{tl}}$. For all other field type forms (t, $t?$, boolean connectives), the $(\cdot)_\ell$ and $(\cdot)_{\mathsf{tl}}$ operations yield the same result by definition, so the induction is straightforward. □

Proposition A.7. *The record atom $r\phi$ is such that for every label ℓ, we have $(r\phi)(\ell) = (r(\ell)\phi)_\ell$. Similarly, the row $R\phi$ is such that, for every label ℓ, we have $(R\phi)(\ell) = (R(\ell)\phi)_\ell$.*

Proof. Let r be a record atom and ϕ be a type substitution. Let $\ell \in \mathcal{L}$ be a label. We show that $(r\phi)(\ell) = (r(\ell)\phi)_\ell$. We have the following cases:

$\ell \in \mathbf{labels}(r)$ We have $(r\phi)(\ell) = (r(\ell)\phi)_\ell$ by the definition of type substitution on record atoms.

$\ell \in \mathbf{labels}(\phi) \setminus \mathbf{labels}(r)$ We have $(r\phi)(\ell) = (\mathsf{tl}(r)\phi)_\ell$ by the definition of type substitution on record atoms. We also have $\mathsf{tl}(r) = r(\ell)$ as $\ell \notin \mathsf{labels}(r)$. We thus have $(r\phi)(\ell) = (r(\ell)\phi)_\ell$.

$\ell \notin \mathbf{labels}(\phi) \cup \mathbf{labels}(r)$ We have $(r\phi)(\ell) = (\mathsf{tl}(r)\phi)_{\mathsf{tl}}$ by the definition of type substitution on record atoms. We also have $\mathsf{tl}(r) = r(\ell)$ as $\ell \notin \mathsf{labels}(r)$. We thus have $(r\phi)(\ell) = (r(\ell)\phi)_{\mathsf{tl}}$. As $\ell \notin \mathsf{labels}(\phi)$, we can apply Lemma A.6, yielding $(r(\ell)\phi)_{\mathsf{tl}} = (r(\ell)\phi)_\ell$, which concludes the proof.

The proof for rows is similar. □

Definition A.8 (Type substitution composition). Given two type substitutions ϕ_1 and ϕ_2, their composition $\phi_2 \circ \phi_1$ is the substitution defined as follows:

$$\forall \alpha \in \mathcal{V}_{\mathsf{Ty}}.\ (\phi_2 \circ \phi_1)(\alpha) = \begin{cases} (\phi_1(\alpha))\phi_2 & \text{if } \alpha \in \mathsf{dom}(\phi_1) \\ \phi_2(\alpha) & \text{if } \alpha \in \mathsf{dom}(s_2) \setminus \mathsf{dom}(s_1) \\ \alpha & \text{otherwise} \end{cases}$$

$$\forall \rho \in \mathcal{V}_{\mathsf{Row}}.\ (\phi_2 \circ \phi_1)(\rho) = \begin{cases} (\phi_1(\rho))\phi_2 & \text{if } \rho \in \mathsf{dom}(\phi_1) \\ \phi_2(\rho) & \text{if } \rho \in \mathsf{dom}(s_2) \setminus \mathsf{dom}(s_1) \\ \langle \mid \rho \rangle & \text{otherwise} \end{cases}$$

A.2 Interpretation

This section includes the proofs relative to Sections 3.2 and 3.3.

Definition A.9 (Quasi-constant functions). Let X, Y denote two sets. A function $F : X \longrightarrow Y$ is quasi-constant if there exists $y \in Y$, called the *default value* of F, denoted by $\mathsf{def}(F)$, such that the set $\mathsf{dom}(F) = \{x \in X \mid F(x) \neq y\}$ is finite. We use $X \twoheadrightarrow Y$ to denote quasi-constant functions from X to Y.

Definition A.10 (Quasi-constant function terms). We use the finite term $\{\!\!\{x_1 = y_1, ..., x_n = y_n, \_ = y\}\!\!\}$, where all the x_i are distinct and all the y_i are different from y, to denote the quasi-constant function F defined by $\forall i \in [1..n].\ F(x_i) = y_i$ and $\forall x \in X \setminus \{x_1, ..., x_n\}.\ F(x) = y$. We have a one-to-one correspondance between such terms and quasi-constant functions.

Definition A.11 (Interpretation domain [11]). The *interpretation domain* $\mathcal{D}$ (respectively *field interpretation domain* $\mathcal{D}_\mathsf{f}$) is the set of finite terms d (respectively δ) produced inductively by the following grammar

$$v ::= c \mid (d, d) \mid \{(d, \partial), \ldots, (d, \partial)\} \mid \{\!\!\{\ell = \delta, \ldots, \ell = \delta, \_ = \delta\}\!\!\}$$
$$d ::= v^L \qquad\qquad \partial ::= d \mid \Omega \qquad\qquad \delta ::= \partial^F$$

where c ranges over the set C of constants, L ranges over finite sets of type variables, F ranges over finite sets of row variables, and where Ω is such that $\Omega \notin C$.

Definition A.12 (Assignment). An assignment η is a function $\mathcal{V}_\mathsf{Ty} \cup \mathcal{V}_\mathsf{Row} \longrightarrow \mathcal{P}(\mathcal{D}) \cup (\mathcal{L} \twoheadrightarrow \mathcal{P}(\mathcal{D}_\mathsf{f}))$ mapping type variables to subsets of $\mathcal{D}$ and row variables to quasi-constant functions from labels to subsets of $\mathcal{D}_\mathsf{f}$. We note $\mathcal{H}$ the set of assignments.

Definition A.13 (Interpretation predicate). Let η be an assignment. We define two binary predicates $(\partial : t)^\eta$ and $(\delta : f)^\eta_\ell$, where $\partial \in \mathcal{D}_\Omega$, $t \in \mathcal{T}$, $\delta \in \mathcal{D}_\mathsf{f}$ and $f \in \mathcal{F}$, by structural induction on the pair (∂, t) and (δ, f) ordered lexicographically. The predicate is defined as follows:

