Polymorphic Records for Dynamic Languages

GIUSEPPE CASTAGNA, CNRS, Université Paris Cité, France
LOÏC PEYROT, Université Paris Cité, France

We define and study *row polymorphism* for a type system with set-theoretic types, specifically union, intersection, and negation types. We consider record types that embed row variables and define a subtyping relation by interpreting types into sets of record values and by defining subtyping as the containment of interpretations. We define a functional calculus equipped with operations for field extension, selection, and deletion, its operational semantics, and a type system that we prove to be sound. We provide algorithms for deciding the typing and subtyping relations.

This research is motivated by the current trend of defining static type system for dynamic languages and, in our case, by an ongoing effort of endowing the Elixir programming language with a gradual type system.

CCS Concepts: • **Theory of computation** → *Type structures*; *Program analysis*; • **Software and its engineering** → *Functional languages.*

1 INTRODUCTION

The goal of this work is to study *row polymorphism* for a type system with set-theoretic types, in particular, union, intersection, and negation types. Row polymorphism was originally introduced by Rémy [1989, 1994] and Wand [1987, 1991] and further studied and developed in several works (e.g., [Gaster and Jones 1996; Leijen 2005; Morris and McKinna 2019; Tang et al. 2023]). However, as we argue in this work, the theories in the current literature cannot be reused for systems with set-theoretic types and, hence, an original one must be developed.

The interest of developing this new theory is grounded in practice. There is a growing effort to develop type systems for dynamic languages: TypeScript [Microsoft 2012] and Flow [Facebook 2015] are two major examples of such an effort and, as most of the existing attempts to add type systems to dynamic languages, they include union and intersection types to accommodate the flexible programming patterns prevalent in such languages. In particular, some widely used dynamic programming languages—such as Luau [Jeffrey 2022; Roblox 2021], and Elixir [2012]—have adopted set-theoretic types as defined in the *semantic subtyping* framework by Frisch et al. [2008]. There, (*i*) types are interpreted as sets of values, (*ii*) unions, intersections, and negation types are interpreted as the corresponding set-theoretic interpretations, and (*iii*) the subtyping between two types is defined as set-containment of their interpretations. Since semantic subtyping is the framework that we use here, our work can be considered as a study on adding row polymorphism to Luau and Elixir. In particular, the rest of this section uses Elixir's syntax to showcase practical motivations.

1.1 A motivating example

To show the interest of having row polymorphism in dynamic languages, let us consider the `logger` module of Elixir's standard library, which exports the following function (lightly rewritten to make its definition less dependent on Elixir idiom: see [Elixir 2024] for the actual version):

```
def add_elixir_domain(x) do
  case x do
    %{domain: y} when is_list(y) -> %{x | domain: [:elixir | y]}
    _ -> Map.put(x, :domain, [:elixir])
  end
end
```

This code snippet defines the function `add_elixir_domain`, whose argument, bound to `x`, is matched in a case expression. The argument must be a record value (in Elixir records are called *maps* and are delimited by curly brackets prefixed by the "`%`" symbol). If it is a record with (at least) a field for the key `:domain` whose content is a list (line 3), then the function returns a copy of the argument in which the field with key `:domain` is updated by consing the atom[1] `:elixir` to the list in the argument. Otherwise, (line 4), the function adds (if absent) or replaces (if present but does not contain a list) in the record `x` a field `:domain` whose content is the singleton list whose only element is the atom `:elixir` (this is performed by the function `Map.put` of Elixir's standard library).

The current proposal for the Elixir type system [Castagna et al. 2024a] does not feature row polymorphism. Hence, `add_elixir_domain` can be given the following type (Elixir uses the symbol `$` to signal that what follows is a type):

```
$ %{...} -> %{domain: list(term()),...}
```

In Elixir the ellipsis "`...`" in a record type means that the record type is "open", that is, that the values of the record type may define other fields besides those specified in the type. Therefore, the type above states that `add_elixir_domain` is a function that accepts any record value (since `%{...}` is the top record type) and returns a record with at least a `:domain` field that contains a list of values (in Elixir `term()` is the top type, which types all values, so `list(term())` is the type of all lists; following the Erlang convention, type names are post-fixed by `()`, e.g., `integer()`, `term()`, ...).

The problem with such a type is well known: the type does not specify that the fields corresponding to the ellipsis in the result are the same as those in the argument. Thus, any static knowledge of these fields is lost after the application. The practical consequence of this loss is that an expression such as `add_elixir_domain(%{file: "foo.txt", line: 42}).line`, which tries to select the field `:line` in the result of the application, is rejected by the type checker despite being correct. This hinders the applicability of the type system to existing code and obliges the programmer to resort to the less precise gradual typing features of Elixir's type system.

The solution to this problem is also well known, and resorts to using *row variables*, which are variables that range over "rows" of fields of a record type. For Elixir, this would correspond to extending the system so that `add_elixir_domain` could be typed as follows:[2]

```
$ %{f} -> %{domain: list(term()), f} when f: fields()
```

where `f` is a *row variable*, as stated by the post-fix declaration `f: fields()` (in Elixir, type variables are post-fixedly quantified by a `when` declaration). Now, when typing the previous expression `add_elixir_domain(%{file: "foo.txt", line: 42}).line`, this variable can be instantiated to include the two fields for the keys `:file` and `:line` in the argument, enabling the system to deduce for the expression the type `integer()`.

The system by Castagna et al. [2024a] for Elixir also features union, intersection, and negation types, denoted by `or`, `and`, and `not`. Combined with row polymorphism, they refine the previous type of `add_elixir_domain` as follows:[3]

```
$ (%{domain: list(a), f} -> %{domain: list(atom() or a), f}) and
  (%{:domain => not(list(term()), g} -> %{domain: list(atom()), g})
  when a: term(), f: fields(), g: fields()
```

[1] Atoms are user-defined constants prefixed by a colon and of type `atom()`. Field keys are atoms, too, in which the starting colon can be omitted if the field is not optional (see later on): `%{domain: 42}` has a field with key `:domain` and value `42`.
[2] The syntax we propose below may differ from the one that will be included in Elixir, if any.
[3] Such a refinement can also be done without row variables, but the problem with forgotten fields is still the same.

The type above uses the whole palette of set-theoretic types. It also uses both parametric and row polymorphism, since the type features a type variable `a` and two row variables `f` and `g`, as stated by the `when` declaration in line 11. The combination of these two features renders a more precise description of the behavior of `add_elixir_domain`:

- The arrow type in line 9 states that when `add_elixir_domain` is applied to a record value formed by a field `:domain` that contains a list of `a` elements, and by some other fields captured by `f`, then the function returns a record that is of the same type as the argument except in the `:domain` field that now contains a list of `atom() or a` elements (union type).
- The arrow type in line 10 specifies as input type a record type with an optional field (denoted by “`=>`”)[4] of type `not(list(term()))`. Thus, the function type in line 10 types functions that when applied to values in which the field `:domain` is either absent or bound to a value that is `not` a list (negation type), then it returns a record where the `:domain` field contains a list of atoms and, thanks to the row variable `g`, where the other fields of the argument are preserved.
- The two arrows above are connected by an intersection (the `and` connective at the end of line 9) meaning that the function has both types and, thus, obeys both specifications.

If `add_elixir_domain` is given the type in lines 9–11 and if `x` is defined as:

```
x = add_elixir_domain(%{domain: [41, 43], file: "foo.txt", line: 42})
```

then the type deduced for the expression `[x.line | x.domain]` is `list(atom() or integer())`.

It is out of the scope of this work to explain why the precision allowed by set-theoretic types is necessary to the typing of dynamic languages. For such an explanation, we invite the reader to refer to [Castagna et al. 2024a] which treats the case of Elixir. We want nevertheless to stress the importance of negation types for a precise typing of pattern matching. This is shown in line 4 of the definition of `add_elixir_domain`, where the type deduced for the variable `x` occurring in that line is obtained by removing, by a negation type, from the type of the input the type of all the values captured by the preceding pattern occurring in line 3.

In the rest of this introduction we explain the details of adding row polymorphism in the presence of set-theoretic types and how to use it to type a language with extensible records in which fields can be added or deleted. We start by explaining why we need a new theory for row variables.

1.2 The need for row polymorphism

Even before considering whether we need a new theory for row polymorphism, we may wonder whether we need row polymorphism at all. The system we are planning to extend features first order (a.k.a., prenex) polymorphism with set-theoretic types and their combination is enough to encode a limited form of bounded polymorphism, which can be used to type the following `bump_counter` functions without losing the static information of the fields of the argument

```
def bump_counter(x), do: %{x | counter: x.counter+1}
```

This function increments the `:counter` field of its argument and can be typed by bounded polymorphism by the type $\forall(\alpha{\leq}\texttt{\{counter=integer(),}\,...\texttt{\}}).\alpha \to \alpha$. The type variable α captures the whole type of the argument, thus also its extra fields. Thanks to that the type system deduces for

[4] A record type such as `%{:bar => t()}` means that in the values of this type, a field for the key `:bar` may be absent, but if present it must contain a value of type `t()`. This is Erlang’s Typespec syntax: in Elixir the keyword `optional` is mandatory, as in `%{optional(:bar) => t()}`, but we omit it to streamline the presentation.

`(bump_counter(%{counter: 1, file: "foo.txt"})).file` the type string. The bounded polymorphic type above is encoded by set-theoretic types as follows (see [Castagna 2024]):[5]

```
14 $ (%{counter: integer(),...} and a  ->  %{counter: integer(),...} and a) when a: term()
```

for which Elixir provides the nifty shorthand `(a -> a) when a: %{counter: integer(),...}` of bounded polymorphism.

The example above may suggest that intersecting record types with type variables could play the same role as row polymorphism. Unfortunately, the example above is one of the few cases in which this works.[6] This technique may work to type functions in which the input and the output have the same fields; but even in that case it is easy to have a partial loss of information. For instance, consider the following function that takes as input a record with a field `foo` and redefines its content (in Elixir, functions are annotated by the `$`-prefixed type that precedes their definition):

```
15 $ (a, b) -> a when b: term(), a: %{foo: b,...}
16 def redefine_foo(x,y), do: %{x | foo: y}
```

even though we do not lose the static type information of any field of the argument (thanks to the type variable `a`), the type for the `:foo` field may lose precision. For example, the type deduced for `redefine_foo(%{foo: 42},true).foo` is the union `integer() or boolean()`—rather than just `boolean()`—since the type variable `b` is unified with the *union* of the type of the second argument and the type of the field `foo:` of the first argument. Furthermore, the technique fails when we try to add a new field to a record or delete an existing field from it. For instance, consider again the function `Map.put`, whose use in line 4 can be abstracted as follows:

```
17 $ (%{...} and a -> %{domain: list(atom()),...} and a) when a: term()
18 def put_domain(x), do: Map.put(x, :domain, [:elixir])
```

One might think that the use of the intersection with the type variable `a` captures all the fields of the argument, but the type declaration is wrong—and, as such, rejected by the type system—because if we instantiate the variable `a` by a type such as `%{domain: integer()}`, then we deduce for `put_domain` the type `%{domain: integer()} -> none()`, where `none()` is the empty type (resulting by simplifying the intersection in the codomain): this type states that the application of `put_domain` to an argument of type `%{domain: integer()}` must diverge (since any returned value must be in the empty type, which contains none), which is clearly wrong. A similar problem happens when we apply the record operations of deleting a field or of adding a field that is not already present.

The problem with the type in line 17 is that for each intersection in it not to be empty, the type variable `a` must be instantiated by a record type that contains all the fields both of the input and of the output, so in particular the field for `:domain`. If, as above, the field `:domain` has different types in the input and in the output, then one of the two intersections results empty, which is unsound. For the typing to work we need the type variable to instantiate all the fields of the input *except the field for* `:domain`. This is exactly the role of row variables, which instantiate "all the other fields" of the record type. The function `put_domain` can be given any of the two following types

```
19 $ %{f}  -> %{domain: list(atom()),f} when f: fields()
20 $ %{:domain => term(), f} -> %{domain: list(atom()), f}  when f: fields()
```

[5] Intersections and unions have a precedence higher than arrows and records, and negation has the highest precedence of all.
[6] Even this is not true: it is just an approximation we did for presentation purposes, since it is unsound to deduce the type `a` for the result type. To see why, try to instantiate `a` with `%{counter: 42}`.

The type in line 19 is syntactic sugar for the one in line 20 whose domain, despite being more verbose, explicitly states that the field `:domain` is either undefined or contains a value of any type. The type in 20, thus, explicitly shows that the row variable `f` captures all fields but `:domain`.

Likewise, we need row variables for typing a function that deletes, say, the `:domain` field since, once again, the type of the deleted field will be different in the input and in the output:

```
21 $ %{:domain => term(), f} -> %{:domain => none(), f}  when f: fields()
22 def del_domain(x), do: Map.delete(x, :domain)
```

The codomain of the type in line 21 states that the field for `:domain` must be absent (i.e., if present, it must contain a value of the empty type `none()`, of which there is none). As before, the type above can be more conveniently written as: `%{f} -> %{:domain => none(), f} when f: fields()`, and again, this works because `f` can only be instantiated to rows that do not contain a field for `:domain`.

Since we have established that we need row polymorphism, the next question is why not use the existing theory? A first reason is that to integrate row polymorphism in a semantic subtyping setting, we need to define subtyping for the polymorphic records, and this requires to interpret the row-polymorphic record types as sets of values which, in our ken, was not done before. A second peculiarity of our setting are optional fields that, as far as we know, are not dealt with by the current approaches (see the discussion on presence polymorphism in Section 5 and Appendix A.1). A third and much harder to tackle reason is that the usual unification techniques used in row polymorphism are not sufficient in this setting. Indeed, consider the following example:

```
23 type figure() = %{shape: "circle", perim: integer(), diam: float()} or
24                 %{shape: "polygon", perim: integer(), edges: integer()}
25 $ {perim: integer(), f} -> {perim: float(), f} when f: fields()
26 def perim_to_float(x), do: %{x | perim: to_float(x.perim)}
```

The first two lines define the type of figures, which are records with an integer field `:perim` and with either a `:diam` or an `:edges` field, according to the value of their `:shape` field. In line 26 we define a function that transforms the integer field `:perim` of the input into float. Its type is given in line 25. Now, if we apply the function `perim_to_float` to an argument of type `figure()` we expect to deduce the for it a type like `figure()` but where the `:perim` field is of type `float()`, that is

```
27 $ %{shape: "circle", perim: float(), diam: float()} or
28   %{shape: "polygon", perim: float(), edges: integer()}
```

but current theories of row polymorphism unify record types component-wise, which in our case would yield the following less precise type:

```
29 $ %{shape: "circle" or "polygon", perim: float(), :diam => float(), :edges => integer()}
```

where the `:shape` field has now a union type and `:diam` and `:edges` have become optional. To deduce the type in lines 27 and 28 the row variable `f` must be expanded into the union of two rows, one for the shape circle and the other for the shape polygon.

Even if the problem above can be solved by particular typing techniques, the presence of negation types in our theory invalidate such techniques in general (see example in Appendix C.1). Therefore, we need to develop an original technique to replace unification, so that it takes into account subtyping and enables substitutions to expand row variables into Boolean combinations of rows.

In conclusion, to embed row polymorphism in a type system featuring semantically defined union, intersection, and negation types, we need new theoretical developments to cope with the semantic interpretation of types and the inference of substitutions for type variables in the presence of subtyping and set-theoretic types. To do that, we proceed as we describe next.

1.3 Overview

In Section 2 we define the syntax of types and the subtyping relation. We abandon Elixir's syntax for maps and introduce in Section 2.1 a more theoretically-oriented one: we denote records types as finite lists of field type specifications of the form $\ell = \tau$, followed by a *tail* ς specifying an infinite row of fields as in $\{\ell_1 = \tau_1, ..., \ell_n = \tau_n \,|\, \varsigma\}$. In this type, each ℓ_i denotes a label (or key) of a field, and labels are pairwise distinct; τ_i is either a type t or the union $t \vee \bot$, meaning that the field is optional with type t (i.e., Elixir's `:key => t()` field) and absent if t is the empty type (i.e., `:key => none()`); ς is either a row variable ρ, or "$..$" (meaning that the record type is open), or "ϵ" (meaning that the record is closed: its values contain all and only the fields specified in the type).

To define the subtyping relation on types, we give in Section 2.2 an interpretation of types as sets of elements of a domain $\mathcal{D}$—whose elements, intuitively, represent the values of the language—and then define subtyping as containment of the interpretations. Following Frisch [2004], records values are interpreted as quasi-constant functions, that is, functions that map all labels into $\bot$ (meaning that the field for that label is undefined) except for a finite set of labels that are mapped into values. Therefore, (ground) record types are interpreted as sets of quasi-constant functions. More subtle is the interpretation for row variables which, as we saw, define the type of *all the labels but a few ones*; as a consequence our interpretation will map them into *partial* quasi-constant functions, requiring a careful handling of their domains.

In Section 2.3 we define an algorithm to decide the subtyping relation just defined. We do so by extending the subtyping algorithm of the language CDuce [Benzaken et al. 2003], on which we base our theory, and monomorphic record types to our polymorphic records. This algorithm essentially works on Boolean combinations of record type atoms put into a specific normal form that we have to adapt to deal with row variables. This normal form is then decomposed by comparing the types of the fields and the tails of the record type atoms that compose it, using appropriate combinatorics. The resulting algorithm has the same order of complexity as the one for monomorphic record types, and we also define a backtracking-free implementation for it. Finally, we give a formal definition for type substitutions in Section 2.4, in particular for the case of row variables which are expanded into Boolean combinations of rows of fields, and prove that its application preserves subtyping.

In Section 3 we define a language with record operations. For record operations, several equivalent choices are possible [Cardelli and Mitchell 1991]. We build records starting from the empty record value, noted `{}`, and adding new fields to it by the expression $\{e \texttt{ with } \ell = e'\}$ which extends the record (resulting from the evaluation of) e with the field $\ell = e'$, provided that a field for ℓ is not already present in e (this constraint is statically enforced by the type system: see rule [Ext] in Fig. 1). The other operations on records are field selection, noted $e.\ell$, which returns the content of the field ℓ in e, and field deletion, noted $e \backslash \ell$ which removes from e the field labeled ℓ, if any. We define an operational semantics and a declarative type system, and we show that the latter is sound in the sense of Wright and Felleisen [1994], by proving that every well-typed expression either diverges or returns a value of the expression's type (Section 3.1). Next, we define an algorithmic type system and prove it to be sound and complete with respect to the declarative one. The system is derived from the declarative one in a standard way: subsumption is embedded in the elimination rules, intersection introduction is essentially embedded in the typing of λ-abstractions, and the rule for applications performs instantiation and expansion by looking for a set of substitutions that make the type of the argument be a subtype of the domain of the function (Section 3.2).

Section 4 studies the tallying problem, which plays the same role as the unification problem in type inference, but for a subtyping—rather than an equality—relation on types. The algorithmic system in Section 3.2 is effective, provided that we produce an algorithm to deduce the type substitutions to apply to the types of the function and the argument when typing an application. Following Castagna et al. [2015] this can be done by solving the tallying problem for our types, namely, the problem of deciding whether given two types, there exists a type substitution that makes one type subtype of the other. Castagna et al. [2015] prove that the problem is decidable for a system with function and product type constructors and set-theoretic types, and give a sound and complete algorithm. However, defining a tallying algorithm for types with row variables is far more difficult. This is because substitutions replace row variables by Boolean combinations of rows of fields. We tackle this problem in Section 4, where we define a tallying algorithm for row polymorphic types. We prove that the algorithm is sound but not complete, and we conjecture completeness for the case in which row variables are substituted by a single row of fields.

We conclude by discussing related work (Section 5) and further research directions (Section 6).

1.4 Contributions and limitations

The overall contribution of this work is threefold, since it provides (i) a theory for a first-order polymorphic type system with row polymorphism and set-theoretic types, (ii) the practical motivations for such a system, as well as (iii) the relevant algorithms to apply it in practice. In particular, all the examples we presented in Sections 1.1 and 1.2 are typed by our system.

The technical contributions can be summarized as follows:

(1) We describe a first-order polymorphic type theory with union, intersection and negation type connectives, and function and record type constructors, where record types can be either closed, open, or specify a row variable, and their fields can be declared optional (Section 2.1). We define a subtyping relation for these types by providing an interpretation where types are interpreted as set of values and subtyping as set containment (Section 2.2).
(2) We prove that the subtyping relation is decidable and provide a backtrack-free algorithm to decide it (Section 2.3).
(3) We define type substitutions that map row variables into Boolean combinations of rows, and prove that the application of type substitutions preserves subtyping (Section 2.4).
(4) We define a declarative type system for a record calculus with record extension, selection, and deletion and prove its soundness (Section 3.1).
(5) We define an algorithmic system that we prove sound and complete with respect to the declarative one (Section 3.2).
(6) We define an algorithm for the *tallying problem*, that is, the problem of deciding whether given two types there exists a type substitution that make one subtype of the other; we prove soundness of the algorithm (Section 4).

The system we define presents some limitations. Some are expected and characteristic of the kind of systems we consider here: the typing relation is not decidable (this is typical of systems with intersection types) and the type system has no principal types (which is already the case both for systems with polymorphic set-theoretic types [Castagna et al. 2015, 2014] and for expressive record type systems [Cardelli and Mitchell 1991]). Other limitations are instead new, in particular that the tallying algorithm is sound but not complete (an example is given in Example 4.5; a complete algorithm exists when record types are kept out of the equation: see Castagna et al. [2015]). We prove that one of the reasons for incompleteness is that we interpret row variables into Boolean combinations of rows rather than into single rows, and we conjecture that completeness can be recovered in the latter case, but at the expenses of the type system which can type fewer expressions (cf. Example C.2).

Finally, from a practical point of view, the main limitation of this system is that it does not feature first class labels, that is, the operations for field selection, extension, and deletion must specify nominal labels which, thus, cannot be obtained as the result of a computation. This important omission might hinder the application of our theory to dynamic languages where such a feature is widely used. This omission, however, is deliberate since we wanted to focus on the problem of row polymorphism, and we consider that having first-class labels is mostly orthogonal to it. We believe that it will not be hard to extend our work on the lines of Castagna [2023] to have first class labels also in our system, and we leave it for future work.

2 TYPES

We introduce the syntax of types (Section 2.1) and their set-theoretic interpretation from which we derive the subtyping relation (Section 2.2). We define the algorithm to decide the subtyping relation (Section 2.3) and prove that subtyping is preserved by type substitutions (Section 2.4).

2.1 Syntax of Types

Definition 2.1 (Rows and Types). *Let $\mathcal{L}$ be a countable set of labels ranged over by ℓ, $\mathcal{V}_t$ be a countable set of* type variables *ranged over by α, $\mathcal{V}_f$ be a countable set of* field-type variables *ranged over by θ, $\mathcal{V}_r$ be a countable set of* row variables *ranged over by ρ, and $\mathcal{B}$ a finite set of basic types (e.g., Int, Bool, ...) ranged over by b. The set of* rows *$\mathcal{R}$, ranged over by r, contains all terms inductively generated by the corresponding grammar below. The set of* types *$\mathcal{T}$, ranged over by t, contains all terms coinductively generated by the corresponding grammar below and that* (1) *have a finite number of different sub-terms (regularity) and* (2) *in which every infinite branch contains an infinite number of occurrences of the record or arrow type constructors (contractivity).*

$$\begin{array}{llll}
\textbf{\textit{Kinds}} & \kappa & ::= & \star \mid \star_\perp \mid \mathrm{Row}(L) \\
\textbf{\textit{Types}} & t & ::= & \alpha \mid b \mid t \to t \mid \{\ell = \tau, \ldots, \ell = \tau \,|\, \varsigma\} \mid t \vee t \mid \neg t \mid \mathbb{0} \\
\textbf{\textit{Field types}} & \tau & ::= & \theta \mid t \mid \perp \mid \tau \vee \tau \mid \neg\tau \\
\textbf{\textit{Tails}} & \varsigma & ::= & \rho \mid \epsilon \mid .. \\
\textbf{\textit{Rows}} & r & ::= & \langle \ell = \tau, \ldots, \ell = \tau \,|\, \varsigma\rangle^L \mid r \vee r \mid \neg r
\end{array} \tag{1}$$

where $L \in \mathcal{P}_{\mathrm{fin}}(\mathcal{L})$ is a finite set of labels and field types are inductively defined.

We use T to range over types, field types, and rows, and define $T_1 \wedge T_2 \stackrel{\text{def}}{=} \neg(\neg T_1 \vee \neg T_2)$ and $T_1 \setminus T_2 \stackrel{\text{def}}{=} T_1 \wedge \neg T_2$. For types, we define $\mathbb{1} \stackrel{\text{def}}{=} \neg\mathbb{0}$. For rows, we use a generic notation $\mathbb{0}$ for $\neg\langle\,|\,..\rangle^L$, whatever the L. Following a mathematical logic terminology, basic types, arrows, and records are called *type constructors* and yield *type atoms*, while unions, intersections, and negations are *type connectives*.

The types of Definition 2.1 are those of Castagna and Xu [2011] where record types replace product types. Coinduction accounts for recursive types and comes with the usual restrictions of regularity (necessary for the decidability of the subtyping relation) and contractivity (which rules out meaningless types such as an infinite tower of negations, while providing a well-founded order for inductive proofs); see [Castagna 2024] for a detailed explanation.

We use R to range over record type atoms, that is, types of the form $\{\ell = \tau, \ldots, \ell = \tau \,|\, \varsigma\}$. These are unordered finite sets of fields, mapping pairwise distinct labels into field types, and followed by a tail. We often use the more compact notation $\{(\ell = \tau_\ell)_{\ell \in L} \,|\, \varsigma\}$ where $L \in \mathcal{P}_{\mathrm{fin}}(\mathcal{L})$ (as in the above definition and hereafter). If $\mathsf{R} = \{(\ell = \tau_\ell)_{\ell\in L} \,|\, \varsigma\}$, then we define $\mathrm{lab}(\mathsf{R}) \stackrel{\text{def}}{=} L$, $\mathsf{tail}(\mathsf{R}) \stackrel{\text{def}}{=} \varsigma$, and similarly for a row $r = \langle(\ell = \tau_\ell)_{\ell\in L} \,|\, \varsigma\rangle^{L'}$. The type R above is said to be *closed* if $\varsigma = \epsilon$, *open* if $\varsigma = ..$, and *polymorphic* if ς is a row variable. We use the notation $\mathsf{R}(\ell)$ to denote the field type associated to ℓ by R. Precisely, if $\mathsf{R} = \{(\ell = \tau_\ell)_{\ell\in L} \mid \varsigma\}$, then:

$$\mathsf{R}(\ell) \stackrel{\text{def}}{=} \begin{cases} \tau_\ell & \text{if } \ell \in L \\ \bot & \text{if } \ell \in \mathcal{L} \setminus L \text{ and } \varsigma = \epsilon \\ \mathbb{1} \vee \bot & \text{otherwise} \end{cases}$$

Record types are largely those of polymorphic CDuce [Castagna et al. 2015, 2014; *CDuce Manual*] with two exceptions: row variables and field types. The theory for the records used first in CDuce and then in other languages (e.g., Ballerina [Clark 2022] and Elixir), was defined by Frisch [2004] and recently extended by Castagna [2023] to account for first class labels. It considers record values as total functions that map every label in $\mathcal{L}$ either into a value or into a distinguished symbol $\bot$ representing undefined. Since record values are defined only on a finite set of labels, then these functions map every label into $\bot$ apart from a finite set of labels. Frisch [2004] calls these functions the *quasi-constant* functions, since they map a co-finite part of their domain into a single image called the default value: see Definition 2.2 for their formal definition.

Record types reflect this interpretation: they are composed by a finite set of field types, which covers a finite set of labels, and by a tail which covers the infinitely many remaining labels including the constant part of the quasi-constant function. The definition of field types given in Definition 2.1 states that when the type of the field is not a variable, then it is morally either t or $t \vee \bot$: in the former case the field is mandatory with type t; in the latter case it is optional with type t (and if $t = \mathbb{0}$, then the field is undefined).

The definition of field types includes Boolean connectives and field-type variables (*field variables* for short). The latter are used in two ways: for presence polymorphism (see Section 5 and Appendix A.1) and to solve the tallying problem (Section 4). The tallying problem generates constraints with Boolean formulas containing variables in arbitrary positions. Since these variables can be instantiated by $t \vee \bot$ we have to differentiate them from type variables which can be instantiated only by types (hence the introduction of field variables θ) and, as a consequence of instantiation, allow $\bot$ to appear anywhere (hence the introduction of Boolean combinations).

We said in the introduction that a row variable in a record type stands *for all the fields not already defined in the record type*. Thus, in $\{(\ell = \tau_\ell)_{\ell\in L} \mid \rho\}$ the row variable ρ represents the fields whose labels are in $\mathcal{L} \setminus L$. Formally, each row variable ρ defines the fields for a cofinite set of labels that we call the *domain* of ρ and note as $\mathrm{dom}(\rho) \in \mathcal{P}_{\text{cofin}}(\mathcal{L})$. Since record types are *total* functions on $\mathcal{L}$, we cannot use them to interpret row variables. We need another syntactic category to denote *partial* functions defined on co-finite sets of labels, that we will use to interpret row variables.

This syntactic category is the one of *rows* also given in Definition 2.1. Rows are of the form $\langle(\ell = \tau_\ell)_{\ell\in L} \mid \varsigma\rangle^{L'}$. The relevant part is the set $L' \in \mathcal{P}_{\text{fin}}(\mathcal{L})$ at the index, which denotes the finite set of labels on which the row is *not* defined. In other terms, the row above is a total function from $\mathcal{L} \setminus L'$ into field types: the τ_ℓ's define the fields for the labels in L and the tail ς the fields for the labels in $\mathcal{L} \setminus (L \cup L')$. Of course, not every row or record type is well-formed, since we must ensure that the various L, L', and $\mathrm{dom}(\rho)$ form a partition of $\mathcal{L}$. They have to verify three properties:

(1) In a type $\{(\ell = \tau_\ell)_{\ell\in L} \mid \rho\}$ we must have $\mathcal{L} \setminus L = \mathrm{dom}(\rho)$;
(2) In a row $\langle(\ell = \tau_\ell)_{\ell\in L} \mid \varsigma\rangle^{L'}$ we must have $L \cap L' = \emptyset$ and, if $\varsigma = \rho$, $\mathcal{L} \setminus (L \cup L') = \mathrm{dom}(\rho)$;
(3) Unions (and, thus, intersections) can only be performed between rows defined on the same set of labels.

These conditions are enforced by a kinding system (whose straightforward definition is given in Fig. 3), which selects the terms that are well-formed in the kind $\star$ of types, the kind $\star_\bot$ of field types, and the kind $\mathrm{Row}(L)$ of rows that are defined on the labels in $\mathcal{L} \setminus L$.

2.2 Subtyping relation

Following Frisch et al. [2008] the subtyping relation for the types of the previous section is defined by interpreting them into subsets of some domain $\mathcal{D}$ whose elements represent the values of the language, and by defining subtyping as the containment of interpretations. Defining an interpretation for our types presents three major challenges: (1) how to interpret function types into a domain $\mathcal{D}$ which, for cardinality reasons, cannot contain its function space; (2) how to interpret type, row, and field variables; (3) how to interpret polymorphic record types.

The techniques to solve the first two problems are already found in the literature on semantic subtyping. Frisch et al. [2008] interpret function types by giving the definition of a *model*, that is, a domain and an interpretation on that domain that satisfy some specific conditions. These conditions are given by an auxiliary function called the *extensional interpretation* and require that the zeros of the interpretation function and of its extensional interpretation are the same. Frisch et al. [2008] then define some specific interpretations and prove that each of them satisfies the conditions of model, thus yielding a subtyping relation. When types include type variables, Castagna and Xu [2011] show that the solution by Frisch et al. [2008] can be used provided that the interpretation function and its extensional interpretation are parametric in the interpretation of the variables, and that models satisfy a further condition called *convexity*. To solve the third challenge, then, one can start from the interpretation of monomorphic records given by Frisch [2004], extend it with the techniques by Castagna and Xu [2011] to interpret type variables, adapt it to interpret field variables, and introduce new techniques to handle row variables.

This is exactly what we do here. We extend the definitions of extensional interpretation and of model given by Castagna and Xu [2011] to cope with polymorphic record types. Then we define a specific interpretation, that is convex by construction, and prove it to satisfy the conditions of a model. To do this proof we need the definition of a second specific interpretation, which is harder to work with (e.g., for proving properties of the subtyping relation) but that can be more easily proved to be a model. Finally, we prove that the latter interpretation is equivalent to (in the sense that it induces the same subtyping relation as) the former. For space reasons, we present here only the specific interpretation that we use to define subtyping and derive its decision procedure. The comprehensive technical development with the definition of the extensional interpretation, of a model, and the proof that the interpretation we present below is indeed a model is moved to Appendix A.2, given as supplemental material of the submission.

To interpret record values we follow Frisch [2004] and represent a record value by a quasi-constant function that maps labels into either values (i.e., the elements of $\mathcal{D}$) or $\bot$. Quasi-constant functions are total functions that map all but a finite set of elements of their domain into the same value (called default value). Thus, record values can be represented by quasi-constant functions whose default value is $\bot$ (see Castagna [2023] for a more detailed explanation). Formally, we have the following definition.

Definition 2.2 ([Frisch 2004]). *Let Z denote some set. A function $r : \mathcal{L} \to Z$ is* quasi-constant *if there exists $z \in Z$ such that the set $\{\ell \in \mathcal{L} \mid r(\ell) \neq z\}$ is finite. We use $\mathcal{L} \twoheadrightarrow Z$ to denote the set of quasi-constant functions from $\mathcal{L}$ to Z and $\{\!\!\{\ell_1 = z_1, \dots, \ell_n = z_n, \_ = z\}\!\!\}$ to denote the quasi-constant function $r : \mathcal{L} \twoheadrightarrow Z$ defined by $r(\ell_i) = z_i$ for $i = 1..n$ and $r(\ell) = z$ for $\ell \in \mathcal{L} \setminus \{\ell_1, \dots, \ell_n\}$.*

Let us write $\mathcal{D}_\bot$ for $\mathcal{D} \cup \{\bot\}$ where $\bot$ is a distinguished element not in $\mathcal{D}$. We represent our record values as quasi-constant functions from $\mathcal{L}$ to $\mathcal{D}_\bot$ and, thus, interpret record types as sets of these values, that is, of functions in $\mathcal{L} \twoheadrightarrow \mathcal{D}_\bot$. But we have also to interpret *rows*, which are defined only on a cofinite *subset* of $\mathcal{L}$. In other terms, rows are *partial* quasi-constant functions from $\mathcal{L}$ to $\mathcal{D}_\bot$, that we note $\mathcal{L} \not\twoheadrightarrow \mathcal{D}_\bot$.

Since a total function is also a partial one, then we need just the latter in our domain to interpret record types and rows. This yields the following definition of domain.

Definition 2.3 (Interpretation Domain). *As* interpretation domain $\mathcal{D}$ *for types, we take the set of finite terms* d *produced inductively by the following grammar, where* c *ranges over the set* C *of constants,* ℓ *over the set* $\mathcal{L}$ *of labels, and* V *over sets of variables contained in* $\mathcal{V} = \mathcal{V}_t \cup \mathcal{V}_f \cup \mathcal{V}_r$*. The interpretation domain* $\mathcal{D}_\perp$ *for fields (resp.* $\mathcal{D}_{\text{row}}$ *for rows) is the set of terms* δ *(resp.* $\dj$*).*

$$
\begin{array}{llr}
d & ::= c^V \mid \{(d,\partial),\ldots,(d,\partial)\}^V \mid \mathrm{Rec}(\dj)^V & \mathrm{dom}(\dj) = \mathcal{L} \\
\dj & ::= \llparenthesis \ell_1 = \delta,\ldots,\ell_n = \delta, \_ = \perp^{\emptyset} \rrparenthesis_L^V & L \in \mathcal{P}_{\text{fin}}(\mathcal{L} \setminus \{\ell_1,\ldots,\ell_n\}) \\
\partial & ::= d \mid \Omega & \\
\delta & ::= d \mid \perp^V &
\end{array}
$$

We use D *for an element that is either* d, $\dj$ *or* δ*. We define* $\mathrm{dom}(\llparenthesis(\ell = \delta_\ell)_{\ell \in L_1}, \_ = \perp^{\emptyset}\rrparenthesis_{L_2}^V) = \mathcal{L} \setminus L_2$*. We use* $\mathrm{tag}(D)$ *to denote the set of variables indexing* D*, that is,* $\mathrm{tag}(c^V) = \mathrm{tag}(\{(d,\partial),\ldots,(d,\partial)\}^V) = \mathrm{tag}(\mathrm{Rec}(\dj)^V) = \mathrm{tag}(\llparenthesis \ell_1 = \delta,\ldots,\ell_n = \delta, \_ = \perp^{\emptyset}\rrparenthesis_L^V) = \mathrm{tag}(\perp^V) = V$*.*

The intuition behind this definition is simple. The elements of the domain are constants C to interpret basic types, sets of finite binary relations $\mathcal{P}_{\text{fin}}(\mathcal{D} \times \mathcal{D}_\Omega)$ to interpret function types, and *partial* quasi constant functions $\mathcal{L} \nrightarrow \mathcal{D}_\perp$ to interpret rows (and record types by the total ones). The fact that functions are finite binary relations is standard in semantic subtyping and corresponds to interpret function spaces into the infinite set of their finite approximations; that these binary relations can yield a distinguished element Ω (which, intuitively, represents a type error) is also a standard technique of semantic subtyping to avoid $\mathbb{1} \to \mathbb{1}$ to be a supertype of all function types: since both aspects not play any specific role in our work we will not further comment on them (see [Frisch et al. 2008] for a detailed explanation or Castagna [2023, Section 3.2] for a shorter one).

If we look more closely at the definition of the row elements in Definition 2.3, we see that they are *partial* quasi-constant functions in $\mathcal{L} \nrightarrow \mathcal{D}_\perp$ with default value $\perp$. More precisely, the row element $\llparenthesis \ell_1 = \delta_1,\ldots,\ell_n = \delta_n, \_ = \perp^{\emptyset}\rrparenthesis_L$ is the quasi-constant function $\{\!\!\{\ell_1 = \delta_1,\ldots,\ell_n = \delta_n, \_ = \perp\}\!\!\}$ in $(\mathcal{L} \setminus L) \to \mathcal{D}_\perp$. When a row is total on $\mathcal{L}$, then it can be wrapped in a Rec() constructor yielding (the interpretation of) a record value (the inverse operation is given in Definition 2.4 below).

All these elements are indexed by a finite set of variables ranged over by V. This standard technique was introduced by Gesbert et al. [2015] to interpret type variables (cf. Definition 2.5), while ensuring that the model we obtain is *convex* in the sense of Castagna and Xu [2011]. The domain of Definition 2.3 is the one by Gesbert et al. [2015], but where records and rows replace pairs.

Definition 2.4. *Let* $\mathsf{R} = \{\!(\ell = \tau_\ell)_{\ell \in L} \mid \varsigma\!\}$*. We define* $\mathrm{row}(\mathsf{R}) = \langle(\ell = \tau_\ell)_{\ell \in L} \mid \varsigma\rangle^{\emptyset}$*. We extend this definition homomorphically to Boolean combinations of record type atoms.*

We have now all ingredients needed to define our set-theoretic interpretation for the types:

Definition 2.5 (Interpretation of Types and Rows as Sets). *Let* $\mathcal{D}$ *be the domain of Definition 2.3 and* $\mathcal{T}$ *the types of Definition 2.1. We define a binary predicate* $(D : T)$ *("the element* D *belongs to* T*") on* $\mathcal{D} \times \mathcal{T} \cup \mathcal{D}_\perp \times \mathcal{T}_\perp \cup \mathcal{D}_{\text{row}} \times \mathcal{R}$ *by induction on the pair* (D, T) *ordered lexicographically. The predicate is only defined if* D *is coherent with the kind of* T*:* $D = d$ *if* $T = t$*,* $D = \delta$ *if* $T = \tau$*, and* $D = \dj$ *if* $T = r$ *and* $\mathrm{dom}(D) = \mathrm{dom}(r)$*.*

- *On types:*

$$(d : \alpha) = \alpha \in \mathrm{tag}(d) \qquad (c^V : b) = c \in \mathbb{B}(b) \qquad (\mathrm{Rec}(\dj)^V : \mathsf{R}) = (\dj : \mathrm{row}(\mathsf{R}))$$

$$(\{(d_1,\partial_1),\ldots,(d_n,\partial_n)\}^V : t_1 \to t_2) = \forall i \in [1..n].\ \text{if } (d_i : t_1) \text{ then } (\partial_i : t_2)$$

- *On fields:*

$$(\delta : \theta) = \theta \in \mathsf{tag}(\delta) \qquad\qquad (\bot^V : \bot) = \mathsf{true}$$

- *On atomic rows:*

$$\begin{aligned}(\llparenthesis(\ell = \delta_\ell)_{\ell\in L_1}, \_ = \bot^\emptyset\rrparenthesis^V_{L_2} : r) = (\forall \ell \in L_1.(\delta_\ell : r(\ell))) \text{ and } (\forall \ell \in \mathrm{dom}(r) \setminus L_1.(\bot^\emptyset : r(\ell)))\\ \text{and } \mathsf{tail}(r) = \rho \implies \rho \in V\end{aligned} \tag{2}$$

- *On types, fields and rows:*

$$\begin{aligned}(D : T_1 \vee T_2) &= (D : T_1) \text{ or } (D : T_2) \textit{ and } T_1, T_2 \textit{ are of the same kind}\\ (D : \neg T) &= \text{not } (D : T) \textit{ and the kinds of } D \textit{ and } T \textit{ correspond}\\ (D : T) &= \text{false} \qquad\qquad \textit{otherwise}\end{aligned}$$

We define the interpretation $[\![\cdot]\!] : \mathcal{T} \to \mathcal{P}(\mathcal{D})$ *as* $[\![t]\!] = \{d \in \mathcal{D} \mid (d : t)\}$.

The interpretation of Definition 2.5 is mostly the same as the one by Gesbert et al. [2015], where the interpretation of a type variables α is the set of all elements that are tagged by α. The only difference with respect to [Gesbert et al. 2015] is the interpretation of rows and, thus, of record types. Equation (2) defines when a row element is in the interpretation of a row r: it requires that all the components of the row element are in the interpretations of the types specified by r (first line) and if the tail of r is a row variable, then it must index the row element (second line).

Definition 2.6 (Subtyping). *Let* $[\![\cdot]\!] : \mathcal{T} \to \mathcal{P}(\mathcal{D})$ *be the interpretation from Definition 2.5. It induces the following subtyping relation in* $\mathcal{T}\times\mathcal{T}$*:*

$$t_1 \leq t_2 \overset{\text{def}}{\iff} [\![t_1]\!] \subseteq [\![t_2]\!]$$

The interpretation also induces the subfield relation in $\mathcal{T}_\bot\times\mathcal{T}_\bot$ *and subrow relation in* $\mathcal{R}\times\mathcal{R}$ *defined as*

$$\tau_1 \leq \tau_2 \overset{\text{def}}{\iff} [\![\tau_1]\!]^{\text{fld}} \subseteq [\![\tau_2]\!]^{\text{fld}} \qquad r_1 \leq r_2 \overset{\text{def}}{\iff} [\![r_1]\!]^{\text{row}} \subseteq [\![r_2]\!]^{\text{row}}$$

where, the interpretation $[\![\cdot]\!]^{\text{fld}} : \mathcal{T}_\bot \to \mathcal{P}(\mathcal{D}_\bot)$ *is defined as* $[\![\tau]\!]^{\text{fld}} = \{\delta \in \mathcal{D}_\bot \mid (\delta : \tau)\}$, *and the interpretation* $[\![\cdot]\!]^{\text{row}} : \mathcal{R} \to \mathcal{P}(\mathcal{D}_{\text{row}})$ *is defined as* $[\![r]\!]^{\text{row}} = \{\eth \in \mathcal{D}_{\text{row}} \mid (\eth : r)\}$.

2.3 Deciding subtyping

Now that subtyping is defined via an interpretation of types of a specific model, we need an effective decision procedure for it. We derive it in two steps: first, we give a formula to decompose subtyping of two record types into several subtyping problems for the subterms of these record types (Lemma 2.1); second, we define an efficient implementation of this decomposition that we prove sound and complete with respect to the model (Lemma 2.2).

From now on, we use the notation $\{L \mid \varsigma\}$ for the record $\{(\ell = \mathbb{1}\vee\bot)_{\ell\in L} \mid \varsigma\}$. In technical developments, we often decompose a type $\{(\ell = \tau_\ell)_{\ell\in L} \mid \varsigma\}$ to the equivalent $\{(\ell = \tau_\ell)_{\ell\in L} \mid ..\} \wedge \{L \mid \varsigma\}$. It is particularly useful to separate row variables into their own atomic record, when $\varsigma \in \mathcal{V}$. We can even decompose further, the previous type being equivalent to $(\bigwedge_{\ell\in L} \{\ell = \tau_\ell \mid ..\}) \wedge \{L \mid \varsigma\}$.

Deciding subtyping amounts to deciding the emptiness of a type, since $t_1 \leq t_2$ is equivalent to $t_1 \wedge \neg t_2 \leq \mathbb{0}$. From [Frisch et al. 2008], we know that any type can be equivalently rewritten into a disjunctive normal form (DNF) of the form $\bigvee_{i\in I}(\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a \wedge \bigwedge_{\alpha\in V_i^p} \alpha \wedge \bigwedge_{\alpha\in V_i^n} \neg\alpha)$ where each intersection contains only atoms a's with the same type constructors: they are all basic types, or all arrows, or all records. Thus, checking emptiness of a type amounts to checking emptiness of all these intersections. Emptiness of these intersections cannot depend on the type variables (unless the same variable appears both positive and negated, in which case emptiness is straightforward), since their intersection is never empty. Thus, we just have to check emptiness for the $\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a$ parts. The procedure is already known for every intersection of atoms but polymorphic records. What is still missing is a formula that characterizes the emptiness of an intersection of the form $\bigwedge_{\mathsf{R}\in P} \mathsf{R} \wedge \bigwedge_{\mathsf{R}\in N} \neg\mathsf{R}$, that is, that decides whether $\bigwedge_{\mathsf{R}\in P} \mathsf{R} \leq \bigvee_{\mathsf{R}\in N} \mathsf{R}$ holds.

The lemma below gives a characterization of the emptiness of $\bigwedge_{r\in P} r \wedge \bigwedge_{r\in N} \neg r$. We prefer to state it on rows, as we will refer back to it in that way for tallying (Section 4.1). The corollary for records follows immediately by $t_1 \leq t_2 \iff \mathrm{row}(t_1) \leq \mathrm{row}(t_2)$ when $t_1, t_2 \leq \{\,|\,..\}$.

Lemma 2.1. *Let P and N be sets of atomic row types r each of domain $\mathcal{L}\smallsetminus L_r$. Let L be a finite set of labels such that $\bigcup_{r\in P\cup N} \mathrm{lab}(r) \subseteq L \subseteq \mathcal{L}\smallsetminus L_r$. Let $P_{\mathcal{V}} = \{r \in P \mid \mathrm{tail}(r) \in \mathcal{V}\}$ and likewise for $N_{\mathcal{V}}$. For every r, we define its default type $\mathrm{def}(r)$ as: $\mathrm{def}(r) = \bot$ if r is closed, and $\mathrm{def}(r) = \mathbb{1} \vee \bot$ otherwise. The relation $\bigwedge_{r\in P} r \leq \bigvee_{r\in N} r$ holds iff $\forall \iota : N \to L \cup \{\_\}$,*

$$\left(\exists \ell \in L.\ \bigwedge_{r\in P} r(\ell) \quad \leq \bigvee_{r\in\iota^{-1}(\ell)} r(\ell)\right) \tag{3}$$

$$\text{or } \left(\exists r_\circ \in \iota^{-1}(\_) \smallsetminus N_{\mathcal{V}}.(\bigwedge_{r\in P} \mathrm{def}(r) \leq \mathrm{def}(r_\circ))\right) \tag{4}$$

$$\text{or } \left(\exists r_\circ \in \iota^{-1}(\_) \cap N_{\mathcal{V}}.\exists r \in P_{\mathcal{V}}.\ \mathrm{tail}(r_\circ) = \mathrm{tail}(r)\right) \tag{5}$$

This lemma generalizes to rows and polymorphic record types the decomposition of monomorphic ones defined by Frisch [2004]. The main difference is the addition the third condition on line (5), that checks whether a row variable appears both in the positive and in the negative fragment.

Subtyping algorithm. Naively implementing the above subtyping formula requires backtracking. Indeed, for all map $\iota : N \to L \cup \{\_\}$, we have to check if subtyping holds on one of the labels $\ell \in L$. On that recursive call, since the types are coinductive, we need to assume that the type we are checking is empty. So, we are collecting a series of emptiness assumptions along the call stack. If later a contradiction arises, we need to backtrack to the point where the wrong assumption was introduced, to then take another branch, in our case, check subtyping for another ℓ. Following [Frisch 2004, Chapter 7], we avoid backtracking by defining a function Φ to compute subtyping more efficiently.

Computing subtyping on a record type amounts to give a decision procedure for the emptiness of $t = \bigwedge_{\mathsf{R}\in P} \mathsf{R} \wedge \bigwedge_{\mathsf{R}\in N} \neg\mathsf{R}$. Before applying function Φ, we preprocess t by normalizing the positive side of the type to isolate row variables. We choose $L = \bigcup_{r\in P\cup N} \mathrm{lab}(r)$. Our starting type t is equivalent to the following intermediate one:

$$\{(\ell = \textstyle\bigwedge_{\mathsf{R}\in P} \mathsf{R}(\ell))_{\ell\in L}\,|\,..\} \wedge \displaystyle\bigwedge_{\mathsf{R}\in P} \{\,\mathrm{lab}(\mathsf{R})\,|\,\mathrm{tail}(\mathsf{R})\} \wedge \bigwedge_{\mathsf{R}\in N} \neg\mathsf{R} \tag{6}$$

In this type, we merge the field types in L into a single atomic record type, and group the tails of positive records in a separate intersection. Next, we are going to rewrite the type in (6) by splitting the middle intersection $\bigwedge_{\mathsf{R}\in P} \{\,\mathrm{lab}(\mathsf{R})\,|\,\mathrm{tail}(\mathsf{R})\}$ in two: an intersection with all the record atoms whose tail is a row variable, and all the others that we will merge with the leftmost record in (6). For that, let us define $\varsigma_\circ$ to represent the intersection of the tails of the records in P whenever this tail is either ϵ or $..$, that is, $\varsigma_\circ = \epsilon$ if there is $\mathsf{R} \in P$ such that $\mathrm{tail}(\mathsf{R}) = \epsilon$, and $\varsigma_\circ = ..$ otherwise. If we take all the records in $\bigwedge_{\mathsf{R}\in P} \{\,\mathrm{lab}(\mathsf{R})\,|\,\mathrm{tail}(\mathsf{R})\}$ whose tail is not a row variable and intersect them with the leftmost record in (6), then we obtain the record type $\mathsf{R}_\circ = \{(\ell = \bigwedge_{\mathsf{R}\in P} \mathsf{R}(\ell))_{\ell\in L}\,|\,\varsigma_\circ\}$. Notice that $\mathsf{R}_\circ$ is a monomorphic record type. For the remaining records in the middle intersection, let us denote by $V_p = \{\rho \mid \exists \mathsf{R} \in P.\ \mathrm{tail}(\mathsf{R}) = \rho\}$ the set of all top-level type variables occurring in P. The intersection of atoms in (6) is then equivalent to the following type, which is the one for which we decide emptiness:

$$\mathsf{R}_\circ \wedge \bigwedge_{\rho\in V_p} \{\mathcal{L}\smallsetminus \mathrm{dom}(\rho)\,|\,\rho\} \wedge \bigwedge_{\mathsf{R}\in N} \neg\mathsf{R} \tag{7}$$

The key ingredient for our subtyping algorithm is the function $\Phi(\mathsf{R}_\circ, V_p, N)$, where $\mathsf{R}_\circ \not\leq \mathbb{0}$:

$$\begin{array}{rcl}\Phi(\mathsf{R}_\circ, V_p, \emptyset) & := & \text{false} \\ \Phi(\mathsf{R}_\circ, V_p, N \cup \{\mathsf{R}\}) & := & \text{if } (\mathrm{tail}(\mathsf{R}) = \mathbf{..} \text{ or } \mathrm{tail}(\mathsf{R}) = \mathrm{tail}(\mathsf{R}_\circ) \text{ or } \mathrm{tail}(\mathsf{R}) \in V_P) \text{ then} \\ & & \quad \forall \ell \in \mathrm{lab}(\mathsf{R}_\circ).\ (\mathsf{R}_\circ(\ell) \leq \mathsf{R}(\ell) \text{ or } \Phi(\mathsf{R}_\circ \wedge \{\ell : \neg\mathsf{R}(\ell) \mid \mathbf{..}\}, V_p, N)) \\ & & \text{else } \Phi(\mathsf{R}_\circ, V_p, N)\end{array}$$

The function must decide whether $\mathsf{R}_\circ \wedge \bigwedge_{\rho \in V_p} \{\mathcal{L} \setminus \mathrm{dom}(\rho) \mid \rho\} \leq \bigvee_{\mathsf{R} \in N} \mathsf{R}$, so it picks an $\mathsf{R} \in N$ and generates the conditions to test the containment. The first clause of the definition states that if we already examined all $\mathsf{R} \in N$, then subtyping does not hold, since $\mathsf{R}_\circ \not\leq \mathbb{0}$ and so its intersection with some row variables is also non empty. If $\mathsf{R}_\circ$ is open and R is closed, or if R is polymorphic, but its row variable is not one in V_P, then the containment cannot come from this particular R, that is discarded. This corresponds to the else branch of second clause. Otherwise, we are in the case in which either $\mathsf{R}_\circ$ is closed, or we are comparing two records types with a common row variable, or R is open. In these cases we compare $\mathsf{R}_\circ$ and R component-wise and for each ℓ we check that either $\mathsf{R}_\circ(\ell) \leq \mathsf{R}(\ell)$ or that the part that is in excess in $\mathsf{R}_\circ(\ell)$ is contained in the records remaining in N.[7]

If we compare this function to its monomorphic version found in [Castagna 2023], we can see that the only addition is the third test ($\mathrm{tail}(\mathsf{R}) \in V_p$), where V_p is constant in the function. This means that the complexity of this function and of its monomorphic version are the same.

Lemma 2.2 (Soundness and completeness of Φ). *Let $\mathsf{R}_\circ$ be an open or closed record type, $V_p \subset \mathcal{V}_r$ and N a set of atomic record types. Then,*

$$\mathsf{R}_\circ \wedge \bigwedge_{\rho \in V_p} \{\mathcal{L} \setminus \mathrm{dom}(\rho) \mid \rho\} \leq \bigvee_{\mathsf{R} \in N} \mathsf{R} \iff \mathsf{R}_\circ \leq \mathbb{0} \text{ or } \Phi(\mathsf{R}_\circ, V_p, N).$$

Proposition 2.3. *The subtyping algorithm terminates. As a corollary, subtyping is decidable.*

2.4 Substitutions

The upcoming descriptions of the type systems and of the inference algorithm rely on type, row, and field substitutions.

Definition 2.7. *Substitutions, ranged over by σ, are total mappings from variables of kind κ to expressions of kind κ (i.e., type variables to types, field variables to field types, and row variables of domain L to rows of domain L) that is the identity everywhere except on a finite set of variables. This set is called the* domain *of the substitution σ and is defined as* $\mathrm{dom}(\sigma) = \{\alpha \mid \sigma(\alpha) \neq \alpha\} \cup \{\theta \mid \sigma(\theta) \neq \theta\} \cup \{\rho \mid \sigma(\rho) \neq \langle\mid\rho\rangle^{\mathcal{L}\setminus\mathrm{dom}(\rho)}\}$.

The application of a substitution σ to a term T is denoted by $T\sigma$. Notice that the application is defined both on field types and on rows, the latter being useful only for tallying. The application of a substitution satisfies the following equalities.

$$\begin{array}{cc}\mathbb{0}\sigma = \mathbb{0} & b\sigma = b \\ (t_1 \to t_2)\sigma = t_1\sigma \to t_2\sigma & (T_1 \vee T_2)\sigma = T_1\sigma \vee T_2\sigma \\ (\neg T)\sigma = \neg(T\sigma) & \bot\sigma = \bot \\ \alpha\sigma = \sigma(\alpha) & \theta\sigma = \sigma(\theta)\end{array}$$

$$\{(\ell = \tau_\ell)_{\ell \in L} \mid \varsigma\}\sigma = \begin{cases}\{(\ell = \tau_\ell\sigma)_{\ell \in L} \mid \sigma(\rho)\}, & \text{if } \varsigma = \rho \\ \{(\ell = \tau_\ell\sigma)_{\ell \in L} \mid \varsigma\}, & \text{otherwise.}\end{cases} \tag{8}$$

[7]This formula generalizes the decomposition for tuples: e.g., if $s_\circ \times t_\circ \not\leq \mathbb{0}$ then $s_\circ \times t_\circ \leq s_1 \times t_1 \vee s_2 \times t_2 \iff (s_\circ \leq s_1$ or $(s_\circ \setminus s_1) \times t_1 \leq s_2 \times t_2)$ and $(t_\circ \leq t_1$ or $s_1 \times (t_\circ \setminus t_1) \leq s_2 \times t_2)$: see [Castagna 2023, Appendix D] for a longer explanation.

Where $\{(\ell = \tau_\ell)_{\ell\in L} \,|\, r\} \overset{\text{def}}{=} \{(\ell = \tau_\ell)_{\ell\in L} \,|\, ..\} \wedge \{L \,|\, r\}$ and:

$$\{L \,|\, \langle(\ell = \tau_\ell)_{\ell\in L'} \,|\, \varsigma\rangle^L\} \overset{\text{def}}{=} \{L, (\ell = \tau_\ell)_{\ell\in L'} \,|\, \varsigma\} \quad \{L \,|\, r_1 \vee r_2\} \overset{\text{def}}{=} \{L \,|\, r_1\} \vee \{L \,|\, r_2\} \quad \{L \,|\, \neg r\} \overset{\text{def}}{=} \neg\{L \,|\, r\}$$

The equalities above are standard, apart from the one in (8) which needs a definition for the notation $\{(\ell = \tau_\ell\sigma)_{\ell\in L} \,|\, \sigma(\rho)\}$, since $\sigma(\rho)$ is a row rather than a tail. The definition is given right after (8) and simply states that $\{(\ell = \tau_\ell\sigma)_{\ell\in L} \,|\, \sigma(\rho)\}$ stands for the record type obtained by recursively decomposing the Boolean combinations of the rows in $\sigma(\rho)$, until we arrive to single rows that are expanded in the record type (recall that rows are inductively defined). Substitution for rows is defined in the same way as for records (just change the delimiting brackets). We give several examples of applications of (row) substitutions in Section 4.1 when discussing constraints.

As expected, if $\text{dom}(\sigma) = \emptyset$, then $T\sigma = T$. If $\sigma(\rho) \leq \mathbb{0}$ and $\text{tail}(\mathsf{R}) = \rho$, then $\mathsf{R}\sigma \leq \mathbb{0}$. As we can show thanks to the parametric interpretation of types, substitution preserves subtyping.

Proposition 2.4. *If $t_1 \leq t_2$, then $t_1\sigma \leq t_2\sigma$ for any row substitution σ.*

3 LANGUAGE

We define the syntax, static and dynamic semantics of a record calculus that we prove to be type sound (Section 3.1) and define a sound and complete typing algorithm for it (Section 3.2).

3.1 Syntax and Semantics

$$\begin{array}{lrcl}
\textbf{Expressions} & e & ::= & c \mid x \mid ee \mid \lambda^{\wedge_{i\in I}(t_i\to s_i)}x.e \mid \{\} \mid \{e \texttt{ with } \ell = e\} \mid e.\ell \mid e\backslash\ell \\
\textbf{Values} & v & ::= & c \mid \lambda^{\wedge_{i\in I}(t_i\to s_i)}x.e \mid \{\} \mid \{v \texttt{ with } \ell = v\} \\
\textbf{Evaluation contexts} & E & ::= & [\,] \mid Ee \mid vE \mid \{e \texttt{ with } \ell = E\} \mid \{E \texttt{ with } \ell = v\} \mid E.\ell \mid E\backslash\ell
\end{array}$$

The syntax above describes a functional language with constants, functions, and records with field selection ($e.\ell$), addition ($\{e \texttt{ with } \ell = e\}$), and deletion ($e\backslash\ell$). As customary in semantic subtyping, λ-abstractions are annotated by their type, which is an intersection of arrow types. We use $\{\ell_1 = e_1, \ldots, \ell_n = e_n\}$ as syntactic sugar for $\{\ldots\{\{\} \texttt{ with } \ell_1 = e_1\}\ldots \texttt{ with } \ell_n = e_n\}$.

The semantics of the language is given by the call-by-value weak reduction defined below:

$$\begin{array}{rcl}
[\mathsf{R}_{\text{app}}] & (\lambda^t x.e)v \rightsquigarrow e[v/x] & \\
[\mathsf{R}^{=}_{\text{sel}}] & \{v \texttt{ with } \ell = v'\}.\ell \rightsquigarrow v' & \\
[\mathsf{R}^{\neq}_{\text{sel}}] & \{v \texttt{ with } \ell' = v'\}.\ell \rightsquigarrow v.\ell & \text{if } \ell \neq \ell' \\
[\mathsf{R}^{=}_{\text{del}}] & \{v \texttt{ with } \ell = v'\}\backslash\ell \rightsquigarrow v\backslash\ell & \\
[\mathsf{R}^{\neq}_{\text{del}}] & \{v \texttt{ with } \ell' = v'\}\backslash\ell \rightsquigarrow \{v\backslash\ell \texttt{ with } \ell' = v'\} & \text{if } \ell \neq \ell' \\
[\mathsf{R}_{\text{emp}}] & \{\}\backslash\ell \rightsquigarrow \{\} & \\
[\mathsf{R}_{\text{ctx}}] & E[e] \rightsquigarrow E[e'] & \text{if } e \rightsquigarrow e'
\end{array}$$

where $e[v/x]$ is the term obtained by standard capture-avoiding substitution of v for x in e, defined modulo α-equivalence. Notice that the deletion of a label ℓ is defined for the empty record $\{\}$ but selection is not: selection requires the presence of the field ℓ while deletion does not.

The terms of the language are typed by the declarative type system in Fig. 1, whose judgments are of the form $\Delta \mid \Gamma \vdash_{\mathcal{D}} e : t$, where $\Delta \subseteq \mathcal{P}_{\text{fin}}(\mathcal{V}_t \cup \mathcal{V}_f \cup \mathcal{V}_r)$ is a set of monomorphic variables (i.e., variables that cannot be instantiated) and Γ is a type environment mapping term variables into types. The rules for the functional part are standard: they are a simplified version of those by Castagna et al. [2014] where we do not track the relabeling of type annotations (these are only necessary in the presence of type-cases that can discriminate on functions of different types: see Castagna et al. [2014, Section 2]).

$$\text{(Const)}\ \frac{}{\Delta \mid \Gamma \vdash_{\mathcal{D}} c : \boldsymbol{b}_c} \qquad \text{(Var)}\ \frac{}{\Delta \mid \Gamma \vdash_{\mathcal{D}} x : \Gamma(x)}\ x \in \mathrm{dom}(\Gamma)$$

$$\text{(Abs)}\ \frac{(\Delta \cup \Delta' \mid \Gamma, x : t_i \vdash_{\mathcal{D}} e : s_i)_{i\in I}}{\Delta \mid \Gamma \vdash_{\mathcal{D}} \lambda^{\bigwedge_{i\in I}(t_i \to s_i)} x.e : \bigwedge_{i\in I}(t_i \to s_i)}\ \Delta' = \mathrm{var}(\textstyle\bigwedge_{i\in I} t_i \to s_i)$$

$$\text{(App)}\ \frac{\Delta \mid \Gamma \vdash_{\mathcal{D}} e_1 : t_1 \to t_2 \qquad \Delta \mid \Gamma \vdash_{\mathcal{D}} e_2 : t_1}{\Delta \mid \Gamma \vdash_{\mathcal{D}} e_1 e_2 : t_2} \qquad \text{(Emp)}\ \frac{}{\Delta \mid \Gamma \vdash_{\mathcal{D}} \{\} : \{\!\mid \epsilon\}}$$

$$\text{(Ext)}\ \frac{\Delta \mid \Gamma \vdash_{\mathcal{D}} e : t \leq \{\ell = \bot \mid ..\} \qquad \Delta \mid \Gamma \vdash_{\mathcal{D}} e' : t'}{\Delta \mid \Gamma \vdash_{\mathcal{D}} \{e \ \mathtt{with}\ \ell = e'\} : \{\ell = t' \mid t \backslash \ell\}}\ \ell \# t \qquad \text{(Del)}\ \frac{\Delta \mid \Gamma \vdash_{\mathcal{D}} e : t \leq \{\mid ..\}}{\Delta \mid \Gamma \vdash_{\mathcal{D}} e \backslash \ell : \{\ell = \bot \mid t \backslash \ell\}}\ \ell \# t$$

$$\text{(Sel)}\ \frac{\Delta \mid \Gamma \vdash_{\mathcal{D}} e : \{\ell = t \mid ..\}}{\Delta \mid \Gamma \vdash_{\mathcal{D}} e.\ell : t} \qquad \text{(Inter)}\ \frac{\Delta \mid \Gamma \vdash_{\mathcal{D}} e : t_1 \qquad \Delta \mid \Gamma \vdash_{\mathcal{D}} e : t_2}{\Delta \mid \Gamma \vdash_{\mathcal{D}} e : t_1 \wedge t_2}$$

$$\text{(Sub)}\ \frac{\Delta \mid \Gamma \vdash_{\mathcal{D}} e : t' \leq t}{\Delta \mid \Gamma \vdash_{\mathcal{D}} e : t} \qquad \text{(Inst)}\ \frac{\Delta \mid \Gamma \vdash_{\mathcal{D}} e : t}{\Delta \mid \Gamma \vdash_{\mathcal{D}} e : t\sigma}\ \mathrm{dom}(\sigma) \cap \Delta = \emptyset$$

Fig. 1. Declarative type system

Constants are typed by a given function $\boldsymbol{b}$ which maps each constant to its basic type (rule (const)).[8] Rule (Abs) checks that a function has all the types declared in its annotation, by checking the body of the function under suitable environments in which all the variables in the annotation are added to the set of monomorphic variables. The rules for intersection introduction (rule (Inter)) and subsumption (rule (Sub)) are standard (we use the notation $\Delta \mid \Gamma \vdash_{\mathcal{D}} e : t \leq t'$ in the premises of a rule, to indicate that the rule has premise $\Delta \mid \Gamma \vdash_{\mathcal{D}} e : t$ and side condition $t \leq t'$). The instantiation rule (Inst), does not instantiate monomorphic variables in Δ, since this would be unsound.

The new rules in this system are those for record expressions and their operations. The empty record value has the closed empty record type (rule (Emp)). Rule (Sel) states that selection is typable only if the selected field is present, in which case its type is given to the select expression. Rule (Ext) is a strict extension of an expression e of type t by the expression e' on label ℓ, only if the field ℓ is undefined in e, that is, the type of e is a subtype of $\{\ell = \bot \mid ..\}$. Rule (Del) states that we can delete a field ℓ from a record expression e provided that the condition $\ell \# t$ is satisfied. This condition is that the label ℓ is not in the domain of any row variable ρ or type variable α that appears at top-level in the record type t: formally, $\ell \# t \overset{\text{def}}{\iff} \forall \rho \in \mathrm{tlv}(\mathrm{row}(t)), \ell \notin \mathrm{dom}(\rho)$ and $\mathrm{tlv}(t) = \emptyset$, where $\mathrm{tlv}(T)$ returns the top-level variables of a term (i.e., any type variable under only type connectives if T is a type, and any row variable under only type connectives and a single record or row constructor if T is a row: see Definition B.1).

The types of the expressions typed by (Ext) and (Del) are computed in similar way. First, we compute the operator $t \backslash \ell$ which returns $\mathrm{row}(t)$ truncated by the field of label ℓ and whose formal definition we give below. If $t \backslash \ell$ is defined (otherwise the expression are not well typed), then we put back the field of label ℓ with the desired field type (t' or $\bot$) using to the operation $\{(\ell = \tau_\ell)_{\ell \in L} \mid r\}$ we defined in Section 2.4 for substitutions (cf. Eq. (8)). As an aside, the side condition $\ell \# t$ of (Del) can always be satisfied by subsuming the type of e to a type t'' obtained from t by replacing with "**..**" every top-level row variable whose domain contains ℓ. But then, the type of the expression would be computed using $t'' \backslash \ell$ and, thus, would not contain any critical row variable.[9]

[8] The functions $\boldsymbol{b}$ and $\mathbb{B}$ used in Definitions 2.5 and A.5 must satisfy $c \in \mathbb{B}(\boldsymbol{b}_c)$ for all $c \in C$.

[9] In practice, this means that we can give the function `def add_delete(x), do: Map.delete(%{x | a: x.a + 1}, :b)` the type `%{a: integer(), f} -> %{a: integer(), :b => none(), ...} when f: fields()` but not the type

$$(\textsc{Abs})\ \frac{\left(\Delta \cup \Delta' \mid \Gamma, x : t_i \vdash_{\mathcal{A}} e : s_i' \sqsubseteq_{\Delta\cup\Delta'} s_i\right)_{i\in I}}{\Delta \mid \Gamma \vdash_{\mathcal{A}} \lambda^{\bigwedge_{i\in I}(t_i \to s_i)} x.e : \bigwedge_{i\in I}(t_i \to s_i)}\ \Delta' = \mathrm{var}(\textstyle\bigwedge_{i\in I} t_i \to s_i)$$

$$(\textsc{App})\ \frac{\Delta \mid \Gamma \vdash_{\mathcal{A}} e_1 : t_1 \qquad \Delta \mid \Gamma \vdash_{\mathcal{A}} e_2 : t_2}{\Delta \mid \Gamma \vdash_{\mathcal{A}} e_1 e_2 : u}\ u \in t_1 \bullet_\Delta t_2$$

$$(\textsc{Ext})\ \frac{\Delta \mid \Gamma \vdash_{\mathcal{A}} e : t \qquad \Delta \mid \Gamma \vdash_{\mathcal{A}} e' : t'}{\Delta \mid \Gamma \vdash_{\mathcal{A}} \{e \ \mathtt{with}\ \ell = e'\} : \{\ell = t' \,|\, r\}}\ r \in t \odot_\Delta^\perp \ell$$

$$(\textsc{Del})\ \frac{\Delta \mid \Gamma \vdash_{\mathcal{A}} e : t}{\Delta \mid \Gamma \vdash_{\mathcal{A}} e\backslash\ell : \{\ell = \bot \,|\, r\}}\ r \in t \odot_\Delta \ell \qquad (\textsc{Sel})\ \frac{\Delta \mid \Gamma \vdash_{\mathcal{A}} e : t}{\Delta \mid \Gamma \vdash_{\mathcal{A}} e.\ell : u}\ u \in \amalg_\Delta^\ell(t)$$

Fig. 2. Algorithmic type system

Finally, let us define the operator $t\backslash\ell$, where $t \leq \{\,|\,..\}$ and $\ell \# t$. The operator $t\backslash\ell$, which returns a row, is defined only if $t \simeq \bigvee_{i\in I} \bigwedge_{\mathsf{R}\in P_i} \mathsf{R} \wedge \bigwedge_{\mathsf{R}\in N_i} \neg\mathsf{R}$, that is, if t is a record type composed only of atoms and negation of atoms that are not intersected at top-level with any type variable.[10] In that case, we define $t\backslash\ell = \bigvee_{i\in I} \bigwedge_{\mathsf{R}\in P_i} \mathsf{R}\backslash\ell \wedge \bigwedge_{\mathsf{R}\in N_i} (\neg\mathsf{R})\backslash\ell$ where on atoms:

$$\begin{array}{rcll} \{\ell = \tau_\ell \,|\, r\}\backslash\ell & = & r & \\ (\neg\{\ell = \mathbb{1} \vee \bot \,|\, r\})\backslash\ell & = & \neg r & \\ (\neg\{\ell = \tau \,|\, r\})\backslash\ell & = & \langle\,|\,..\rangle^{\{\ell\}} & \text{if } \tau \neq \mathbb{1} \vee \bot \end{array}$$

The three definitions above cover all possible cases. On an atom R, since $\ell\#\mathsf{R}$, then R has the shape $\{\ell = \tau_\ell \,|\, r\}$, where r is an atomic row of domain $\mathcal{L}\smallsetminus\{\ell\}$ containing the other elements of the record. We intuitively want $\mathsf{R}\backslash\ell = r$. Given a negative atom R, $\ell\#\mathsf{R}$ also implies that $\neg\mathsf{R}$ is of the shape $\neg\{\ell = \tau_\ell \,|\, r\}$. If $\tau_\ell \neq \mathbb{1} \vee \bot$, then the type $\neg\mathsf{R}$ contains, among others, all row elements such that the field ℓ is not of type τ_ℓ (since $\tau_\ell \neq \mathbb{1} \vee \bot$, then there exists at least one such element), and every other field is of arbitrary value. Hence, the set obtained from removing the field ℓ from these values gives all quasi-constant functions on $\mathcal{L} \smallsetminus \{\ell\}$, without restriction, that is the set entailed by $\langle\,|\,..\rangle^{\{\ell\}}$. When $\tau_\ell = \mathbb{1} \vee \bot$, it is impossible to have a value of a type different from τ_ℓ, hence the constraints on the type must be on the other fields, so on $\neg r$.

The language satisfies the property of soundness. Its proof is routine (subject reduction and progress, using inversion and generation lemmas) and is given in Appendix B.

Theorem 3.1 (Type soundness). *Let e be a well-typed closed expression, that is, $\emptyset \mid \emptyset \vdash_{\mathcal{D}} e : t$ for some t. Then either e diverges or it reduces to a value of type t.*

3.2 Algorithmic type system

The system in Fig. 1 is not algorithmic: it is not syntax-directed and some of its rules are not analytic[11]. A sound and complete algorithmic system for the functional part of the system in Fig. 1

`%{a: integer(), f} -> %{a: integer(), :b => none(), f} when f: fields()` unless this type is considered syntactic sugar for `%{a: integer(), :b => term(), f} -> %{a: integer(), :b => none(), f} when f: fields()`. The row variable `f` cannot appear in the type of the result if `b` is in its domain.

[10] Requiring that the type of a record in which we add or remove a field is not intersected by a type variable is not restrictive in practice: intersections with top-level type variables are used in practice only to implement bounded polymorphism which, as argued in the introduction, cannot be used for extensions and deletions, these requiring instead the use of row variables.

[11] A rule is *analytic* (as opposed to *synthetic*) when the input (i.e., Γ and e) of the judgment at the conclusion is sufficient to determine the inputs of the judgments at the premises (cf. [Martin-Löf 1994; Types mailing list 2019]).

is given in Fig. 2 (from which we omit the rules that are the same as in Fig. 1). The system includes the algorithmic counterparts of the typing rules for record operations which, apart from (Emp), must be changed. In fact, for (Ext) and (Del) we could use in practice the same rules as in the declarative systems, but the system would not be complete. Indeed, in the declarative system if $x : \alpha$ and $\alpha \notin \Delta$, then $x.\ell : \mathsf{Int}$ can be deduced by instantiating α to $\{\ell = \mathsf{Int}\,|\,..\}$. For all record operations, the algorithmic system needs to perform a possible instantiation of the type of the record. We use operators in the side conditions of the record rules, which instantiate the type of the record to match the conditions in the declarative system:

$$\begin{aligned}
\amalg^{\ell}_{\Delta}(t) &= \{u \mid [\sigma_i]_{i\in I} \Vdash t \sqsubseteq_\Delta \{\ell = \mathbb{1}\,|\,..\} \text{ and } u = (\textstyle\bigwedge_{i\in I} t\sigma_i).\ell\}\\
t \odot_\Delta \ell &= \{r \mid [\sigma_i]_{i\in I} \Vdash t \sqsubseteq_\Delta \{\,|\,..\} \text{ and } \ell\# t \text{ and } r = (\textstyle\bigwedge_{i\in I} t\sigma_i)\backslash\ell\}\\
t \odot^{\perp}_\Delta \ell &= \{r \mid [\sigma_i]_{i\in I} \Vdash t \sqsubseteq_\Delta \{\ell = \perp\,|\,..\} \text{ and } \ell\# t \text{ and } r = (\textstyle\bigwedge_{i\in I} t\sigma_i)\backslash\ell\}
\end{aligned}$$

The interest of these side conditions is only theoretical (to satisfy completeness) since they cover only the case of record expressions returning polymorphic values, which never happens in practice. In practice, the sets in the side conditions are never computed, the rules (Ext) and (Del) of Fig. 1 are used, while for selection the rule used in practice has premises $\Delta \mid \Gamma \vdash_{\mathcal{A}} e : t$, side-condition $t \leq \{\ell = \mathbb{1}\,|\,..\}$, and conclusion $\Delta \mid \Gamma \vdash_{\mathcal{A}} e.\ell : t.\ell$: we loose completeness, but only in theory, and we gain efficiency (see Castagna et al. [2015, Appendix B.3] for a detailed discussion of this point).

In any case, selection uses a new type operator $t.\ell$ (theoretically, in its side condition to compute u, in practice in its conclusion). Since the algorithmic type system does not include a subsumption rule, we cannot assume that the type t deduced for the expression e in (Sel) will be a record type atom of the form required by the declarative system (i.e., $\{\ell = t\,|\,..\}$): in general, t will be a union of intersections of such atoms, type variables, and their negations. Thus the rule checks that t is a record type in which the field ℓ is surely defined, and delegates to the operator $t.\ell$ (defined below) the computation of the type of the result.

Definition 3.1 (Field Selection). *Let $t \leq \{\ell = \mathbb{1}\,|\,..\}$ be a DNF. We define the selection of the field ℓ of t as $(\bigvee_{i\in I} t_i).\ell \stackrel{\text{def}}{=} \bigvee_{i\in I} t_i.\ell$ and*

$$\Big(\bigwedge_{\mathsf{R}\in P}\mathsf{R} \wedge \bigwedge_{\mathsf{R}\in N}\neg\mathsf{R} \wedge \bigwedge_{\alpha\in V_p}\alpha \wedge \bigwedge_{\alpha\in V_n}\neg\alpha\Big) \stackrel{\text{def}}{=} \bigvee_{N'\subseteq N}\left(\bigwedge_{\mathsf{R}\in P}\mathsf{R}(\ell) \wedge \bigwedge_{\mathsf{R}\in N'}\neg\mathsf{R}(\ell)\right)$$

The condition $t \leq \{\ell = \mathbb{1}\,|\,..\}$ assures that that $t.\ell \leq \mathbb{1}$, so that selection always returns a type (and not a generic field type). The transformation applied to t to derive the definition above and the proof that $t.\ell$ is equivalent to $\min\{u \mid t \leq \{\ell = u\,|\,..\}\}$ are given in Appendix B.2. Once more, the presence of top-level intersections with type variables does not play any role in selection.