$$
\begin{array}{llcll}
\boxed{\textbf{Types:}} & (d : \mathbb{1})^\eta & = & \mathsf{true} & \\
& (d : \alpha)^\eta & = & d \in \eta(\alpha) & \\
& (c^L : b)^\eta & = & c \in \mathbb{B}(b) & \\
& ((d_1, d_2)^L : t_1 \times t_2)^\eta & = & (d_1 : t_1)^\eta \text{ and } (d_2 : t_2)^\eta & \\
& (\{(d_1, \partial_1), ..., (d_n, \partial_n)\}^L : t_1 \to t_2)^\eta & = & \forall i \in [1..n].\ \text{if } (d_i : t_1)^\eta \text{ then } (\partial_i : t_2)^\eta & \\
& (\{\!\!\{\ell_1 = \delta_1, \ldots, \ell_n = \delta_n, \_ = \delta\}\!\!\}^L : r)^\eta & = & \forall i \in [1..n].\ (\delta_i : r(\ell_i))^\eta_{\ell_i} & \\
& & & \quad \text{and } \ \forall \ell \in \mathcal{L} \setminus \{\ell_1, ..., \ell_n\}.\ (\delta : r(\ell))^\eta_\ell & \\
& (d : t_1 \wedge t_2)^\eta & = & (d : t_1)^\eta \text{ and } (d : t_2)^\eta & \\
& (d : t_1 \vee t_2)^\eta & = & (d : t_1)^\eta \text{ or } (d : t_2)^\eta & \\
& (d : \neg t)^\eta & = & \text{not } (d : t)^\eta & \\
& (\partial : t)^\eta & = & \mathsf{false} & \text{otherwise} \\
\boxed{\textbf{Fields:}} & (\delta : \rho)^\eta_\ell & = & \delta \in \eta(\rho)(\ell) & \\
& (d^F : t)^\eta_\ell = (d^F : t?)^\eta_\ell & = & (d : t)^\eta & \\
& (\Omega^F : t?)^\eta_\ell & = & \mathsf{true} & \\
& (\delta : f_1 \wedge f_2)^\eta_\ell & = & (\delta : f_1)^\eta_\ell \text{ and } (\delta : f_2)^\eta_\ell & \\
& (\delta : f_1 \vee f_2)^\eta_\ell & = & (\delta : f_1)^\eta_\ell \text{ or } (\delta : f_2)^\eta_\ell & \\
& (\delta : \neg f)^\eta_\ell & = & \text{not } (\delta : f)^\eta_\ell & \\
& (\delta : f)^\eta_\ell & = & \mathsf{false} & \text{otherwise}
\end{array}
$$

where $\mathbb{B}$ denotes the function that assigns to each base type the set of constants of that type.

Definition A.14. We define the *set-theoretic interpretation* $\llbracket t \rrbracket^\eta$ of a type t, and the *set-theoretic interpretation* $\llbracket f \rrbracket^\eta_\ell$ of a field type f, as follows:

$$\llbracket t \rrbracket^\eta = \{d \in \mathcal{D} \mid (d : t)^\eta\}$$
$$\llbracket f \rrbracket^\eta_\ell = \{\delta \in \mathcal{D}_\mathsf{f} \mid (\delta : f)^\eta_\ell\}$$

Definition A.15 (Identity assignment). We define the identity assignment $\eta_\circ$ as follows:

$$\forall \alpha \in \mathcal{V}_{\mathsf{Ty}}.\ \eta_\circ(\alpha) = \{v^L \in \mathcal{D} \mid \alpha \in L\}$$
$$\forall \rho \in \mathcal{V}_{\mathsf{Row}}.\ \forall \ell \in \mathcal{L}.\ \eta_\circ(\rho)(\ell) = \{\partial^F \in \mathcal{D}_\mathsf{f} \mid \rho \in F\}$$

Proposition A.16. *For any f, ℓ and ℓ', we have $\llbracket f \rrbracket^{\eta_\circ}_\ell = \llbracket f \rrbracket^{\eta_\circ}_{\ell'}$.*

Proof. Trivial, as the definition of $\eta_\circ(\rho)(\ell)$ does not depend on ℓ. □

Definition A.17 (Set-theoretic interpretation of types). We define the *set-theoretic interpretation* $\llbracket . \rrbracket : \mathcal{T} \to \mathcal{P}(\mathcal{D})$ as $\llbracket t \rrbracket = \llbracket t \rrbracket^{\eta_\circ}$, and $\llbracket . \rrbracket : \mathcal{F} \to \mathcal{P}(\mathcal{D}_\mathsf{f})$ as $\llbracket f \rrbracket = \llbracket f \rrbracket^{\eta_\circ}_\ell$, where ℓ can be any label (according to Proposition 3.15, the interpretation is the same whichever label ℓ we choose).

Definition A.18 (Interpretation of type substitutions). The *set-theoretic interpretation* $\llbracket \phi \rrbracket$ of a type substitution ϕ is the assignment η defined as follows:

$$\forall \alpha \in \mathcal{V}_{\mathsf{Ty}}.\ \eta(\alpha) = \llbracket \phi(\alpha) \rrbracket$$
$$\forall \rho \in \mathcal{V}_{\mathsf{Row}}.\ \forall \ell \in \mathcal{L}.\ \eta(\rho)(\ell) = \llbracket \phi(\rho)(\ell) \rrbracket$$

Definition A.19 (Subtyping relation). We define the *subtyping* relation $\leq$ and the *subtyping equivalence* relation $\simeq$ as follows:

$$t_1 \leq t_2 \overset{\text{def}}{\iff} \llbracket t_1 \rrbracket \subseteq \llbracket t_2 \rrbracket \quad \text{and} \quad t_1 \simeq t_2 \overset{\text{def}}{\iff} \llbracket t_1 \rrbracket = \llbracket t_2 \rrbracket$$

Definition A.20 (Type substitutions equivalence). We define the equivalence relation $\simeq$ over type substitutions as follows:

$$\phi_1 \simeq \phi_2 \overset{\text{def}}{\iff} \llbracket \phi_1 \rrbracket = \llbracket \phi_2 \rrbracket$$

Lemma A.21. *If $(d : t\phi)^{\eta_\circ}$, then $(d : t)^{\llbracket \phi \rrbracket}$. If $(\delta : (f\phi)_\ell)^{\eta_\circ}_\ell$, then $(\delta : f)^{\llbracket \phi \rrbracket}_\ell$.*

Proof. Note: the two statements of this lemma and of Lemma A.22 are proved by simultaneous (mutual) induction on the lexicographic pair (d, t) or (δ, f). The arrow and negation cases of each lemma invoke the other lemma at a strictly smaller first component, so the mutual recursion is well-founded. We have several cases:

$t = \mathbb{1}$ $\mathbb{1}\phi = \mathbb{1}$ and $(d : \mathbb{1})^\eta =$ true for any η. Trivial.

$t = \mathbb{0}$ The premise $(d : \mathbb{0}\phi)^{\eta_\circ} = (d : \mathbb{0})^{\eta_\circ} =$ false is vacuously false. Trivial.

$t = b$ **(base type)** $b\phi = b$ and $(c^L : b)^\eta$ does not depend on η. Trivial.

$t = \alpha$ **(type variable)** $\alpha\phi = \phi(\alpha)$ and $(d : \phi(\alpha))^{\eta_\circ}$ means $d \in \llbracket \phi(\alpha) \rrbracket$. By definition of $\llbracket \phi \rrbracket$, we have $(\llbracket \phi \rrbracket)(\alpha) = \llbracket \phi(\alpha) \rrbracket$, so $d \in (\llbracket \phi \rrbracket)(\alpha)$, i.e., $(d : \alpha)^{\llbracket \phi \rrbracket}$.