To finish explaining the algorithmic system, we need to introduce the notations $\Vdash s \sqsubseteq_\Delta t$ and $t \bullet_\Delta s$, whose definitions are taken verbatim from [Castagna et al. 2015]:

Definition 3.2 ([Castagna et al. 2015]). *Let s and t be two types and Δ a set of variables. We define the following relations:*

$$\begin{aligned}
[\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta t &\stackrel{\text{def}}{\iff} \textstyle\bigwedge_{i\in I} s\sigma_i \leq t \text{ and } \forall i \in I.\, \mathrm{dom}(\sigma_i) \cap \Delta = \emptyset\\
\Vdash s \sqsubseteq_\Delta t &\stackrel{\text{def}}{\iff} \exists[\sigma_i]_{i\in I} \textit{ such that } [\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta t
\end{aligned}$$

Definition 3.3 ([Castagna et al. 2015]). *Let s and t be two types and Δ a set of variables. We define $t \bullet_\Delta s$ as the set of types for which there exist two sets of type substitutions (for variables not in*

Δ) that make s compatible with the domain of t:

$$t \bullet_\Delta s \stackrel{\text{def}}{=} \left\{ u \;\middle|\; \begin{array}{l} [\sigma_j]_{j\in J} \Vdash t \sqsubseteq_\Delta \mathbb{0} \to \mathbb{1} \\ [\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta \mathbf{dom}(\bigwedge_{j\in J} t\sigma_j) \\ u = \bigwedge_{j\in J} t\sigma_j \cdot \bigwedge_{i\in I} s\sigma_i \end{array} \right\}$$

Where $t \cdot s \stackrel{\text{def}}{=} \min\{u \mid t \leq s \to u\}$. For an arrow type $t \leq \mathbb{0} \to \mathbb{1}$, we have $t \simeq \bigvee_{i\in I}(\bigwedge_{p\in P_i}(s_p \to t_p) \wedge \bigwedge_{n\in N_i} \neg(s_n \to t_n) \wedge \bigwedge_{\alpha\in V_i^p} \alpha \wedge \bigwedge_{\alpha\in V_i^n} \neg\alpha)$. Then, $\mathbf{dom}(t) = \bigwedge_{i\in I} \bigvee_{p\in P_i} s_p$ (see [Castagna et al. 2014]).

These two definitions are used in the rule (App), the key rule for the algorithmic system, which again is taken verbatim from [Castagna et al. 2015] where all details can be found. Essentially, (App) merges together intersection elimination (in this case the standard terminology is *expansion*), instantiation, and subsumption. For the application $e_1 e_2$ to be well typed, the type of the function must be a functional type (i.e., a subtype of $\mathbb{0} \to \mathbb{1}$, the type of all functions) whose domain is a supertype of the type of the argument. Therefore, the rule looks for two finite sets of type substitutions for the variables not in Δ, that make the type of the function subtype of $\mathbb{0} \to \mathbb{1}$ and the type of the argument subtype of the function's domain. This search is collapsed in the definition of $t_1 \bullet_\Delta t_2$. Concretely, this operation finds two sets of substitutions $[\sigma_j]_{j\in J}$ and $[\sigma_i]_{i\in I}$ such that (1) $\bigwedge_{j\in J} t_1\sigma_j \leq \mathbb{0} \to \mathbb{1}$ (this corresponds to the notation $t_1 \sqsubseteq_\Delta \mathbb{0} \to \mathbb{1}$ of Definition 3.2) and (2) $\bigwedge_{i\in I} t_2\sigma_i$ is a subtype of the domain of $\bigwedge_{j\in J} t_1\sigma_j$. It then returns all the types of the result of the application of such two types. See [Castagna et al. 2015] for details.

While the typing rules (Abs) and (App) themselves are not new, the process behind $s \sqsubseteq_\Delta t$ and $t \bullet_\Delta s$ are. The novelty is that we infer type substitutions that range not only over types, but also over rows and field types. In the next section, we describe how to adapt the existing algorithms to our framework.

As expected, the algorithmic type system is sound and complete with respect to the declarative one, as stated by the following theorems (proofs are given in Appendix B.2):

Theorem 3.2 (Soundness). *If $\Delta \mid \Gamma \vdash_{\mathcal{A}} e : t$, then $\Delta \mid \Gamma \vdash_{\mathcal{D}} e : t$.*

Theorem 3.3 (Completeness). *If $\Delta \mid \Gamma \vdash_{\mathcal{D}} e : t$, then there is s such that $\Delta \mid \Gamma \vdash_{\mathcal{A}} e : s$ and $s \sqsubseteq_\Delta t$.*

4 TALLYING

The algorithmic type system we have defined in the last section is parametric in the decision procedure $\sqsubseteq_\Delta$. While this problem has been tackled for type variables by Castagna et al. [2015], here we need to extend it to row and field variables.

The main part of the design of a procedure to decide $\sqsubseteq_\Delta$ is the *tallying* algorithm. Tallying is a unification problem with subtyping instead equality. Given an initial set of subtyping constraints, tallying looks for a substitution that satisfies these constraints. As explained in detail by Castagna et al. [2015, §3.2.2-3.2.3], deciding both $s \sqsubseteq_\Delta t$ and $t \bullet_\Delta s$ can be reduced to solving a sequence of tally problems that are generated by varying the cardinality of the set of substitutions we are looking for. For instance, looking for a set $[\sigma_i]_{i=1,2}$ such that $[\sigma_i]_{i=1,2} \Vdash s \sqsubseteq_\Delta t$, is equivalent to solving the tallying problem for $s_1 \wedge s_2 \leq t$ where each s_i is obtained from s by replacing all variables not in Δ by fresh ones. It then suffices to test all cardinalities of the set of substitutions, following a dove-tail order for the two sets sought for $t \bullet_\Delta s$.

We define the solving procedure for the type tallying of a constraint-set as an extension of the existing one for type variables. The procedure follows the same steps given by Castagna et al. [2015] (plus an additional one), namely: (1) normalization (Appendix C.4), (2) merging and saturation (Appendix C.5), (3) harmonization (which is new and specific to row variables: Appendix C.6), (4) transformation of constraints into equations (Appendix C.7), and (5) creation of the substitution solutions (Appendix C.8). For space reasons, in what follows we detail only the most important step, that is, step (1) normalization. The other steps are more straightforward and, in the case of steps (2), (4), and (5), they are similar to the corresponding steps defined by Castagna et al. [2015]. Therefore, we just hint at them in Section 4.2, and move their definition to the appendix.

We begin with giving few definitions, starting with the definition for constraints. Although the typing rules need to solve the tallying problem on types, types will be decomposed in their subterms, thus generating constraints on field types and rows, too.

Definition 4.1 (Constraints). *A constraint* (T_1, c, T_2) *is a triple such that* $c \in \{\leq, \geq\}$ *and* $(T_1, T_2) \in (\mathcal{T}\times\mathcal{T}) \cup (\mathcal{T}_\perp\times\mathcal{T}_\perp) \cup (\mathcal{R}\times\mathcal{R})$. T_1 *and* T_2 *must be of the same kind which, in particular, implies* $\mathrm{dom}(T_1) = \mathrm{dom}(T_2)$ *if* T_1 *and* T_2 *are rows. Let* $\mathcal{C}$ *denote the set of all constraints. Given a constraint-set* $C \subseteq \mathcal{C}$*, the* set of variables *occurring in* C *is defined as* $\mathrm{var}(C) = \bigcup_{(T_1,c,T_2)\in C} \mathrm{var}(T_1) \cup \mathrm{var}(T_2)$.

The presence of subtyping, and in particular of the empty type, implies that to solve a single constraint-set C we need to generate several constraint sets, yielding different solutions (e.g., solving $\{s_1{\times}s_2 \leq t_1{\times}t_2\}$ generates three independent subproblems: $\{s_1{\leq}\mathbb{0}\}$, $\{s_2{\leq}\mathbb{0}\}$, and $\{s_1{\leq}t_1, s_2{\leq}t_2\}$). Thus, we consider sets $\mathcal{S}$ of constraint-sets, each set representing a possible solution. Given two such sets $\mathcal{S}_1, \mathcal{S}_2 \subseteq \mathcal{P}(\mathcal{C})$, we define their union as $\mathcal{S}_1 \sqcup \mathcal{S}_2 = \mathcal{S}_1 \cup \mathcal{S}_2$ and their intersection as $\mathcal{S}_1 \sqcap \mathcal{S}_2 = \{C_1 \cup C_2 \mid C_1 \in \mathcal{S}_1, C_2 \in \mathcal{S}_2\}$.

We create a solution by decomposing types into elementary constraints. For records and rows, it might be necessary to decompose the row variables across different fields. For instance, the record $\{\mid \rho_0\}$ might need to be decomposed over labels ℓ_1 and ℓ_2. For this, we define two constructors: $\rho.\ell$, that we treat as a field variable, and $\rho\backslash L$, that we treat as a row variable of domain $\mathrm{dom}(\rho)\smallsetminus L$. Using these constructors the row variable ρ_0 can be decomposed into $\rho_0.\ell_1$, $\rho_0.\ell_2$ and $\rho_0\backslash\{\ell_1, \ell_2\}$, so that $\{\mid \rho_0\}$ can be equivalently written $\{\ell_1 = \rho_0.\ell_1, \ell_2 = \rho_0.\ell_2 \mid \rho_0\backslash\{\ell_1, \ell_2\}\}$. Substitution is extended to $(\rho.\ell)\sigma = \tau$ if $\sigma(\rho) \simeq \langle \ell = \tau \mid r\rangle^{L'}$, and $(\rho\backslash L)\sigma = r$ if $\sigma(\rho) \simeq \langle (\ell = \tau_\ell)_{\ell\in L\cap\mathrm{dom}(\rho)} \mid r\rangle^{L'}$. Substitution is undefined if $\sigma(\rho)$ does not have the desired shape. For ρ_0, the tallying algorithm might give constraints over $\rho_0.\ell_1$, $\rho_0.\ell_2$ and $\rho_0\backslash\{\ell_1, \ell_2\}$, and we will expect a solution of the shape $\sigma(\rho_0) = \langle \ell_1 = \tau_1, \ell_2 = \tau_2 \mid r\rangle^{\emptyset}$, where τ_i has been obtained from the constraints on $\rho_0.\ell_i$, and r from the ones over $\rho_0\backslash\{\ell_1, \ell_2\}$.

In this section, the name ρ ranges over row variables and $\rho\backslash L$ constructors, the name θ ranges over field variables and $\rho.\ell$ constructors, and the name X ranges over all kinds of variables plus these new constructors. We consider $\rho\backslash L$ up to the equivalence generated by identifying $\rho \in \mathcal{V}_r$ with $\rho\backslash L$ for all $L \subseteq \mathcal{L}\smallsetminus \mathrm{dom}(\rho)$, and $(\rho\backslash L_1)\backslash L_2$ with $\rho\backslash(L_1 \cup L_2)$. With an abuse of notation we write ρ for the row $\langle\mid \rho\rangle^{\mathcal{L}\smallsetminus\mathrm{dom}(\rho)}$—in particular in rows constraints (e.g., (ρ, c, r))—when no confusion arises.

Definition 4.2 (Constraint solution). *Let* $C \subseteq \mathcal{C}$ *be a constraint-set. A solution to* C *is a substitution* σ *such that* $\forall(T_1, \leq, T_2) \in C. T_1\sigma \leq T_2\sigma$ *and* $\forall(T_1, \geq, T_2) \in C. T_1\sigma \geq T_2\sigma$ *hold. If* σ *is a solution to* C*, we write* $\sigma \Vdash C$*. In particular,* $T_1\sigma$ *and* $T_2\sigma$ *must be defined, that is, for all* $\rho \in \mathrm{dom}(\sigma)$*:*

- $\forall \ell \in \mathcal{L}.\ \rho.\ell \in \mathrm{vars}(C) \implies \exists\tau, r. \sigma(\rho) \simeq \langle \ell = \tau \mid r\rangle^{\mathcal{L}\smallsetminus\mathrm{dom}(\rho)}$*, and*
- $\forall L \subseteq \mathcal{L}.\ \rho\backslash L \in \mathrm{vars}(C) \implies \exists(\tau_\ell)_{\ell\in L}, r. \sigma(\rho) \simeq \langle (\ell = \tau_\ell)_{\ell\in L} \mid r\rangle^{\mathcal{L}\smallsetminus\mathrm{dom}(\rho)}$.

where $\mathrm{vars}(C)$ *is the set of all variables occurring in* C *(see Definition C.1):*

The tallying algorithm is parametric on a total order O (Definition C.2) which is used to ensure that the step (5) of tallying produces (contractive) types. A set of constraints is *well-ordered* if for all constraints of the shape (X, c, T) and for all $X' \in \mathrm{tlv}(T)$, $O(X) < O(X')$ holds.

4.1 Constraint normalization

The first step of tallying decomposes the initial constraints on types into a set of normalized constraint-sets $\mathcal{S}$. This one only contains sets $C \in \mathcal{S}$ where all constraints are of the shape (X, c, T), with X and T of the same kind. This step is implemented by a recursive function $\mathrm{norm}(t, M)$ that takes a type t as input and returns a set of normalized constraint-sets necessary for $t \leq \mathbb{0}$ to hold. The argument M is a set of visited types, that guarantees termination of recursion on infinitary types. A formal description of normalization as inference rules is given in Figs. 5 to 7 in Appendix C.

The general idea of normalization is the following. We first rewrite each constraint (T, c, T) into a set of constraints $\{(T_i, \leq, \mathbb{0})\}_{i \in I}$, where T_i is a conjunction of atoms (basic types, arrows, records), type and field variables (in the latter case, $\perp$ can also be present), or their negations. This first constraint-set is obtained by transforming the type into DNF and putting each summand of the outer union into a separate constraint. For each constraint $(T_i, \leq, \mathbb{0})$ we then isolate the smallest (w.r.t., the order O) top-level type variable or field variable X not in Δ, to obtain a constraint (X, c, T_i') that gives a lower (i.e., c is $\geq$ when X is negated in T_i) or an upper (i.e., c is $\leq$ otherwise) bound on that variable, where T_i' is obtained from T_i simply by removing X.

There may be no variable in T_i, or they all may be in Δ. A variable in Δ is monomorphic, cannot be instantiated and is treated as a constant. In that case, we erase monomorphic variables because they cannot help to satisfy the constraint (see case (Ntlv) in Lemma C.8). On basic types, we can directly see if the constraint holds. For arrow constructors, we use the subtyping formula, that decomposes the types into constraints on their subtypes. Until now, all these steps (apart from dealing with field variables) are similar to those of the existing tallying algorithm for type variables by Castagna et al. [2015]. For a conjunction of record atoms, the technique is more elaborated.

First, we start by defining the normalization of a conjunction of records t as the normalization of its underlying row so that we have an auxiliary procedure on rows.

$$\mathrm{norm}(t, M) = \mathrm{norm}_{\mathrm{row}}(\mathrm{row}(t), M \cup \{t\})$$

In the literature where rows are all atomic, rather than Boolean combinations, unification of a row variable with another row is often done component-wise by introducing a series of fresh type variables. In our setting, these variables would be named $\rho.\ell_1, \ldots, \rho.\ell_n$ and $\rho \backslash \{\ell_1, \ldots, \ell_n\}$.

Example 4.1. With component-wise unification, the constraint-set $\{(\langle \mathsf{log} = \mathsf{String} \,|\, \rho \rangle^{\emptyset}, \leq, \langle \mathsf{log} = \mathsf{String}, \mathsf{succ} = \mathsf{True}, \mathsf{val} = \mathbb{1} \,|\, \epsilon \rangle^{\emptyset})\}$ is normalized to $\{(\mathsf{String}, \leq, \mathsf{String}), (\rho.\mathsf{succ}, \leq, \mathsf{True}), (\rho.\mathsf{val}, \leq, \mathbb{1}), (\rho \backslash L, \leq, \langle \,|\, \epsilon \rangle^{L})\}$, where $L = \{\mathsf{log}, \mathsf{succ}, \mathsf{val}\}$. Putting the solutions for each parts together yields the solution $\sigma(\rho) = \langle \mathsf{succ} = \mathsf{True}, \mathsf{val} = \mathbb{1} \,|\, \epsilon \rangle^{\{\mathsf{log}\}}$.

As the example above shows, component-wise unification works well with atomic rows. However, it fails when considering Boolean combination of rows.

Example 4.2. Let $\mathsf{result} = \langle \mathsf{log} = \mathsf{String}, \mathsf{succ} = \mathsf{True}, \mathsf{val} = \mathbb{1} \,|\, \epsilon \rangle^{\emptyset} \vee \langle \mathsf{log} = \mathsf{String}, \mathsf{succ} = \mathsf{False}, \mathsf{val} = \perp \,|\, \epsilon \rangle^{\emptyset}$. Applying an argument of type result to a function of type $\langle \mathsf{log} = \mathsf{String} \,|\, \rho \rangle^{\emptyset} \to \langle \mathsf{log} = \mathsf{String} \,|\, \rho \rangle^{\emptyset}$ gives the following constraint: $(\langle \mathsf{log} = \mathsf{String} \,|\, \rho \rangle^{\emptyset}, \leq, \mathsf{result})$. A component-wise unification gives the constraint-set $\{(\mathsf{String}, \leq, \mathsf{String}), (\rho.\mathsf{succ}, \leq, \mathsf{Bool}), (\rho.\mathsf{val}, \leq, \mathbb{1} \vee \perp), (\rho \backslash L, \leq, \langle \,|\, \epsilon \rangle^{L})\}$, where $L = \{\mathsf{log}, \mathsf{succ}, \mathsf{val}\}$. This entails the solution $\sigma(\rho) = \langle \mathsf{succ} = \mathsf{Bool}, \mathsf{val} = \mathbb{1} \vee \perp \,|\, \epsilon \rangle^{\{\mathsf{log}\}}$, which is not the most precise one: since the type of the function is essentially the type of an identity function, we would have expected the application to have type result.

In some cases, component-wise unification is not even sound.

Example 4.3. From the constraint $(\langle \mathsf{val} = \mathbb{1} \vee \perp \,|\, \rho' \rangle^{\emptyset}, \leq, \mathsf{result})$, component-wise unification gives $\sigma(\rho') = \langle \mathsf{log} = \mathsf{String}, \mathsf{succ} = \mathsf{Bool} \,|\, \epsilon \rangle^{\{\mathsf{val}\}}$ as a solution, which does not verify the constraint (the type obtained as "solution" contains record values in which succ is True and val is undefined, which are not included in result).

To obtain a sound decomposition of rows, we can adapt the subtyping formula from Lemma 2.1. Given a constraint C on DNFs of rows, we consider the set L of all top-level labels in the DNF. The formula decomposes the constraints into independent constraints over the fields with labels in L and constraints over the rest of the rows. Since we rely on Lemma 2.1, we find no solution for the constraint in Example 4.3, which thus, correctly, cannot be satisfied.

Although this method yields a correct set of solutions, this set is far from complete. In fact, since it decomposes records over the set $L = \{\mathsf{log}, \mathsf{succ}, \mathsf{val}\}$ of all top-level labels, the solution found for the constraint of Example 4.2 is the same as the one given there, as it will decompose ρ unnecessarily into $\rho.\mathsf{succ}$, $\rho.\mathsf{val}$ and $\rho\backslash L$. To solve this, we give a more general decomposition formula, where the set L of labels over which we decompose can be as small as desired.

Example 4.4. From the same constraint as in Example 4.2: $(\langle \mathsf{log} = \mathsf{String} \,|\, \rho\rangle^{\emptyset}, \leq, \mathsf{result})$, we take $L = \{\mathsf{log}\}$ and obtain the constraint-set: $\{(\mathsf{String}, \leq, \mathsf{String}), (\rho, \leq, \langle \mathsf{succ} = \mathsf{True}, \mathsf{val} = \mathbb{1} \,|\, \epsilon\rangle^{\{\mathsf{log}\}} \vee \langle \mathsf{succ} = \mathsf{False}, \mathsf{val} = \bot \,|\, \epsilon\rangle^{\{\mathsf{log}\}})$. This entails the desired solution.

The general decomposition formula is given in Lemma C.4. The auxiliary function for normalization of constraints on rows makes use of two operators.

Definition 4.3. *Let $r = \langle (\ell = \tau_\ell)_{\ell \in L_1} \,|\, \varsigma\rangle^{L_2}$, Δ be a set of variables, $\ell \in \mathcal{L}\setminus L_2$ and $L \in \mathcal{P}_{\mathrm{fin}}(\mathcal{L})$.*

- *$r[\ell] \stackrel{\mathrm{def}}{=} \rho.\ell$ if $\varsigma = \rho \notin \Delta$ and $\ell \in \mathrm{dom}(\rho)$, and $r[\ell] \stackrel{\mathrm{def}}{=} r(\ell)$ otherwise.*
- *$\langle (\ell = \tau_\ell)_{\ell \in L_1} \,|\, \varsigma\rangle^{L_2} \backslash^{\Delta} L \stackrel{\mathrm{def}}{=} \langle (\ell = \tau_\ell)_{\ell \in L_1 \setminus L} \,|\, \varsigma'\rangle^{L_2 \cup L}$; where if $\varsigma = \rho$ and $\mathrm{dom}(\rho) \cap L \neq \emptyset$: $\varsigma' = ..$ if $\rho \in \Delta$ and $\varsigma' = \rho\backslash L$ otherwise; and $\varsigma' = \varsigma$ otherwise.*

Let us consider a row in DNF $r_0 = \bigwedge_{r \in P} r \wedge \bigwedge_{r \in N} \neg r$. Let ρ_0 be the smallest top-level variable of r_0 not in Δ and $L = \mathrm{dom}(r_0) \setminus \mathrm{dom}(\rho_0)$, or $L = \emptyset$ if there is no such variable. Let $S_\Delta = \{r \in S \mid \mathrm{tail}(r) = \rho \in \Delta \text{ and } \mathrm{dom}(\rho) \cap L \neq \emptyset\}$, for $S = P, N$. Normalization $\mathrm{norm}_{\mathrm{row}}(r_0, M)$ is defined as:

$$\prod_{\iota : N \to L \cup \{\_\}} \left(\bigsqcup_{\ell \in L} \mathrm{norm}_{\mathrm{fld}} \left(\bigwedge_{r \in P} r[\ell] \wedge \bigwedge_{r \in \iota^{-1}(\ell)} \neg r[\ell], M \right) \sqcup \prod_{N' \in \mathcal{N}} \left(\mathrm{norm}_{\mathrm{tl}} \left(\bigwedge_{r \in P} (r \backslash^{\Delta} L) \wedge \bigwedge_{r \in \iota^{-1}(\_) \setminus N' \in S} \neg (r \backslash^{\Delta} L), M \right) \right) \right)$$

where $\mathcal{N} = \{N' \subseteq \iota^{-1}(\_) \cap N_\Delta \mid \bigwedge_{r \in P_\Delta} \langle \mathrm{lab}(r) \,|\, \mathrm{tail}(r)\rangle \not\leq \bigvee_{r \in N'} \langle \mathrm{lab}(r) \,|\, \mathrm{tail}(r)\rangle\}$.

Here, the idea of the algorithm is the following. If there is no polymorphic top-level variable (these can only be row variables), we let $L = \emptyset$, which does not decompose the row, only calls the function $\mathrm{norm}_{\mathrm{tl}}(r_0, M)$ recursively. In that case, this function will decompose the record over the whole set of top-level labels using the formula for subtyping. We do not loose solutions doing so: due to the absence of top-level polymorphic variables, the emptiness of r_0 can only be satisfied by the emptiness of one of the components.

If there is a smallest polymorphic top-level variable ρ_0, we take $L = \mathrm{dom}(r_0) \setminus \mathrm{dom}(\rho_0)$. In other words, we take L to be the set of labels on the left of the variable ρ_0. In Example 4.4, we take L to be $\{\mathsf{log}\}$. In this way, we decompose the row on one side over the elements in L, which does not affect ρ_0. These elements are handled by the auxiliary function $\mathrm{norm}_{\mathrm{fld}}$, that normalizes constraints on fields in a similar way than is done on types. On the other side, we obtain constraints over ρ_0, where ρ_0 is in a row of the shape $\langle \,|\, \rho_0\rangle^{\mathcal{L} \setminus \mathrm{dom}(\rho_0)}$. Then, the recursive call to $\mathrm{norm}_{\mathrm{tl}}$ singles out the variable ρ_0 in order to obtain an upper or lower bound for ρ_0, as we do for type and field variables. The definitions of $\mathrm{norm}_{\mathrm{fld}}$ and $\mathrm{norm}_{\mathrm{tl}}$ are in Appendix C.3.

While ρ_0 is not affected by the decomposition thanks to our choice of L, there can be polymorphic top-level variables ρ' such that $\mathrm{dom}(\rho') \cap L \neq \emptyset$. This is why we have introduced the two new operators $r[\ell]$ and $r\backslash^{\Delta} L$.

By correction of the decomposition formula, the normalization function is sound. However, the procedure is still not complete.

Example 4.5 (Incompleteness). Let $O(\rho_1) < O(\rho_2)$. From the constraint-set $\{(\langle \mathsf{log} = \mathsf{String} \,|\, \rho_2 \rangle^{\emptyset} \to \langle \mathsf{log} = \mathsf{String} \,|\, \rho_2 \rangle^{\emptyset}, \geq, \mathsf{result} \to \mathsf{result}), (\langle \mathsf{log} = \mathsf{String} \,|\, \rho_2 \rangle^{\emptyset} \to \langle \,|\, .. \rangle^{\emptyset}, \geq, \langle \mathsf{succ} = \mathbb{1} \vee \bot, \mathsf{val} = \mathbb{1} \vee \bot \,|\, \rho_1 \rangle^{\emptyset} \to \langle \mathsf{succ} = \mathbb{1} \vee \bot, \mathsf{val} = \mathbb{1} \vee \bot \,|\, \rho_1 \rangle^{\emptyset})\}$, by a decomposition over $L = \{\mathsf{log}\}$ in the first constraint and $L = \{\mathsf{succ}, \mathsf{val}\}$ in the second constraint, we derive the constraint-set (omitting trivial constraints) $\{(\mathsf{String}, \leq, \rho_1.\mathsf{log}), (\rho_2 \backslash \{\mathsf{succ}, \mathsf{val}\}, \leq, \rho_1 \backslash \{\mathsf{log}\}), (\rho_2, \leq, \mathsf{result}), (\rho_2, \geq, \mathsf{result})\}$. As we describe in Section 4.2 below, a further step of the tallying algorithm harmonizes the decomposition of the row variables across all constraints. In particular, it decomposes ρ_2 over $\{\mathsf{succ}, \mathsf{val}\}$ in the constraints $(\rho_2, \leq, \mathsf{result})$ and $(\rho_2, \geq, \mathsf{result})$. Since when decomposing ρ_2 in this way, no solution might apply to both of these constraints (ρ_2 needs to be instantiated exactly to the union type result), tallying fails. The solution mapping ρ_2 to result is not found.

4.2 Other Steps

Constraint merging and saturation. After normalization a constraint set may have for the same variable X different constraints of the form $T_i \leq X$ or $X \leq T'_j$ for $i \in I$ and $j \in J$: we replace them by two constraints $\bigvee_{i \in I} T_i \leq X$ and $X \leq \bigwedge_{j \in j} T'_j$, add $\bigvee_{i \in I} T_i \leq \bigwedge_{j \in j} T'_j$ to (i.e., saturate) the constraint set, and normalize again.

Harmonization. As mentioned in Example 4.5, harmonization of constraint-set rewrites the constraints of the shape $(\rho \backslash L_1, c, r)$ into $(\rho \backslash L, c, r)$ and feeds this constraint to normalization again. This happens when $L_1 \subsetneq L$ and L is the set of labels appearing within terms $\rho'.\ell$ or $\rho' \backslash L_2$. The goal is to have a homogeneous domain, where for a given row variable ρ all occurrences of $\rho \backslash L$ are defined with the same L.

Equations generation. At this point, for each type and field variable X there is a unique (double) constraint of the form $T \leq X \leq T'$, that we transform into the equation $X = (T \vee X') \wedge T'$ with X' fresh. For a row variable ρ, there is a set of labels L and constraints of the form $\tau_1 \leq \rho.\ell \leq \tau_2$ for each $\ell \in L$, and a constraint $r_1 \leq \rho \backslash L \leq r_2$. We define the terms $\tau_\ell = (\tau_1 \vee \theta^{\ell}_{\rho}) \wedge \tau_2$ and $r = (r_1 \vee \rho') \wedge r_2)$, with fresh variables θ^{ℓ}_{ρ} and ρ'. We finally create an equation $\rho = \langle (\ell = \tau_\ell)_{\ell \in L} \,|\, r \rangle^{\mathcal{L} \setminus \mathrm{dom}(\rho)}$.

Solution. We solve the equations in the order O of the variables on the left-hand side, by collecting the equation $X = T$ and replacing in all other equations X by $\mu X'.(T\{X'/X\})$ with X' fresh. Thanks to the order O, the type $\mu X'.(T\{X'/X\})$ is contractive (cf. Proposition C.22).

4.3 Properties of the algorithm

We call $\mathrm{Sol}_\Delta(C)$ the solving procedure for the type tallying of C. We write $\mathrm{Sol}_\Delta(C) \rightsquigarrow \Theta$ if $\mathrm{Sol}_\Delta(C)$ terminates with Θ, and we call Θ the solution of the type tallying problem for C.

Theorem 4.6 (Soundness). *Let C be a constraint-set. If $\mathrm{Sol}_\Delta(C) \rightsquigarrow \Theta$, then for all $\sigma \in \Theta$, $\sigma \Vdash C$.*

Theorem 4.7 (Termination). *Let C be a constraint-set. Then $\mathrm{Sol}_\Delta(C)$ terminates.*

Proposition 4.8. *Let C be a constraint-set and $\mathrm{Sol}_\Delta(C) \rightsquigarrow \Theta$. Then (1) Θ is finite and (2) for all $\sigma \in \Theta$ and for all $X \in \mathrm{dom}(\sigma)$, the types in $\sigma(X)$ are contractive.*

5 RELATED WORK

Row polymorphism. Our formalization of records as quasi-constant total functions and the inclusion of row polymorphism are directly inspired from the formalism of Rémy [1989, 1994]. Remy's work contains neither set-theoretic types nor subtyping, and therefore commutation of fields is obtained by structural equations. Unlike our case, the types in [Rémy 1994] are not recursive. An extension of this system to recursive types is given by Rémy and Vouillon [1998] when describing a type system for objects.

Before Rémy, Wand [1987] introduced row polymorphism to type object inheritance. His type inference algorithm, corrected in [Wand 1991], considers free record extension (i.e., right priority record concatenation) but lacks principal solutions. Instead, it deduces a finite set of solutions of which all types of the term under consideration are instances, as we also do. However, unlike us, Wand's type grammar lacks intersection types, so he is unable to merge the multiple solutions into a single type, as we instead do. An earlier attempt of our work considered free record extension, as it is present both in CDuce and in its generalization by Castagna [2023], but this made the theory much more involved, in particular tallying. It would be worthwhile to study this possibility again, now that the theory with strict extension is precisely laid down.

(Syntactic) subtyping is present in the work of Cardelli and Mitchell [1991]. Operations on records (deletion, selection, extension) are directly defined within the syntax of types. In our system instead, we compute these operations on the types during typing. It is thus currently impossible to postpone the extension or deletion of a field with label ℓ that is affected by a row variable, until the point where the row variable will be instantiated and, in that case, we must resort to an approximation. Cardelli and Mitchell must however define syntactic equivalence relations on operators and fields, and their system lacks principal typing, as well.

Row polymorphism and extensible records are implemented using predicate on types by Harper and Pierce [1991], later by Gaster and Jones [1996] under the name qualified types. In the latter, positive information is given in the type and negative information (absent field) in the predicate. Morris and McKinna [2019] use qualified type with uninterpreted predicates for concatenation and membership of fields. Their system can be instantiated to most of the standard approaches of the literature. We aimed to minimize changes to the type syntax of CDuce and Elixir, opting to refrain from incorporating qualified types.

A convenient way to define extensible records is with scoped labels [Leijen 2005], where labels may appear several times in a record, the most recent occurrence shadowing earlier ones. Paszke and Xie [2023] recently extended the formalism to deal with first-class labels, first-class rows and concatenation. This gives a simple formulation of types, yet too syntactic for our semantic approach to typing and subtyping, and our desire to interpret records as quasi-constant functions.

Variants, the dual of records, are studied in a semantic subtyping setting by Castagna et al. [2016], where they show that adopting full-fledged union (and intersection and negation) types as well as let-polymorphism gives a more intuitive and expressive theory of polymorphic variants, which is why here we focused only on polymorphism for records.

Presence polymorphism. In [Rémy 1994] record fields can be either present or absent, and this is indicated by an annotation. Rémy [1994] shows how to add presence polymorphism over these annotations, to have records parametric in the absence or presence of some fields. In our work, field variables that range over types augmented with $\bot$ give a natural notion of presence polymorphism. They can be instantiated with $\bot$ for an absent field, a type t for a mandatory field, but also by $t \vee \bot$ for an optional one. To our knowledge, our work is the first to allow presence polymorphism over optional fields: fields on polymorphic records in existing literature can just be either present or absent. For a comparison between presence polymorphism of [Rémy 1994] and ours, see Appendix A.1.

Relation between different kinds of polymorphism. Our type system features three kinds of parametric polymorphism: on types, rows, and field types. It also features subtype polymorphism and ad-hoc polymorphism via intersection and union types. The question naturally arises: how are these concepts interconnected, and where do they overlap? It is folklore that unrestricted intersections combined with parametric polymorphism encode a form of bounded quantification, as described in the introduction (see also [Castagna 2024, Section 2]). Xie et al. [2020] show that row and bounded polymorphism can be encoded with disjoint polymorphism, obtained by adding parametric polymorphism to a system with *disjoint* intersections, and having a disjointedness predicate in the quantification of types. Contrary to our intersections that are uninhabited if applied to separate types, disjoint intersections type a merge operator that generalizes the disjoint concatenation of extensible data types, like records, to arbitrary types. The deletion operator of extensible data types can also be generalized to arbitrary types using disjoint polymorphism and a merge operator. This operator once again differs from our set exclusion operation $t_1 \setminus t_2 = t_1 \wedge \neg t_2$.

Tang et al. [2023] formally compare the expressiveness of row and presence polymorphism to *structural* subtyping, for calculi with records and variants. More precisely, they encode diverse subtyping using relevant polymorphic systems. They take special care in framing the complexity of the translation, for instance when it can be done by changing only types.

Practice. Efficient compilation of polymorphic or extensible records has been widely explored by researchers such as Gaster and Jones [1996] and Ohori [1995]. These works advocate moving away from Rémy's formalism. Ohori's calculus in particular stores information in elaborated kinds. It was expanded to extensible records by Alves and Ramos [2021], but with no mention of the compilation method. Yet, Hillerström and Lindley [2016] provide a compelling abstract machine for a calculus employing this formalism, serving as a foundation for their language Links. Regarding general-purpose languages, several of them propose either a flavor of set-theoretic types (like Typescript [Microsoft 2012] or Flow [Facebook 2015]), sometimes based on the theory of semantic subtyping (CDuce, Luau, Elixir or Ballerina), or polymorphic extensible records (like Purescript [Freeman 2013] or OCaml) but to our knowledge, none of them offers both of these features.

6 CONCLUSION

We designed a type system featuring set-theoretic types and semantic subtyping for record calculi with row and presence polymorphism. We instantiated this type system on a specific λ-calculus incorporating record selection, extension, and deletion, and devised a unification (*tallying*) algorithm.

Our next goal is implementing these results in the CDuce compiler, thereby enhancing the language with row polymorphism. We closely adhered to the theory used for CDuce, in that record types were simply extended with row variables and presence polymorphism, leveraging the union connective and an existing constant for undefinedness. Our model and algorithms, specifically for subtyping and tallying, naturally extend the existing ones. While we did not address the problem of type reconstruction, Castagna et al. [2024b] provide the theory and an implementation of type reconstruction for an ML-like language, that uses CDuce's tallying library as a black box. We are confident that plugging our tallying algorithm into that system and adding records to the language should yield, with some extra effort, a reconstruction system with polymorphic records.

CDuce is of course more complex than our record calculus. Likewise, any type system for a dynamic language needs to account for features like pattern matching, type cases, guards, or type narrowing. At first sight, and considering the implementation of CDuce, these features seem mostly orthogonal to the introduction of row polymorphism. Our hope is to be able to integrate the latter seamlessly in any existing set-theoretic type system with semantic subtyping like Elixir [Castagna et al. 2024a], Ballerina [Ballerina 2024], Luau [Jeffrey 2022] or Erlang [Schimpf et al. 2022], and we closely monitor the ongoing efforts to port CDuce's tallying algorithm into the Elixir's compiler.

Our record calculus lacks first-class labels. Castagna [2023] defines a unique type system in which records can be used both as "structs" (i.e., records without first-class labels) and as dictionaries/maps with first-class labels. His work is carried out in the same setting as ours, set-theoretic types with semantic subtyping and records as quasi-constant functions, but without row-polymorphism. However, some solutions proposed by Castagna [2023] are explicitly motivated by having a system that could be easily extended with row and presence polymorphism, which is why we believe that merging the two systems should not pose any fundamental issue. The actual expressiveness of a system with first-class labels where operations on records are not part of the syntax of types remains to be investigated.

REFERENCES

Sandra Alves and Miguel Ramos. Dec. 2021. "An ML-style Record Calculus with Extensible Records". *Electronic Proceedings in Theoretical Computer Science*, 351, (Dec. 2021), 1–17. DOI: 10.4204/eptcs.351.1.

Ballerina. 2024. *Structural Typing*. Retrieved Feb. 2024 from https://ballerina.io/learn/by-example/structural-typing/.

Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. 2003. "CDuce: an XML-Centric General-Purpose Language". In: *ICFP '03, 8th ACM International Conference on Functional Programming*. ACM Press, Uppsala, Sweden, 51–63.

Luca Cardelli and John C. Mitchell. Mar. 1991. "Operations on records". *Mathematical Structures in Computer Science*, 1, 1, (Mar. 1991), 3–48. DOI: 10.1017/s0960129500000049.

Giuseppe Castagna. Mar. 2024. "Programming with union, intersection, and negation types". In: *The French School of Programming*. Ed. by Bertrand Meyer. ISBN 978-3-031-34517-3. To appear. Preprint at arXiv:2111.03354. Springer, (Mar. 2024).

Giuseppe Castagna. Aug. 2023. "Typing Records, Maps, and Structs". *Proceedings of the ACM on Programming Languages*, 7, ICFP, (Aug. 2023), 215–258. DOI: 10.1145/3607838.

Giuseppe Castagna, Guillaume Duboc, and José Valim. 2024a. "The Design Principles of the Elixir Type System". *The Art, Science, and Engineering of Programming*, 8, 2. DOI: 10.22152/programming-journal.org/2024/8/4.

Giuseppe Castagna, Mickaël Laurent, and Kim Nguyen. Jan. 2024b. "Polymorphic Type Inference for Dynamic Languages". *Proceedings of the ACM on Programming Languages*, 8, POPL, (Jan. 2024), 1179–1210. DOI: 10.1145/3632882.