$t = t_1 \vee t_2$ $(t_1 \vee t_2)\phi = t_1\phi \vee t_2\phi$. The premise gives $(d : t_1\phi)^{\eta_\circ}$ or $(d : t_2\phi)^{\eta_\circ}$. By induction, $(d : t_1)^{\llbracket \phi \rrbracket}$ or $(d : t_2)^{\llbracket \phi \rrbracket}$, hence $(d : t_1 \vee t_2)^{\llbracket \phi \rrbracket}$.

$t = t_1 \wedge t_2$ Analogous: both $(d : t_1\phi)^{\eta_\circ}$ and $(d : t_2\phi)^{\eta_\circ}$ yield by induction $(d : t_1 \wedge t_2)^{\llbracket \phi \rrbracket}$.

$t = \neg t'$ The premise gives not $(d : t'\phi)^{\eta_\circ}$. Suppose for contradiction that $(d : t')^{\llbracket \phi \rrbracket}$. By Lemma A.22 (mutual induction, strictly smaller d not needed here since the type is smaller), we would get $(d : t'\phi)^{\eta_\circ}$, a contradiction. Hence not $(d : t')^{\llbracket \phi \rrbracket}$, i.e., $(d : \neg t')^{\llbracket \phi \rrbracket}$.

$t = r$ **(record atom)** $d = F^L$ is a quasi-constant function. We have $(F^L : r\phi)^{\eta_\circ}$, and we want to show that $(F^L : r)^{[\![\phi]\!]}$. By definition, the quasi-constant function F is such that $\forall \ell \in \mathcal{L}.\ (F(\ell) : (r\phi)(\ell))_\ell^{\eta_\circ}$, and we need to show that $\forall \ell \in \mathcal{L}.\ (F(\ell) : r(\ell))_\ell^{[\![\phi]\!]}$. Let $\ell \in \mathcal{L}$. By Proposition A.7, we have $(r\phi)(\ell) = (r(\ell)\phi)_\ell$, and thus we have $(F(\ell) : (r(\ell)\phi)_\ell)_\ell^{\eta_\circ}$. By the induction hypothesis, we get $(F(\ell) : r(\ell))_\ell^{[\![\phi]\!]}$.

$t = t_1 \times t_2$ Let $d = (d_1, d_2)^L$. The premise gives $(d_1 : t_1\phi)^{\eta_\circ}$ and $(d_2 : t_2\phi)^{\eta_\circ}$. By induction: $(d_1 : t_1)^{[\![\phi]\!]}$ and $(d_2 : t_2)^{[\![\phi]\!]}$, hence $((d_1, d_2)^L : t_1 \times t_2)^{[\![\phi]\!]}$.

$t = t_1 \to t_2$ Let $d = \{(d_i, \partial_i)\}_{i\in I}^L$. The premise gives: $\forall i.\ (d_i : t_1\phi)^{\eta_\circ} \Rightarrow (\partial_i : t_2\phi)^{\eta_\circ}$. We want: $\forall i.\ (d_i : t_1)^{[\![\phi]\!]} \Rightarrow (\partial_i : t_2)^{[\![\phi]\!]}$. Fix i and assume $(d_i : t_1)^{[\![\phi]\!]}$. By Lemma A.22 applied to (d_i, t_1), we get $(d_i : t_1\phi)^{\eta_\circ}$, and thus by hypothesis $(\partial_i : t_2\phi)^{\eta_\circ}$. By induction on (∂_i, t_2), we get $(\partial_i : t_2)^{[\![\phi]\!]}$.

$f = \rho$ We have $(\delta : (\rho\phi)_\ell)_\ell^{\eta_\circ}$, and we want to show that $(\delta : \rho)_\ell^{[\![\phi]\!]}$. By definition of type substitution on row variables, we have $(\rho\phi)_\ell = \phi(\rho)(\ell)$, and thus $(\delta : \phi(\rho)(\ell))_\ell^{\eta_\circ}$. By definition of $[\![.]\!]$, this can be rewritten $\delta \in [\![\phi(\rho)(\ell)]\!]_\ell^{\eta_\circ}$, that is, $\delta \in [\![\phi(\rho)(\ell)]\!]$. Consequently, by definition of the interpretation of type substitutions, we have $\delta \in [\![\phi]\!](\rho)(\ell)$, which implies, by definition, $(\delta : \rho)_\ell^{[\![\phi]\!]}$.

$f = \bar{t}$ For option types: if $f = t$ then $(d^F : t\phi)_\ell^{\eta_\circ} = (d : t\phi)^{\eta_\circ}$ and we apply the type case above; if $f = t?$ then $(\Omega^F : t\phi?)_\ell^{\eta_\circ} =$ true and $(\Omega^F : t?)_\ell^{[\![\phi]\!]} =$ true (trivial), while $(d^F : t\phi?)_\ell^{\eta_\circ} = (d : t\phi)^{\eta_\circ}$ reduces to the type case.

Other cases For field types, the boolean connective cases ($\neg f$, $f_1 \vee f_2$, $f_1 \wedge f_2$) are analogous to those for types.

□

Lemma A.22. *If $(d : t)^{[\![\phi]\!]}$, then $(d : t\phi)^{\eta_\circ}$. If $(\delta : f)_\ell^{[\![\phi]\!]}$, then $(\delta : (f\phi)_\ell)_\ell^{\eta_\circ}$.*

Proof. As noted for Lemma A.21, both lemmas are proved by simultaneous induction on the lexicographic pair (d, t) or (δ, f). We have several cases:

$t = \mathbb{1}$ $(d : \mathbb{1})^\eta =$ true always. Trivial.

$t = \mathbb{0}$ $(d : \mathbb{0})^{[\![\phi]\!]} =$ false; premise is vacuously false. Trivial.

$t = b$ $b\phi = b$ and the predicate is independent of η. Trivial.

$t = \alpha$ $(d : \alpha)^{[\![\phi]\!]}$ means $d \in ([\![\phi]\!])(\alpha) = [\![\phi(\alpha)]\!]$, i.e., $(d : \phi(\alpha))^{\eta_\circ} = (d : \alpha\phi)^{\eta_\circ}$.

$t = t_1 \vee t_2$ The premise gives $(d : t_1)^{[\![\phi]\!]}$ or $(d : t_2)^{[\![\phi]\!]}$. By induction, $(d : t_1\phi)^{\eta_\circ}$ or $(d : t_2\phi)^{\eta_\circ}$, hence $(d : (t_1 \vee t_2)\phi)^{\eta_\circ}$.

$t = t_1 \wedge t_2$ Analogous.