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. Jan. 2015. "Polymorphic Functions with Set-Theoretic Types". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, (Jan. 2015). DOI: 10.1145/2676726.2676991.

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. Jan. 2014. "Polymorphic functions with set-theoretic types". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, (Jan. 2014). DOI: 10.1145/2535838.2535840.

Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. Sept. 2016. "Set-theoretic types for polymorphic variants". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ACM, (Sept. 2016). DOI: 10.1145/2951913.2951928.

Giuseppe Castagna and Zhiwu Xu. Sept. 2011. "Set-theoretic foundation of parametric polymorphism and subtyping". In: *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*. ACM, (Sept. 2011). DOI: 10.1145/2034773.2034788.

[SW] CDuce, *CDuce Manual*. URL: https://www.cduce.org/manual.html.

James Clark. 2022. *Ballerina Language Specification, 2022R4*. Ed. by WSO2. Retrieved Feb. 21, 2024 from https://ballerina.io/spec/lang/2022R4/.

[SW] Elixir, *Elixir Language* (Coord.by The Elixir Team), 2012. URL: https://elixir-lang.org/.

[SW] Elixir, *Function* `add_elixir_domain` *in* `logger.ex` *module* 2024. URL: https://github.com/elixir-lang/elixir/blob/af2b21d67d4865313807e485e43d4e38a1676a54/lib/logger/lib/logger.ex#L940-L944.

[SW] Facebook, *Flow* 2015. URL: https://flow.org/.

[SW] Phil Freeman, *PureScript* 2013. URL: https://www.purescript.org/.

Alain Frisch. Dec. 2004. "Théorie, conception et réalisation d'un langage de programmation adapté à XML". PhD thesis. Université Paris 7 - Denis Diderot, (Dec. 2004).

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Sept. 2008. "Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types". *Journal of the ACM*, 55, 4, (Sept. 2008), 1–64. DOI: 10.1145/1391289.1391293.

Benedict R. Gaster and Mark P. Jones. Nov. 1996. *A Polymorphic Type System for Extensible Records and Variants*. Tech. rep. Department of Computer Science, University of Nottingham, (Nov. 1996).

Nils Gesbert, Pierre Genevès, and Nabil Layaïda. Oct. 2015. "A Logical Approach to Deciding Semantic Subtyping". *ACM Transactions on Programming Languages and Systems*, 38, 1, (Oct. 2015), 1–31. DOI: 10.1145/2812805.

Robert Harper and Benjamin Pierce. 1991. “A record calculus based on symmetric concatenation”. In: *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL ’91* (POPL ’91). ACM Press. DOI: 10.1145/99583.99603.

Daniel Hillerström and Sam Lindley. Sept. 2016. “Liberating effects with rows and handlers”, (Sept. 2016). DOI: 10.1145/2976022.2976033.

Alan Jeffrey. Nov. 2022. *Semantic Subtyping in Lua*u. Blog post. Accessed on May 6th 2023. (Nov. 2022). https://blog.roblox.com/2022/11/semantic-subtyping-luau.

Daan Leijen. 2005. “Extensible Records with Scoped Labels”. *Revised Selected Papers from the Sixth Symposium on Trends in Fuctional Programming, TFP 2005*. Ed. by Marko C. J. D. van Eekelen.

[SW] Roblox, *Lua*u 2021.

Ed. by Paolo Parrini. “Analytic and Synthetic Judgements in Type Theory”. *Kant and Contemporary Epistemology*. Springer Netherlands, Dordrecht, 87–99. ISBN: 978-94-011-0834-8. DOI: 10.1007/978-94-011-0834-8_5.

[SW] Microsoft, *TypeScript* 2012. URL: https://www.typescriptlang.org/.

J. Garrett Morris and James McKinna. Jan. 2019. “Abstracting extensible data types: or, rows by any other name”. *Proceedings of the ACM on Programming Languages*, 3, POPL, (Jan. 2019), 1–28. DOI: 10.1145/3290325.

Atsushi Ohori. Nov. 1995. “A polymorphic record calculus and its compilation”. *ACM Transactions on Programming Languages and Systems*, 17, 6, (Nov. 1995), 844–895. DOI: 10.1145/218570.218572.

Adam Paszke and Ningning Xie. Aug. 2023. “Infix-Extensible Record Types for Tabular Data”. In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Type-Driven Development* (TyDe ’23). ACM, (Aug. 2023). DOI: 10.1145/3609027.3609406.

Tommaso Petrucciani. Mar. 2019. “Polymorphic Set-Theoretic Types for Functional Languages”. PhD thesis. Università di Genova, Université Paris Diderot, (Mar. 2019).

Didier Rémy. 1989. “Type checking records and variants in a natural extension of ML”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL ’89* (POPL ’89). ACM Press, Austin, Texas, USA, 77–88. ISBN: 0897912942. DOI: 10.1145/75277.75284.

“Type inference for records in natural extension of ML”. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. MIT Press, Cambridge, MA, USA, 67–95. ISBN: 026207155X.

Didier Rémy and Jérôme Vouillon. 1998. “Objective ML: An effective object-oriented extension to ML”. *Theory and Practice of Object Systems*, 4, 1, 27–50. DOI: 10.1002/(sici)1096-9942(1998)4:1<27::aid-tapo3>3.0.co;2-4.

Albert Schimpf, Stefan Wehr, and Annette Bieniusa. Aug. 2022. “Set-theoretic Types for Erlang”. In: *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages* (IFL 2022). ACM, (Aug. 2022). DOI: 10.1145/3587216.3587220.

Wenhao Tang, Daniel Hillerström, James McKinna, Michel Steuwer, Ornela Dardha, Rongxiao Fu, and Sam Lindley. Oct. 2023. “Structural Subtyping as Parametric Polymorphism”. *Proceedings of the ACM on Programming Languages*, 7, OOPSLA2, (Oct. 2023), 1093–1121. DOI: 10.1145/3622836.

Types mailing list. June 2019. *What exactly should we call syntax-directed inference rules?* http://lists.seas.upenn.edu/pipermail/types-list/2019/002138.html. (June 2019).

Mitchell Wand. 1987. “Complete Type Inference for Simple Objects”. In: *Proceedings of the IEEE Symposium on Logic in Computer Science*.

Mitchell Wand. July 1991. “Type inference for record concatenation and multiple inheritance”. *Information and Computation*, 93, 1, (July 1991), 1–15. DOI: 10.1016/0890-5401(91)90050-c.

Andrew K. Wright and Matthias Felleisen. 1994. “A Syntactic Approach to Type Soundness”. *Information and Computation*, 115, 1, 38–94. DOI: 10.1006/inco.1994.1093.

Ningning Xie, Bruno C. d. S. Oliveira, Xuan Bi, and Tom Schrijvers. 2020. “Row and Bounded Polymorphism via Disjoint Polymorphism”. en. In: Schloss Dagstuhl - Leibniz-Zentrum für Informatik. DOI: 10.4230/LIPICS.ECOOP.2020.27.

A APPENDIX FOR TYPES

The kinding rules for types are given in Fig. 3.

$$\frac{}{\mathbb{0} : \kappa} \qquad \frac{t : \kappa}{\neg t : \kappa} \qquad \frac{t_1 : \kappa \qquad t_2 : \kappa}{t_1 \vee t_2 : \kappa} \qquad \frac{}{\alpha : \star} \qquad \frac{}{b : \star} \qquad \frac{t_1 : \star \qquad t_2 : \star}{t_1 \to t_2 : \star}$$

$$\frac{\langle (\ell = \tau_\ell)_{\ell \in L} \,|\, \varsigma \rangle^{\emptyset} : \mathrm{Row}(\emptyset)}{\{ (\ell = \tau_\ell)_{\ell \in L} \,|\, \varsigma \} : \star} \qquad \frac{\forall \ell \in L_1 . \tau_\ell : \star_\bot}{\langle (\ell = \tau_\ell)_{\ell \in L_1} \,|\, \varsigma \rangle^{L_2} : \mathrm{Row}(L_2)} \; \begin{array}{l} \varsigma \in \{\epsilon, ..\} \\ L_1 \cap L_2 = \emptyset \end{array}$$

$$\frac{\forall \ell \in L_1 . \tau_\ell : \star_\bot}{\langle (\ell = \tau_\ell)_{\ell \in L_1} \,|\, \rho \rangle^{L_2} : \mathrm{Row}(L_2)} \; \begin{array}{l} L_1 \cap L_2 = \emptyset \\ \mathrm{dom}(\rho) = \mathcal{L} \smallsetminus (L_1 \cup L_2) \end{array} \qquad \frac{}{\theta : \star_\bot} \qquad \frac{}{\bot : \star_\bot} \qquad \frac{t : \star}{t : \star_\bot}$$

Fig. 3. Kinding rules

A.1 Example of a presence polymorphic type

Presence polymorphism has been introduced by Rémy [1994] to let a field be polymorphic in its presence. A presence variable θ can be instantiated to one of {abs, pre}. Our calculus also supports presence polymorphism thanks to field variables. Let us reproduce an example of a presence polymorphic type declaration from [Rémy 1994], that we will then transcribe to our setting.

```
type tree(θ) = Leaf of Int | Node of {left:pre.tree(θ), right:pre.tree(θ), annot:θ.int}
```

Instantiating θ to abs gives the type of trees with no annotations on the nodes, while instantiating it with pre gives the type of trees with integers annotated on the nodes.

But there are several problems: adding yet another kind of polymorphism adds a bit of work, and most of all, even absent fields must have a type attached. In [Rémy 1994], this can cause losing unification of two semantically equivalent records, if for instance one has a field ℓ of type abs.int, and the other a field ℓ of type abs.bool. To avoid this problem, Tang et al. [2023] requires every record value to be annotated with its type, so that they can forget about absent types, and also so that they can have deterministic typing. So for instance the superfluous type $\{\ell_1 = M; \ell_2 = N\}^{\{\ell_1:\mathsf{pre}.A;\ell_2:\mathsf{abs}.B\}}$ is equivalent to $\{\ell_1 = M\}^{\{\ell_1:\mathsf{pre}.A\}}$.

In CDuce, there is no presence polymorphism at all, and the tree type defined previously cannot be expressed without code redundancy. In our formalism, presence polymorphism is simply obtained by means of field variables, and adding support for it in CDuce would make it possible to write the following (note that we use unions instead of tagged unions):

```
type tree(θ) = Int | {left:tree(θ), right:tree(θ), annot:θ}
```

Fields do not have any superfluous type information when they are absent, so we do not have the redundant record expressions, and do not need to put type annotations directly on the values. Moreover, in CDuce and in our type system, fields can be optional. This is achieved thanks to union types (an optional field is encoded as $t \vee \bot$). We do not know of any presence polymorphic type system dealing with optional types.

We could instantiate the type of the example to any field type, such as Int for a tree with all nodes annotated by an integer, $\mathsf{Int} \vee \bot$ for a tree with some nodes annotated by an integer, $\bot$ for a tree with no annotation, and even for instance to a type variable α, to create a type parametric in the type of its annotation, but where the annotation is mandatory on every node. In Rémy's system, having polymorphism also on the type of the annotations would require having two parameter variables (for presence and for type). From a technical point of view, the theory behind field variables is mostly the same as for type variables, and field variables can generally be described in the same way as type variables.

A.2 Models

In this section we give the detailed technical development to define our subtyping relation.

To interpret record values we follow Frisch [2004] and represent a record value by a quasi-constant function that maps labels into either values (i.e., the elements of $\mathcal{D}$) or $\bot$. Quasi-constant functions are total functions that map all but a finite set of elements of their domain into the same value (called default value). Thus record values can be represented by quasi-constant functions whose default value is $\bot$ (see Castagna [2023] for a more detailed explanation). Formally, let us write $\mathcal{D}_\bot$ for $\mathcal{D} \cup \{\bot\}$ where $\bot$ is a distinguished element not in $\mathcal{D}$. We represent our record values as quasi-constant functions from $\mathcal{L}$ to $\mathcal{D}_\bot$ and, thus, interpret record types as sets of these functions.

The formal definition of quasi-constant function has been given in Definition 2.2 and is repeated below for convenience.

Definition 2.2 ([Frisch 2004]). *Let Z denote some set. A function $r : \mathcal{L} \to Z$ is* quasi-constant *if there exists $z \in Z$ such that the set $\{\ell \in \mathcal{L} \mid r(\ell) \neq z\}$ is finite. We use $\mathcal{L} \rightharpoonup Z$ to denote the set of quasi-constant functions from $\mathcal{L}$ to Z and $\{\!\!\{\ell_1 = z_1, \ldots, \ell_n = z_n, \_ = z\}\!\!\}$ to denote the quasi-constant function $r : \mathcal{L} \rightharpoonup Z$ defined by $r(\ell_i) = z_i$ for $i = 1..n$ and $r(\ell) = z$ for $\ell \in \mathcal{L} \setminus \{\ell_1, \ldots, \ell_n\}$.*

Although this notation is not univocal (unless we require $z_i \neq z$ and the ℓ_i's to be pairwise distinct), this is largely sufficient for the purposes of this work. If $(Z_\ell)_{\ell \in \mathcal{L}}$ is a family of subsets of Z indexed by $\mathcal{L}$, we denote by $\prod_{\ell \in \mathcal{L}} Z_\ell$ the subset of $\mathcal{L} \rightharpoonup Z$ formed by all quasi-constant functions r such that $r(\ell) \in Z_\ell$ for all $\ell \in \mathcal{L}$ (intuitively, $\prod_{\ell \in \mathcal{L}} Z_\ell$ is a "type" of quasi-constant functions).

Next we have to give an interpretation for the variables. Castagna and Xu [2011] tell us that type variables must be interpreted as sets in the domain $\mathcal{D}$. Therefore, an interpretation for type variables is a function in $\mathcal{V}_t \to \mathcal{P}(\mathcal{D})$. Field variables are not much harder, since the only difference with type variables is that their interpretation can contain $\bot$, and therefore it is a function in $\mathcal{V}_f \to \mathcal{P}(\mathcal{D}_\bot)$. More difficult is the interpretation of row variables, since these are mapped into *rows*, that is, *partial* quasi-constant functions on $\mathcal{L}$. Let us write $\mathcal{L} \nrightarrow \mathcal{D}_\bot$ for the partial quasi-constant functions from $\mathcal{L}$ to $\mathcal{D}_\bot$. Thus, an interpretation of row variables must map an element of $\mathcal{V}_r$ into a set of functions in $\mathcal{L} \nrightarrow \mathcal{D}_\bot$. However, for a given ρ we cannot consider any element in $\mathcal{P}(\mathcal{L} \nrightarrow \mathcal{D}_\bot)$: we need that the functions in the interpretation of ρ are total on $\mathrm{dom}(\rho)$. Formally, we have:

Definition A.1 (Well-Kinded Interpretation). *Let η be a function in $\mathcal{V}_r \to \mathcal{P}(\mathcal{L} \nrightarrow \mathcal{D}_\bot)$. We say that η is* well kinded *if for every $\rho \in \mathcal{V}_r$ and for every $f \in \eta(\rho)$, f is a (total) quasi-constant function in $\mathrm{dom}(\rho) \rightharpoonup \mathcal{D}_\bot$. We denote by $\mathcal{V}_r \twoheadrightarrow \mathcal{P}(\mathcal{L} \nrightarrow \mathcal{D}_\bot)$ the set of well-kinded functions.*

In conclusion our interpretation of types will be parametric in an assignment η for the variables, which will be a function in

$$\mathcal{H} \stackrel{\text{def}}{=} (\mathcal{V}_t \to \mathcal{P}(\mathcal{D})) \cup (\mathcal{V}_f \to \mathcal{P}(\mathcal{D}_\bot)) \cup (\mathcal{V}_r \twoheadrightarrow \mathcal{P}(\mathcal{L} \nrightarrow \mathcal{D}_\bot))$$

The next step is to define the domain $\mathcal{D}$ in which to give the interpretation of types. This is quite simple for us since it suffices to take the model defined by Castagna and Xu [2011] and replace products by quasi-constant functions. The hard problem for defining this model, and thus the interpretation of types, is to give an interpretation of the function spaces, but this problem was solved by Frisch et al. [2008] whose solution is reused by Castagna and Xu [2011]. In a nutshell we want to define an interpretation function $[\![.]\!] : \mathcal{T} \to \mathcal{H} \to \mathcal{P}(\mathcal{D})$. Since the elements of $\mathcal{D}$ represent the values of the language, then $\mathcal{D}$ must contain the set C of constants of the language, the quasi-constant functions (to represent record values), and the functions from $\mathcal{D}$ to $\mathcal{D}$, but the last containment is impossible for cardinality reasons. The solution by [Frisch et al. 2008] is to associate to every domain $\mathcal{D}$ and function $[\![.]\!] : \mathcal{T} \to \mathcal{H} \to \mathcal{P}(\mathcal{D})$ a unique *extensional interpretation* $\mathbb{E}(\cdot) : \mathcal{T} \to \mathcal{H} \to \mathcal{P}(\mathbb{E}_{\mathcal{D}})$ which fixes the semantic of the type constructors, and then to accept as a valid interpretation of the types only the pairs $([\![.]\!], \mathcal{D})$ such that for all η, $[\![t]\!]_\eta = \emptyset \iff \mathbb{E}(t)_\eta = \emptyset$.

We invite the reader to refer to Castagna and Xu [2011, Section 2.2] for a more detailed explanation of how the extensional interpretation works and to Frisch et al. [2008] for full details. Henceforth, we just present how to extend the extensional interpretation of Castagna and Xu [2011] to include quasi-constant functions and the interpretation of row-variables, and pinpoint the differences between the two definitions. We suppose to be given an interpretation $\mathbb{B} : \mathcal{B} \to \mathcal{P}(C)$ of basic types into sets of constants. Given a set S we use the notation $\overline{S}$ to denote its complement in an appropriate universe: this notation is in particular used for the set $\mathcal{P}(\overline{[\![t_1]\!]_\eta \times \overline{[\![t_2]\!]_\eta}})$ which corresponds to interpreting the elements in $t_1 \to t_2$ as binary relations, namely as elements of the set $\{ f \subseteq \mathcal{D}^2 \mid \text{for all } (d_1, d_2) {\in} f,\ \text{if } d_1 {\in} [\![t_1]\!]_\eta \text{ then } d_2 {\in} [\![t_2]\!]_\eta \}$.

Definition A.2 (Extensional interpretation). *Let $\mathcal{D}$ be a set. The* extensional domain *of $\mathcal{D}$ is defined as: $\mathbb{E}_{\mathcal{D}} = C + \mathcal{D} + \mathcal{P}(\mathcal{D} \times \mathcal{D}_\Omega) + (\mathcal{L} \twoheadrightarrow \mathcal{D}_\perp)$ where Ω and $\perp$ are two different distinguished elements not in $\mathcal{D}$.*

Let $[\![\cdot]\!] : \mathcal{T} \to \mathcal{H} \to \mathcal{P}(\mathcal{D})$ be an interpretation of types parametric in a well-kinded interpretation of variables. The associated extensional interpretation *of types is the unique function $\mathbb{E}(\cdot) : \mathcal{T} \to \mathcal{H} \to \mathcal{P}(\mathbb{E}_{\mathcal{D}})$ such that:*

$$\begin{aligned}
\mathbb{E}(\mathbb{0})_\eta &= \emptyset\\
\mathbb{E}(\alpha)_\eta &= \eta(\alpha)\\
\mathbb{E}(b)_\eta &= \mathbb{B}(b)\\
\mathbb{E}(\neg t)_\eta &= \mathbb{E}_{\mathcal{D}} \setminus \mathbb{E}(t)_\eta\\
\mathbb{E}(t_1 \vee t_2)_\eta &= \mathbb{E}(t_1)_\eta \cup \mathbb{E}(t_2)_\eta\\
\mathbb{E}(t_1 \to t_2)_\eta &= \mathcal{P}(\overline{[\![t_1]\!]_\eta \times \overline{[\![t_2]\!]_\eta}})
\end{aligned}
\qquad
\mathbb{E}(\mathsf{R})_\eta = \begin{cases} \bigcup_{\bar{d} \in \eta(\rho)} \left(\mathop{\triangleright\!\!\!\mid}_{\ell \in \mathrm{lab}(\mathsf{R})} [\![\mathsf{R}(\ell)]\!]^{\mathrm{fld}}_\eta \sqcup \bar{d} \right) & \textit{if } \mathrm{tail}(\mathsf{R}) = \rho\\ \mathop{\triangleright\!\!\!\mid}_{\ell \in \mathcal{L}} [\![\mathsf{R}(\ell)]\!]^{\mathrm{fld}}_\eta & \textit{otherwise} \end{cases}$$

where

$$\begin{aligned}
[\![t]\!]^{\mathrm{fld}}_\eta &= [\![t]\!]_\eta \quad \textit{if } t \neq \neg t' \textit{ and } t \neq t_1 \vee t_2\\
[\![\theta]\!]^{\mathrm{fld}}_\eta &= \eta(\theta)\\
[\![\perp]\!]^{\mathrm{fld}}_\eta &= \{\perp\}
\end{aligned}
\qquad
\begin{aligned}
[\![\tau_1 \vee \tau_2]\!]^{\mathrm{fld}}_\eta &= [\![\tau_1]\!]^{\mathrm{fld}}_\eta \cup [\![\tau_2]\!]^{\mathrm{fld}}_\eta\\
[\![\neg\tau]\!]^{\mathrm{fld}}_\eta &= (\mathbb{E}_{\mathcal{D}} \cup \{\perp\}) \setminus [\![\tau]\!]^{\mathrm{fld}}_\eta
\end{aligned}$$

Notice that the induction used in the definition is well-founded thanks to the contractivity condition in the definition of types and the fact that field types are inductively defined.

The extensional interpretation is defined with respect to some domain $\mathcal{D}$ and interpretation $[\![.]\!]$, and maps types into a domain $\mathbb{E}_{\mathcal{D}}$ that contains $\mathcal{D}$, constants C to interpret basic types, sets of binary relations $\mathcal{P}(\mathcal{D} \times \mathcal{D}_\Omega)$ to interpret function types, and quasi constant functions $\mathcal{L} \twoheadrightarrow \mathcal{D}_\perp$ to interpret record types. The fact that functions are binary relations that can yield a distinguished element Ω (which, intuitively, represents a type error) is a standard technique of semantic subtyping to avoid $\mathbb{1} \to \mathbb{1}$ to be a supertype of all function types: since it does not play any specific role in our work we will not further comment on it (see [Frisch et al. 2008] for a detailed explanation or Castagna [2023, Section 3.2] for a shorter one).

The definitions for the extensional interpretation given on the right-hand side in Definition A.2 are the same as those by Castagna and Xu [2011]. They state that the empty type is interpreted as the empty set, the interpretation of the type variables is given by η, that unions and negations are interpreted as set-theoretic unions and complements, and that functions types are interpreted as sets of binary relations whose output is in the codomain if the input is in the domain.

The novelty of our definition is the interpretation of record types given on the left-hand side. There are two cases. The easy case is when the tail of the record type is either ϵ or $..$: in that case the interpretation of the record type is the set of all quasi constant functions in $\mathcal{L} \twoheadrightarrow \mathcal{P}(\mathcal{D}_\perp)$ that map a label ℓ into an element of the interpretation of $\mathsf{R}(\ell)$ (recall that for $\ell \notin \mathrm{lab}(\mathsf{R})$, $\mathsf{R}(\ell)$ is $\perp$ for $\mathrm{tail}(\mathsf{R}) = \epsilon$ and $\mathbb{1} \vee \perp$ for $\mathrm{tail}(\mathsf{R}) = ..$). If instead $\mathrm{tail}(\mathsf{R})$ is a row variable ρ, then $\mathcal{L}$ is partitioned in two, the sets $\mathrm{lab}(\mathsf{R})$ and—by well-kindedness—$\mathrm{dom}(\rho)$, and the interpretation of R will be the set of quasi-constant functions in $\mathcal{L} \twoheadrightarrow \mathcal{D}_\perp$ obtained by unioning two partial functions: a function in $\bowtie_{\ell\in\mathrm{lab}(\mathsf{R})} [\![\mathsf{R}(\ell)]\!]^{\mathrm{fld}}_\eta$ for the $\mathrm{lab}(\mathsf{R})$ labels of $\mathcal{L}$, and a function in $\eta(\rho)$ for the remaining labels of $\mathcal{L}$. There is a caveat: fields can map labels both into values and $\perp$. Therefore, the interpretation of field types must be slightly different from that of types, since it must map $\perp$ into $\{\perp\}$ and the negation of a field type is the complement with respect to $\mathcal{D}_\perp$, rather than $\mathcal{D}$: the $[\![.]\!]^{\mathrm{fld}}$ does just that.

Given a domain $\mathcal{D}$ and a set-theoretic interpretation of the types into this domain, they form a *model* if the interpretation and the associated extensional interpretation have the same zeros:

Definition A.3 (Model). *Let $\mathcal{D}$ be a domain and $[\![\cdot]\!] : \mathcal{T} \to \mathcal{H} \to \mathcal{P}(\mathcal{D})$. The pair $(\mathcal{D}, [\![\cdot]\!])$ is a* model *if and only, if for all $t \in \mathcal{T}$ and $\eta \in \mathcal{H}$, $[\![t]\!]_\eta = \emptyset \iff \mathbb{E}(t)_\eta = \emptyset$.*

Every model induces a subtyping relation on types[12]:

Definition A.4 (Subtyping). *If $(\mathcal{D}, [\![\cdot]\!])$ is a model, then it induces a subtyping relation defined as follows:*

$$t_1 \leq t_2 \quad \stackrel{\mathrm{def}}{\iff} \quad \forall\eta. [\![t_1]\!]_\eta \subseteq [\![t_2]\!]_\eta$$

As explained by Frisch [2004, Section 2.6], the interest of defining of a *model* is that we can work with the interpretation of the model "*as if*" the interpretation of the type constructors (in particular, the function type constructor) were defined as their extensional interpretation. So when deducing the properties for the subtyping relation of a model—and just for the subtyping relation—we can assume that $\mathcal{P}(\overline{[\![t_1]\!]_\eta \times \overline{[\![t_2]\!]_\eta}})$, even if this is impossible for cardinality reasons.

Definition A.3 specifies which characteristics a model must have to induce a subtyping relation (that behaves "*as if*"), but it does not define any particular model nor, thus, any particular subtyping relation. In what follows we define a concrete interpretation domain $\mathcal{D}$ (whose elements are defined by induction) and two specific interpretations, and prove that they satisfy the conditions to be models, since they both have the same zeros as the extensional interpretation (of one of them). This yields two equivalent definitions of a concrete subtyping relation we are going to use in the rest of this presentation. We will define:

- An interpretation $[\![t]\!]^{\mathrm{q}}_\eta$ parametrized by an assignment η, for which subtyping $t_1 \leq^{\mathrm{q}} t_2$ is defined as $\forall\eta. [\![t_1]\!]^{\mathrm{q}}_\eta \subseteq [\![t_2]\!]^{\mathrm{q}}_\eta$; (the index q stands for ***quantified***, since subtyping is quantified on all variable interpretations);
- An interpretation of types directly into sets $[\![t]\!]$, that avoids quantification over η, and for which subtyping $t_1 \leq t_2$ is defined directly as $[\![t_1]\!] \subseteq [\![t_2]\!]$ (notice the absence of a variable interpretation argument η).

The interpretation of types is mutually recursive with interpretations of rows and field types. Both of these will also give rise to subtyping relations. We will use the same notation $\leq$ for the relations in $\mathcal{T} \times \mathcal{T}$, $\mathcal{T}_\perp \times \mathcal{T}_\perp$ and $\mathcal{R} \times \mathcal{R}$.

This interpretation in Definition 2.5 induces a subtyping relation that it is easy to work with, since it got rid of the interpretation η for the variables. We can consider the interpretation $[\![\cdot]\!]$:

[12]Actually, a model must be convex: see Castagna and Xu [2011]. We omit this detail since it is not relevant to our presentation.

$\mathcal{T} \to \mathcal{P}(\mathcal{D})$ as a function in $\mathcal{T} \to \mathcal{H} \to \mathcal{P}(\mathcal{D})$ that is constant on its second argument: if we apply Definition A.4 to it, then the subtyping relation is defined as simply as $s \leq t \overset{\text{def}}{\iff} [\![s]\!] \subseteq [\![t]\!]$. But getting rid of η makes it difficult to prove that this interpretation is a model and, thus, that when considering the properties of this subtyping relation, we can work "as if" the interpretation of type constructors were as in the extensional interpretation. To overcome this difficulty we define a second interpretation, on the same domain, but this interpretation disregards the indexes of the elements and uses an assignment η to interpret the variables.

Definition A.5 (Parametrized intepretation of types and rows). *We define a ternary predicate $(D : T)^{\mathsf{q}}_\eta$ ("the element D belongs to T under assignment η"), by induction on the pair (D, T) ordered lexicographically. The only differences with the predicate $(D : T)$ (apart from recursive calls to the appropriate predicate), are:*

$$(d : \alpha)^{\mathsf{q}}_\eta = d \in \eta(\alpha)$$
$$(\delta : \theta)^{\mathsf{q}}_\eta = \delta \in \eta(\theta)$$
$$\begin{aligned}(\langle\!\!| (\ell = \delta_\ell)_{\ell \in L_1}, \_ = \bot^\emptyset |\!\!\rangle^V_{L_2} : r)^{\mathsf{q}}_\eta = {} & (\forall \ell \in L_1.(\delta_\ell : r(\ell))^{\mathsf{q}}_\eta) \text{ and } (\forall \ell \in \mathrm{dom}(r) \setminus L_1.(\bot^\emptyset : r(\ell))^{\mathsf{q}}_\eta) \\ & \text{and } \mathrm{tail}(r) = \rho \implies \langle\!\!| (\ell = \delta_\ell)_{\ell \in L_1 \cap \mathrm{dom}(\rho)}, \_ = \bot^\emptyset |\!\!\rangle^V_{\mathcal{L} \setminus \mathrm{dom}(\rho)} \in \eta(\rho)\end{aligned}$$

We define the interpretations $[\![\cdot]\!]^{\mathsf{q}}_\eta$, $[\![\cdot]\!]^{\mathsf{qfld}}_\eta$ and $[\![\cdot]\!]^{\mathsf{qrow}}_\eta$ as expected.

While the interpretation of type and field variables is straightforwardly given by η, the interpretation of row variables is less evident. The first line of the interpretation of a row is the same as in Definition 2.5: in both definitions this line deals with the case when the tail of r is not a row variable. The second line covers the case for $\mathrm{tail}(r) = \rho$: it checks that $\bar{d} \in \eta(\rho)$, where $\bar{d}$ is obtained by restricting the quasi-constant function on the left to the domain of ρ.

Our goal is to prove that both interpretations give a model of types. Formally, this corresponds to proving the following equivalences: Let $\mathbb{E}(\cdot)$ be the extensional interpretation of $[\![.]\!]^{\mathsf{q}}$. For all $t \in \mathcal{T}$:

$$(\forall \eta . \mathbb{E}(t)_\eta = \emptyset) \iff (\forall \eta . [\![t]\!]^{\mathsf{q}}_\eta = \emptyset) \iff [\![t]\!] = \emptyset \tag{9}$$

The leftmost "iff" proves that $(\mathcal{D}, [\![.]\!]^{\mathsf{q}})$ is a model, while the rightmost one proves that the subtyping relation induced by $[\![.]\!]$ is the same as the one induced by the model $[\![.]\!]^{\mathsf{q}}$ (since $\forall \eta . [\![s]\!]^{\mathsf{q}}_\eta \subseteq [\![t]\!]^{\mathsf{q}}_\eta \iff \forall \eta . [\![s]\!]^{\mathsf{q}}_\eta \cap (\mathcal{D} \setminus [\![t]\!]^{\mathsf{q}}_\eta) = \emptyset \iff \forall \eta . [\![s \wedge \neg t]\!]^{\mathsf{q}}_\eta = \emptyset \iff [\![s \wedge \neg t]\!] = \emptyset \iff [\![s]\!] \subseteq [\![t]\!]$).

For the first equivalence, we prove the following, more precise, statement.

Lemma A.1. *For all type t, for all η,*

$$[\![t]\!]^{\mathsf{q}}_\eta = \emptyset \iff \mathbb{E}(t)_\eta = \emptyset$$

Proof. For all d, we show $(d : t)^{\mathsf{q}}_\eta \iff d \in \mathbb{E}(t)_\eta$ by induction on t in both directions. This induction is well-founded because the cases for type constructors do not use induction, $\mathbb{E}(t)_\eta$ is defined on top of $[\![t]\!]^{\mathsf{q}}_\eta$, and the number of type connectives is finite by regularity of the types.

We start with the left-to-right implication and detail the case $t = \mathsf{R}$. By hypothesis there is $\bar{d} = \langle\!\!| (\ell = \delta_\ell)_{\ell \in L}, \_ = \bot^\emptyset |\!\!\rangle^V_\emptyset$ such that $(\mathrm{Rec}(\bar{d})^{V'} : \mathsf{R})^{\mathsf{q}}_\eta$. By hypothesis, $\forall \ell \in L.(\delta_\ell : \mathsf{R}(\ell))^{\mathsf{q}}_\eta$, so $\delta_\ell \in [\![\mathsf{R}(\ell)]\!]^{\mathsf{qfld}}_\eta$, and $\forall \ell \in \mathcal{L} \setminus L.(\bot^\emptyset : \mathsf{R}(\ell))^{\mathsf{q}}_\eta$, so $\bot^\emptyset \in [\![\mathsf{R}(\ell)]\!]^{\mathsf{qfld}}_\eta$. It is easy to see that this implies respectively $\delta_\ell \in [\![\mathsf{R}(\ell)]\!]^{\mathsf{fld}}_\eta$ and $\bot^\emptyset \in [\![\mathsf{R}(\ell)]\!]^{\mathsf{fld}}_\eta$. Moreover, if $\mathrm{tail}(\rho) = \rho$, then $\langle\!\!| (\ell = \delta_\ell)_{\ell \in L \cap \mathrm{dom}(\rho)}, \_ = \bot^\emptyset |\!\!\rangle^V_{\mathcal{L} \setminus \mathrm{dom}(\rho)} \in \eta(\rho)$. So $\bar{d} \in \mathbb{E}_\eta(\mathsf{R})$.

Now, for the right-to-left implication. Let $\bar{d} = \langle\!\!| (\ell = \delta_\ell)_{\ell \in L}, \_ = \bot^\emptyset |\!\!\rangle^V_\emptyset \in \mathbb{E}_\eta(\mathsf{R})$. For all $\ell \in \mathrm{lab}(\mathsf{R})$, we have by hypothesis $\delta \in [\![\mathsf{R}(\ell)]\!]^{\mathsf{fld}}$, which implies $(\delta : \mathsf{R}(\ell))^{\mathsf{qfld}}_\eta$. Let $\ell \notin \mathrm{lab}(\mathsf{R})$. If $\ell \in L_1$, then $(\delta_\ell : \mathsf{R}(\ell))^{\mathsf{q}}_\eta$ holds. If $\ell \notin L_1$: if $\mathrm{tail}(\mathsf{R}) \in \mathcal{V}$, by definition $\mathsf{R}(\ell) = \mathbb{1} \vee \bot$, and if $\mathrm{tail}(\mathsf{R}) \notin \mathcal{V}$,

$\mathsf{R}(\ell) = \mathbb{1} \vee \bot$, or $\mathsf{R}(\ell) = \bot$. In any case, $(\bot^{\emptyset} : \mathsf{R}(\ell))^{\mathrm{q}}_{\eta}$ holds. Finally, if $\mathrm{tail}(\mathsf{R}) = \rho \in \mathcal{V}$, then $\langle\!\langle (\ell = \delta_\ell)_{\ell \in L \cap \mathrm{dom}(\rho)}, \_ = \bot^{\emptyset} \rangle\!\rangle^{V}_{\mathcal{L} \setminus \mathrm{dom}(\rho)} \in \eta(\rho)$. □

We have thus shown that the subtyping relation generated by $[\![\cdot]\!]^{\mathrm{q}}$ has the expected properties described by $\mathbb{E}(\cdot)$. In particular, it is a *set-theoretic* model because type operators are interpreted as set operators.

Since it will be easier to work directly with interpretations as sets rather than to quantify over η, we now show that the subtyping relation generated by $[\![\cdot]\!]$ is equivalent to the parametrized one. We show that equivalence not only on types, but also on field types and rows. The main element of the proof is the canonical assignment $\hat{\eta}$, defined as

$$\hat{\eta}(\alpha) = \{d \in \mathcal{D} \mid \alpha \in \mathrm{tag}(d)\} \tag{10}$$

$$\hat{\eta}(\theta) = \{\delta \in \mathcal{D}_\bot \mid \theta \in \mathrm{tag}(\delta)\} \tag{11}$$

$$\hat{\eta}(\rho) = \{\partial\!\!\!/ \in \mathcal{D}_{\mathrm{row}} \mid \mathrm{dom}(\partial\!\!\!/) = \mathrm{dom}(\rho) \text{ and } \rho \in \mathrm{tag}(\partial\!\!\!/)\} \tag{12}$$

Lemma A.2. *For every $T \in \mathcal{T}_\bot \cup \mathcal{R}$, $[\![T]\!] = [\![T]\!]^{\mathrm{q}}_{\hat{\eta}}$.*

Proof. For any D and T we prove that $(D : T) \iff (D : T)^{\mathrm{q}}_{\hat{\eta}}$ by induction on (D, T). The only interesting case is when $T = r$ and $D = \langle\!\langle (\ell = \delta_\ell)_{\ell \in L_1}, \_ = \bot^{\emptyset} \rangle\!\rangle^{V}_{L_2}$. For all $\ell \in L_1$ we have $(\delta_\ell : r(\ell)) \iff (\delta_\ell : r(\ell))^{\mathrm{q}}_{\hat{\eta}}$ by induction hypothesis. Let $\ell \notin L_1$. We show that $(\bot^{\emptyset} : r(\ell)) \iff (\bot^{\emptyset} : r(\ell))^{\mathrm{q}}_{\eta}$ by induction on $r(\ell)$. This induction is well-founded : for $r(\ell) = \bot$, both propositions are true and they are false for any other type constructor (in particular $r(\ell) = \theta$). The inductive cases on type operators are straightforward. If $\mathrm{tail}(r) \notin \mathcal{V}$, we are done. Otherwise, let $\mathrm{tail}(r) = \rho$. On the left side, we have $\rho \in V$. On the right side, we have $\langle\!\langle (\ell = \delta_\ell)_{\ell \in L \cap \mathrm{dom}(\rho)}, \_ = \bot^{\emptyset} \rangle\!\rangle^{V}_{\mathcal{L} \setminus \mathrm{dom}(\rho)} \in \hat{\eta}(\rho)$. By definition of $\hat{\eta}(\rho)$, this is equivalent to $\rho \in V$. □

Lemma A.3. *Let $W \in \mathcal{P}_{\mathrm{fin}}(\mathcal{V})$ and $T_W = \{T \in \mathcal{T}_\bot \cup \mathcal{R} \mid \mathrm{vars}(T) \subseteq W\}$. For every $T \in T_W$,*

$$[\![T]\!]^{\mathrm{q}}_{\hat{\eta}} = \emptyset \iff \forall \eta . [\![T]\!]^{\mathrm{q}}_{\eta} = \emptyset.$$

Proof. The right-to-left implication is trivial, by instantiation of the quantifier by η'. The left-to-right implication is by contraposition: for an arbitrary W and T_W, we prove $\forall T \in T_W . (\exists \eta . [\![T]\!]^{\mathrm{q}}_{\eta} \neq \emptyset \implies [\![T]\!]^{\mathrm{q}}_{\hat{\eta}} \neq \emptyset)$. For this, we define the functions $F^{\eta}_W : \mathcal{D}_\bot \cup \mathcal{D}_{\mathrm{row}} \cup \{\Omega\} \to \mathcal{D}_\bot \cup \mathcal{D}_{\mathrm{row}} \cup \{\Omega\}$ as $F^{\eta}_W(\Omega) = \Omega$ and:

$$F^{\eta}_W(D) = \begin{cases} c^{\hat{V}(D)} & \text{if } D = c; \\ \{(F^{\eta}_W(d_1), F^{\eta}_W(\partial_1)), \ldots, (F^{\eta}_W(d_n), F^{\eta}_W(\partial_n))\}^{\hat{V}(D)} & \text{if } D = \{(d_1, \partial_1), \ldots, (d_n, \partial_n)\}^{V}; \\ \mathrm{Rec}(\partial\!\!\!/)^{\hat{V}(D)} & \text{if } D = \mathrm{Rec}(\partial\!\!\!/)^{V}; \\ \bot^{\hat{V}(D)} & \text{if } D = \bot^{V}; \\ \langle\!\langle (\ell = F^{\eta}_W(\delta_\ell))_{\ell \in L_1}, \_ = \bot^{\emptyset} \rangle\!\rangle^{\hat{V}(D)}_{L_2} & \text{if } D = \langle\!\langle (\ell = \delta_\ell)_{\ell \in L_1}, \_ = \bot^{\emptyset} \rangle\!\rangle^{V}_{L_2} \end{cases}$$

where $\hat{V}(D) = \{\alpha \in W \mid D \in \eta(\alpha)\} \cup \{\theta \in W \mid D \in \eta(\theta)\} \cup \{\rho \in W \mid D \in \eta(\rho)\}$. The finiteness of W ensures that $\hat{V}$ is finite. We prove the following statement, for an arbitrary η and by induction on (D, T) ordered lexicographically:

$$\forall T \in T_W . \forall D \in \mathcal{D}_\bot \cup \mathcal{R} . (D : T)^{\mathrm{q}}_{\eta} \implies (F^{\eta}_W(D) : T)^{\mathrm{q}}_{\hat{\eta}}$$

- $T = \alpha$. We have $(F^{\eta}_W(D) : \alpha)^{\mathrm{q}}_{\hat{\eta}} \iff \alpha \in \mathrm{tag}(F^{\eta}_W(D)) \iff \alpha \in \hat{V}(D) \iff D \in \eta(\alpha)$ and $\alpha \in W \iff (D : \alpha)^{\mathrm{q}}_{\eta}$. The last equivalence holds by the hypothesis that $T \in T_W$. The case for $T = \theta$ is similar.