$t = \neg t'$ The premise gives not $(d : t')^{[\![\phi]\!]}$. Suppose for contradiction $(d : t'\phi)^{\eta_\circ}$. By Lemma A.21 (mutual induction), we get $(d : t')^{[\![\phi]\!]}$, a contradiction.

$t = r$ **(record atom)** $d = F^L$ is a quasi-constant function. We have $(F^L : r)^{[\![\phi]\!]}$, and we want to show that $(F^L : r\phi)^{\eta_\circ}$. By definition, the quasi-constant function F is such that $\forall \ell \in \mathcal{L}.\ (F(\ell) : r(\ell))_\ell^{[\![\phi]\!]}$, and we need to show that $\forall \ell \in \mathcal{L}.\ (F(\ell) : (r\phi)(\ell))_\ell^{\eta_\circ}$. By the induction hypothesis, we get $(F(\ell) : (r(\ell)\phi)_\ell)_\ell^{\eta_\circ}$. By Proposition A.7, we have $(r\phi)(\ell) = (r(\ell)\phi)_\ell$, and thus we have $(F(\ell) : (r\phi)(\ell))_\ell^{\eta_\circ}$.

$t = t_1 \times t_2$ Let $d = (d_1, d_2)^L$. From $(d_1 : t_1)^{[\![\phi]\!]}$ and $(d_2 : t_2)^{[\![\phi]\!]}$, by induction $(d_1 : t_1\phi)^{\eta_\circ}$ and $(d_2 : t_2\phi)^{\eta_\circ}$, hence $((d_1, d_2)^L : t_1\phi \times t_2\phi)^{\eta_\circ}$.

$t = t_1 \to t_2$ Let $d = \{(d_i, \partial_i)\}_{i\in I}^L$. The premise gives: $\forall i.\ (d_i : t_1)^{[\![\phi]\!]} \Rightarrow (\partial_i : t_2)^{[\![\phi]\!]}$. We want: $\forall i.\ (d_i : t_1\phi)^{\eta_\circ} \Rightarrow (\partial_i : t_2\phi)^{\eta_\circ}$. Fix i and assume $(d_i : t_1\phi)^{\eta_\circ}$. By Lemma A.21 applied to

(d_i, t_1), we get $(d_i : t_1)^{[\![\phi]\!]}$, then by hypothesis $(\partial_i : t_2)^{[\![\phi]\!]}$, then by induction on (∂_i, t_2): $(\partial_i : t_2\phi)^{\eta_\circ}$.

$f = \rho$ We have $(\delta : \rho)_\ell^{[\![\phi]\!]}$ and we want to show that $(\delta : (\rho\phi)_\ell)_\ell^{\eta_\circ}$. By definition, we have $\delta \in [\![\phi]\!](\rho)(\ell)$. Consequently, by definition of the interpretation of type substitutions, we get $\delta \in [\![\phi(\rho)(\ell)]\!]$, that is $\delta \in [\![\phi(\rho)(\ell)]\!]_\ell^{\eta_\circ}$. This can be rewritten $(\delta : \phi(\rho)(\ell))_\ell^{\eta_\circ}$ by definition of $[\![.]\!]$. By definition of type substitution on row variables, we have $(\rho\phi)_\ell = \phi(\rho)(\ell)$, and thus $(\delta : (\rho\phi)_\ell)_\ell^{\eta_\circ}$.

Other cases The other field type cases are analogous to those of Lemma A.21.

□

Proposition A.23. *For any type t and type substitution ϕ, $[\![t\phi]\!] = [\![t]\!]^{[\![\phi]\!]}$.*

Proof. Direct consequence of Lemmas A.21 and A.22. □

Lemma A.24. *Let P and N two finite sets of types. Let $d \in \mathcal{D}$ and $\eta \in \mathcal{H}$. If $\forall t \in P.\ (d : t)^\eta$ and $\forall t \in N.$ not $(d : t)^\eta$, then there exists $d' \in \mathcal{D}$ such that $\forall t \in P.\ (d' : t)^{\eta_\circ}$ and $\forall t \in N.$ not $(d' : t)^{\eta_\circ}$.*

Let P and N two finite sets of types of field types. Let $\delta \in \mathcal{D}_\mathrm{f}$, $\eta \in \mathcal{H}$, and $\ell \in \mathcal{L}$. If $\forall f \in P.\ (\delta : f)_\ell^\eta$ and $\forall f \in N.$ not $(\delta : f)_\ell^\eta$, then there exists $\delta' \in \mathcal{D}_\mathrm{f}$ such that $\forall f \in P.\ (\delta' : f)_\ell^{\eta_\circ}$ and $\forall f \in N.$ not $(\delta' : f)_\ell^{\eta_\circ}$.

Proof. We proceed by induction on the triple $(d, \sum_{t\in P} s(t)+\sum_{t\in N} s(t), |P|+|N|)$—or $(\delta, \sum_{f\in P} s(f)+\sum_{f\in N} s(f), |P|+|N|)$—ordered lexicographically, where s counts the number of top-level set connectives ($\wedge$, $\vee$, and $\neg$) in a type of field type (not counting those appearing inside a type constructor).

If at least one $t \in P$ is a negation $\neg t'$ We use the induction hypothesis on d and on the sets of types $P \setminus \{t\}$ and $N \cup \{t'\}$.

If at least one $t \in N$ is a negation $\neg t'$ We use the induction hypothesis on d and on the sets of types $P \cup \{t'\}$ and $N \setminus \{t\}$.

If at least one $t \in P$ is a union $t_1 \vee t_2$ From $(d : t)^\eta$, we deduce that there exists $i \in \{1, 2\}$ such that $(d : t_i)^\eta$ and use the induction hypothesis on d and on the sets of types $P \setminus \{t\} \cup \{t_i\}$ and N.

If at least one $t \in N$ is a union $t_1 \vee t_2$ We use the induction hypothesis on d and on the sets of types P and $N \setminus \{t\} \cup \{t_1\} \cup \{t_2\}$.

If at least one $t \in P$ is an intersection $t_1 \wedge t_2$ We use the induction hypothesis on d and on the sets of types $P \setminus \{t\} \cup \{t_1\} \cup \{t_2\}$ and N.

If at least one $t \in N$ is an intersection $t_1 \wedge t_2$ From not $(d : t)^\eta$, we deduce that there exists $i \in \{1, 2\}$ such that not $(d : t_i)^\eta$ and use the induction hypothesis on d and on the sets of types P and $N \setminus \{t\} \cup \{t_i\}$.

If every $t \in P \cup N$ is a literal (i.e. a type variable or type constructor)

If $\alpha \in P$ and $\alpha \in N$ for some type variable α Impossible since $(d : \alpha)^\eta$ and not $(d : \alpha)^\eta$ cannot both hold.