- $T = \bot$ and $D = \bot^V$. $(F^\eta_W(\bot^V) : \bot)^{\mathrm{q}}_{\hat\eta} = (\bot^{\hat V(D)} : \bot)^{\mathrm{q}}_\eta$ holds.
- $T = \mathsf{R}$ and $D = \mathrm{Rec}(\mathfrak{d})^V$. By hypothesis, we have $(\mathfrak{d} : \mathrm{row}(\mathsf{R}))^{\mathrm{q}}_\eta$. By induction, this implies $(F^\eta_W(\mathfrak{d}) : \mathrm{row}(\mathsf{R}))^{\mathrm{q}}_{\hat\eta}$ and thus $(F^\eta_W(D) : \mathsf{R})^{\mathrm{q}}_{\hat\eta}$.
- $d = \langle\!\langle (\ell = \delta_\ell)_{\ell\in L_1}, \_ = \bot^{\emptyset} \rangle\!\rangle^V_{L_2}$ and $t = r$. The statement holds for labels in and outside of L_1 by induction hypothesis. If $\mathrm{tail}(r) \notin \mathcal{V}$, we are done. Let $\mathrm{tail}(r) = \rho$. By hypothesis, $\rho \in W$. By definition of the predicate, there is $\mathfrak{d} = \langle\!\langle (\ell = \delta_\ell)_{\ell\in L_1\cap \mathrm{dom}(\rho)}, \_ = \bot^{\emptyset} \rangle\!\rangle^V_{\mathcal{L}\setminus \mathrm{dom}(\rho)} \in \eta(\rho)$. As in the case for type variables, we show: $(F^\eta_W(\mathfrak{d}) : \rho)^{\mathrm{q}}_{\hat\eta} \iff \rho \in \mathrm{tag}(F^\eta_W(\mathfrak{d})) \iff \rho \in \hat V(\mathfrak{d}) \iff D \in \eta(\rho)$ and $\rho \in W$.
- Other cases can be found in [Petrucciani 2019, Lemma 2.8] or are direct by falsity of the premise. □

Lemma A.4. *For all t_1 and t_2, $t_1 \leq^{\mathrm{q}} t_2 \iff t_1 \leq t_2$.*

Proof. By definition, Lemma A.3 and Lemma A.2, we show:

$$t_1 \leq^{\mathrm{q}} t_2 \iff \forall\eta.[\![t_1 \setminus t_2]\!]^{\mathrm{q}}_\eta = \emptyset \iff [\![t_1\setminus t_2]\!]^{\mathrm{q}}_{\hat\eta} = \emptyset \iff [\![t_1\setminus t_2]\!] = \emptyset \iff t_1 \leq t_2$$ □

A.3 Subtyping relation

Lemma A.5. *Let r be an atomic row of domain $\mathcal{L} \setminus L_r$ and L such that $\mathrm{lab}(r) \subseteq L \subseteq \mathrm{dom}(r)$. Then,*

$$r \simeq \langle(\ell = r(\ell) \,|\, ..\rangle^{L_r} \wedge \langle L' \,|\, \mathrm{tail}(r)\rangle^{L_r} \simeq \bigwedge_{\ell\in L} \langle \ell = r(\ell) \,|\, ..\rangle^{L_r} \wedge \langle L' \,|\, \mathrm{tail}(r)\rangle^{L_r}$$

where $L' = \mathrm{lab}(r)$ if $\mathrm{tail}(r) \in \mathcal{V}$ and $L' = L$ otherwise.

Proof. Straightforward by the definition of the models. □

Lemma 2.1. *Let P and N be sets of atomic row types r each of domain $\mathcal{L}\setminus L_r$. Let L be a finite set of labels such that $\bigcup_{r\in P\cup N} \mathrm{lab}(r) \subseteq L \subseteq \mathcal{L}\setminus L_r$. Let $P_{\mathcal{V}} = \{r \in P \mid \mathrm{tail}(r) \in \mathcal{V}\}$ and likewise for $N_{\mathcal{V}}$. For every r, we define its default type $\mathrm{def}(r)$ as: $\mathrm{def}(r) = \bot$ if r is closed, and $\mathrm{def}(r) = \mathbb{1} \vee \bot$ otherwise. The relation $\bigwedge_{r\in P} r \leq \bigvee_{r\in N} r$ holds iff $\forall\iota : N \to L \cup \{\_\}$,*

$$\left(\exists \ell \in L. \bigwedge_{r\in P} r(\ell) \quad \leq \bigvee_{r\in\iota^{-1}(\ell)} r(\ell)\right) \tag{3}$$

$$\text{or } \left(\exists r_\circ \in \iota^{-1}(\_) \setminus N_{\mathcal{V}}.(\bigwedge_{r\in P} \mathrm{def}(r) \leq \mathrm{def}(r_\circ))\right) \tag{4}$$

$$\text{or } \left(\exists r_\circ \in \iota^{-1}(\_) \cap N_{\mathcal{V}}.\exists r \in P_{\mathcal{V}}.\, \mathrm{tail}(r_\circ) = \mathrm{tail}(r)\right) \tag{5}$$

Proof. In the following, we let $L_i = \mathrm{lab}(r_i)$ and $\varsigma_i = \mathrm{tail}(r_i)$. Using Lemma A.5, we decompose the conjunction into:

$$\bigwedge_{r_p\in P} (\langle(\ell = r_p(\ell))_{\ell\in L} \,|\, ..\rangle^{L_r} \wedge \langle L'_p \,|\, \varsigma_p\rangle^{L_r}) \wedge \bigwedge_{r_n\in N} (\vee_{\ell\in L} \langle \ell = \neg r_n(\ell) \,|\, ..\rangle^{L_r} \vee \neg\langle L'_n \,|\, \varsigma_n\rangle^{L_r}) \tag{13}$$

Where $L'_i = L_i$ if $r_i \in \mathcal{V}$ and $L'_i = L$ otherwise. We can distribute the intersection of the elements of N on the right of (13) over the unions in the second brackets. We obtain a union of intersections of, each time, $|N|$ elements, where each intersection is a possible combination of the individual rows present in the second line. Each combination is described by a function $\iota : N \to L \cup \{\_\}$, where $\iota(r_n) = \ell$ means that the element $\langle \ell = \neg r_n(\ell) \,|\, ..\rangle^{L_r}$ is present in the combination given by ι, while

$\iota(r_n) = \_$ means that the element $\neg\langle L'_n \,|\, \varsigma_n\rangle^{L_r}$ is present in the combination. For each $r_n \in N$, let us write $r^n_\ell = \langle \ell = \neg r_n(\ell) \,|\, ..\rangle^{L_r}$ and $r^n_\_ = \neg\langle L'_n \,|\, \varsigma_n\rangle^{L_r}$. Therefore the row in (13) is equivalent to:

$$\bigwedge_{r_p \in P} (\langle(\ell = r_p(\ell))_{\ell\in L} \,|\, ..\rangle^{L_r} \wedge \langle L'_p \,|\, \varsigma_p\rangle^{L_r}) \wedge \bigvee_{\iota:N\to L\cup\{\_\}} (\bigwedge_{r_n\in N} r^n_{\iota(r_n)}) \tag{14}$$

By distributing the intersection over the union we obtain

$$\bigvee_{\iota:N\to L\cup\{\_\}} \left(\bigwedge_{r_p \in P} (\langle(\ell = r_p(\ell))_{\ell\in L} \,|\, ..\rangle^{L_r} \wedge \langle L'_p \,|\, \varsigma_p\rangle^{L_r}) \wedge \bigwedge_{r_n\in N} r^n_{\iota(r_n)} \right) \tag{15}$$

A union is empty if and only if each summand of the union is empty. Therefore the row above is empty if and only if for all $\iota : N \to L \cup \{\_\}$, the following is empty:

$$\begin{aligned}
&\bigwedge_{r_p \in P} (\langle(\ell = r_p(\ell))_{\ell\in L} \,|\, ..\rangle^{L_r} \wedge \langle L'_p \,|\, \varsigma_p\rangle^{L_r}) \wedge \bigwedge_{r_n\in N} r^n_{\iota(r_n)} \\
&\simeq \bigwedge_{r_p \in P} (\langle(\ell = r_p(\ell))_{\ell\in L} \,|\, ..\rangle^{L_r} \wedge \langle L'_p \,|\, \varsigma_p\rangle^{L_r}) \wedge \bigwedge_{\ell\in L\cup\{_\}} \bigwedge_{r_n\in\iota^{-1}(\ell)} r^n_\ell \\
&\simeq \bigwedge_{r_p \in P} (\langle(\ell = r_p(\ell))_{\ell\in L} \,|\, ..\rangle^{L_r} \wedge \langle L'_p \,|\, \varsigma_p\rangle^{L_r}) \wedge \bigwedge_{r_n\in\iota^{-1}(_)} r^n__ \wedge \bigwedge_{\ell\in L} \bigwedge_{r_n\in\iota^{-1}(\ell)} r^n_\ell \\
&= \bigwedge_{r_p \in P} (\langle(\ell = r_p(\ell))_{\ell\in L} \,|\, ..\rangle^{L_r} \wedge \langle L'_p \,|\, \varsigma_p\rangle^{L_r}) \wedge \bigwedge_{r_n\in\iota^{-1}(_)} \neg\langle L'_n \,|\, \varsigma_n\rangle^{L_r} \wedge \bigwedge_{\ell\in L} \bigwedge_{r_n\in\iota^{-1}(\ell)} \langle \ell = \neg r_n(\ell) \,|\, ..\rangle^{L_r} \\
&\simeq \bigwedge_{r_p \in P} (\langle(\ell = r_p(\ell))_{\ell\in L} \,|\, ..\rangle^{L_r} \wedge \langle L'_p \,|\, \varsigma_p\rangle^{L_r}) \wedge \bigwedge_{r_n\in\iota^{-1}(_)} \neg\langle L'_n \,|\, \varsigma_n\rangle^{L_r} \wedge \langle(\ell = \textstyle\bigwedge_{r_n\in\iota^{-1}(\ell)} \neg r_n(\ell))_{\ell\in L} \,|\, ..\rangle^{L_r} \\
&\simeq \langle(\ell = \bigwedge_{r_p \in P} r(\ell) \wedge \bigwedge_{r_n\in\iota^{-1}(\ell)} \neg r(\ell))_{\ell\in L} \,|\, ..\rangle^{L_r} \wedge \bigwedge_{r_p\in P} \langle L'_p \,|\, \varsigma_p\rangle^{L_r} \wedge \bigwedge_{r_n\in\iota^{-1}(_)} \neg\langle L'_n \,|\, \varsigma_n\rangle^{L_r} \\
&\simeq \langle(\ell = \bigwedge_{r_p \in P} r_p(\ell) \wedge \bigwedge_{r_n\in\iota^{-1}(\ell)} \neg r_n(\ell))_{\ell\in L} \,|\, ..\rangle^{L_r} \\
&\quad \wedge \bigwedge_{r_p\in P_{\bar{\mathcal{V}}}} \langle L \,|\, \varsigma_p\rangle^{L_r} \wedge \bigwedge_{r_n\in\iota^{-1}(_)\cap N_{\bar{\mathcal{V}}}} \neg\langle L \,|\, \varsigma_n\rangle^{L_r} \wedge \bigwedge_{r_p\in P_{\mathcal{V}}} \langle L_p \,|\, \varsigma_p\rangle^{L_r} \wedge \bigwedge_{r_n\in\iota^{-1}(_)\cap N_{\mathcal{V}}} \neg\langle L_n \,|\, \varsigma_n\rangle^{L_r}
\end{aligned}$$

Let:

- $r^1_\iota = \langle(\ell = \bigwedge_{r_p\in P} r_p(\ell) \wedge \bigwedge_{r_n\in\iota^{-1}(\ell)} \neg r_n(\ell))_{\ell\in L} \,|\, ..\rangle^{L_r}$;
- $r^2_\iota = \bigwedge_{r_p\in P_{\bar{\mathcal{V}}}} \langle L \,|\, \varsigma_p\rangle^{L_r} \wedge \bigwedge_{r_n\in\iota^{-1}(\_)\cap N_{\bar{\mathcal{V}}}} \neg\langle L \,|\, \varsigma_n\rangle^{L_r}$;
- $r^3_\iota = \bigwedge_{r_p\in P_{\mathcal{V}}} \langle L_p \,|\, \varsigma_p\rangle^{L_r} \wedge \bigwedge_{r_n\in\iota^{-1}(\_)\cap N_{\mathcal{V}}} \neg\langle L_n \,|\, \varsigma_n\rangle^{L_r}$.

We can see that r^1_ι is empty *iff* condition (3) holds, r^2_ι is empty *iff* condition (4) does (in the case where $P_{\bar{\mathcal{V}}}$ is empty, notice that the intersection is equal to $\mathbb{1} \vee \bot$), and r^3_ι is empty *iff* condition (5) holds. We directly obtain that if one of the conditions holds, then the row r_ι is empty. We now show that if r_ι is empty, then there is $1 \le i \le 3$ such that r^i_ι is empty.

For this, we suppose that none of the subtypes is empty and build an element $\text{đ} \in [\![r_\iota]\!]^{\text{row}}$.

(1) Since r^1_ι is not empty, for all $\ell \in L$ there is an element $\delta^1_\ell \in [\![\bigwedge_{r_p\in P} r_p(\ell) \wedge \bigwedge_{r_n\in\iota^{-1}(\ell)} \neg r_n(\ell)]\!]^{\text{fld}}$.
(2) Since r^2_ι is not empty, there is an element $\{\!|(\ell = \delta^2_\ell)_{\ell\in L_2}, \_ = \bot^{\emptyset}|\!\}^{V_2} \in [\![r^2_\iota]\!]^{\text{row}}$.
(3) Since r^3_ι is not empty, there is an element $\text{đ}_3 \in [\![r^3_\iota]\!]^{\text{row}}$. However, the restrictions on the set of elements in $[\![r^3_\iota]\!]^{\text{row}}$ only concern their tags so that any element $\text{đ}'$ with $\text{tag}(\text{đ}') = \text{tag}(\text{đ}_3)$ and $\text{dom}(\text{đ}') = \text{dom}(\text{đ}_3)$ is in $[\![r_3]\!]^{\text{row}}$. Let $V_3 = \text{tag}(\text{đ}_3)$.

We build the element $\langle\!|(\ell = \delta^1_\ell)_{\ell\in L}, (\ell = \delta^2_\ell)_{\ell\in L_2\setminus L}, \_ = \bot^{\emptyset}|\!\rangle^{V_3}$. This element belongs to $[\![r_\iota]\!]^{\text{row}}$, which is a contradiction. □

A.4 Subtyping algorithm

Lemma 2.2 (Soundness and completeness of Φ). *Let* $\mathsf{R}_\circ$ *be an open or closed record type,* $V_p \subset \mathcal{V}_r$ *and* N *a set of atomic record types. Then,*

$$\mathsf{R}_\circ \wedge \bigwedge_{\rho\in V_p} \{\mathcal{L}\setminus \mathrm{dom}(\rho)\,|\,\rho\} \leq \bigvee_{\mathsf{R}\in N} \mathsf{R} \iff \mathsf{R}_\circ \leq \mathbb{0} \text{ or } \Phi(\mathsf{R}_\circ, V_p, N).$$

Proof. If $\mathsf{R}_\circ \simeq \mathbb{0}$, the result holds. Otherwise, we prove this by induction on the cardinality of N. In the following, given a variable $\rho \in V_p$, let us write L_ρ for $\mathcal{L} \setminus \mathrm{dom}(\rho)$.

If $N = \emptyset$, the union $\bigvee_{\mathsf{R}\in N} \mathsf{R}$ is empty. So the statement holds if and only if $\mathsf{R}_\circ \simeq \mathbb{0}$, since $\Phi(\mathsf{R}_\circ, V_p, \emptyset) = \texttt{false}$, and $\bigwedge_{\rho\in V_p} \{L_\rho\,|\,\rho\}$ is never empty.

Now, let $N = N' \cup \{\mathsf{R}\}$. Let $L = \mathrm{lab}(\mathsf{R}_\circ)$. R can be decomposed as $\bigwedge_{\ell\in L} \{\ell = \mathsf{R}(\ell)\,|\,..\} \wedge \{L'\,|\,\varsigma\}$, where $L' = L$ if $\varsigma \notin \mathcal{V}$ and $L' = \mathrm{lab}(\mathsf{R})$ otherwise. The left-hand side of the statement is thus equivalent to $\mathsf{R}_\circ \wedge \bigwedge_{\rho\in V_p} \{L_\rho\,|\,\rho\} \wedge (\bigvee_{\ell\in L} \{\ell = \neg\mathsf{R}(\ell)\,|\,..\} \vee \neg\{L'\,|\,\varsigma\}) \leq \bigvee_{n\in N'} \mathsf{R}_n$. We can distribute the intersection over the unions. We must then prove an equivalence between ($\mathsf{R}_\circ \simeq \emptyset$ or $\Phi(\mathsf{R}_\circ, V_p, N)$) and:

$$\forall \ell \in L. \mathsf{R}_\circ \wedge \{\ell = \neg\mathsf{R}(\ell)\,|\,..\} \wedge \bigwedge_{\rho\in V_p} \{L_\rho\,|\,\rho\} \leq \bigvee_{n\in N'} \mathsf{R}_n \tag{16}$$

$$\text{and } \mathsf{R}_\circ \wedge \bigwedge_{\rho\in V_p} \{L_\rho\,|\,\rho\} \wedge \neg\{L'\,|\,\varsigma\} \leq \bigvee_{n\in N'} \mathsf{R}_n \tag{17}$$

We now have to verify that each of these statements are equivalent to $\Phi(\mathsf{R}_\circ, V_p, N)$.

We start with (16). Let $\ell \in L$ and let $\mathsf{R}^\ell_\circ = \mathsf{R}_\circ \wedge \{\ell = \neg\mathsf{R}(\ell)\,|\,..\}$. By the induction hypothesis, we have that $\mathsf{R}^\ell_\circ \wedge \bigwedge_{\rho\in V_p} \{L_\rho\,|\,\rho\} \leq \bigvee_{n\in N'} \mathsf{R}_n \iff \mathsf{R}^\ell_\circ \simeq \mathbb{0}$ or $\Phi(\mathsf{R}^\ell_\circ, V_p, N')$. Since $\mathsf{R}_\circ \not\simeq \mathbb{0}$, $\mathsf{R}^\ell_\circ \simeq \mathbb{0} \iff \mathsf{R}_\circ(\ell) \leq \mathsf{R}(\ell)$.

We continue with (17). We define $\Psi(\mathsf{R}_\circ, V_p, \{L'\,|\,\varsigma\})$ as ($\varsigma = ..$ or $\varsigma = \mathrm{tail}(\mathsf{R}_\circ)$ or $\varsigma \in V_p$). We show that (17) is equivalent to: $\Psi(\mathsf{R}_\circ, V_p, \{L'\,|\,\varsigma\})$ or ($\neg\Psi(\mathsf{R}_\circ, V_p, \{L'\,|\,\varsigma\})$ and $\Phi(\mathsf{R}_\circ, V_p, N')$). It is easy to show using Lemma 2.1 that $\mathsf{R}_\circ \wedge \bigwedge_{\rho\in V_p} \{L_\rho\,|\,\rho\} \wedge \neg\{L'\,|\,\varsigma\}$ is empty *iff* $\Psi(\mathsf{R}_\circ, V_p, \{L'\,|\,\varsigma\})$ holds. In particular, the first condition of Lemma 2.1 never holds since $L' = L$ when $\varsigma \notin \mathcal{V}$. Then, there are two cases.

(1) $\mathsf{R}_\circ \wedge \bigwedge_{\rho\in V_p} \{L_\rho\,|\,\rho\} \wedge \neg\{L'\,|\,\varsigma\} \leq \mathbb{0}$. This means that $\Psi(\mathsf{R}_\circ, V_p, \{L'\,|\,\varsigma\})$ holds, and also that $\mathsf{R}_\circ \wedge \bigwedge_{\rho\in V_p} \{L_\rho\,|\,\rho\} \wedge \neg\{L'\,|\,\varsigma\} \leq \bigvee_{n\in N'} \mathsf{R}_n$.
(2) $\mathsf{R}_\circ \wedge \bigwedge_{\rho\in V_p} \{L_\rho\,|\,\rho\} \wedge \neg\{L'\,|\,\varsigma\} \not\leq \mathbb{0}$. This means that $\Psi(\mathsf{R}_\circ, V_p, \{L'\,|\,\varsigma\})$ does not hold. Thus, there are two possible cases: either (a) $\mathrm{tail}(\mathsf{R}_\circ) = \epsilon$ and $\varsigma = ..$, or (b) $\varsigma = \rho \notin V_p$. We show that $\mathsf{R}_\circ \wedge \bigwedge_{\rho\in V_p} \{L_\rho\,|\,\rho\} \wedge \neg\{L'\,|\,\varsigma\} \leq \bigvee_{\mathsf{R}\in N'} \mathsf{R}$ is equivalent to $\mathsf{R}_\circ \wedge \bigwedge_{\rho\in V_p} \{L_\rho\,|\,\rho\} \leq \bigvee_{\mathsf{R}\in N'} \mathsf{R}$. From this, we use the induction hypothesis to obtain the equivalence with $\Phi(\mathsf{R}_\circ, V_p, N')$ (since $\mathsf{R}_\circ \not\simeq \mathbb{0}$).
The right-to-left implication of the equivalence is trivial. For the converse implication, we use Lemma 2.1 on $\mathsf{R}_\circ \wedge \bigwedge_{\rho\in V_p} \{L_\rho\,|\,\rho\} \leq \neg\{L'\,|\,\varsigma\} \vee \bigvee_{\mathsf{R}\in N'} \mathsf{R}$. By the hypothesis (a) and (b) in the corresponding cases, the second and third conditions of that lemma never hold. Thus, we have by hypothesis that $\forall\iota : N' \cup \{L'\,|\,\varsigma\} \to L \cup \{\_\}.\exists\ell \in L.\mathsf{R}_\circ(\ell) \leq \bigvee_{\mathsf{R}\in\iota^{-1}(\ell)} \mathsf{R}(\ell)$. The implication holds because $\forall\ell \in L.\{L'\,|\,\varsigma\}(\ell) = \mathbb{1} \vee \bot$.

Summing up, we have proved by induction that if $\mathsf{R}_\circ \wedge \bigwedge_{\rho\in V_p} \{L_\rho\,|\,\rho\}$ is not empty, checking that it is a subtype of $\bigvee_{n\in N} \mathsf{R}_n$ is equivalent to checking both these two propositions:

(1) $\forall\ell \in L\ .\ (\ (\mathsf{R}_\circ(\ell) \leq \mathsf{R}(\ell)) \quad \text{or} \quad \Phi(\mathsf{R}_\circ \wedge \{\ell = \neg\mathsf{R}(\ell)\,|\,..\}, V_p, N')\ )$

(2) $(\Psi(\mathsf{R}_\circ, V_p, \{L' \,|\, \varsigma\}))$ or $(\neg\Psi(\mathsf{R}_\circ, V_p, \{L' \,|\, \varsigma\})$ and $\Phi(\mathsf{R}_\circ, V_p, N'))$

To conclude, notice that if $(\neg\Psi(\mathsf{R}_\circ, V_p, \{L' \,|\, \varsigma\})$ and $\Phi(\mathsf{R}_\circ, V_p, N'))$ holds, then by induction hypothesis we have $\mathsf{R}_\circ \wedge_{\rho \in V_p} \{L_\rho \,|\, \rho\} \leq \bigvee_{n \in N'} \mathsf{R}_n$ and, *a fortiori*, $\mathsf{R}_\circ \wedge \bigwedge_{\rho \in V_p} \{L_\rho \,|\, \rho\} \leq \mathsf{R} \vee \bigvee_{n \in N'} \mathsf{R}_n$. It is therefore useless to check the other proposition (1) above, which thus must be checked only when $\Psi(\mathsf{R}_\circ, V_p, \{L' \,|\, \varsigma\})$ holds. This yields

$$\begin{array}{l} \quad (\; \neg\Psi(\mathsf{R}_\circ, V_p, \{L' \,|\, \varsigma\}) \text{ and } \; \Phi(\mathsf{R}_\circ, V_p, N') \;) \\ \text{or } (\; \Psi(\mathsf{R}_\circ, V_p, \{L' \,|\, \varsigma\}) \text{ and } \; (\forall \ell \in L \,.\, (\mathsf{R}_\circ(\ell) \leq \mathsf{R}(\ell) \;\text{ or }\; \Phi(\mathsf{R}_\circ \wedge \{\ell = \neg\mathsf{R}(\ell) \,|\, ..\}, V_p, N'))) \end{array}$$

which corresponds to the second clause of the definition of Φ. ☐

Proposition 2.3. *The subtyping algorithm terminates. As a corollary, subtyping is decidable.*

Proof. The number of disjoint types in a DNF is finite. Also, the preprocessing of a conjunction of record types can be defined for all such types and computed in a finite number of steps. Finally, in function Φ, the number of elements in the third parameter decreases at each recursive call. Moreover, the subtyping relation on non-record types is decidable [Castagna and Xu 2011], from which we get decidability of the subtyping relation on field types. ☐

A.5 Substitutions

Lemma A.6. *Let $[\![\cdot]\!]^{\boldsymbol{q}}_\eta$ be the appropriate interpretation among $[\![\cdot]\!]^{\mathrm{q}}_\eta$, $[\![\cdot]\!]^{\mathrm{qfld}}_\eta$ and $[\![\cdot]\!]^{\mathrm{qrow}}_\eta$. For every T, σ and η, if η' is defined by $\eta'(X) = [\![\sigma(X)]\!]^{\boldsymbol{q}}_\eta$, then $[\![T\sigma]\!]^{\boldsymbol{q}}_\eta = [\![T]\!]^{\boldsymbol{q}}_{\eta'}$.*

Proof. For an arbitrary σ and η, we show that

$$\forall T \in \mathcal{T}_\perp \cup \mathcal{R}. \forall D \in \mathcal{D}_\perp \cup \mathcal{D}_{\mathrm{row}}. (D : T\sigma)^{\mathrm{q}}_\eta \iff (D : T)^{\mathrm{q}}_{\eta'}$$

by induction on (D, T) and with η' defined as before. We detail two cases, the others are straightforward.

- $T = \alpha$ and $D = d$. On the left, we have $(d : \alpha\sigma)^{\mathrm{q}}_\eta = (d : \sigma(\alpha))^{\mathrm{q}}_\eta$ and on the right $(d : \alpha)^{\mathrm{q}}_{\eta'} = d \in \eta'(\alpha) = d \in [\![\sigma(\alpha)]\!]^{\mathrm{q}}_\eta = (d : \sigma(\alpha))^{\mathrm{q}}$.
- $T = \{(\ell = \tau_\ell)_{\ell \in L} \,|\, \rho\}$ and $D = \mathrm{Rec}\ \{\!\!\langle (\ell = \delta_\ell)_{\ell \in L_d}, \_ = \perp^{\emptyset} \rangle\!\!\}^V$. The case for rows is similar. We have $T\sigma = \{(\ell = \tau_\ell)_{\ell \in L} \,|\, ..\} \wedge \{L \,|\, \sigma(\rho)\}$. Let $\bar{d} = \{\!\!\langle (\ell = \delta_\ell)_{\ell \in L_d \setminus L}, \_ = \perp^{\emptyset} \rangle\!\!\}^V_L$. On the left, we have:

  $$(D : T\sigma)^{\mathrm{q}}_\eta = (\forall \ell \in L_d. (\delta_\ell : T(\ell)\sigma)^{\mathrm{q}}_\eta) \text{ and } (\forall \ell \notin L_d. (\perp^{\emptyset} : T(\ell)\sigma)^{\mathrm{q}}_\eta \text{ and } (\bar{d} : \sigma(\rho))^{\mathrm{q}}_\eta$$

  By induction hypothesis, the first two conditions are equivalent to $\forall \ell \in L_d. (\delta_\ell : T(\ell))^{\mathrm{q}}_{\eta'}$ and $\forall \ell \notin L_d. (\perp^{\emptyset} : T(\ell))^{\mathrm{q}}_{\eta'}$. By the same reasoning as in the previous case, the last condition is equivalent to $(\bar{d} : \rho)^{\mathrm{q}}_{\eta'}$, which altogether give $(D : T)^{\mathrm{q}}_{\eta'}$. ☐

Proposition 2.4. *If $t_1 \leq t_2$, then $t_1\sigma \leq t_2\sigma$ for any row substitution σ.*

Proof. By definition, $t_1 \leq t_2 \iff [\![t_1 \setminus t_2]\!] = \emptyset$. By Lemma A.4, this is equivalent to $\forall \eta. [\![t_1 \setminus t_2]\!]^{\mathrm{q}}_\eta$. In particular, this holds for η' defined as in Lemma A.6, so that $[\![t_1 \setminus t_2]\!]^{\mathrm{q}}_{\eta'} = \emptyset$. By Lemma A.6, this implies $[\![(t_1 \setminus t_2)\sigma]\!]^{\mathrm{q}}_\eta = \emptyset$ which means $t_1\sigma \leq t_2\sigma$. ☐

B APPENDIX FOR LANGUAGE

B.1 Syntax and semantics

Definition B.1 (Top-level variables). *The top-level variables of a type (resp. field type, row) are defined as* $\mathrm{tlv}(t) = \mathrm{tlv}'(t) \cap \mathcal{V}_t$ *(resp.* $\mathrm{tlv}(\tau) = \mathrm{tlv}'(\tau) \cap \mathcal{V}_f$, $\mathrm{tlv}(r) = \mathrm{tlv}'(r) \cap \mathcal{V}_r$*).*

$$
\begin{array}{ll}
\mathrm{tlv}'(\alpha) = \{\alpha\} & \\
\mathrm{tlv}'(\theta) = \{\theta\} & \\
\mathrm{tlv}'(\langle \ell = \tau, \ldots, \ell = \tau \,|\, \rho \rangle^L) = \mathrm{tlv}'(\{\ell = \tau, \ldots, \ell = \tau \,|\, \rho\}) = \{\rho\} & \\
\mathrm{tlv}'(T_1 \vee T_2) = \mathrm{tlv}'(T_1) \cup \mathrm{tlv}'(T_2) & \\
\mathrm{tlv}'(\neg T) = \mathrm{tlv}'(T) & \\
\mathrm{tlv}'(T) = \emptyset & \textit{otherwise}
\end{array}
$$

Lemma B.1. *If* $t_1 \leq t_2$ *with* $t_2 \leq \{\,|\,..\}$ *and* $\ell \# t_1$ *and* $\ell \# t_2$*, then* $t_1 \backslash \ell \leq t_2 \backslash \ell$*.*

Proof. We show that for any $t \leq \{\,|\,..\}$, we have $[\![t\backslash\ell]\!] = \{ɗ\backslash\ell \mid (\mathrm{Rec}(ɗ)^V : t)\}$, where $\langle\!|(\ell = \tau_\ell)_{\ell\in L}, \_ = \bot^\emptyset |\!\rangle^V_\emptyset \backslash \ell = \langle\!|(\ell = \tau_\ell)_{\ell \in L\smallsetminus\ell}, \_ = \bot^\emptyset |\!\rangle^V_{\{\ell\}}$. We start with $\{ɗ\backslash\ell \mid (\mathrm{Rec}(ɗ)^V : t)\} \subseteq [\![t\backslash\ell]\!]$. Let $(\mathrm{Rec}(ɗ)^V : t)$ and t in DNF. The proof is by induction on the top-level type connectives of t. Let $ɗ = \langle\!|(\ell = \delta_\ell)_{\ell\in L}, \_ = \bot^\emptyset |\!\rangle^V_\emptyset$, with $\ell \in L$.

- $t = \{\ell = \tau_\ell \,|\, r\}$ with r atomic. Then $t\backslash\ell = r$ and it is clear that $(ɗ\backslash\ell : r)$.
- $t = \neg\{\ell = \tau_\ell \,|\, r\}$ with r atomic. If $\tau_\ell = \mathbb{1} \vee \bot$, then $t\backslash\ell = \neg r$ and there is $\ell' \neq \ell$ such that $(\delta_\ell : r(\ell))$ is false, or $(\bot^\emptyset : r(\ell))$ is false. Thus, $(ɗ\backslash\ell : \neg r)$. Otherwise, $t\backslash\ell = \langle\,|\,..\rangle^\ell$ and any $(ɗ\backslash\ell : t\backslash\ell)$.
- If $t = t_1 \wedge t_2$, we have $t\backslash\ell = t_1\backslash\ell \wedge t_2\backslash\ell$. By induction hypothesis, we have $(ɗ\backslash\ell : t_i\backslash\ell)$ for $i \in \{1,2\}$, so $(ɗ\backslash\ell : t\backslash\ell)$.
- If $t = t_1 \vee t_2$, we have $t\backslash\ell = t_1\backslash\ell \vee t_2\backslash\ell$. By induction hypothesis, there is $i \in \{1,2\}$ such that $(ɗ\backslash\ell : t_i\backslash\ell)$. By induction hypothesis, we have $(ɗ\backslash\ell : t\backslash\ell)$.

Now, we consider $[\![t\backslash\ell]\!] \subseteq \{ɗ\backslash\ell \mid (\mathrm{Rec}(ɗ)^V : t)\}$. Let $(ɗ_\ell : t\backslash\ell)$. Let $t = \bigvee_{i\in I} \bigwedge_{\mathsf{R}\in P_i} \mathsf{R} \wedge \bigwedge_{\mathsf{R}\in N_i} \neg\mathsf{R}$. By a reasoning similar to Lemma C.4 and since $\ell\#t$, we have $t \simeq \bigvee_{N'\subseteq N_i} t_{N'} = \bigvee_{i\in I}\bigvee_{N'\subseteq N_i}(\{\ell = \bigwedge_{\mathsf{R}\in P_i}\mathsf{R}(\ell) \wedge \bigwedge_{\mathsf{R}\in N'}\neg\mathsf{R}(\ell) \,|\, ..\} \wedge \bigwedge_{\mathsf{R}\in P_i}\{\ell \,|\, \mathsf{R}\backslash\ell\} \wedge \bigwedge_{\mathsf{R}\in N_i\smallsetminus N'}\neg\{\ell \,|\, \mathsf{R}\backslash\ell\})$. For each N', we have $t_{N'}\backslash\ell = \bigwedge_{\mathsf{R}\in P_i}\{\ell \,|\, \mathsf{R}\backslash\ell\} \wedge \bigwedge_{\mathsf{R}\in N_i\smallsetminus N'}\neg\{\ell \,|\, \mathsf{R}\backslash\ell\})$. If $t \leq \mathbb{0}$, then by definition $t\backslash\ell$ is also. Otherwise, by hypothesis, there are $i \in I$ and N' such that $(ɗ_\ell : t_{N'}\backslash\ell)$. Let δ be such that $(\delta_\ell : \bigwedge_{\mathsf{R}\in P_i}\mathsf{R}(\ell) \wedge \bigwedge_{\mathsf{R}\in N'}\neg\mathsf{R}(\ell))$. We take $ɗ$ to be $ɗ_\ell$ completed by $\ell = \delta$ and we have $(\mathrm{Rec}(ɗ)^{\mathrm{tag}(ɗ_\ell)} : t')$. Now, we show that for all $t \leq \{\,|\,..\}$, $t\backslash\ell \simeq \mathrm{split}(t)\backslash\ell$, where $\mathrm{split}(t)$ is the type obtained by the previous decomposition. First, let $t = \bigwedge_{\mathsf{R}\in P}\mathsf{R} \wedge \bigwedge_{\mathsf{R}\in N}\neg\mathsf{R}$. The proof is by induction on $|N|$.

- $N = \emptyset$. Then, $\mathrm{split}(t) = \{\ell = \bigwedge_{\mathsf{R}\in P}\mathsf{R}(\ell) \,|\, ..\} \wedge \bigwedge_{R\in P}\{\ell \,|\, \mathsf{R}\backslash\ell\}$ and $\mathrm{split}(t)\backslash = \bigwedge_{R\in P}\mathsf{R}\backslash\ell = t\backslash\ell$.
- $N = N_0 \cup \{\neg\{\ell = \tau \,|\, r\}\}$. Let $t_0 = \bigwedge_{\mathsf{R}\in P}\mathsf{R} \wedge \bigwedge_{\mathsf{R}\in N_0}\neg\mathsf{R}$. The type t is decomposed as follows:

$$
\begin{aligned}
t \simeq \bigvee_{N'\subseteq N_0} \Big(& \{\ell = \bigwedge_{\mathsf{R}\in P}\mathsf{R}(\ell) \wedge \bigwedge_{\mathsf{R}\in N'}\mathsf{R}(\ell) \wedge \neg\tau \,|\, ..\} \wedge \bigwedge_{\mathsf{R}\in P}\{\ell \,|\, \mathsf{R}\backslash\ell\} \wedge \bigwedge_{\mathsf{R}\in N_0\smallsetminus N'}\neg\{\ell \,|\, \mathsf{R}\backslash\ell\} \\
\vee\, & \{\ell = \bigwedge_{\mathsf{R}\in P}\mathsf{R}(\ell) \wedge \bigwedge_{\mathsf{R}\in N'}\mathsf{R}(\ell) \,|\, ..\} \wedge \bigwedge_{\mathsf{R}\in P}\{\ell \,|\, \mathsf{R}\backslash\ell\} \wedge \bigwedge_{\mathsf{R}\in N_0\smallsetminus N'}\neg\{\ell \,|\, \mathsf{R}\backslash\ell\} \wedge \neg\{\ell \,|\, r\}\Big)
\end{aligned}
$$

There are two cases.