Otherwise We have $d = v^L$. Set $L' = \{\alpha \in \mathcal{V}_\mathrm{Ty} \mid \alpha \in P\}$. Under $\eta_\circ$, the predicate $(v'^{L'} : \alpha)^{\eta_\circ}$ holds if and only if $\alpha \in L'$, independently of v': hence for every $\alpha \in P$ it holds, and for every $\alpha \in N$ it fails (since $\{\alpha \in \mathcal{V}_\mathrm{Ty} \mid \alpha \in P\} \cap \{\alpha \in \mathcal{V}_\mathrm{Ty} \mid \alpha \in N\} = \varnothing$ by assumption). Let us now build v' such that $\forall t \in P.\ (v'^{L'} : t)^{\eta_\circ}$ and $\forall t \in N.$ not $(v'^{L'} : t)^{\eta_\circ}$.

If d is a constant c^L Trivial.

If d is a relation $\{(d_1, \partial_1), \cdot, (d_n, \partial_n)\}^L$ Let $i \in 1 \mathinner{..} n$. Let us collect the sets of types P_i and N_i that characterize d_i: for each $(s \to t) \in P \cup N$, we have either $(d_i : s)^\eta$ or not $(d_i : s)^\eta$; in the first case, we add s in P_i, otherwise we add s in N_i. Likewise, if $\partial_i \neq \Omega$, we collect the sets of types P'_i and N'_i that characterize

∂_i: for each $(s \to t) \in P \cup N$, we have either $(\partial_i : t)^\eta$ or not $(\partial_i : t)^\eta$; in the first case, we add t in P'_i, otherwise we add t in N'_i. By using the induction hypothesis on d_i, P_i and N_i, we get some d'_i such that $\forall s \in P_i.\ (d'_i : s)^{\eta_\circ}$ and $\forall s \in N_i.$ not $(d'_i : s)^{\eta_\circ}$. Likewise, by using the induction hypothesis on ∂_i, P'_i and N'_i, we get some ∂'_i such that $\forall t \in P'_i.\ (\partial'_i : t)^{\eta_\circ}$ and $\forall t \in N'_i.$ not $(\partial'_i : t)^{\eta_\circ}$. Let $v' = \{(d'_1, \partial'_1), \cdot, (d'_n, \partial'_n)\}$. We can easily check that $\forall (s \to t) \in P.\ (v'^{L'} : s \to t)^{\eta_\circ}$ and $\forall (s \to t) \in N.$ not $(v'^{L'} : s \to t)^{\eta_\circ}$, which concludes.

If d is a pair $(d_1, d_2)^L$ Similar to above.

If d is a quasi-constant function $\{\!\![\ell 1 = \delta_1, \dots, \ell_n = \delta_n, \_ = \delta]\!\!\}$ Similar to above.

We detail the base case for field types (the boolean connective cases are identical to the type case). When every $f \in P \cup N$ is a literal (i.e. a row variable or an option type):

If $\rho \in P$ and $\rho \in N$ for some row variable ρ Impossible since $(\delta : \rho)^\eta_\ell$ and not $(\delta : \rho)^\eta_\ell$ cannot both hold.

Otherwise We have $\delta = \partial^F$. Set $F' = \{\rho \in \mathcal{V}_{\text{Row}} \mid \rho \in P\}$. Under $\eta_\circ$, the predicate $(\partial'^{F'} : \rho)^{\eta_\circ}_\ell$ holds if and only if $\rho \in F'$, independently of ∂': hence for every $\rho \in P$ it holds, and for every $\rho \in N$ it fails (since $\{\rho \in \mathcal{V}_{\text{Row}} \mid \rho \in P\} \cap \{\rho \in \mathcal{V}_{\text{Row}} \mid \rho \in N\} = \varnothing$ by assumption). For ∂', note that $(\partial'^{F'} : \bar{t})^{\eta_\circ}_\ell$ depends only on ∂':

- If P contains a non-optional type (t), or N contains an optional type $(t'?)$: the hypothesis forces $\partial \in \mathcal{D}$ (otherwise $(\Omega^F : \bar{t})^\eta_\ell$ = false for a non-optional $\bar{t} \in P$, or $(\Omega^F : \bar{t})^\eta_\ell$ = true for an optional $\bar{t} \in N$, both contradicting the premise). Moreover, $(\partial : \text{get}(\bar{t}))^\eta$ for all $\bar{t} \in P$, and not $(\partial : \text{get}(\bar{t}))^\eta$ for all $\bar{t} \in N$. By the induction hypothesis on $(\partial, \{\text{get}(\bar{t}) \mid \bar{t} \in P\}, \{\text{get}(\bar{t}) \mid \bar{t} \in N\})$, there exists $\partial' \in \mathcal{D}$ such that $(\partial' : \text{get}(\bar{t}))^{\eta_\circ}$ for all $\bar{t} \in P$ and not $(\partial' : \text{get}(\bar{t}))^{\eta_\circ}$ for all $\bar{t} \in N$.
- Otherwise (all option types in P are optional and all in N are non-optional): take $\partial' = \Omega$; then $(\Omega^{F'} : t?)^{\eta_\circ}_\ell$ = true for all $(t?) \in P$, and $(\Omega^{F'} : t)^{\eta_\circ}_\ell$ = false for all $t \in N$.

In both cases, $\delta' = \partial'^{F'}$ satisfies all constraints in P and violates all constraints in N under $\eta_\circ$.

□

Proposition A.25. *For any type t, $[\![t]\!] = \varnothing \Rightarrow \forall \eta.\ [\![t]\!]^\eta = \varnothing$.*

Proof. Direct consequence of Lemma A.24. □

Proposition A.26 (Preservation of subtyping by type substitutions).

$$\forall t_1, t_2, \phi.\ t_1 \leq t_2 \Rightarrow t_1\phi \leq t_2\phi$$

Proof. Direct consequence of Propositions A.23 and A.25. □

A.3 Subtyping algorithm

This section includes the proofs relative to Section 4.2.

Proposition A.27 (Field type subtyping).

$$\bigwedge_{i\in I} \rho_i \ \wedge \ \bigwedge_{j\in J} \neg\rho'_j \ \wedge \ \bigwedge_{k\in K} \bar{t}_k \ \wedge \ \bigwedge_{l\in L} \neg\bar{t}'_l \simeq \mathbb{0} \ \textit{(where } \forall i.\ \forall j.\ \rho_i \neq \rho'_j\textit{)} \iff \bigwedge_{k\in K} \bar{t}_k \leq \bigvee_{l\in L} \bar{t}'_l$$

Proof. This property relies on the fact that our interpretation of types is convex, as defined in [8]. The $(\Leftarrow)$ direction is trivial (recall that $\bigwedge_k \bar{t}_k \leq \bigvee_l \bar{t}'_l$ can be rewritten $(\bigwedge_k \bar{t}_k) \wedge (\bigwedge_l \neg\bar{t}'_l) \simeq \mathbb{0}$). Let us prove the $(\Rightarrow)$ direction. Let $\ell \in \mathcal{L}$ and $\delta \in \mathcal{D}_\mathsf{f}$ such that $(\delta : \bigwedge_k \bar{t}_k \ \wedge \ \bigwedge_l \neg\bar{t}'_l)^{\eta_\circ}_\ell$. We want to show that there exists $\delta' \in \mathcal{D}_\mathsf{f}$ such that $(\delta' : \bigwedge_i \rho_i \ \wedge \ \bigwedge_j \neg\rho'_j \ \wedge \ \bigwedge_k \bar{t}_k \ \wedge \ \bigwedge_l \neg\bar{t}'_l)^{\eta_\circ}_\ell$.

We have $\delta = \partial^F$. As $(\partial^F : \bigwedge_k \bar{t}_k \ \wedge\ \bigwedge_l \neg\bar{t}'_l)^{\eta_\circ}_\ell$ does not depend on F, we can deduce that for any $F' \subseteq \mathcal{V}_{\text{Row}}$, $(\partial^{F'} : \bigwedge_k \bar{t}_k \ \wedge\ \bigwedge_l \neg\bar{t}'_l)^{\eta_\circ}_\ell$ holds. Conversely, the predicate $(\partial^{F'} : \bigwedge_i \rho_i \ \wedge\ \bigwedge_j \neg\rho'_j)^{\eta_\circ}_\ell$ does not depend on ∂. By choosing $F' = \{\rho_i\}_{i\in I}$ and posing $\delta' = \partial^{F'}$, we thus have $(\delta' : \bigwedge_i \rho_i \wedge \bigwedge_j \neg\rho'_j)^{\eta_\circ}_\ell$ and $(\delta' : \bigwedge_k \bar{t}_k \ \wedge\ \bigwedge_l \neg\bar{t}'_l)^{\eta_\circ}_\ell$. □

Proposition A.28 (Option type subtyping). *For any option types $\{\bar{t}_p\}_{p\in P}$ and $\{\bar{t}'_n\}_{n\in N}$, we have:*

$$\bigwedge_{p\in P} \bar{t}_p \leq \bigvee_{n\in N} \bar{t}'_n \iff (\forall p \in P.\ \mathrm{opt}(\bar{t}_p) \Rightarrow \exists n \in N.\mathrm{opt}(\bar{t}'_n)) \textit{ and } \bigwedge_{p\in P} \mathrm{get}(\bar{t}_p) \leq \bigvee_{n\in N} \mathrm{get}(\bar{t}'_n)$$

where $\forall t.\ \mathrm{opt}(t?) = \mathrm{true}, \mathrm{opt}(t) = \mathrm{false}$, *and* $\mathrm{get}(t) = \mathrm{get}(t?) = t$.

Proof. We can rewrite this proposition as follows:

$$(\bigwedge_{p\in P} \bar{t}_p) \wedge (\bigwedge_{n\in N} \neg\bar{t}'_n) \simeq \mathbb{0} \iff (\forall p \in P.\ \mathrm{opt}(\bar{t}_p) \Rightarrow \exists n \in N.\mathrm{opt}(\bar{t}'_n)) \text{ and}$$
$$(\bigwedge_{p\in P} \mathrm{get}(\bar{t}_p)) \wedge (\bigwedge_{n\in N} \neg\mathrm{get}(\bar{t}'_n)) \simeq \mathbb{0}$$

Both directions are then trivial. □

Proposition A.29 (Record containment). *Let $(X_p)_{p\in P}$ and $(X_n)_{n\in N}$ be two families of elements of $\mathcal{L} \twoheadrightarrow \mathcal{P}(\mathcal{D}_\mathrm{f})$. Let $L = \bigcup_{i\in P\cup N} \mathrm{dom}(X_i)$. Then, $(\bigcap_{p\in P} \boxtimes_{\ell\in\mathcal{L}} X_p(\ell)) \subseteq (\bigcup_{n\in N} \boxtimes_{\ell\in\mathcal{L}} X_n(\ell))$ if and only if either $\bigcap_{p\in P} \mathrm{def}(X_p) = \varnothing$, or for every map $\iota : N \to L \cup \{\_\}$*

$$\left(\exists \ell \in L. \left(\bigcap_{p\in P} X_p(\ell) \subseteq \bigcup_{n\in N|\iota(n)=\ell} X_n(\ell)\right)\right) \textit{ or } \left(\exists n_\circ \in N.(\iota(n_\circ) = \_) \textit{ and } \left(\bigcap_{p\in P} \mathrm{def}(X_p) \leq \mathrm{def}(X_{n_\circ})\right)\right)$$

Proof. A full proof is available in the PhD manuscript of Alain Frisch [10, Lemma 9.1] (originally in french, translated in english by Guillaume Duboc). □

Proposition A.30 (Record subtyping). *For any record atoms $\{r_p\}_{p\in P}$ and $\{r_n\}_{n\in N}$, we have:*

$$\bigwedge_{p\in P} r_p \leq \bigvee_{n\in N} r_n \iff \bigwedge_{p\in P} \mathrm{tl}(r_p) \leq \mathbb{0} \textit{ or } \forall \iota : N \to L \cup \{\_\}.$$
$$\left(\exists \ell \in L.\ (\bigwedge_{p\in P} r_p(\ell) \leq \bigvee_{n\in N|\iota(n)=\ell} r_n(\ell))\right) \textit{ or } \left(\exists n_\circ \in N.(\iota(n_\circ) = \_) \textit{ and } \left(\bigwedge_{p\in P} \mathrm{tl}(r_p) \leq \mathrm{tl}(r_{n_\circ})\right)\right)$$

where $L = \bigcup_{i\in P\cup N} \mathrm{labels}(r_i)$.

Proof. Immediate consequence of Proposition A.29. □

A.4 Tallying algorithm

This section includes the proofs relative to Section 4.3.

The full description of the original tallying algorithm—as well as the proofs of correctness, completeness, and termination—can be found in [5, Appendix C] (appendices are available in the supplementary material).