(1) $\tau = \mathbb{1} \vee \bot$. Then, $(\neg\{\ell \,|\, r\})\backslash\ell = \neg r$ and

$$
\mathrm{split}(t) = \bigvee_{N'\subseteq N_0}\Big(\{\ell = \bigwedge_{\mathsf{R}\in P}\mathsf{R}(\ell) \wedge \bigwedge_{\mathsf{R}\in N'}\mathsf{R}(\ell) \,|\, ..\} \wedge \bigwedge_{\mathsf{R}\in P}\{\ell \,|\, \mathsf{R}\backslash\ell\} \wedge \bigwedge_{\mathsf{R}\in N_0\smallsetminus N'}\neg\{\ell \,|\, \mathsf{R}\backslash\ell\} \wedge \neg\{\ell \,|\, r\}\Big)
$$

So by induction hypothesis,

$$\text{split}(t)\backslash\ell = \bigvee_{N'\subseteq N_0}\Big(\bigwedge_{\mathsf{R}\in P}\mathsf{R}\backslash\ell \wedge \bigwedge_{\mathsf{R}\in N_0\smallsetminus N'}\neg\mathsf{R}\backslash\ell \wedge \neg r\Big) \simeq \text{split}(t_0)\backslash\ell \wedge \neg r \simeq t_0\backslash\ell \wedge \neg r = t\backslash\ell$$

(2) $\tau \neq \mathbb{1} \vee \bot$. Then, $(\neg\{\ell = \tau\,|\,r\})\backslash\ell = \langle|\,..\rangle^{\{\ell\}}$ and by induction hypothesis

$$\text{split}(t)\backslash\ell = \bigvee_{N'\subseteq N_0}\Big(\bigwedge_{\mathsf{R}\in P}\{\ell\,|\,\mathsf{R}\backslash\ell\} \wedge \bigwedge_{\mathsf{R}\in N_0\smallsetminus N'}\neg\{\ell\,|\,\mathsf{R}\backslash\ell\} \vee \bigwedge_{\mathsf{R}\in P}\{\ell\,|\,\mathsf{R}\backslash\ell\} \wedge \bigwedge_{\mathsf{R}\in N_0\smallsetminus N'}\neg\{\ell\,|\,\mathsf{R}\backslash\ell\} \wedge \neg\{\ell\,|\,r\}\Big)$$

$$\simeq \text{split}(t_0)\backslash\ell \simeq t_0\backslash\ell \simeq t_0\backslash\ell \wedge \langle|\,..\rangle^{\{\ell\}} = t\backslash\ell$$

Now, if $t = \bigvee_{i\in I} t_i$ where the t_i's are conjunctions, we have $t\backslash\ell = \bigvee_{i\in I} t_i\backslash\ell \simeq \bigvee_{i\in I}\text{split}(t_i)\backslash\ell = \text{split}(t)\backslash\ell$. □

Lemma B.2 (Inversion). *Let $v = \{v_1 \texttt{ with } \ell = v_2\}$. If there is a derivation $\Delta \mid \Gamma \vdash_{\mathcal{D}} v : t$, then there are derivations $\Delta \mid \Gamma \vdash_{\mathcal{D}} v_1 : t_1 \leq \{\ell = \bot\,|\,..\}$ and $\Delta \mid \Gamma \vdash_{\mathcal{D}} v_2 : t_2$ such that $\{\ell = t_2\,|\,t_1\backslash\ell\} \leq t$.*

Proof. By induction on $\Delta \mid \Gamma \vdash_{\mathcal{D}} v : t$, with a case analysis on the last rule used, that has to be of (Ext), (Inter) or (Sub).

(Ext) Straightforward.

(Inter) We apply the induction hypothesis twice. Since both types obtained are supertypes of $\{\ell = t_2\,|\,t_1\backslash\ell\}$, their intersection is also.

(Sub) By induction hypothesis and transitivity of subtyping. □

Lemma B.3 (Subject reduction). *Let e be an expression and t a type. If $\Delta \mid \Gamma \vdash_{\mathcal{D}} e : t$ and $e \rightsquigarrow e'$, then $\Delta \mid \Gamma \vdash_{\mathcal{D}} e' : t$.*

Proof. The proof is by induction on the derivation of $\Delta \mid \Gamma \vdash_{\mathcal{D}} e : t$ and by a case analysis on the last rule used in the derivation of $\Delta \mid \Gamma \vdash_{\mathcal{D}} e : t$. We detail the cases related to the rules for records, for the rest, see *e.g.* [Frisch et al. 2008].

(Emp) $e = \{\}$, so it does not reduce.

(Ext) $e = \{e_1 \texttt{ with } \ell = e_2\}$. Necessarily, we have $e' = \{e_1' \texttt{ with } \ell = e_2\}$ or $e' = \{e_1 \texttt{ with } \ell = e_2'\}$ and this is direct by induction hypothesis.

(Sel) If $e = e_0.\ell \rightsquigarrow e_0'.\ell = e'$ or if $e = \{e_1 \texttt{ with } \ell' = e_2\}.\ell$ and the reduction occurs in e_1 or e_2, this is direct by induction hypothesis. Otherwise, we have $e = \{v \texttt{ with } \ell' = v'\}.\ell$ and $\Delta \mid \Gamma \vdash_{\mathcal{D}} \{v \texttt{ with } \ell' = v'\} : \{\ell = t\,|\,..\}$. By Lemma B.2, there are derivations $\Delta \mid \Gamma \vdash_{\mathcal{D}} v : t_1 \leq \{\ell' = \bot\,|\,..\}$ and $\Delta \mid \Gamma \vdash_{\mathcal{D}} v' : t_2$ such that $\{\ell' = t_2\,|\,t_1\backslash\ell'\} \leq \{\ell = t\,|\,..\}$.
- If $\ell = \ell'$, then $e' = v'$. Since $\{\ell = t_2\,|\,t_1\backslash\ell\} \leq \{\ell = t\,|\,..\}$, we have $t_2 \leq t$ and we conclude by (Sub).
- If $\ell \neq \ell'$, then $e' = v.\ell$. Since $\{\ell' = t_2\,|\,t_1\backslash\ell'\} \leq \{\ell = t\,|\,..\}$, we have $t_1 \leq \{\ell = t, \ell' = \bot\,|\,..\}$. We conclude by rules (Sub) and (Sel).

(Del) If $e = \{\}\backslash\ell \rightsquigarrow \{\}$, since $\{\}\backslash\ell = \{\}$, we can use the same derivation. If $e = \{e_1 \texttt{ with } \ell' = e_2\}\backslash\ell$ and the reduction occurs in e_1 or e_2, this is direct by induction hypothesis. Otherwise, we have $e = \{v \texttt{ with } \ell' = v'\}\backslash\ell$. By Lemma B.2, there are derivations $\Delta \mid \Gamma \vdash_{\mathcal{D}} v : t_1 \leq \{\ell' = \bot\,|\,..\}$ and $\Delta \mid \Gamma \vdash_{\mathcal{D}} v' : t_2$ such that $\{t_1 \texttt{ with } \ell' = t_2\} \leq t$.
- If $\ell = \ell'$, then $e' = v\backslash\ell$. Since $t_1 \leq \{\ell = \bot\,|\,..\}$, v can be typed with $t_1 = t_1\backslash\ell = \{t_1 \texttt{ with } \ell = t_2\}\backslash\ell$. We conclude because $\{t_1 \texttt{ with } \ell = t_2\} \leq t$.
- If $\ell \neq \ell'$, then $e' = \{v\backslash\ell \texttt{ with } \ell' = v'\}$. Since $\ell\#t_1$, there is a derivation $\Delta \mid \Gamma \vdash_{\mathcal{D}} v\backslash\ell : t_1\backslash\ell$. Since $\{t_1\backslash\ell \texttt{ with } \ell' = t_2\} = \{t_1 \texttt{ with } \ell' = t_2\}\backslash\ell$, we conclude by rule (Ext).

(Inst) Direct by induction hypothesis. □

Lemma B.4 (Generation for values). *Let v be a value such that $\Delta \mid \Gamma \vdash_{\mathscr{D}} v : t$ with $t \leq \{\mid ..\}$. Then v has one of the forms* `{}` *or* $\{v_1$ `with` $\ell = v_2\}$.

Proof. It is easy to see that a derivation for v is obtained by a rule (Emp) followed by rules (Ext), (Inter) or (Sub). Remark that there are no rules (Inst) because it is impossible to derive a polymorphic type t for v, in particular since for instance $\{\mid \epsilon\} \not\leq \{\mid \rho\}$. Moreover, we can show by induction on the depth of the derivation that if $\Delta \mid \Gamma \vdash_{\mathscr{D}} v : t$ is derivable, then $t \not\simeq \mathbb{0}$. The proof is by induction on the derivation.

(Emp) This is the base case, where $v = \{\}$.

(Ext) v is of the second form.

(Inter) t_1 and t_2 are reducible to disjunctive normal forms $t_1^{\mathsf{R}} \wedge t_1'$ and $t_2^{\mathsf{R}} \wedge t_2'$, such that $t_1^{\mathsf{R}}, t_2^{\mathsf{R}} \leq \{\mid ..\}$ and by hypothesis, $t_1' \wedge t_2' \leq \mathbb{0}$. We can show by induction on the derivation of v that this last property does not hold if v is not a record expression.

(Sub) We have $\Delta \mid \Gamma \vdash_{\mathscr{D}} v : t' \leq t$. $t' \not\leq \mathbb{0}$ since v is a value, so we can apply the induction hypothesis. □

Lemma B.5 (Progress). *Let e be a well-typed closed expression, that is, $\emptyset \mid \emptyset \vdash_{\mathscr{D}} e : t$ for some t. If e is not a value, then there exists an expression e' such that $e \rightsquigarrow e'$.*

Proof. The proof is by induction on the derivation of $\Delta \mid \Gamma \vdash_{\mathscr{D}} e : t$ and by a case analysis on the last rule used in the derivation of $\Delta \mid \Gamma \vdash_{\mathscr{D}} e : t$. We detail the cases related to records, for the rest, see *e.g.* [Frisch et al. 2008].

(Emp) $e = \{\}$ is a value.

(Ext) $e = \{e_1$ `with` $\ell = e_2\}$. If e_1 or e_2 can be reduced, e can also. Otherwise, e_1 and e_2 are values by induction and so is e.

(Del) $e = e_0 \backslash \ell$. If e_0 can be reduced, so can e. Otherwise, we have by induction hypothesis that e_0 is a value. By Lemma B.4, either $e_0 = \{v$ `with` $\ell' = v'\}$ and e reduces with $[\mathsf{R}_{\mathsf{del}}^{=}]$ or $[\mathsf{R}_{\mathsf{del}}^{\neq}]$, or $e_0 = \{\}$ and e reduces with $[\mathsf{R}_{\mathsf{emp}}]$.

(Sel) $e = e_0.\ell$. If e_0 can be reduced, so can e. Otherwise, we have by induction hypothesis that e_0 is a value. By Lemma B.4, $e_0 = \{v$ `with` $\ell' = v'\}$ or $e_0 = \{\}$. In the first case, e reduces with $[\mathsf{R}_{\mathsf{del}}^{=}]$ or $[\mathsf{R}_{\mathsf{del}}^{\neq}]$. The second case is impossible, since there is no derivation for e_0 of type $\{\ell = t \mid ..\}$.

(Inst) Directly by induction hypothesis. □

Theorem 3.1 (Type soundness). *Let e be a well-typed closed expression, that is, $\emptyset \mid \emptyset \vdash_{\mathscr{D}} e : t$ for some t. Then either e diverges or it reduces to a value of type t.*

Proof. Consequence of Lemmas B.3 and B.5. □

B.2 Algorithmic type system

B.2.1 Field selection. We remind here the definition of the field selection operator.

Definition 3.1 (Field Selection). *Let $t \leq \{\ell = \mathbb{1} \mid ..\}$ be a DNF. We define the selection of the field ℓ of t as $(\bigvee_{i\in I} t_i).\ell \overset{\text{def}}{=} \bigvee_{i\in I} t_i.\ell$ and*

$$(\bigwedge_{\mathsf{R}\in P} \mathsf{R} \wedge \bigwedge_{\mathsf{R}\in N} \neg\mathsf{R} \wedge \bigwedge_{\alpha\in V_p} \alpha \wedge \bigwedge_{\alpha\in V_n} \neg\alpha) \overset{\text{def}}{=} \bigvee_{N'\subseteq N} \left(\bigwedge_{\mathsf{R}\in P} \mathsf{R}(\ell) \wedge \bigwedge_{\mathsf{R}\in N'} \neg\mathsf{R}(\ell) \right)$$

For an arbitrary type $t \leq \{\ell = \mathbb{1} \mid ..\}$, we define $t.\ell \overset{\text{def}}{=} (\mathrm{dnf}(t \wedge \{\mid ..\}))$.

Lemma B.6. *Let $t = \bigwedge_{\mathsf{R}\in P} \mathsf{R} \wedge \bigwedge_{\mathsf{R}\in N} \neg\mathsf{R}$ and $\rho \in \mathcal{V}_r$ such that $\rho \notin \mathrm{tlv}(\mathrm{row}(t))$. Then, $t \wedge \{\mathcal{L} \setminus \mathrm{dom}(\rho) \mid \rho\} \leq \mathbb{0} \iff t \leq \mathbb{0}$.*

Proof. Because $\rho \notin \mathrm{tlv}(\mathrm{row}(t))$, the set of elements $\mathrm{Rec}(\mathit{d}) \in [\![t..]\!]$ is exactly the elements of $[\![t]\!]$, where ρ is added from $\mathrm{tag}(\mathit{d})$. ◻

Lemma B.7. *Let* $t \leq \{\ell = \mathbb{1} \mid ..\}$. *Then, for all* u, $t \leq \{\ell = u \mid ..\} \iff t.\ell \leq u$. *In particular,* $t.\ell \leq \mathbb{1}$ *and* $t \leq \{\ell = t.\ell \mid ..\}$.

Proof. By Lemma C.4, t is equivalent to:

$$\bigvee_{i\in I} \bigvee_{N' \subseteq N_i} \bigvee_{N'_{\mathcal{V}} \subseteq N' \cap N_{\mathcal{V}}} \Bigg(\{\ell = \bigwedge_{\mathsf{R}\in P} \mathsf{R}(\ell) \wedge \bigwedge_{\mathsf{R}\in N \setminus N'} \neg \mathsf{R}(\ell) \mid ..\} \\ \wedge \bigwedge_{\mathsf{R}\in P} \{\ell = \mathbb{1} \vee \bot \mid \mathsf{R} \backslash \ell\} \wedge \bigwedge_{\mathsf{R}\in N' \setminus N'_{\mathcal{V}}} \neg \{\ell = \mathbb{1} \vee \bot \mid \mathsf{R}\backslash \ell\} \\ \wedge \bigwedge_{\mathsf{R}\in P_{\mathcal{V}}} \{\,\mathrm{lab}(\mathsf{R}) \mid \mathrm{tail}(\mathsf{R})\} \wedge \bigwedge_{\mathsf{R}\in N'_{\mathcal{V}}} \neg \{\,\mathrm{lab}(\mathsf{R}) \mid \mathrm{tail}(\mathsf{R})\} \Bigg)$$

where $P_{\mathcal{V}} = \{\mathsf{R} \in P \mid \mathrm{tail}(\mathsf{R}) = \rho \text{ and } \ell \in \mathrm{dom}(\rho)\}$, similarly for $N_{\mathcal{V}}$, and because for any atom R, $\mathrm{row}(\mathsf{R}) \backslash \{\ell\} = \mathsf{R}\backslash \ell$. Then, for any u it is clear that $t \leq \{\ell = u \mid ..\}$ is equivalent to $t.\ell \leq u$. ◻

Corollary B.8. *Let* $t \leq \{\ell = \mathbb{1} \mid ..\}$ *and* $[\sigma_i]_{i\in I}$ *be a set of substitutions. Then* $(\bigwedge_{i\in I} t\sigma_i).\ell \leq \bigwedge_{i\in I} t.\ell\sigma$.

B.2.2 Taming non-structural rules. To prove soundness and completeness of the algorithmic type system, we go through an intermediate type system, where the intersection rule is n-ary, and the introduction of a term variable can perform a renaming of polymorphic variables. The introduction of a renaming aims at eliminating trivial instantiations (only renamings) in the uses of (Inst). More details are given by Castagna et al. [2024b, Section I.1].

$$(\textsc{Var})\ \frac{}{\Delta \mid \Gamma \vdash_m x : \Gamma(x)\sigma}\ \begin{array}{l} x \in \mathrm{dom}(\Gamma) \\ \mathrm{dom}(\sigma) \cap \Delta = \emptyset \\ \text{and } \sigma \text{ is a renaming} \end{array} \qquad (\textsc{Inter})\ \frac{(\Delta \mid \Gamma \vdash_m e : t_i)_{i\in I}}{\Delta \mid \Gamma \vdash_m e : \bigwedge_{i\in I} t_i}\ |I| > 0$$

Rules different from (Inst) and (Var) are the same as in the declarative type system.

Lemma B.9. $\Delta \mid \Gamma \vdash_m e : t \iff \Delta \mid \Gamma \vdash_{\mathcal{D}} e : t$.

Proof. The left-to-right direction is straightforward since the rules in the intermediate system generalize the ones of the declarative system. The right-to-left direction is obtained by replacing instances of (Var) by instances of (Var) followed by (Inst) in the declarative system, and by replacing occurences of n-ary intersections by $n - 1$ (Inter) nodes. ◻

Next, we want to restrict derivations in the intermediate system to a canonical form, where the apparition of (Inst) and (Sub) nodes is controlled. For convenience, we introduce the following rule macro:

$$(\sqsubseteq)\ \frac{\Delta \mid \Gamma \vdash_m e : s}{\Delta \mid \Gamma \vdash_m e : t}\ s \sqsubseteq_\Delta t$$

which is a stands for (Sub) when $s \leq t$ and otherwise for:

$$\text{(Sub)}\ \dfrac{\text{(Inter)}\ \dfrac{\text{(Inst)}\ \dfrac{\Delta \mid \Gamma \vdash_m e : s}{\Delta \mid \Gamma \vdash_m e : s\sigma_i}\ \forall i \in I}{\Delta \mid \Gamma \vdash_m e : \bigwedge_{i\in I} s\sigma_i}}{\Delta \mid \Gamma \vdash_m e : t}$$

Definition B.2 (Canonical derivation). *A derivation is* canonical *if every (Inst) node it contains is part of a ($\sqsubseteq$) pattern and every ($\sqsubseteq$) and (Sub) nodes are either:*

- *The premise of an (Del) or (Sel) node, or*
- *The first premise of a (Ext) node, or*
- *One of the premises of an (Abs) or (App) node.*

Lemma B.10. *A derivation of $\Delta \mid \Gamma \vdash_m e : t$ can be transformed into a derivation $\Delta \mid \Gamma \vdash_m e : t\sigma$, for any renaming σ such that $\mathrm{dom}(\sigma) \cap \Delta = \emptyset$, without changing the structure of the derivation.*

Proof. As in [Castagna et al. 2024b, Lemma I.14]. □

Lemma B.11. *Let I, Δ and $t'_i \sqsubseteq_\Delta t_i$ for all $i \in I$. Then, $\bigwedge_{i\in I} t'_i \sqsubseteq_\Delta \bigwedge_{i\in I} t_i$.*

Proof. As in [Castagna et al. 2024b, Proposition I.15]. □

Lemma B.12. *Let $s' \sqsubseteq_\Delta s$, ℓ and r of domain $\mathcal{L} \setminus \ell$ such that $\mathrm{vars}(s') \cap \mathrm{vars}(r) \subseteq \Delta$. Then, $\{\ell = s' \,|\, r\} \sqsubseteq_\Delta \{\ell = s \,|\, r\}$.*

Proof. Let $\{\sigma_i\}_{i\in I}$ such that $\bigwedge_{i\in I} s'\sigma_i \leq s$, with $\mathrm{dom}(\sigma) \subseteq \mathrm{vars}(s')$. We have $\bigwedge_{i\in I} \{\ell = s' \,|\, r\}\sigma_i \simeq \{\ell = \bigwedge_{i\in I} s'\sigma_i \,|\, r\} \leq \{\ell = s \,|\, r\}$. □

Lemma B.13. *Any derivation of $\Delta \mid \Gamma \vdash_m e : t$ can be transformed into a canonical derivation of $\Delta \mid \Gamma \vdash_m e : t'$, where $t' \sqsubseteq_\Delta t$.*

Proof. By induction on the size of the derivation and through a case analysis on the root of the derivation tree used.

(Inst) or (Sub) We remove the root and let its premise be the new one.

(Inter) Let $t = \bigwedge_{i\in I} s_i$. By induction hypothesis, for all $i \in I$ we have derivations $\Delta \mid \Gamma \vdash_m e : s'_i$ with $s'_i \sqsubseteq_\Delta s_i$. By rule (Inter), we have a derivation of $t' = \bigwedge_{i\in I} s'_i$, and $t' \sqsubseteq_\Delta t$ is verified by Lemma B.11.

(Const), (Var), (Emp) Alredy canonical.

(Ext) Let $e = \{e_1 \texttt{ with } \ell = e_2\}$ and $t = \{\ell = t_2 \,|\, t_1 \setminus \ell\}$. By induction hypothesis, we have canonical derivations $\Delta \mid \Gamma \vdash_m e_1 : t'_1$ and $\Delta \mid \Gamma \vdash_m e_2 : t'_2$ with $t'_1 \sqsubseteq_\Delta t_1$ and $t'_2 \sqsubseteq_\Delta t_2$. By rule ($\sqsubseteq$), we derive $\Delta \mid \Gamma \vdash_m e_1 : t_1$. Let $t' = \{\ell = t'_2 \,|\, \ell\}t_1$. Rule (Ext) node gives a canonical derivation of $\Delta \mid \Gamma \vdash_m e : t'$. We can suppose that the variables in t'_2 and t_1 are disjoint (otherwise we use Lemma B.10), and conclude $t' \sqsubseteq_\Delta t$ by Lemma B.12.

(Abs) By induction hypothesis, for each $i \in I$ we have derivations $\Delta \cup \mathrm{vars}(t) \mid \Gamma, x : t_i \vdash_m e : s'_i$ with $s'_i \sqsubseteq_\Delta s_i$. By rule ($\sqsubseteq$), we have derivations $\Delta \cup \mathrm{vars}(t) \mid \Gamma, x : t_i \vdash_m e : s_i$ and we conclude by rule (Abs).

(App), (Sel), (Del) Similar to the previous case. □

B.2.3 Soundness and completeness.

Theorem 3.2 (Soundness). *If* $\Delta \mid \Gamma \vdash_{\mathcal{A}} e : t$*, then* $\Delta \mid \Gamma \vdash_{\mathcal{D}} e : t$*.*

Proof. By induction on the algorithmic typing derivation. By Lemma B.9, it suffices to give a derivation $\Delta \mid \Gamma \vdash_m e : t$. We proceed by a case analysis on the last rule used in the derivation.

(Const), (Var) Straightforward.

(Abs) By induction hypothesis, for each $i \in I$ we have $\Delta \cup \Delta' \mid \Gamma, x : t_i \vdash_m e : s'_i$. Since $s'_i \sqsubseteq_{\Delta\cup\Delta'} s_i$, we derive $\Delta \cup \Delta' \mid \Gamma, x : t_i \vdash_m e : s_i$ by rule ($\sqsubseteq$). We conclude by rule (Abs) in the declarative system.

(App) By hypothesis, we have $u \in t_1 \bullet_\Delta t_2$, so u is such that there are two substitution sets with $\bigwedge_{j\in J} t_1\sigma_j \leq \bigwedge_{i\in I} t_2\sigma_i \to u$. By induction hypothesis and ($\sqsubseteq$), we obtain $\Delta \mid \Gamma \vdash_m e_1 : \bigwedge_{j\in J} t_1\sigma_j$ and $\Delta \mid \Gamma \vdash_m e_2 : \bigwedge_{i\in I} t_2\sigma_i$. By (Sub), we have $\Delta \mid \Gamma \vdash_m e_1 : \bigwedge_{i\in I} t_2\sigma_i \to u$. We conclude by rule (App) in the declarative system.

(Emp) Straightforward.

(Ext) By induction hypothesis, we have $\Delta \mid \Gamma \vdash_m e : t$, $\Delta \mid \Gamma \vdash_m e' : t'$ and sets of substitutions $[\sigma_i]_{i\in I}$ such that $\bigwedge_{i\in I} t\sigma_i \leq \{\,|\,..\}$ and $r = (\bigwedge_{i\in I} t\sigma_i)\backslash\ell$. By rules (Inst) and (Inter), we have $\Delta \mid \Gamma \vdash_m e : \bigwedge_{i\in I} t\sigma_i$, and we conclude with rule (Del).

(Del) Similar to the case for (Ext), without the derivation of t'.

(Sel) By induction hypothesis, we have $\Delta \mid \Gamma \vdash_m e : t$ and a set of substitutions $[\sigma_i]_{i\in I}$ such that $\bigwedge_{i\in I} t\sigma_i \leq \{\ell = \mathbb{1} \,|\, ..\}$ and $u = (\bigwedge_{i\in I} t\sigma_i).\ell$. By Lemma B.7, we have $\bigwedge_{i\in I} t\sigma_i \leq \{\ell = u \,|\, ..\}$, so we conclude with rule ($\sqsubseteq$) and (Sel). □

Theorem 3.3 (Completeness). *If* $\Delta \mid \Gamma \vdash_{\mathcal{D}} e : t$*, then there is* s *such that* $\Delta \mid \Gamma \vdash_{\mathcal{A}} e : s$ *and* $s \sqsubseteq_\Delta t$*.*

Proof. By Lemmas B.9 and B.13 we transform the input derivation into a canonical derivation $\Delta \mid \Gamma \vdash_m e : t'$, where $t' \sqsubseteq_\Delta t$. The proof is by induction on the derivation (where we use ($\sqsubseteq$) instead of the corresponding pattern). In the end, we obtain $\Delta \mid \Gamma \vdash_{\mathcal{A}} e : s$ with $s \sqsubseteq_\Delta t'$ and thus conclude since by transitivity of $\sqsubseteq_\Delta$, we have $s \sqsubseteq_\Delta t$.

(Const), (Var) Straightforward.

(Abs) By induction hypothesis, for each $i \in I$, we have $\Delta \cup \Delta' \mid \Gamma, x : t_i \vdash_{\mathcal{A}} e : s_i$ with $s'_i \sqsubseteq_\Delta s_i$. We conclude with rule (Abs) in the algorithmic system.

(App) By induction hypothesis, we have $\Delta \mid \Gamma \vdash_{\mathcal{A}} e_1 : t$ and $\Delta \mid \Gamma \vdash_{\mathcal{A}} e_2 : s$, where $t \sqsubseteq_\Delta t_1 \to t_2$ and $s \sqsubseteq_\Delta t_1$. So $t_2 \in t \bullet_\Delta s$ since $(t_1 \to t_2) \cdot t_1 = t_2$.

(Emp) Straightforward.

(Ext) By induction hypothesis, we have $\Delta \mid \Gamma \vdash_{\mathcal{A}} e : s$, a set of substitutions such that $[\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta t \leq \{\ell = \bot \,|\, ..\}$, $\Delta \mid \Gamma \vdash_{\mathcal{A}} e' : s'$ and $s' \sqsubseteq_\Delta t'$. Thus, we have $[\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta \{\ell = \bot \,|\, ..\}$. Since $t \leq \{\ell = \bot \,|\, ..\}$. By hypothesis, we also have $\ell\#t$. Let $r = (\bigwedge_{i\in I} t\sigma_i)\backslash\ell$. By Lemma B.1, we have $r \leq t\backslash\ell$, so that $\{\ell = s' \,|\, r\} \sqsubseteq_\Delta \{\ell = t' \,|\, ..\}t\backslash\ell$.

(Del) By induction hypothesis, we have $\Delta \mid \Gamma \vdash_{\mathcal{A}} e : s$ and a set of substitutions such that $[\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta t \leq \{\,|\,..\}$. Thus, we have $[\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta \{\,|\,..\}$. By hypothesis, we also have $\ell\#t$. Let $r = (\bigwedge_{i\in I} t\sigma_i)\backslash\ell$. By Lemma B.1, we have $r \leq t\backslash\ell$, so that $\{\ell = \bot \,|\, r\} \sqsubseteq_\Delta \{\ell = \bot \,|\, t\backslash\ell\}$.

(Sel) By induction hypothesis, we have $\Delta \mid \Gamma \vdash_{\mathcal{A}} e : s$ and a set of substitutions such that $[\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta \{\ell = t \,|\, ..\}$. Thus, we have $[\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta \{\ell = \mathbb{1} \,|\, ..\}$. Let $u = (\bigwedge_{i\in I} s\sigma_i).\ell$. We have $u \in \amalg^\ell_\Delta(s)$, and by Lemma B.7 $u \leq t$ so $u \sqsubseteq_\Delta t$.

($\sqsubseteq$) Straightforward by induction hypothesis and transitivy of $\sqsubseteq$.

(Sub) Straightforward by induction hypothesis, inclusion of $\leq$ in $\sqsubseteq$ and transitivity of $\sqsubseteq$.

(Inter) By hypothesis, there is I and derivations $\Delta \mid \Gamma \vdash_m e : s_i$ for all $\{s_i\}_{i\in I}$, with $t = \bigwedge_{i\in I} s_i$. Since the derivation is canonical, we know that each of these derivations ends with (the

$$(\textsc{Ext})\ \frac{\Delta \mid \Gamma \vdash e : t \leq \{\ell = \bot \,|\, ..\} \qquad \Delta \mid \Gamma \vdash e' : t'}{\Delta \mid \Gamma \vdash \{e \texttt{ with } \ell = e'\} : \{\ell = t' \,|\, t \backslash \ell\}} \qquad (\textsc{Del})\ \frac{\Delta \mid \Gamma \vdash e : t \leq \{\,|\, ..\}}{\Delta \mid \Gamma \vdash e \backslash \ell : \{\ell = \bot \,|\, t \backslash \ell\}}$$

$$(\textsc{Sel})\ \frac{\Delta \mid \Gamma \vdash e : t}{\Delta \mid \Gamma \vdash e.\ell : t.\ell}$$

The rules (Const), (Var), (Abs), (App), (Emp) are the same as in the algorithmic system.

Fig. 4. Alternative algorithmic system

same kind of) structural rule. According to the other cases, for all $i \in I$ we have derivations $\Delta \mid \Gamma \vdash_m e : s'_i$, where $s'_i \sqsubseteq_\Delta s_i$. By Lemma B.11 we have $t' = \bigwedge_{i \in I} s'_i \sqsubseteq_\Delta \bigwedge_{i \in I} s_i = t$. □

B.2.4 Alternative incomplete type system.

Lemma B.14. *The type system in Fig. 4 is sound with respect to the declarative type system.*

Proof. Similar to the proof of Theorem 3.2. □

C APPENDIX FOR TALLYING

Definition C.1. *Given a type term T, we write* $\mathrm{vars}(T)$ *the set of variables occurring in it. The following equalities hold.*

$$\begin{array}{rcc}
\mathrm{vars}(\alpha) = \{\alpha\} & \mathrm{vars}(T_1 \to T_2) = \mathrm{vars}(T_1) \cup \mathrm{vars}(T_2) & \mathrm{vars}(b) = \emptyset \\
\mathrm{vars}(\theta) = \{\theta\} & \mathrm{vars}(T_1 \vee T_2) = \mathrm{vars}(T_1) \cup \mathrm{vars}(T_2) & \mathrm{vars}(\bot) = \emptyset \\
\mathrm{vars}(\langle (\ell = \tau_\ell)_{\ell \in L_1} \,|\, \varsigma \rangle^{L_2}) = \{\varsigma\} \cap \mathcal{V}_r & \mathrm{vars}(\neg T) = \mathrm{vars}(T) & \mathrm{vars}(\mathbb{0}) = \emptyset \\
\mathrm{vars}(\mathsf{R}) = \mathrm{vars}(\mathrm{row}(\mathsf{R})) & &
\end{array}$$

Definition C.2 (Ordering). *Let V and Δ be sets of variables and L a set of labels. An ordering O_X on V is an injective map from V to $\mathbb{N}$, an ordering O_ℓ on L is an injective map from L to $\mathbb{N}$. An ordering O on V and L is defined as a lexicographic ordering on $\mathbb{N} \times \mathbb{N}$ according to O_X and O_ℓ, where: $O(\rho.\ell) = (O_X(\rho), O_\ell(\ell))$, $O(\rho \backslash L) = (O_X(\rho), O'(\rho, L))$ and $O(X) = (O_X(X), 0))$ otherwise; for all $X_1 \notin \Delta$ and $X_2 \in \Delta$ of the same kind, $O(X_1) < O(X_2)$; and $O'(\rho, L)$ is an integer obtained in a canonical way from the set $L \cup \mathrm{dom}(\rho)$ and different from any $O_\ell(\ell)$.*

C.1 Examples regarding the restriction of solutions to atomic rows

We give two examples where we discuss considering only atomic rows instead of Boolean combinations of them in the grammar. The first one shows that we can find tallying solutions even for unions of records thanks to the unification technique. The second demonstrates that this technique is not enough to recover all desired solutions, and therefore that Boolean combinations of rows, as we adopt them in our system, are welcome.

Example C.1. Let us consider the types of the example on Page 5, that we rewrite as follows: $t = \{p = \mathsf{Int} \,|\, \rho\} \to \{p = \mathsf{Float} \,|\, \rho\}$ and $u = \{s = \texttt{"circle"}, p = \mathsf{Int}, d = \mathsf{Float} \,|\, \epsilon\} \vee \{s = \texttt{"polygon"}, p = \mathsf{Int}, e = \mathsf{Int} \,|\, \epsilon\}$. We would like to be able to unify the parameter of the function with the argument, even though the latter is a union. As explained in [Castagna et al. 2015, §C.2.1], the problem of computing $t \bullet_\emptyset u$ can be reduced to solving $\{(t', \leq, u \to \alpha)\}$, where α is a fresh variable, and $t' = \bigwedge_{i \in I} t\sigma_i$, where the σ_i are renamings of ρ. The cardinality of I will be increased during the search for a solution.

With a cardinality $|I| = 1$, we need in particular to find a solution for the constraint $(\{p = \mathsf{Int} \,|\, \rho\}, \geq, u)$. A component-wise unification gives the most precise solution (since we restrict ourselves to atomic rows): $\sigma(\rho) = \langle s = \texttt{"circle"} \vee \texttt{"polygon"}, d = \mathsf{Float} \vee \bot, e = \mathsf{Int} \vee \bot \,|\, \epsilon \rangle^{\{s\}}$.

This is however not the solution we want. Incrementing the cardinality of I, we now look for a solution of $\{((\{p = Int \mid \rho_1\} \to \{p = \mathsf{Int} \mid \rho_1\}) \wedge (\{p = Int \mid \rho_2\} \to \{p = \mathsf{Int} \mid \rho_2\}), \leq, u \to \alpha)\}$. Thus, we look in particular for a solution to the constraint $(\{p = Int \mid \rho_1\} \vee \{p = Int \mid \rho_2\}, \geq, u)$. A component-wise unification gives the solution $\sigma(\rho_1) = \langle s = \texttt{"circle"}, d = \mathsf{Float} \mid \epsilon\rangle^{\{s\}}, \sigma(\rho_2) = \langle s = \texttt{"polygon"}, e = \mathsf{Int} \mid \epsilon\rangle^{\{s\}}$, thanks to which we retrieve the desired solution, even with the restriction that rows are all atomic.

Example C.2 (Necessity of connectives on rows). This second examples illustrates why considering only atomic rows is not satisfactory. Take $t = \{\mid \rho\} \to \{\mid \rho\}$ and $s = \{a = \bot \mid ..\} \wedge \neg\{\mid \epsilon\}$ and let us look for a solution of $t \bullet_\emptyset s$. As in the previous example, this problem can be reduced to solving $\{(t', \leq, s \to \alpha)\}$, where α is a fresh variable, and $t' = \bigwedge_{i\in I} t\sigma_i$, where the σ_i are renamings of ρ.

At first, we try $|I| = 1$, so we look for a solution of $\{(\{\mid \rho\} \to \{\mid \rho\}, \leq, \{a = \bot \mid ..\} \wedge \neg\{\mid \epsilon\} \to \alpha)\}$. After normalization, we obtain the constraint-set $\{(\alpha, \geq, \{\mid \rho\}), (\rho, \geq, \langle a = \bot \mid ..\rangle^{\emptyset} \wedge \neg\langle \mid \epsilon\rangle^{\emptyset})\}$. If we were to consider only atomic rows, we could only give the solution $\sigma(\rho) = \langle \mid ..\rangle^{\emptyset}$ and $\sigma(\alpha) = \{\mid ..\}$.

To try to find a more precise solution, we run tallying again after incrementing the cardinal of I. This yields the following constraint-set: $\{((\{\mid \rho_1\} \to \{\mid \rho_1\}) \wedge (\{\mid \rho_2\} \to \{\mid \rho_2\}), \leq, s \to \alpha)\}$, which normalizes first to $\{(\alpha, \geq, \{\mid \rho_1\} \wedge \{\mid \rho_2\}), (\rho_1 \vee \rho_2, \geq, \langle a = \bot \mid ..\rangle^{\emptyset} \wedge \neg\langle \mid \epsilon\rangle^{\emptyset})\}$, and then (assuming $O(\rho_1) \leq O(\rho_2)$) to $\{(\alpha, \geq, \{\mid \rho_1\} \wedge \{\mid \rho_2\}), (\rho_1, \geq, \langle a = \bot \mid ..\rangle^{\emptyset} \wedge \neg\langle \mid \epsilon\rangle^{\emptyset} \wedge \neg\rho_2)\}$, With atomic rows only, we still do not have a satisfactory solution, and further expansions do not help.

C.2 General decomposition of rows

Definition C.3. *Let $r = \langle (\ell = \tau_\ell)_{\ell\in L_1} \mid \varsigma\rangle^{L_2}$ and L a finite set of labels. We define $r\backslash L = \langle (\ell = \tau_\ell)_{\ell\in L_1\setminus L} \mid \varsigma'\rangle^{L_2\cup L}$, where $\varsigma' = ..$ if $\varsigma \in \mathcal{V}$ and $L \nsubseteq L_1 \cup L_2$, and $\varsigma' = \varsigma$ otherwise.*

Lemma C.3. *Let r be an atomic row and L a set of labels. Let $r' = \langle L \cap \mathrm{lab}(r) \mid r\backslash(L \cap \mathrm{lab}(r))\rangle$ if $\mathrm{tail}(r) = \rho$ and $\mathrm{dom}(\rho) \cap L \neq \emptyset$, and $r' = \langle L \mid r\backslash L\rangle$ otherwise.*

(1) $r \simeq \langle (\ell = r(\ell))_{\ell\in L} \mid ..\rangle \wedge r' \simeq \bigwedge_{\ell\in L} \langle \ell = r(\ell) \mid ..\rangle \wedge r'$

(2) $\langle L \cap \mathrm{lab}(r) \mid r\backslash(L \cap \mathrm{lab}(r))\rangle \simeq \langle L \mid r\backslash L\rangle \wedge \langle\, \mathrm{lab}(r) \mid \mathrm{tail}(r)\rangle$ if $\mathrm{tail}(r) \in \mathcal{V}$.

Proof. If $r' = \langle L \mid r\backslash L\rangle$, the proof is straightforward. Otherwise, let $r = \langle (\ell = \tau_\ell)_{\ell\in L_1} \mid \rho\rangle^{L_2}$, with $L \nsubseteq L_1 \cup L_2$. For the first item, since $r(\ell) = \mathbb{1} \vee \bot$ for any $\ell \notin L_1 \cap L_2$, we must show $r \simeq \langle (\ell = \tau_\ell)_{\ell\in L\cap L_1}, (\ell = \mathbb{1} \vee \bot)_{\ell\in L\setminus L_1} \mid ..\rangle \wedge \langle L \cap L_1, (\ell = \tau_\ell)_{\ell\in L_1\setminus L} \mid \rho\rangle$. For the second item, we must show $\langle L \cap L_1, (\ell = \tau_\ell)_{\ell\in L_1\setminus L} \mid \rho\rangle \simeq \langle L, (\ell = \tau_\ell)_{\ell\in L_1\setminus L} \mid ..\rangle \wedge \langle\, \mathrm{lab}(r) \mid \rho\rangle$. Both of them are straightforward. ☐

Lemma C.4. *Let P and N be sets of atomic rows of the same domain and L be a finite set of labels. Let $P_{\mathcal{V}} = \{r \in P \mid \mathrm{tail}(r) = \rho \text{ and } L \cap \mathrm{dom}(\rho) \neq \emptyset\}$, similarly for $N_{\mathcal{V}}$. The relation $\bigwedge_{r\in P} r \leq \bigvee_{r\in N} r$ holds iff for every map $\iota : N \to L \cup \{\_\}$, for every $N' \subseteq \iota^{-1}(\_) \cap N_{\mathcal{V}}$:*

$$\left(\exists \ell \in L. \bigwedge_{r\in P} r(\ell) \leq \bigvee_{r\in\iota^{-1}(\ell)} r(\ell)\right) \text{ or } \left(\bigwedge_{r\in P} r\backslash L \leq \bigvee_{r\in\iota^{-1}(\_)\setminus N'} r\backslash L\right)$$
$$\text{or } \left(\bigwedge_{r\in P_{\mathcal{V}}} \langle\, \mathrm{lab}(r) \mid \mathrm{tail}(r)\rangle \leq \bigvee_{r\in N'} \langle\, \mathrm{lab}(r) \mid \mathrm{tail}(r)\rangle\right)$$

Proof. For each $r \in P_{\mathcal{V}} \cup N_{\mathcal{V}}$, let $r_- = \langle L \cap \mathrm{lab}(r) \mid r\backslash(L \cap \mathrm{lab}(r)\rangle$ and for each $r \in (P \cup N) \setminus (P_{\mathcal{V}} \cup N_{\mathcal{V}})$, let $r_- = \langle L \mid r\backslash L\rangle$.

Using Lemma C.3, we decompose the type in the statement into:

$$\bigwedge_{r\in P} (\langle (\ell = r(\ell))_{\ell\in L} \mid ..\rangle \wedge r_- \wedge \bigwedge_{r\in N} (\bigvee_{\ell\in L} \neg\langle \ell = r(\ell) \mid ..\rangle \vee \neg r_-) \tag{18}$$

We can distribute the intersection of the elements of N on the right of (18) over the unions in the second brackets. We obtain a union of intersections of, each time, $|N|$ elements, where each intersection is a possible combination of the individual types present in the second line. Each combination is described by a function $\iota : N \to L \cup \{\_\}$, where $\iota(n) = \ell$ means that the element $\langle \ell = \neg r(\ell) \mathbin{|} ..\rangle$ is present in the combination given by ι, while $\iota(n) = \_$ means that the element $\neg r_{\_}$ is present in the combination. For each $r \in N$ and $\ell \in L$, let us write $r_\ell = \langle \ell = r(\ell) \mathbin{|} ..\rangle$. Therefore the type in (18) is equivalent to:

$$\bigwedge_{r\in P}(\langle(\ell = r(\ell))_{\ell\in L} \mathbin{|} ..\rangle \wedge r_{\_}) \wedge \bigvee_{\iota:N\to L\cup\{\_\}} (\bigwedge_{r\in N} \neg r_{\iota(r)}) \tag{19}$$

By distributing the intersection over the union we obtain

$$\bigvee_{\iota:N\to L\cup\{\_\}} \left(\bigwedge_{r\in P}(\langle(\ell = r(\ell))_{\ell\in L} \mathbin{|} ..\rangle \wedge r_{\_}) \wedge \bigwedge_{r\in N} \neg r_{\iota(r)}\right) \tag{20}$$

A union is empty if and only if each summand of the union is empty. Therefore the type above is empty if and only if for all $\iota : N \to L \cup \{\_\}$, the following type is empty:

$$\begin{aligned}
&\bigwedge_{r\in P}(\langle(\ell = r(\ell))_{\ell\in L} \mathbin{|} ..\rangle \wedge r_{_}) \wedge \bigwedge_{r\in N} \neg r_{\iota(r)} \\
\simeq{}& \bigwedge_{r\in P}(\langle(\ell = r(\ell))_{\ell\in L} \mathbin{|} ..\rangle \wedge r_{_}) \wedge \bigwedge_{\ell\in L\cup\{_\}} \bigwedge_{r\in\iota^{-1}(\ell)} \neg r_\ell \\
\simeq{}& \bigwedge_{r\in P}(\langle(\ell = r(\ell))_{\ell\in L} \mathbin{|} ..\rangle \wedge r_{_}) \wedge \bigwedge_{r\in\iota^{-1}(_)} \neg r_{_} \wedge \bigwedge_{\ell\in L} \bigwedge_{r\in\iota^{-1}(\ell)} \neg r_\ell \\
\simeq{}& \bigwedge_{r\in P}(\langle(\ell = r(\ell))_{\ell\in L} \mathbin{|} ..\rangle \wedge r_{_}) \wedge \bigwedge_{r\in\iota^{-1}(_)} \neg r_{_} \wedge \langle(\ell = \textstyle\bigwedge_{r\in\iota^{-1}(\ell)} \neg r(\ell))_{\ell\in L} \mathbin{|} ..\rangle \\
\simeq{}& \langle(\ell = \textstyle\bigwedge_{r\in P} r(\ell) \wedge \bigwedge_{r\in\iota^{-1}(\ell)} \neg r(\ell))_{\ell\in L} \mathbin{|} ..\rangle \wedge \displaystyle\bigwedge_{r\in P} r_{_} \wedge \bigwedge_{r\in\iota^{-1}(_)} \neg r_{_}
\end{aligned}$$

Let $r_1 = \langle(\ell = \bigwedge_{r\in P} r(\ell) \wedge \bigwedge_{r\in\iota^{-1}(\ell)} \neg r(\ell))_{\ell\in L} \mathbin{|} ..\rangle$. The last type is equivalent to

$$\begin{aligned}
& r_1 \wedge \bigwedge_{r\in P} \langle L \mathbin{|} r\backslash L\rangle \wedge \bigwedge_{r\in P_{\mathcal{V}}} \langle \operatorname{lab}(r) \mathbin{|} \operatorname{tail}(r)\rangle \\
& \wedge \bigwedge_{r\in\iota^{-1}(_)\setminus N_{\mathcal{V}}} \neg\langle L \mathbin{|} r\backslash L\rangle \wedge \bigwedge_{r\in\iota^{-1}(_)\cap N_{\mathcal{V}}} (\neg\langle L \mathbin{|} r\backslash L\rangle \vee \neg\langle \operatorname{lab}(r) \mathbin{|} \operatorname{tail}(r)\rangle) \\
\simeq{}& r_1 \wedge \bigwedge_{r\in P} \langle L \mathbin{|} r\backslash L\rangle \wedge \bigwedge_{r\in P_{\mathcal{V}}} \langle \operatorname{lab}(r) \mathbin{|} \operatorname{tail}(r)\rangle \wedge \bigwedge_{r\in\iota^{-1}(_)\setminus N_{\mathcal{V}}} \neg\langle L \mathbin{|} r\backslash L\rangle \\
& \wedge \bigvee_{N'\subseteq\iota^{-1}(_)\cap N_{\mathcal{V}}} (\bigwedge_{r\in(\iota^{-1}(_)\cap N_{\mathcal{V}})\setminus N'} \neg\langle L \mathbin{|} r\backslash L\rangle \wedge \bigwedge_{r\in N'} \neg\langle \operatorname{lab}(r) \mathbin{|} \operatorname{tail}(r)\rangle) \\
\simeq{}& \bigvee_{N'\subseteq\iota^{-1}(_)\cap N_{\mathcal{V}}} \left(r_1 \wedge \bigwedge_{r\in P} \langle L \mathbin{|} r\backslash L\rangle \wedge \bigwedge_{r\in\iota^{-1}(_)\setminus N'} \neg\langle L \mathbin{|} r\backslash L\rangle \right.\\
& \left. \wedge \bigwedge_{r\in P_{\mathcal{V}}} \langle \operatorname{lab}(r) \mathbin{|} \operatorname{tail}(r)\rangle \wedge \bigwedge_{r\in N'} \neg\langle \operatorname{lab}(r) \mathbin{|} \operatorname{tail}(r)\rangle \right)
\end{aligned}$$

This type is empty if and only if the conjunctions are all empty for each ι and $N' \subseteq \iota^{-1}(\_)\cap N_{\mathcal{V}}$. Take ι and N' and let $r_2 = \bigwedge_{r\in P} \langle L \mathbin{|} r\backslash L\rangle \wedge \bigwedge_{r\in\iota^{-1}(\_)\setminus N'} \neg\langle L \mathbin{|} r\backslash L\rangle$ and $r_3 = \wedge \bigwedge_{r\in P_{\mathcal{V}}} \langle \operatorname{lab}(r) \mathbin{|} \operatorname{tail}(r)\rangle \wedge$

$\bigwedge_{r\in N'} \neg\langle\!| \operatorname{lab}(r) \,|\, \operatorname{tail}(r) |\!\rangle$. Let $r_\iota = \bigwedge_{1\le i\le 3} r_i$. It is immediate that r_1 is empty *iff* the first condition of the statement holds, r_2 is empty *iff* the second does, and r_3 is empty *iff* the third does. We directly obtain that if one of the conditions holds, then the type r_ι is empty. We now show that if r_ι is empty, then there is $1 \le i \le 3$ such that r_i is empty.

For this, we suppose that none of the subtypes is empty and build an element $d \in [\![r_\iota]\!]^{\mathrm{row}}$.

(1) Since r_1 is not empty, for all $\ell \in L$ there is an element $\delta^1_\ell \in [\![\bigwedge_{r\in P} r(\ell) \wedge \bigwedge_{r\in\iota^{-1}(\ell)} \neg r(\ell)]\!]^{\mathrm{fld}}$.
(2) Since r_2 is not empty, there is an element $\{\!|(\ell = \delta^2_\ell)_{\ell\in L_2}, \_ = \bot^{\emptyset}|\!\}^{V_2} \in [\![r_2]\!]^{\mathrm{row}}$.
(3) Since r_3 is not empty, there is an element $d_3 \in [\![r_3]\!]^{\mathrm{row}}$. However, the restrictions on the set of elements in $[\![r_3]\!]^{\mathrm{row}}$ only concern their tags so that any element d' with $\operatorname{tag}(d') = \operatorname{tag}(d_3)$ and $\operatorname{dom}(d') = \operatorname{dom}(d_3)$ is in $[\![r_3]\!]^{\mathrm{row}}$. Let $V_3 = \operatorname{tag}(d_3)$.

We build the element $\{\!|(\ell = \delta^1_\ell)_{\ell\in L}, (\ell = \delta^2_\ell)_{\ell\in L_2\setminus L}, \_ = \bot^{\emptyset}|\!\}^{V_3}$. This element belongs to $[\![r_\iota]\!]^{\mathrm{row}}$, which is a contradiction. □

C.3 Normalization of fields and tails

In this subsection, we define the functions $\operatorname{norm}_{\mathrm{fld}}(\tau, M)$ and $\operatorname{norm}_{\mathrm{tl}}(r, M)$ that are mentioned in Section 4.1. The formal definition of the whole algorithm is given below in Appendix C.4.

Fields. A field type is always equivalent to a DNF that is a disjunction of conjunctions of either of the shape $\tau \wedge \bigwedge_{\theta\in P} \theta \wedge \bigwedge_{\theta\in N} \theta$, where τ is either $\bot$ or a type t. If there is a smallest variable $\theta_0 \in P \cup N$ not in Δ, we single out this variable, in the same way as is done for type variables in our algorithm and in [Castagna et al. 2015]. If all top-level field variables are monomorphic:

- If $\tau = t$, τ can be instantiated to an empty type only if t can, so we apply $\operatorname{norm}(t, M)$.
- If $\tau = \bot$, τ can never be instantiated to an empty type since $\bot \not\le \mathbb{0}$, so normalization fails.

We use the notation $X \lightning T$ to indicate that X is the smallest top-level variable in T.

$$\operatorname{norm}_{\mathrm{fld}}(\tau, M) = \begin{cases} \{\{(\theta_0, \le, \neg\tau \vee \bigvee_{\theta\in P\setminus\{\theta_0\}} \neg\theta \vee \bigvee_{\theta\in N} \theta)\}\}, & \text{if } \exists\theta_0 \in P.\theta_0 \lightning \tau \\ \{\{(\tau \wedge \bigwedge_{\theta\in P} \theta \wedge \bigwedge_{\theta\in N\setminus\{\theta_0\}} \neg\theta \le \theta_0)\}\}, & \text{if } \exists\theta_0 \in N.\theta_0 \lightning \tau \\ \operatorname{norm}(t, M), & \text{if } \tau' = t \text{ and } (P\cup N)\setminus\Delta = \emptyset \\ \emptyset, & \text{if } \tau' = \bot \text{ and } (P\cup N)\setminus\Delta = \emptyset \end{cases}$$

Tails. By design, the input of this function is a row such that:

- Either there is a polymorphic top-level variable that is an a row r_0 with $\operatorname{lab}(r) = \emptyset$. Then, we single out this variable on the left of a new constraint.
- Or there is no polymorphic top-level row variable. In that case, we decompose the row over all the labels using the subtyping formula.

$$\operatorname{norm}_{\mathrm{tl}}(r, M) = \begin{cases} \{\{(\rho_0, \le, \bigvee_{r\in P\setminus\{r_0\}} \neg r \vee \bigvee_{r\in N} r)\}\}, & \text{if } r_0 \in P \\ \{\{(\rho_0, \ge, \bigwedge_{r\in P} r \wedge_{r\in N\setminus\{\rho\}} \neg r)\}\}, & \text{if } r_0 \in N \\ \bigsqcap_{\iota\in I} \bigsqcup_{\ell\in L} \operatorname{norm}_{\mathrm{fld}}\Big(\bigwedge_{r\in P} r(\ell) \wedge \bigwedge_{r\in\iota^{-1}(\ell)} \neg r(\ell), M\Big), & \text{if } \operatorname{tlv}(r) \subseteq \Delta \end{cases}$$

where in the two first cases, there is $\rho_0 \in (P \cup N) \setminus \Delta$ such that there is $r_0 \in P \cup N$ with $r_0 = \langle\!| \rho_0 |\!\rangle^{\mathcal{L}\setminus\operatorname{dom}(r)}$, and where in the last case:

- $L = \bigcup_{r\in P\cup N} \operatorname{lab}(r)$;
- $P_\Delta = \{r \in P \mid \operatorname{tail}(r) \in \mathcal{V}\}$ and $N_\Delta = \{r \in N \mid \operatorname{tail}(r) \in \mathcal{V}\}$;
- $I = \{\iota : N \to L \cup \{\_\} \mid (\forall r_\circ \in \iota^{-1}(\_) \setminus N_\Delta. \bigwedge_{r\in P} \operatorname{def}(r) \not\le \operatorname{def}(r_\circ))$ and $(\forall r_0 \in \iota^{-1}(\_) \cap N_\Delta.\forall r_p \in P_\Delta. \operatorname{tail}(r_\circ) \ne \operatorname{tail}(r_p))\}$, where $\operatorname{def}(r)$ is defined as in Lemma 2.1.

$$\frac{}{\Sigma \vdash_{\mathcal{N}} \emptyset \rightsquigarrow \{\emptyset\}}\ (\textsc{Nempty}) \qquad \frac{(\Sigma \vdash_{\mathcal{N}} \{(T_i, c_i, T_i')\} \rightsquigarrow \mathcal{S}_i)_{i\in I}}{\Sigma \vdash_{\mathcal{N}} \{(T_i, c_i, T_i') \mid i \in I\} \rightsquigarrow \sqcap_{i\in I}\mathcal{S}_i}\ (\textsc{Njoin})$$

$$\frac{\Sigma \vdash_{\mathcal{N}} (T, \leq, T') \rightsquigarrow \mathcal{S}}{\Sigma \vdash_{\mathcal{N}} (T', \geq, T) \rightsquigarrow \mathcal{S}}\ (\textsc{Nsym}) \qquad \frac{\Sigma \vdash_{\mathcal{N}} \{(T \wedge \neg T', \leq, \mathbb{0})\} \rightsquigarrow \mathcal{S} \qquad T' \neq \mathbb{0}}{\Sigma \vdash_{\mathcal{N}} \{(T, \leq, T')\} \rightsquigarrow \mathcal{S}}\ (\textsc{Nzero})$$

$$\frac{\Sigma \vdash_{\mathcal{N}} \{(\mathrm{dnf}(T), \leq, \mathbb{0})\} \rightsquigarrow \mathcal{S} \quad T \neq \mathrm{dnf}(T)}{\Sigma \vdash_{\mathcal{N}} \{(T, \leq, \mathbb{0})\} \rightsquigarrow \mathcal{S}}\ (\textsc{Ndnf}) \qquad \frac{\Sigma \vdash_{\mathcal{N}} \{(T_i, \leq, \mathbb{0}) \mid i \in I\} \rightsquigarrow \mathcal{S}}{\Sigma \vdash_{\mathcal{N}} \{(\vee_{i\in I} T_i, \leq, \mathbb{0})\} \rightsquigarrow \mathcal{S}}\ (\textsc{Nunion})$$

Where T_i in (Nunion) are single normal forms.

Fig. 5. Normalization rules for all kinds

$$\frac{t \in \Sigma \qquad \mathrm{tlv}(t) = \emptyset}{\Sigma \vdash_{\mathcal{N}} \{(t, \leq, \mathbb{0})\} \rightsquigarrow \{\emptyset\}}\ (\textsc{Nhyp}) \qquad \frac{\Sigma \cup \{t\} \vdash_{\mathcal{N}^\star} \{(t, \leq, \mathbb{0})\} \rightsquigarrow \mathcal{S} \qquad t \notin \Sigma}{\Sigma \vdash_{\mathcal{N}} \{(t, \leq, \mathbb{0})\} \rightsquigarrow \mathcal{S}}\ (\textsc{Nassum})$$

$$\frac{\mathrm{tlv}(T) = \emptyset \qquad X' \not\lessdot_O P \cup N \qquad \mathcal{S} = \begin{cases} \{\mathrm{single}(X', T_0)\} & X' \notin \Delta \\ \Sigma \vdash_{\mathcal{N}} \{(T, \leq, \mathbb{0})\} & X' \in \Delta \end{cases}}{\Sigma \vdash_{\mathcal{N}} \{(T_0 = \bigwedge_{X\in P} X \wedge \bigwedge_{X\in N} \neg X \wedge T, \leq, \mathbb{0})\} \rightsquigarrow \mathcal{S}}\ (\textsc{Ntlv})$$

$$\frac{}{\Sigma \vdash_{\mathcal{N}} \{(\bot, \leq, \mathbb{0})\} \rightsquigarrow \emptyset}\ (\textsc{Nopt}) \qquad \frac{\mathcal{S} = \begin{cases} \{\emptyset\} & \text{if } t, \leq, \mathbb{0} \\ \emptyset & \text{if } t \not\leq \mathbb{0} \end{cases}}{\Sigma \vdash_{\mathcal{N}^\star} \{(t = \bigwedge_{i\in P} b_i \wedge \bigwedge_{j\in N} \neg b_j, \leq, \mathbb{0})\} \rightsquigarrow \mathcal{S}}\ (\textsc{Nbasic})$$

$$\frac{\exists j \in N. \forall P' \subseteq P. \begin{cases} \Sigma \vdash_{\mathcal{N}} \{t_j^1 \wedge \bigwedge_{i\in P'} \neg t_i^1, \leq, \mathbb{0}\} \rightsquigarrow \mathcal{S}_{P'}^1 \\ \begin{cases} \Sigma \vdash_{\mathcal{N}} \{ \bigwedge_{i\in P\setminus P'} t_i^2 \wedge \neg t_j^2, \leq, \mathbb{0}\} \rightsquigarrow \mathcal{S}_{P'}^2 & P' \neq P \\ \mathcal{S}_{N'}^2 = \emptyset & \text{otherwise} \end{cases} \end{cases}}{\Sigma \vdash_{\mathcal{N}^\star} \{(\bigwedge_{i\in P} (t_i^1 \to t_i^2) \wedge \bigwedge_{j\in N} \neg(t_j^1 \to t_j^2), \leq, \mathbb{0})\} \rightsquigarrow \bigsqcup_{j\in N} \bigsqcap_{P'\subseteq P} (\mathcal{S}_{P'}^1 \sqcup \mathcal{S}_{P'}^2)}\ (\textsc{Narrow})$$

$$\frac{\Sigma \vdash_{\mathcal{N}} \{(\mathrm{row}(\bigwedge_{\mathsf{R}\in P} \mathsf{R} \wedge \bigwedge_{\mathsf{R}\in N} \neg\mathsf{R}), \leq, \mathbb{0})\} \rightsquigarrow \mathcal{S}}{\Sigma \vdash_{\mathcal{N}^\star} \{(\bigwedge_{\mathsf{R}\in P} \mathsf{R} \wedge \bigwedge_{\mathsf{R}\in N} \neg\mathsf{R}, \leq, \mathbb{0})\} \rightsquigarrow \mathcal{S}}\ (\textsc{Nrec})$$

Fig. 6. Normalization rules for type and field single normal forms

C.4 Constraint normalization

We formalize normalization as a judgment $\Sigma \vdash_N C \rightsquigarrow \mathcal{S}$, which states that under the environment Σ (which, informally, contains the types that have already been processed at this point), C is normalized to $\mathcal{S}$. The main judgment is derived according to the rules in Figs. 5 to 7. Given a type, field or row variable X and a conjunction of types, field types or rows respectively, we define $\text{single}(X, T \wedge X) = \{(X, \leq, \neg T)\}$ and $\text{single}(X, T \wedge \neg X) = \{(X, \geq, T)\}$. We call single normal form a DNF that has no topmost disjunction.

If $\emptyset \vdash_N C \rightsquigarrow \mathcal{S}$, then $\mathcal{S}$ is the result of the normalization of C. We now prove soundness and termination of the constraint normalization algorithm.

Definition C.4. *We define the family $(\leq_n)_{n\in\mathbb{N}}$ of subtyping relations as*

$$t \leq_n s \quad \stackrel{\text{def}}{\iff} \quad \forall \eta. [\![t]\!]_n \eta \subseteq [\![s]\!]_n \eta$$

where $[\![\cdot]\!]_n$ is the rank n interpretation of a type, defined as

$$[\![t]\!]_n \eta = \{d \in [\![t]\!]_\eta \mid \text{height}(d) \leq n\}$$

and $\text{height}(d)$ *is the height of an element in $\mathcal{D}$, defined as*

$$\ldots$$

$$\text{height}(\text{Rec}(\bar{d})^V) = 1 + \text{height}(\bar{d})$$

$$\text{height}(\langle\!|(\ell = \delta_\ell)_{\ell\in L}, \_ = \bot^{\emptyset}|\!\rangle_{L'}^{V}) = \max(1, (\text{height}(\delta_\ell))_{\ell\in L})$$

Lemma C.5. *Let $t \leq \{\,|\,..\}$. Then, $t \leq_{n+1} \mathbb{0} \iff \text{row}(t) \leq_n \mathbb{0}$.*

Proof. By definition and a trivial well-founded induction on type operators, we have $[\![t]\!]_\eta^{\text{q}} = \{\text{Rec}(\bar{d})^V \mid \bar{d} \in [\![\text{row}(t)]\!]_\eta^{\text{q}}\}$. Thus, by definition of height, we have $[\![t]\!]_{n+1}^{\text{q}}\eta = \{\text{Rec}(\bar{d})^V \mid \bar{d} \in [\![\text{row}(t)]\!]_n^{\text{q}}\eta\}$. □

Definition C.5. *Given a constraint-set C and a substitution σ, we define the rank n satisfaction predicate $\Vdash_n$ as*

$$\sigma \Vdash_n C \quad \stackrel{\text{def}}{\iff} \quad \forall (T_1, \leq, T_2) \in C. T_1 \leq_n T_2 \text{ and } \forall (T_1, \geq, T_2) \in C. T_1 \geq_n T_2$$

Lemma C.6. *(1) $\sigma \Vdash_0 C$ for all σ and C.*
(2) $\sigma \Vdash C \iff \forall n. \sigma \Vdash_n C$.

Proof. Consequence of [Castagna et al. 2015, Lemma C.7] and Lemma C.5. □

Definition C.6 (Marshalling). *Given a conjunction of atomic rows $r_0 = \bigwedge_{r\in P} r \wedge \bigwedge_{r\in N} r$, a finite set of labels and a set of variables, we define marshalling as*

$$\text{marsh}(r, L, \Delta) = \bigwedge_{r\in P\setminus P_0} r \wedge \bigwedge_{r\in N\setminus N_0} \neg r \wedge \bigwedge_{r\in P_0} \langle (\ell = r[\ell])_{\ell\in L} \,|\, r\backslash^{\Delta} L\rangle \wedge \bigwedge_{r\in N_0} \neg\langle (\ell = r[\ell])_{\ell\in L} \,|\, r\backslash^{\Delta} L\rangle$$

where $P_0 = \{r \in P \mid \text{tail}(r) = \rho \notin \Delta \text{ and } \text{dom}(\rho) \cap L \neq \emptyset\}$, similarly for N_0.

Lemma C.7. *If $\sigma \Vdash \{(\text{marsh}(r, L, \Delta), \leq, \mathbb{0}\}$, then $\sigma \Vdash \{(r, \leq, \mathbb{0})\}$.*

Proof. We show that $\text{marsh}(r_0, L, \Delta)\sigma = r_0\sigma$. Let $r = \langle (\ell = r(\ell))_{\ell\in L_1} \,|\, \rho\rangle \in P_0 \cup N_0$ and $r' = \langle (\ell = r[\ell]_{\ell\in L} \,|\, r\backslash^{\Delta} L\rangle$. Then, $r' = \langle (\ell = r(\ell)_{\ell\in L_1}, (\ell = \rho.\ell)_{\ell\in L\setminus L_1} \,|\, \rho\backslash L\rangle$. We have $\sigma(\rho) \simeq \langle (\ell = \tau_\ell)_{\ell\in L\setminus L_1} \,|\, r''\rangle^{\mathcal{L}\setminus\text{dom}(\rho)}$ because σ is a solution for $\{\text{marsh}(r_0, L, \Delta), \leq, \mathbb{0}\}$. Hence, $r\sigma = \text{marsh}(r, L, \Delta)\sigma$. □

Given a set Σ of types and rows, we write $C(\Sigma)$ for the constraint-set $\{(T, \leq, \mathbb{0}) \mid T \in \Sigma\}$.

Lemma C.8 (Soundness). *Let C be a constraint-set. If $\emptyset \vdash_N C \rightsquigarrow \mathcal{S}$, then for all constraint-set $C' \in \mathcal{S}$ and all substitution σ, we have $\sigma \Vdash C' \implies \sigma \Vdash C$.*

Proof. We prove the following stronger statements.

(1) Assume $\Sigma \vdash_N C \rightsquigarrow \mathcal{S}$. For all $C' \in \mathcal{S}$, σ and n, if $\sigma \Vdash_n C(\Sigma)$ and $\sigma \Vdash_n C'$, then $\sigma \Vdash_n C$.
(2) Assume $\Sigma \vdash_{N^\star} C \rightsquigarrow \mathcal{S}$. For all $C' \in \mathcal{S}$, σ and n, if $\sigma \Vdash_n C(\Sigma)$ and $\sigma \Vdash_n C'$, then $\sigma \Vdash_{n+1} C$.

The cases (Nempty), (Njoin), (Nsym), (Nzero), (Ndnf), (Nunion), (Nhyp), (Nassum), (Nbasic) and (Narrow) are given in [Castagna et al. 2015, Lemma C.10].

(Ntlv) Let $T_0 = T \wedge \bigwedge_{X \in P} X \wedge \bigwedge_{X \in N} \neg X$ and X' be the smallest type variable with respect to the order in $P \cup N$. If $X' \in P \setminus \Delta$, then we have $\sigma \Vdash_n \{(X', \leq, \neg T_{X'})\}$ with $T_{X'} = T \wedge \bigwedge_{X \in P \setminus X} X \wedge \bigwedge_{X \in N} \neg X$, thus $(X')\sigma \leq_n \neg T_{X'}\sigma$. This is equivalent to $T\sigma \leq_n \mathbb{0}$, and we conclude $\sigma \Vdash_n \{(T, \leq, \mathbb{0})\}$. If $X' \in N \setminus \Delta$, the result follows as well. If $X' \in \Delta$, then $P \cup N \subseteq \Delta$ by Definition C.2. We have $[\![T_0]\!] = \{D \in [\![T]\!] \mid P \subseteq \mathrm{tag}(D) \text{ and } N \cap \mathrm{tag}(D) = \emptyset\}$. Since T_0 is non empty, the variables in P and N are different, and since those variables cannot be instantiated, we can satisfy $T_0 \leq \mathbb{0}$ if and only if $T \leq \mathbb{0}$ is satisfied.

(Nopt) Direct by emptiness of $\mathcal{S}$.

(Nrec) Let $t = \bigwedge_{\mathsf{R} \in P} \mathsf{R} \wedge \bigwedge_{\mathsf{R} \in N} \neg \mathsf{R}$. By induction, we have $\sigma \Vdash_n \{(\mathrm{row}(t) \leq \mathbb{0})\}$. This is by definition equivalent to $\mathrm{row}(t)\sigma \leq_n \mathbb{0}$, thus $\mathrm{row}(t\sigma) \leq_n \mathbb{0}$ and by Lemma C.5 $t\sigma \leq_{n+1} \mathbb{0}$, which concludes $\sigma \Vdash_{n+1} \{(t \leq \mathbb{0})\}$.

(Nrow) The result is direct if $\mathcal{S} = \emptyset$. Otherwise, we have $C' = \bigcup_{\iota \in N \to L \cup \{\_\}} C'_\iota$, where $C'_\iota \in \bigsqcup_{\ell \in L} \mathcal{S}^\iota_\ell \sqcup \bigsqcap_{N' \in \mathcal{N}} \mathcal{S}^\iota_{N'}$. For all $\iota : N \to L \cup \{\_\}$, there are two cases.

(1) In the first case, there is $\ell \in L$ such that $C'_\iota \in \mathcal{S}^\ell_\iota$. Then, by induction,

$$\sigma \Vdash_n \{(\bigwedge_{r \in P} r[\ell] \wedge \bigwedge_{r \in \iota^{-1}(\ell)} \neg r[\ell] \leq \mathbb{0})\}$$

(2) In the second case, we have $C'_\iota = \bigcup_{N'} C^{N'}_\iota$, where $C^{N'}_\iota \in \mathcal{S}^\iota_{N'}$. For all N', there are two subcases.

(a) In the first subcase, $N' \in \mathcal{N}$. Then, by induction,

$$\sigma \Vdash_n \{(\bigwedge_{r \in P} r \backslash^\Delta L \wedge \bigwedge_{r \in \iota^{-1}(\_) \setminus N'} \neg(r \backslash^\Delta L) \leq \mathbb{0})\}$$

(b) In the second subcase, $N' \in \iota^{-1}(\_) \cap N_\Delta \setminus \mathcal{N}$. Then we have

$$\sigma \Vdash_n \{(\bigwedge_{r \in P_\Delta} \langle \mathrm{lab}(r) \mid \mathrm{tail}(r) \rangle \wedge \bigwedge_{r \in N'} \neg \langle \mathrm{lab}(r) \mid \mathrm{tail}(r) \rangle, \leq, \mathbb{0})\}$$

In other words, $\forall \iota : N \to L \cup \{\_\}. \forall N' \subseteq \iota^{-1}(\_) \cap N_\Delta$:

$$\left(\exists \ell \in L. \bigwedge_{r \in P} r[\ell]\sigma \leq_n \bigvee_{r \in \iota^{-1}(\ell)} r[\ell]\sigma\right) \text{ or } \left(\bigwedge_{r \in P} (r \backslash^\Delta L)\sigma \leq_n \bigvee_{r \in \iota^{-1}(\_) \setminus N'} (r \backslash^\Delta L)\sigma\right)$$
$$\text{or } \left(\bigwedge_{r \in P_\Delta} \langle \mathrm{lab}(r) \mid \mathrm{tail}(r) \rangle \wedge \bigwedge_{r \in N'} \neg \langle \mathrm{lab}(r) \mid \mathrm{tail}(r) \rangle\right)$$

Let $P_0 = \{r \in P \mid \mathrm{tail}(r) = \rho \notin \Delta \text{ and } \mathrm{dom}(\rho) \cap L \neq \emptyset\}$, same for N_0. By Lemma C.4, the definition of substitution and the fact that $r[\ell] = r(\ell)$ and $r \backslash^\Delta L = r \backslash L$ for all $r \notin P_0 \cup N_0$, we have $\mathrm{marsh}(r_0, L, \Delta)\sigma \leq_n \mathbb{0}$, that is $\sigma \Vdash \{\mathrm{marsh}(r_0, L, \Delta), \leq, \mathbb{0})\}$, By Lemma C.7, we conclude $\sigma \Vdash_n \{(r_0 \leq \mathbb{0})\}$.

(Ntail-mono) The result is direct if $\bigsqcap_{\iota\in I}\bigsqcup_{\ell\in L}\mathcal{S}^{\iota}_{\ell}=\emptyset$. Otherwise, we have $C'=\bigcup_{\iota\in I}C'_{\iota}$, where $\mathcal{S}'_{\iota}\in\bigsqcup_{\ell\in L}\mathcal{S}^{\ell}_{\iota}$. By definition of $\sqcup$, for all $\iota\in I$, there is $\ell\in L$ such that $C'_{\iota}\in\mathcal{S}^{\ell}_{\iota}$. By induction, $\sigma\Vdash_n\{(\bigwedge_{r\in P}r(\ell)\wedge\bigwedge_{r\in\iota^{-1}(\ell)}\neg r(\ell))\}$. In other words,

$$\forall\iota\in I.\exists\ell\in L.\bigwedge_{r\in P}r(\ell)\sigma\wedge\bigwedge_{r\in\iota^{-1}(\ell)}\neg r(\ell)\sigma\leq_n\mathbb{0}$$

Moreover, by hypothesis that $L\subseteq\bigcup_{r\in P\cup N}\mathrm{lab}(r)$. Also, for each $\iota\notin I$, one of the two conditions (4) or (5) of Lemma 2.1 is satisfied. By this corollary, $\bigwedge_{r\in P}r\sigma\wedge\bigwedge_{r\in N}\neg r\sigma\leq_n\mathbb{0}$. We conclude $\sigma\Vdash_n\{(\bigwedge_{r\in P}r\wedge\bigwedge_{r\in N}\neg r\leq\mathbb{0})\}$.

(Ntail-tlv) Let $r_0=\bigwedge_{r\in P}r\wedge\bigwedge_{r\in N}r$. By hypothesis, there is $r'=\langle\!|\,\rho\backslash L_1\rangle^{L_2}\in P\cup N$ such that $\rho\nleq_O r'$. There are two similar cases, we detail the one where $r'\in P$. Let $r'_0=\bigwedge_{r\in P\smallsetminus\{r'\}}\neg r\wedge\bigwedge_{r\in N}r$. By hypothesis, we have $\sigma\Vdash_n\{(r'\sigma\leq\neg r'_0)\}$, thus $r'\sigma\leq_n r'_0\sigma$. This is equivalent to $r_0\sigma\leq_n\mathbb{0}$, and we conclude $\sigma\Vdash_n\{(r_0\leq\mathbb{0})\}$. □

We introduce a notion of plinth generalizing the one of Frisch [2004] to types, field types and rows. This notion is used to prove termination of the algorithm.

Definition C.7 (Plinth). *A plinth $\sqsupset\subset\mathcal{T}_{\bot}\cup\mathcal{R}$ is a set of types, field types and rows with the following properties:*

- *$\sqsupset$ is finite;*
- *$\sqsupset$ contains $\mathbb{0}$, $\mathbb{1}$, $\bot$, $\langle\!|\,..\rangle^{\emptyset}$ and is closed under Boolean connectives ($\vee,\wedge,\neg$);*
- *for all type $t_1\to t_2\in\sqsupset$, we have $t_1\in\sqsupset$ and $t_2\in\sqsupset$;*
- *for all type $\mathsf{R}\in\sqsupset$, we have $\mathrm{row}(\mathsf{R})\in\sqsupset$;*
- *for all row $r\in\sqsupset$ of domain $\mathcal{L}\smallsetminus L_r$, let L be the set of labels appearing explicitely in $\sqsupset$ and V the set of row variables in $\sqsupset$, we have:*
  - *for all $\ell\in L$, $r(\ell)\in\sqsupset$*
  - *for all $L'\subseteq L$, $\forall V'\subseteq V$, $r\backslash^{V'}L'\in\sqsupset$ and $\mathrm{marsh}(r,L',V')\in\sqsupset$.*

Every finite set of types, field types and types is included in a plinth. Indeed, for a regular type t, the set of its subtrees S is finite, while rows and field types are inductively defined. The definition of the plinth ensures that the closure of S under Boolean connectives is also finite. Moreover, if a type, field type or row belongs in a plinth, the set of its subtrees also does. Finally, if a record or a row belongs to a plinth, the rows obtained by marshalling or by removing fields also belongs to the plinth, but only if these operations are done with respect to the set of labels present in the plinth, in order to guarantee finiteness of the plinth.

Lemma C.9 (Termination). *Let C be a finite constraint set. The normalization of C terminates.*

Proof. Let B be the set of types occuring in C. As C is finite, B is finite as well. Let $\sqsupset$ be a plinth such that $B\subseteq\sqsupset$. Then, when we normalize a constraint $(t,\leq,\mathbb{0})$ during the process of $\emptyset\vdash_N C$, t would belong to $\sqsupset$. We prove the lemma by induction on $(|\sqsupset\smallsetminus\Sigma|,U,|C|)$, U is the number of unions $\vee$ occurring in the constraint-set C (over any kind) plus the number of constraints $(T_1,\leq,T_2)$ where $T_2\neq\mathbb{0}$ or T_1 is not in DNF, and C is the constraint-set to be normalized. We detail the original cases, others are described in the proof [Castagna et al. 2015, Lemma C.14].