Definition A.31 (Quasi-constant substitutions for a set of labels). Let L be a finite set of labels. The set of all quasi-constant substitutions for L, noted $\mathcal{S}_L$, is the following set:

$$\mathcal{S}_L = \{\phi \in \mathcal{S} \mid \mathrm{labels}(\phi) \subseteq L\}$$

Theorem A.32 (Soundness).

$$\forall\phi \in \mathsf{tally}(S,\Delta,\varnothing).\quad \phi\in\mathcal{S}_\varnothing \quad \textit{and} \quad \mathsf{dom}(\phi)\cap\Delta=\varnothing \quad \textit{and} \quad \forall(s,t)\in S.\ s\phi\le t\phi$$

Proof. Straightforward extension of the proof of soundness of the original tallying algorithm, available in [5, Lemma C.10]. The case for row variables is similar to the case [NTLV] for type variables. The case for records is similar to the case [NProd] for products and directly follows from Proposition A.30. □

Theorem A.33 (Completeness).

$$\begin{aligned}\forall\phi\in\mathcal{S}_\varnothing.\quad &(\mathsf{dom}(\phi)\cap\Delta=\varnothing \quad \textit{and} \quad \forall(s,t)\in S.\ s\phi\le t\phi)\\ &\Rightarrow\ (\exists\phi'\in\mathsf{tally}(S,\Delta,\varnothing).\ \exists\phi''\in\mathcal{S}_\varnothing.\ \phi\simeq\phi''\circ\phi')\end{aligned}$$

Proof. Straightforward extension of the proof of completeness of the original tallying algorithm, available in [5, Lemma C.12]. The case for row variables is similar to the case [NTLV] for type variables. The case for records is similar to the case [NProd] for products and directly follows from Proposition A.30. The proof of termination is available in [5, Lemma C.14]. □

Definition A.34 (Quasi-constant-row tallying). Let L be a finite set of labels and Δ be the set of monomorphic type variables and row variables. Let S be a set of constraints and V the set of row variables appearing in S and that are not in Δ. We define $\mathsf{tally}(S,\Delta,L)$ as follows:

$$\begin{aligned}\mathsf{tally}(S,\Delta,L) &= \{(\phi''\circ\phi'\circ\phi)|_{V\cup\mathcal{V}_{\mathsf{Ty}}} \mid \phi'\in\mathsf{tally}(\{(s\phi,t\phi)\mid(s,t)\in S\},\Delta,\varnothing)\}\\ \phi &= \{\rho\rightsquigarrow\langle\ell_1:\rho_1,\ \ldots,\ \ell_n:\rho_n\mid\rho\rangle\}_{\rho\in V} \text{ for } L=\{\ell_1,\ldots,\ell_n\} \text{ with all } \{\rho_i\}_{\rho\in V,i\in1..n} \text{ fresh}\\ \phi'' &= \{\rho_i\rightsquigarrow\langle\ \mid\rho\rangle\}_{\rho\in V,i\in1..n}\end{aligned}$$

Theorem A.35 (Soundness).

$$\forall\phi\in\mathsf{tally}(S,\Delta,L).\quad \phi\in\mathcal{S}_L \quad \textit{and} \quad \mathsf{dom}(\phi)\cap\Delta=\varnothing \quad \textit{and} \quad \forall(s,t)\in S.\ s\phi\le t\phi$$

Proof. Let L be a finite set of labels and Δ be a set of type variables and row variables. Let S be a set of constraints and V the set of row variables appearing in S and that are not in Δ. Let $\phi_\circ\in\mathsf{tally}(S,\Delta,L)$. We want to show that $\phi_\circ\in\mathcal{S}_L$, $\mathsf{dom}(\phi_\circ)\cap\Delta=\varnothing$, and $\forall(s,t)\in S.\ s\phi_\circ\le t\phi_\circ$. The first two properties are trivial ; we focus on proving $\forall(s,t)\in S.\ s\phi_\circ\le t\phi_\circ$.

Let $(s,t)\in S$. By reusing the notations of Definition A.34, we have $s\phi_\circ\simeq s((\phi''\circ\phi'\circ\phi)|_{V\cup\mathcal{V}_{\mathsf{Ty}}})\simeq s(\phi''\circ\phi'\circ\phi)$ (since $\mathsf{dom}(\phi''\circ\phi'\circ\phi)\setminus(V\cup\mathcal{V}_{\mathsf{Ty}})$ only contains the $\{\rho_i\}_{\rho\in V,i\in1..n}$ which all are fresh row variables, and thus do not appear in s), where $\phi'\in\mathsf{tally}(\{(s\phi,t\phi)\mid(s,t)\in S\},\Delta,\varnothing)$. Consequently, we have $s\phi_\circ\simeq((s\phi)\phi')\phi''$. Similarly, we have $t\phi_\circ\simeq((t\phi)\phi')\phi''$. We know by applying Theorem A.32 on $\phi'\in\mathsf{tally}(\{(s\phi,t\phi)\mid(s,t)\in S\},\Delta,\varnothing)$ that $(s\phi)\phi'\le(t\phi)\phi'$. By Proposition A.26, we thus have $((s\phi)\phi')\phi''\le((t\phi)\phi')\phi''$, that is, $s\phi_\circ\le t\phi_\circ$. □

Before proving the completeness of our quasi-constant-row tallying algorithm, we need an additional lemma (Lemma A.37) about the constant-row tallying algorithm: a row variable that only appears under a field ℓ in the initial constraints S cannot appear under a different field in the solutions returned by the tallying algorithm.

Definition A.36. Let r be a record atom (resp. R be a row). We say that the row variable ρ appears in r (resp. R) under the field ℓ if and only if ρ appears in $r(\ell)$ at top-level (i.e. not under a type constructor).

Lemma A.37 (Preservation of context). *Let S be a set of constraints and Δ be a set of type variables and row variables. Let $\phi\in\mathsf{tally}(S,\Delta,\varnothing)$. For any row variable ρ appearing in ϕ under a field ℓ, there exists a record atom r in S such that ρ appears in r under the field ℓ.*

Proof. It is straightforward to show that the normalization process preserves this property by induction on the normalization derivation: for any row variable ρ appearing in $C \in \texttt{normalize}(t)$ (resp. $C \in \texttt{normalize}(f)$) under a field ℓ, there exists a record atom r in t (resp. f) such that ρ appears in r under the field ℓ. The key point of this induction is that normalization only decomposes record constraints via Proposition A.30: when $r_1 \leq r_2$ is split into per-label constraints $r_1(\ell) \leq r_2(\ell)$, any row variable ρ that appears under field ℓ in the resulting constraints already appeared under field ℓ in r_1 or r_2. Boolean connectives (union/intersection/negation) distribute over record atoms without changing which row variable appears under which field. The property then lifts from normalization to the tallying algorithm, which only generates new constraints by calling the normalization process on an existing constraint. □

Theorem A.38 (Completeness).

$$\begin{aligned}\forall \phi \in \mathcal{S}_L.\quad &(\mathsf{dom}(\phi) \cap \Delta = \varnothing \quad \textit{and} \quad \forall (s,t) \in S.\ s\phi \leq t\phi)\\ &\Rightarrow\ (\exists \phi' \in \mathsf{tally}(S, \Delta, L).\ \exists \phi'' \in \mathcal{S}_L.\ \phi \simeq \phi'' \circ \phi')\end{aligned}$$

Proof. Let L be a finite set of labels and Δ be a set of type variables and row variables. Let S be a set of constraints and $V = (V' \cup \mathcal{V}_{\mathsf{Ty}}) \setminus \Delta$ where V' is the set of row variables appearing in S. Let $\phi_\bullet \in \mathcal{S}_L$ such that $\mathsf{dom}(\phi_\bullet) \cap \Delta = \varnothing$ and $\forall (s,t) \in S.\ s\phi_\bullet \leq t\phi_\bullet$.

We want to show that there exists $\phi_\circ \in \mathsf{tally}(S, \Delta, L)$ and $\phi'_\circ \in \mathcal{S}_L$ such that $\phi_\bullet \simeq \phi'_\circ \circ \phi_\circ$.

We can assume without loss of generality that $\mathsf{dom}(\phi_\bullet) \subseteq V$. We introduce the substitutions ϕ and ϕ'' from Definition A.34.

We start by observing that $(\phi'' \circ \phi)|_V$ is the identity substitution. Indeed, for $\rho \in V$: $\phi(\rho) = \langle \ell_1 : \rho_1,\ \ldots,\ \ell_n : \rho_n \mid \rho \rangle$. Applying ϕ'' (which maps each fresh ρ_i to $\langle \mid \rho \rangle$): each explicit field ℓ_i gives $(\rho_i\phi'')_{\ell_i} = \phi''(\rho_i)(\ell_i) = \langle \mid \rho \rangle(\ell_i) = \rho$, and the tail gives $(\rho\phi'')_{\mathsf{tl}} = \mathsf{tl}(\phi''(\rho)) = \rho$ (since ρ is not in the domain of ϕ''). Thus $(\phi'' \circ \phi)(\rho) = \langle \ell_1 : \rho,\ \ldots,\ \ell_n : \rho \mid \rho \rangle \simeq \langle \mid \rho \rangle$, which is the identity row for ρ. Consequently, by posing $\phi'_\bullet = \phi \circ \phi_\bullet \circ \phi''$, we get a substitution $\phi'_\bullet \in \mathcal{S}_L$ such that $(\phi'' \circ \phi'_\bullet \circ \phi)|_V \simeq (\phi'' \circ \phi \circ \phi_\bullet \circ \phi'' \circ \phi)|_V \simeq \phi_\bullet$ and $\mathsf{dom}(\phi'_\bullet) \subseteq V$.

We now build a type substitution $\phi''_\bullet \in \mathcal{S}_\varnothing$ such that $(\phi'' \circ \phi''_\bullet \circ \phi)|_V \simeq (\phi'' \circ \phi'_\bullet \circ \phi)|_V$. This substitution $\phi''_\bullet$ is defined by $\mathsf{dom}(\phi''_\bullet) \subseteq \mathsf{dom}(\phi'_\bullet)$ and

$$\begin{aligned}&\forall \alpha \in \mathcal{V}_{\mathsf{Ty}}.\ \phi''_\bullet(\alpha) = \phi'_\bullet(\alpha)\\ &\forall \rho \in V.\ \phi''_\bullet(\rho) = \langle \mid \mathsf{tl}(\phi'_\bullet(\rho)) \rangle\\ &\forall \rho \in V, i \in 1\,..\,n.\ \phi''_\bullet(\rho_i) = \langle \mid \phi'_\bullet(\rho)(\ell_i) \rangle\end{aligned}$$

It is straightforward to check that $(\phi'' \circ \phi''_\bullet \circ \phi)|_V \simeq (\phi'' \circ \phi'_\bullet \circ \phi)|_V$, and thus $(\phi'' \circ \phi''_\bullet \circ \phi)|_V \simeq \phi_\bullet$.

Our hypothesis $\forall (s,t) \in S.\ s\phi_\bullet \leq t\phi_\bullet$ can be rewritten $\forall (s,t) \in S.\ s(\phi'' \circ \phi''_\bullet \circ \phi) \leq t(\phi'' \circ \phi''_\bullet \circ \phi)$, or equivalently, $\forall (s,t) \in S.\ (s\phi)(\phi'' \circ \phi''_\bullet) \leq (t\phi)(\phi'' \circ \phi''_\bullet)$ where $(\phi'' \circ \phi''_\bullet) \in \mathcal{S}_\varnothing$. Consequently, by applying Theorem A.33, we get some $\phi_\Delta \in \mathsf{tally}(\{(s\phi, t\phi) \mid (s,t) \in S\}, \Delta, \varnothing)$ and $\phi'_\Delta \in \mathcal{S}_\varnothing$ such that $\phi'' \circ \phi''_\bullet \simeq \phi'_\Delta \circ \phi_\Delta$.

Let us define $\phi_\circ = (\phi'' \circ \phi_\Delta \circ \phi)|_V$ (thus satisfying $\phi_\circ \in \mathsf{tally}(S, \Delta, L)$) and $\phi'_\circ = (\phi'_\Delta \circ \phi)|_V$ (satisfying $\phi'_\circ \in \mathcal{S}_L$), and show that they satisfy $\phi_\bullet \simeq \phi'_\circ \circ \phi_\circ$ (which will conclude the proof). First, notice that for any $\rho \in V$, we have:

$$(\phi \circ \phi'')(\rho) = \langle \ell_1 : \rho_1,\ \ldots,\ \ell_n : \rho_n \mid \rho \rangle$$

and for any $\rho \in V$ and $i \in 1..n$, we have:

$$(\phi \circ \phi'')(\rho_i) = \langle \ell_1 : \rho_1,\ \ldots,\ \ell_n : \rho_n \mid \rho \rangle$$

We can thus deduce that, if a type t (resp. row R) is such that

(i) no $\rho \in V$ appear under a field $\ell \in L$ in t (resp. R), and

(ii) no ρ_i ($\rho \in V$, $i \in 1..n$) appear under a field other than ℓ_i in t (resp. R),

then we have $t(\phi \circ \phi'') \simeq t$ (resp. $R(\phi \circ \phi'') \simeq R$). Moreover, by applying Lemma A.37 on $\phi_\triangle \in \mathsf{tally}(\{(s\phi, t\phi) \mid (s,t) \in S\}, \Delta, \varnothing)$, we can deduce that for any $\alpha \in V$ (resp. $\rho \in V$), $(\phi_\triangle \circ \phi)(\alpha)$ (resp. $(\phi_\triangle \circ \phi)(\rho)$) satisfies (i) and (ii). We thus have $\phi'_\circ \circ \phi_\circ \simeq (\phi'_\triangle \circ \phi)|_V \circ (\phi'' \circ \phi_\triangle \circ \phi)|_V \simeq \phi'_\triangle|_V \circ (\phi \circ \phi'' \circ \phi_\triangle \circ \phi)|_V \simeq \phi'_\triangle|_V \circ (\phi_\triangle \circ \phi)|_V \simeq (\phi'_\triangle \circ \phi_\triangle \circ \phi)|_V \simeq (\phi'' \circ \phi''_\bullet \circ \phi)|_V \simeq \phi_\bullet$, which concludes the proof. □