(Nopt) Terminates immediately.

(Nrec) None of the indices decreases, but the next rule to apply must be one of (Nrow), (Ntail-mono) or (Ntail-tlv).

(Nrow) Although $(|\sqsupset\smallsetminus\Sigma|,U,|C|)$ may not change, the next rule to apply must be one of (Ndnf), (Nhyp), (Nassum), (Ntlv), (Nopt) for $\mathcal{S}^{\iota}_{\ell}$ and (Ntail-tlv) for $\mathcal{S}^{\iota}_{N'}$.

(Ntail-mono) Although $(|\sqsupset \setminus \Sigma|, U, |C|)$ may not change, the next rule to apply must be one of (Ndf), (Nhy), (Nassum), (Ntlv), (Nopt).
(Ntail-tlv) Terminates immediately. □

Definition C.8 (Normalized constraint). *A constraint is said to be* normalized *if it is of the shape* (X, c, T)*, where* X *and* T *are of the same kind. A constraint-set is* normalized *if all constraints are.*

Lemma C.10. *Let* C *be a constraint-set and* $\emptyset \vdash_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$*. Then, all constraint-sets* $C' \in \mathcal{S}$ *are normalized.*

Proof. Straightforward by induction on the algorithm derivation. □

Lemma C.11 (Finiteness). *Let* C *be a constraint-set and* $\emptyset \vdash_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$*. Then,* $\mathcal{S}$ *is finite.*

Proof. It is easy to prove that each normalizing rule generates a finite set of finite sets of normalized constraints. □

Definition C.9. *Let* C *be a normalized constraint-set and* O *an ordering on* $\mathrm{var}(C)$ *and on the labels occurring in* C*. We say* C *is well-ordered if for all normalized constraint* $(X_1, c, T) \in C$ *and for all* $X_2 \in \mathrm{tlv}(T)$*,* $O(X_1) < O(X_2)$ *holds.*

Lemma C.12. *Let* C *be a constraint-set and* $\emptyset \vdash_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$*. Then for all normalized constraint-set* $C' \in \mathcal{S}$*,* C' *is well-ordered.*

Proof. There are two different ways to generate normalized constraints:

(Ntlv) We single out the type variable X' whose order is minimum, because $\mathrm{tlv}(T) = \emptyset$.
(Ntail-tlv) We single out the row variable ρ whose order is minimum. □

C.5 Constraint merging and saturation

After normalization, we have a set of constraint-sets where the same variable might have several upper and lower bounds given by different constraints on that variable. After this step, we wish to obtain unique upper and lower bounds for the same variable. This is done in two phases. For each normalized constraint-set C and following the order on variables, we:

- Merge two constraints $(X, \leq, T_1)$ and $(X, \leq, T_2)$ into $(X, \leq, T_1 \wedge T_2)$.
- Merge two constraints $(X, \geq, T_1)$ and $(X, \geq, T_2)$ into $(X, \geq, T_1 \vee T_2)$.

Once the set C has been transformed into C' where there are no such constraints left, we saturate the set to verify that the lower bound is indeed a subtype of the upper bound, once again following the order on variables. From two constraints $(T_1, \leq, X)$ and $(X, \leq, T_2)$, we normalize the constraint $(T_1 \wedge \neg T_2, \leq, \mathbb{0})$. We obtain a set of constraint-sets $\mathcal{S}$. We add the new resulting constraint-sets accordingly to the existing ones with $\mathcal{S} \sqcap \{C'\}$, then apply the step of merging and saturation again on each constraint-sets in $\mathcal{S} \sqcap \{C'\}$. Termination of this step is assured as in the step of constraint normalization by an additional argument M to the merge function, saving visited types.

Formally, this part is almost exactly the same as in the original algorithm in [Castagna et al. 2015, Section C.1.2]. One simply needs to replace occurrences of α and t by the more general meta-variables X and T. The rules are given in Figs. 8 and 9. Termination is proved thanks to the generalized definition of plinths (Definition C.7).

$$\forall \iota : N \to L \cup \{\_\}. \begin{cases} \forall \ell \in L. \Sigma \vdash_N \{(\bigwedge_{r \in P} r[\ell] \wedge \bigwedge_{r \in \iota^{-1}(\ell)} \neg r[\ell], \leq, \mathbb{0})\} \rightsquigarrow \mathcal{S}^{\iota}_{\ell} \\ \forall N' \in \mathcal{N}. \Sigma \vdash_N \{(\bigwedge_{r \in P} r \backslash^{\Delta} L \wedge \bigwedge_{r \in \iota^{-1}(\_) \setminus N'} \neg(r \backslash^{\Delta} L), \leq, \mathbb{0})\} \rightsquigarrow \mathcal{S}^{\iota}_{N'} \end{cases}$$

$$\rho \nleq_O \mathrm{tlv}(r_0) \qquad \rho \notin \Delta \qquad L = \mathrm{dom}(r_0) \setminus \mathrm{dom}(\rho) \qquad L \neq \emptyset$$

$$P_\Delta = \{r \in P \cap \Delta \mid \mathrm{tail}(r) = \rho_p \text{ and } \mathrm{dom}(\rho_p) \cap L \neq \emptyset\}$$

$$N_\Delta = \{r \in N \cap \Delta \mid \mathrm{tail}(r) = \rho_n \text{ and } \mathrm{dom}(\rho_n) \cap L \neq \emptyset\}$$

$$\frac{\mathcal{N} = \{N' \subseteq \iota^{-1}(\_) \cap N_\Delta \mid \bigwedge_{r \in P_\Delta} \langle\, \mathrm{lab}(r) \mid \mathrm{tail}(r) \rangle \wedge \bigwedge_{r \in N'} \neg\langle\, \mathrm{lab}(r) \mid \mathrm{tail}(r) \rangle \nleq \mathbb{0}\}}{\Sigma \vdash_N \{(r_0 = \bigwedge_{r \in P} r \wedge \bigwedge_{r \in N} \neg r, \leq, \mathbb{0})\} \rightsquigarrow \bigsqcap_{\iota : N \to L \cup \{\_\}} \left(\bigsqcup_{\ell \in L} \mathcal{S}^{\iota}_{\ell} \sqcup \bigsqcap_{N' \in \mathcal{N}} \mathcal{S}^{\iota}_{N'} \right)} \text{(NRow)}$$

$$\forall \iota \in I_1 \cap I_2. \forall \ell \in L. \Sigma \vdash_N \{(\bigwedge_{r \in P} r(\ell) \wedge \bigwedge_{r \in \iota^{-1}(\ell)} \neg r(\ell), \leq, \mathbb{0})\} \rightsquigarrow \mathcal{S}^{\iota}_{\ell}$$

$$I_1 = \{\iota : N \to L \cup \{\_\} \mid \forall r_\circ \in \iota^{-1}(\_) \setminus N_\Delta. \bigwedge_{r \in P \setminus P_\Delta} \mathrm{def}(r) \nleq \mathrm{def}(r_\circ)\}$$

$$I_2 = \{\iota : N \to L \cup \{\_\} \mid \forall r_\circ \in \iota^{-1}(\_) \cap N_\Delta. \forall r_p \in P_\Delta. \mathrm{tail}(r_\circ) \neq \mathrm{tail}(r_p)\}$$

$$\mathrm{tlv}(P \cup N) \subseteq \Delta \qquad L = \bigcup_{r \in P \cup N} \mathrm{lab}(r)$$

$$\frac{P_\Delta = \{r \in P \mid \mathrm{tail}(r) \in \mathcal{V}\} \qquad N_\Delta = \{r \in N \mid \mathrm{tail}(r) \in \mathcal{V}\}}{\Sigma \vdash_N \{(\bigwedge_{r \in P} r \wedge \bigwedge_{r \in N} \neg r, \leq, \mathbb{0})\} \rightsquigarrow \bigsqcap_{\iota \in I_1 \cap I_2} \bigsqcup_{\ell \in L} \mathcal{S}^{\iota}_{\ell}} \text{(NTail-mono)}$$

$$\frac{\rho \nleq_O \mathrm{tlv}(P \cup N) \qquad \rho \notin \Delta \qquad \exists r' \in P \cup N. r' = \langle\, \mid \rho \rangle^L}{\Sigma \vdash_N \{(r_0 = \bigwedge_{r \in P} r \wedge \bigwedge_{r \in N} \neg r, \leq, \mathbb{0})\} \rightsquigarrow \{\mathrm{single}(\langle\, \mid \rho \rangle^L, r_0)\}} \text{(NTail-tlv)}$$

Fig. 7. Normalization rules for row single normal forms

$$\frac{\forall i \in I. (X, \geq, T_i) \in C \qquad |I| \geq 2}{\vdash_M C \rightsquigarrow (C \setminus \{(X, \geq, T_i) \mid i \in I\} \cup \{(X, \geq, \bigvee_{i \in I} T_i)\})} \text{(MLB)}$$

$$\frac{\forall i \in I. (X, \leq, T_i) \in C \qquad |I| \geq 2}{\vdash_M C \rightsquigarrow (C \setminus \{(X, \leq, T_i) \mid i \in I\} \cup \{(X, \leq, \bigvee_{i \in I} T_i)\})} \text{(MUB)}$$

Fig. 8. Merging rules

$$\frac{\Sigma_p, C_\Sigma \cup \{(X, \geq, T_1), (X, \leq, T_2)\} \vdash_S C \rightsquigarrow \mathcal{S} \qquad (T_1, T_2) \in \Sigma_p}{\Sigma_p, C_\Sigma \vdash_S \{(X, \geq, T_1), (X, \leq, T_2)\} \cup C \rightsquigarrow \mathcal{S}} \text{(Shyp)}$$

$$\frac{\begin{array}{c}(T_1, T_2) \notin \Sigma_p \qquad \emptyset \vdash_N \{(T_1, \leq, T_2)\} \rightsquigarrow \mathcal{S} \\ \mathcal{S}' = \{\{(X, \geq, T_1), (X, \leq, T_2)\} \cup C \cup C_\Sigma\} \sqcap \mathcal{S} \\ \forall C' \in \mathcal{S}'.\Sigma_p \cup \{(T_1, T_2)\}, \emptyset \vdash_{MS} C' \rightsquigarrow \mathcal{S}_{C'}\end{array}}{\Sigma_p, C_\Sigma \vdash_S \{(X, \geq, T_1), (X, \leq, X_2)\} \cup C \rightsquigarrow \bigsqcup_{C' \in \mathcal{S}'} \mathcal{S}_{C'}} \text{(Sassum)}$$

$$\frac{\forall X, T_1, T_2.\nexists\{(X, \geq, T_1), (X, \leq, T_2)\} \subseteq C}{\Sigma_p, C_\Sigma \vdash_S C \rightsquigarrow \{C \cup C_\Sigma\}} \text{(Sdone)}$$

Where $\Sigma_p, C_\Sigma \vdash_{MS} C \rightsquigarrow \mathcal{S}$ means that there exists C' such that $\vdash_M C \rightsquigarrow C'$ and $\Sigma_p, C_\Sigma \vdash_S C' \rightsquigarrow \mathcal{S}$.

Fig. 9. Saturation rules

C.6 Harmonization

Thanks to the last step, we now have a set of constraint-sets such that in all constraint-sets, each variable in the domain of the constraint-set has at most one upper bound and one lower bound. Yet, for a row variable ρ of the original type constraints, we can have several constraints for the derived constructors $\rho\backslash L$, for instance a constraint-set containing both $(\rho\backslash L_1, \leq, T_1)$ and $(\rho\backslash L_2, \leq, T_2)$, with $L_1 \cap \mathrm{dom}(\rho) \neq L_2 \cap \mathrm{dom}(\rho)$. There could also be occurrences of another construction $\rho\backslash L_3$ in T_1 with a different L_3. We want a unique decomposition of ρ in each constraint-set, with a unique set L_0 such that only $\rho\backslash L_0$ may appear in the domain of the constraint-set. From this, we will build a solution such that $\sigma(\rho) \simeq (\ell = \tau_\ell)_{\ell \in L_0} r$. There can be occurrences of $\rho\backslash L'$ not in the domain, or of $\rho.\ell'$, but substitution will be correctly defined since we take L_0 to cover all such occurrences.

Formally, a harmonized constraint is defined as follow.

Definition C.10 (Harmonized constraint). *Let $C \subseteq \mathcal{C}$ be a saturated constraint-set. We say C is* harmonized *if for each type variable $\rho \in \mathrm{dom}(C)$, there is a finite set of labels L such that:*

(1) $\forall(\rho\backslash L_i, c, r_i) \in C.\ L_i \setminus \mathrm{dom}(\rho) = L \setminus \mathrm{dom}(\rho)$;
(2) $\forall \rho\backslash L' \in \mathrm{var}(C).\ L' \subseteq L$;
(3) $\forall \rho.\ell \in \mathrm{vars}(C).\ \ell \in L \cap \mathrm{dom}(\rho)$.

If a constraint-set is not harmonized, then it is of the shape $C \cup \{\langle\!\!\langle\, | \, \rho\backslash L \rangle\!\!\rangle^L, c, r\}$, where $L_0 \not\subseteq L$, when we define $L_0 = \bigcup_{\rho.\ell \in \mathrm{vars}(C) \cup \mathrm{vars}(r)} \{\ell\} \cup \bigcup_{\rho\backslash L' \in \mathrm{vars}(C) \cup \mathrm{vars}(r)} L'$. To harmonize the decomposition of the variable row, we normalize the constraint-set $\{(\langle\!\!\langle (\ell = \rho.\ell)_{\ell \in L_0 \setminus L} \, | \, \rho\backslash L_0 \rangle\!\!\rangle^L, c, r)\}$. We integrate the obtained set of constraint-sets $\mathcal{S}$ to the existing constraints with $\{C\} \sqcap \mathcal{S}$ and apply merging and harmonization recursively. The rules are given in Fig. 10.

Lemma C.13 (Soundness). *Let C be a finite saturated constraint-set. If $\emptyset \vdash_{\mathcal{H}} C \rightsquigarrow \mathcal{S}$, then for all constraint-set $C' \in \mathcal{S}$ and all substitution σ, we have $\sigma \Vdash C' \implies \sigma \Vdash C$.*

Proof. The proof is by induction on the derivation tree. We prove the more general statement for all Σ. The base case $\Sigma \vdash_{\mathcal{H}} C \rightsquigarrow \{C\}$ is trivial. In the inductive case, let $C = C_0 \cup \{(\langle\!\!\langle\, | \, \rho\backslash L \rangle\!\!\rangle^L, c, r)\}$. By definition of $\sqcup$, there are $C_n \in \{C\} \sqcap \mathcal{S}$ and $C_m \in \mathcal{S}_n$ such that $C' \in \mathcal{S}_n^m$. Since $\sigma \Vdash C'$, we have by induction hypothesis on $\Sigma \cup C_m \vdash_{\mathcal{H}} C_m \rightsquigarrow \mathcal{S}_n^m$ that $\sigma \Vdash C_m$. By definition of $\sqcup$ again, there is $C_n \in \{C_0\} \sqcap \mathcal{S}$ such that $C_m \in \mathcal{S}_n$. Since $\sigma \Vdash C_m$, we have by soundness of constraint merging on

$$\frac{C \text{ is harmonized}}{\Sigma \vdash_{\mathcal{H}} C \rightsquigarrow \{C\}} \text{ (HDONE)}$$

$$\frac{\begin{array}{c} L_0 = \bigcup_{\rho.\ell \in \text{vars}(C) \cup \text{vars}(r)} \{\ell\} \cup \bigcup_{\rho \backslash L' \in \text{vars}(C) \cup \text{vars}(r)} L' \qquad L_0 \not\subseteq L \\ \emptyset \vdash_{\mathcal{N}} \{(\langle (\ell = \rho.\ell)_{\ell \in L_0 \setminus L} \mid \rho \backslash L_0 \rangle^L, c, r)\} \rightsquigarrow \mathcal{S} \\ \forall C_n \in \{C\} \sqcap \mathcal{S}.(\emptyset, \emptyset \vdash_{\mathcal{MS}} C_n \rightsquigarrow \mathcal{S}_n) \\ \forall C_m \in \mathcal{S}_n \text{ and } C_m \notin \Sigma.(\Sigma \cup C_m \vdash_{\mathcal{H}} C_m \rightsquigarrow \mathcal{S}_n^m) \end{array}}{\Sigma \vdash_{\mathcal{H}} C \cup \{(\langle \mid \rho \backslash L \rangle^L, c, r)\} \rightsquigarrow \bigsqcup_{C_n \in \{C\} \sqcap \mathcal{S}} \bigsqcup_{C_m \in \mathcal{S}_n} \mathcal{S}_n^m} \text{ (HARM)}$$

Fig. 10. Harmonization rules

$\emptyset, \emptyset \vdash_{\mathcal{MS}} C_n \rightsquigarrow \mathcal{S}_n$ that $\sigma \Vdash C_n$. We have $C_n = C_0 \cup C_n'$ with $C_n' \in \mathcal{S}$. Since $\sigma \Vdash C_n$, we have $\sigma \Vdash C_n'$ and by Lemma C.8 $\sigma \Vdash \{(\langle (\ell = \rho.\ell)_{\ell \in L_0 \setminus L} \mid \rho \backslash L_0 \rangle^L, c, r)\}$. Thus, $\sigma(\rho) \simeq \langle (\ell = \tau_\ell)_{\ell \in L \cap \text{dom}(\rho)}, (\ell = \tau_\ell)_{\ell \in L_0 \setminus L} \mid r' \rangle^{\mathcal{L} \setminus \text{dom}(\rho)}$. Then, $(\langle (\ell = \rho.\ell)_{\ell \in L_0 \setminus L} \mid \rho \backslash L_0 \rangle^L)\sigma = (\langle \mid \rho \backslash L \rangle^L)\sigma = \langle (\ell = \tau_\ell)_{\ell \in L_0 \setminus L} \mid r' \rangle^L$. Therefore, $\sigma \Vdash \{(\langle \mid \rho \backslash L \rangle^L, c, r)\}$. We conclude $\sigma \Vdash C_0$ as well because $\sigma \Vdash C_n$ and $C_0 \subseteq C_n$. □

Lemma C.14 (Termination). *Let C be a finite saturated constraint-set. The harmonization of C terminates.*

Proof. Let B be the set of types, type fields and rows in C, finite since C is finite. Let $\sqsupset$ be a plinth such $B \subseteq \sqsupset$. When adding a set of constraint C_m to Σ during harmonization, every constraint of C_m belongs to $(\sqsupset \times \{\leq, \geq\} \times \sqsupset)$. The proof is by induction on $(|(\sqsupset \times \{\leq, \geq\}, \times \sqsupset)| - |\Sigma|, |C|)$ lexicographically ordered.

(Hdone) Terminates immediatly.

(Harm) Normalization, merging and saturation all terminate. In the recursive step of harmonization that are applied, $|(\sqsupset \times \{\leq, \geq\} \times \sqsupset)| - |\Sigma|$ decreases. □

Lemma C.15 (Finiteness). *Let C be a finite saturated constraint-set and $\emptyset \vdash_{\mathcal{H}} C \rightsquigarrow \mathcal{S}$. Then $\mathcal{S}$ is finite.*

Proof. By induction on the derivation and by Lemma C.11 and finiteness of constraint merging. □

Lemma C.16. *Let C be a finite saturated constraint-set and $\vdash_{\mathcal{H}} C \rightsquigarrow \mathcal{S}$. Then for all constraint-set $C' \in \mathcal{S}$, C' is harmonized.*

Proof. Direct by induction on the derivation, finite by Lemma C.14. □

Lemma C.17. *Let C be a well-ordered saturated constraint-set and $\emptyset \vdash_{\mathcal{H}} C \rightsquigarrow \mathcal{S}$. Then for all harmonized constraint-set $C' \in \mathcal{S}$, C' is well-ordered.*

Proof. Consequence of Lemma C.12 and conservation of well-orderedness by merging and saturation. □

C.7 From constraints to equations

Once normalization, merging and harmonization are done, we have a set of well-ordered constraint-sets at hand, where all variables have unique lower and upper bounds, and for each row variable ρ, there is at most a unique occurrence of $\rho\backslash L$ (where L can be empty if ρ has not been decomposed), such that also every occurrence of $\rho.\ell$ has $\ell \in L$. We are now able to rewrite each constraint-set C into an equivalent equation system.

Definition C.11 (Equation system). *An equation system E is a set of equations of the form $X = T$ such that there exists at most one equation in E for every variable X, and X and T are of the same kind. We define the domain of E, written* $\mathrm{dom}(E)$*, as the set* $\{X \mid \exists T.X = T \in E\}$.

Definition C.12 (Equation system solution). *Let E be an equation system. A solution to E is a substitution σ such that $\forall(X = T) \in E.\sigma(X) \simeq T\sigma$ holds. If σ is a solution to E, we write $\sigma \Vdash E$.*

Given a constraint-set C, we will use the notation $(T_1 \leq X \leq T_2) \in C$ to indicate $\{(T_1, \leq, X), (X, \leq, T_2)\} \subseteq C$. We assume that every variable and every term $\rho.\ell$, $\rho\backslash L$ in $\mathrm{dom}(C)$ have an upper and a lower bound, without loss of generality because a constraint with bottom or top types can always be added if needed:

- For a type variable α, we can add constraints $(\mathbb{0}, \leq, \alpha)$ or $(\alpha, \leq, \mathbb{1})$;
- For a field variable θ (or $\rho.\ell$), we can add constraints $(\mathbb{0}, \leq, \theta)$ or $(\theta, \leq, \mathbb{1} \vee \perp)$;
- For a row variable $\rho\backslash L$ (or ρ), we can add constraints $(\mathbb{0}, \leq, \rho\backslash L)$ or $(\rho\backslash L, \leq, \langle\!| ..\rangle^{\mathcal{L}\smallsetminus(\mathrm{dom}(\rho)\cup L)})$.

We rewrite the set C to a set of equations with a function $\mathrm{solve}(C)$, where α', θ', θ_ℓ and ρ' are fresh variables. We write $(T_1 \leq X \leq T_2)$ for the constraints $\{(T_1, \leq, X), (X, \leq, T_2)\} \subseteq C$.

$$
\begin{aligned}
\mathrm{solve}(C) = {} & \{\alpha = (t_1 \vee \alpha') \wedge t_2 \mid (t_1 \leq \alpha \leq t_2) \in C\} \\
& \cup \{\theta = (\tau_1 \vee \theta') \wedge \tau_2 \mid (\tau_1 \leq \theta \leq \tau_2) \in C \text{ and } \theta \neq \rho.\ell\} \\
& \cup \{\rho = \langle\!(\ell = \tau_\ell)_{\ell\in L} \mid r\rangle^{\mathcal{L}\smallsetminus \mathrm{dom}(\rho)} \\
& \quad \mid (\text{if } \exists L'.(r_1 \leq \rho\backslash L' \leq r_2) \in C, \text{ then } L = L' \cap \mathrm{dom}(\rho) \text{ and } r = (r_1 \vee \rho') \wedge r_2, \\
& \quad \text{else } L = \{\ell \mid \rho.\ell \in \mathrm{vars}(C)\} \text{ and } r = \rho') \\
& \quad \text{and } (\forall \ell \in L.\ \text{if } (\tau^1_\ell \leq \rho.\ell \leq \tau^2_\ell) \in C \text{ then } \tau_\ell = (\tau^1_\ell \vee \theta_\ell) \wedge \tau^2_\ell \text{ else } r = \theta_\ell)\}
\end{aligned}
$$

where α', θ', and ρ' are fresh variables.

For type and field variables (not generated by the decomposition of a row variable), we obtain an equation by means of the type connectives, where the union entails a lower bound and the intersection an upper bound. For a row variable ρ, there is L and constraints $(\tau^1_\ell \leq \rho.\ell \leq \tau^2_\ell) \in C$ for all $\ell \in L$ and $(r_1 \leq \rho\backslash L \leq r_2)$, with the potentially missing constraints obtained with the default values. Since we have decomposed ρ into the labels in L and a part of domain $\mathcal{L} \smallsetminus (\mathrm{dom}(\rho) \cup L)$, we build an equation for ρ by concatenating the independent types for $\rho.\ell$ and $\rho\backslash\ell$ together.

To prove soundness of the transformation, we define the rank n satisfaction predicate $\Vdash_n$ for equation systems, which is similar to the one for constraint-sets.

Lemma C.18 (Soundness). *Let $C \subseteq \mathcal{C}$ be a well-ordered saturated constraint-set and E its transformed equation system. Then for all substitutions σ, if $\sigma \Vdash E$, then $\sigma \Vdash C$.*

Proof. We write $O(C_1) < O(C_2)$ if $O(X_1) < O(X_2)$ for all $X_1 \in \mathrm{dom}(C_1)$ and all $X_2 \in \mathrm{dom}(C_2)$. We prove a stronger statement:

(*) For all σ, n and $C_\Sigma \subseteq C$, if $\sigma \Vdash_n E$, $\sigma \Vdash_n C_\Sigma$, $\sigma \Vdash_{n-1} C \smallsetminus C_\Sigma$ and $O(C \smallsetminus C_\Sigma) < O(C_\Sigma)$, then $\sigma \Vdash_n C \smallsetminus C_\Sigma$.

Here C_Σ denotes the set of constraints that have been checked. The proof proceeds by induction on $|C \smallsetminus C_\Sigma|$, and is similar to the proof of [Castagna et al. 2015, Lemma C.33] for type variables only. The base case $C \smallsetminus C_\Sigma = \emptyset$ is straightforward. Let $C \smallsetminus C_\Sigma \neq \emptyset$ and let us consider the case of row variables. Take ρ with the maximal order in $\mathrm{dom}(C \smallsetminus C_\Sigma)$. There are a set L and corresponding equations $\rho\backslash L = r = (r_1 \wedge \rho') \wedge r_2$ and $(\rho.\ell = \tau_\ell = (\tau_\ell^1 \vee \theta_\ell) \wedge \tau_\ell^2)_{\ell\in L}$. As $\sigma \Vdash_n E$, we have $\sigma(\rho) \simeq_n (\langle \ell = \tau_\ell \,|\, r\rangle^{L_\rho}\sigma)$, where $L_\rho = \mathcal{L} \setminus \mathrm{dom}(\rho)$. Then, for all $\ell \in L$:

$$(\rho.\ell)\sigma \wedge \neg\tau_\ell^2\sigma \simeq_n ((\tau_\ell^1 \vee \theta_\ell) \wedge \tau_\ell^2)\sigma \wedge \neg\tau_\ell^2\sigma \simeq_n \mathbb{0}$$

And similarly for $(\rho\backslash L)\sigma \wedge \neg(\rho\backslash L)\sigma$. On the other hand, for all $\ell \in L$, we have:

$$\tau_\ell^1\sigma \wedge \neg(\rho.\ell)\sigma \simeq_n \tau_\ell^1 \wedge \neg((\tau_\ell^1 \vee \theta_\ell) \wedge \tau_\ell^2)\sigma \simeq_n \tau_\ell^1\sigma \wedge \neg\tau_\ell^2\sigma$$

And similarly for $r_1\sigma \wedge \neg(\rho\backslash L)\sigma$. It remains to show that $\tau_\ell^1\sigma \leq_n \tau_\ell^2\sigma$ holds for all $\ell \in L$ and that $r_1\sigma \leq_n r_2\sigma$ hold, that is $\sigma \Vdash_n \{(\tau_\ell^1 \leq \tau_\ell^2)_{\ell\in L}, (r_1 \leq r_2\}$. The rest of the proof goes as in [Castagna et al. 2015, Lemma C.33]. We use the fact that the order of $\rho.\ell$ and $\rho\backslash L$ is directly superior to the order of ρ, and thus maximal in $\mathrm{dom}(C \setminus C_\Sigma)$. □

Lemma C.19 (Completeness). *Let $C \subseteq \mathcal{C}$ be a saturated normalized constraint-set and E its transformed equation system. Then for all substitution σ, if $\sigma \Vdash C$, there exists σ' such that* $\mathrm{dom}(\sigma') \cup \mathrm{dom}(\sigma) = \emptyset$ *and $\sigma \cup \sigma' \Vdash E$.*

Proof. Let $\sigma' = \{\sigma(\alpha)/\alpha' \mid \alpha \in \mathrm{dom}(C)\} \cup \{\sigma(\theta)/\theta' \mid \theta \in \mathrm{dom}(C) \text{ and } \theta \neq \rho.\ell\} \cup \{(\rho\backslash L)\sigma/\rho'\} \cup \{(\rho.\ell)\sigma/\theta_\ell^\rho \mid \rho.\ell \in \mathrm{dom}(C)\}$, where L is obtained as in the definition of solve either from $\rho\backslash L' \in \mathrm{dom}(C)$ and $L = L' \cap \mathrm{dom}(\rho)$, or by $L = \{\ell \mid \rho.\ell \in \mathrm{vars}(C)\}$. The case for type and field variables is as in [Castagna et al. 2015, Lemma C.34]. Let us consider an equation $\rho = \langle(\ell = \tau_\ell)_{\ell\in L} \,|\, r\rangle^{\mathcal{L}\setminus\mathrm{dom}(\rho)} \in E$. Correspondingly, there exist $(r_1 \leq \rho\backslash L \leq r_2) \in C$ and for all $\ell \in L$, there exist $(\tau_\ell^1 \leq \rho.\ell \leq \tau_\ell^2) \in C$ (without loss of generality, we suppose C to be saturated with default values). As $\sigma \Vdash C$, then $r_1\sigma \leq (\rho\backslash L)\sigma \leq r_2\sigma$, and for all $\ell \in L$, $\tau_\ell^1\sigma \leq (\rho.\ell)\sigma \leq r_2\sigma$, and the operations $(\rho\backslash L)\sigma$ and $(\rho.\ell)\sigma$ are defined. Thus,

$$\begin{aligned}
\langle(\ell = \tau_\ell)_{\ell\in L} \,|\, r\rangle(\sigma\cup\sigma') &= \langle(\ell = (\tau_\ell^1(\sigma\cup\sigma') \vee (\sigma\cup\sigma')(\theta_\ell^\rho)) \wedge \tau_\ell^2(\sigma\cup\sigma'))_{\ell\in L} \,|\\
&\qquad (r_1(\sigma\cup\sigma') \vee (\sigma\cup\sigma')(\rho')) \wedge r_2(\sigma\cup\sigma')\rangle\\
&= \langle(\ell = (\tau_\ell^1\sigma \vee (\rho.\ell)\sigma) \wedge \tau_\ell^2\sigma)_{\ell\in L} \,|\, (r_1\sigma \vee (\rho\backslash L)\sigma \wedge r_2\sigma\rangle\\
&\simeq \langle(\ell = (\rho.\ell)\sigma \wedge \tau_\ell^2\sigma)_{\ell\in L} \,|\, (\rho\backslash L)\sigma \wedge r_2\sigma\rangle\\
&\simeq \langle(\ell = (\rho.\ell)\sigma)_{\ell\in L} \,|\, (\rho\backslash L)\sigma\rangle\\
&= \sigma(\rho)
\end{aligned}$$

The last line is justified by σ being a solution to C, so being of the shape $\sigma(\rho) = \langle(\ell = \tau'_\ell)_{\ell\in L} \,|\, r'\rangle$. □

Definition C.13. *Let E be an equation system and O an ordering on* $\mathrm{dom}(E)$ *and on the labels occurring in E. We say that E is well-ordered if for all $X = T_X \in E$ and $X' \in \mathrm{tlv}(T_X) \cap \mathrm{dom}(E)$, we have $O(X_1) < O(X')$.*

Lemma C.20. *Let C be a well-ordered saturated normalized constraint-set and E its transformed equation system. Then E is well-ordered.*

Proof. We have $\mathrm{dom}(E) = \mathrm{dom}(C) \cap (\mathcal{V}_t \cup \mathcal{V}_f) \cup \{\rho \mid \exists\ell.\rho.\ell \in \mathrm{dom}(C) \text{ or } \exists L.\rho\backslash L \in \mathrm{dom}(C)\}$. The case for type and field variables is as in [Castagna et al. 2015, Lemma C.36], but uses the fact that $(\mathrm{tlv}(T_1) \cup \mathrm{tlv}(T_2)) \cap \mathrm{dom}(E) = (\mathrm{tlv}(T_1) \cup \mathrm{tlv}(T_2)) \cap \mathrm{dom}(C)$. Now, consider $\rho = r_0$ with $r_0 = \langle(\ell = \tau_\ell)_{\ell\in L} \,|\, r\rangle$ obtained from $(r_1 \leq \rho\backslash L \leq r_2) \in C$ with $r = (r_1 \wedge \rho') \vee r_2$, and for all $\ell \in L$ from $(\tau_\ell^1 \leq \rho.\ell^2) \in C$. We have $\mathrm{tlv}(r_0) = \mathrm{tlv}(r)$. Since C is well-ordered, for all $(\rho_2 \in \mathrm{tlv}(r_1) \cup \mathrm{tlv}(r_2)) \cap \mathrm{dom}(C)$, $O(\rho) < O(\rho_2)$. Moreover, ρ' is a fresh row variable in C, that is $\rho' \notin \mathrm{dom}(C)$. And then $\rho' \notin \mathrm{dom}(E)$. Therefore, $\mathrm{tlv}(r) \cap \mathrm{dom}(E) = (\mathrm{tlv}(r_1) \cup \mathrm{tlv}(r_2)) \cap \mathrm{dom}(C)$ and the result follows. □

C.8 Solution of equation systems

We have now obtained a set of equation-sets E, that we must each transform into a substitution σ. We do this in the same way as in [Castagna et al. 2015, §3.2.2], but with all kinds of variables rather than just type variables. In the set of equations, there is no construction $\rho.\ell$ or $\rho\backslash L$ anymore. We define a function $\mathrm{Unify}(E)$ as $\mathrm{Unify}(\emptyset) = \{\}$, and otherwise:

(1) Select in E the equation $X = T$ for the smallest X w.r.t. the order;
(2) Let E' be the set of equations obtained by replacing in $E\setminus\{X = T\}$ every occurrence of X by $\mu X'.(T\{X'/X\})$ (X' fresh);
(3) Let $\sigma = \mathrm{Unify}(E')$ and return $\{X = (\mu X'.T\{X'/X\})\sigma\} \cup \sigma$.

The ordering on the variables guarantees the regularity of the obtained types. For the elements $\sigma(X) = \mu X'.T$ where $X' \notin \mathrm{vars}(T)$, we can remove the introduced μ-abstraction.

It is straightforward to extend the proofs and definitions.

Definition C.14 (General solution). *Let E be an equation system. A general solution to E is a substitution σ from $\mathrm{dom}(E)$ to $\mathcal{T}_\perp \cup \mathcal{R}$ such that $\forall X \in \mathrm{dom}(\sigma).\,\mathrm{var}(\sigma(X)) \cap \mathrm{dom}(\sigma) = \emptyset$ and $\forall X = T \in E.\sigma(X) \simeq T\sigma$ holds.*

Definition C.15 (Equivalent substitutions). *Let σ, σ' be two substitutions. We say $\sigma \simeq \sigma'$ if and only if $\forall X.\sigma(X) \simeq \sigma'(X)$.*

Proposition C.21. *Let E be a well-ordered equation system. Let $\mathrm{Unify}(E)$ be the procedure described by Castagna et al. [2015] to build a substitution.*

Soundness *If $\sigma = \mathrm{Unify}(E)$, then $\sigma \Vdash E$.*

Completeness *For all substitution σ, if $\sigma \Vdash E$, then there exist σ_0 and σ' such that $\sigma_0 = \mathrm{Unify}(E)$ and $\sigma \simeq \sigma' \circ \sigma_0$.*

Termination *The algorithm $\mathrm{Unify}(E)$ terminates.*

The last property we verified is well-formedness. As in [Castagna et al. 2015], a type is well-formed if and only if the recursion traverses a constructor, and this property is guaranteed thanks to the order on variables.

Proposition C.22 (Well-formedness). *If $\sigma = \mathrm{Unify}(E)$, then for all $X \in \mathrm{dom}(\sigma)$, $\sigma(X)$ is well-formed.*

Proof. Assume that there exists an ill-formed $\sigma(X)$. That is, $\sigma(X) = \mu x.t$ where x occurs at the top-level of t. According to the algorithm $\mathrm{Unify}()$, there exists a sequence of equations $(X =)X_0 = T_{X_0}, \ldots, X_n = T_{X_n}$ such that X_i is at top-level in $T_{X_{i-1}}$ and X_0 is at top-level in T_{X_n} and where $i \in \{1, \ldots, n\}$ and $n \geq 0$. We must necessarily have all the X_i of the same kind. Indeed, for type (resp. row) variables only types (resp. row) variables can appear at top-level. Now, if X_0 is a field variable, there can be a type variable X_i at top-level in the field type T_{i-1}. But then, X_0 cannot be at top-level in X_n since T_i is a type, and field variables cannot appear at top-level in a type. Since all X and T must be of the same kind and according to Definition C.13, we have $O(X_{i-1}) < O(X_i)$ and $O(X_n) < O(X_0)$. Therefore, we have $O(X_0) < O(X_1) < \cdots < O(X_n) < O(X_0)$, which is impossible. Thus the result follows. ☐

C.9 The complete algorithm

The procedure $\mathrm{Sol}_\Delta(C)$ to solve type tallying of a constraint-set C proceeds as follows.

(1) C is normalized into a finite set $\mathcal{S}$ of well-ordered normalized constraint-sets (Section 4.1).
(2) Each constraint-set $C_i \in \mathcal{S}$ is merged and saturated into a finite set $\mathcal{S}_{C_i}$ of well-ordered constraint-sets. Then, all these sets are collected into another set $\mathcal{S}'$ (*i.e.*, $\mathcal{S}' = \bigsqcup_{C_i \in \mathcal{S}} \mathcal{S}_{C_i}$) (Appendix C.5).

(3) Each constraint-set $C'_i \in \mathcal{S}'$ is harmonized into a finite set $\mathcal{S}_{C'_i}$ of well-ordered harmonized constraint-sets. Then, all these sets are collected into another set $\mathcal{S}''$ (*i.e.*, $\mathcal{S}'' = \bigsqcup_{C'_i \in \mathcal{S}'} \mathcal{S}_{C'_i}$) (Appendix C.6). This step is specific to row variables.
(4) For each constraint-set $C''_i \in \mathcal{S}''$, we transform C''_i into an equation system E_i and then construct a general solution σ_i from E_i (Appendix C.7).
(5) Finally, we collect all the solutions σ_i, yielding a set Θ of solutions to C (Appendix C.8).

In the original algorithm for type variables, failing at the step of normalization means that there is no solution overall, even when increasing the cardinality of the substitution sets sought by dove-tail order in the general algorithm described in the beginning of Section 4 (see [Castagna et al. 2015, §3.2.3]). Whether here a failure in the step of normalization or harmonization means the absence of a solution overall is still an open question.

We write $\mathrm{Sol}_\Delta(C) \rightsquigarrow \Theta$ if $\mathrm{Sol}_\Delta(C)$ terminates with Θ, and we call Θ the solution of the type tallying problem for C.

Theorem 4.6 (Soundness). *Let C be a constraint-set. If $\mathrm{Sol}_\Delta(C) \rightsquigarrow \Theta$, then for all $\sigma \in \Theta$, $\sigma \Vdash C$.*

Proof. Consequence of Lemma C.8, soundness of merging, Lemma C.13, Lemma C.18 and soundness of Unify (Proposition C.21). □

Theorem 4.7 (Termination). *Let C be a constraint-set. Then $\mathrm{Sol}_\Delta(C)$ terminates.*

Proof. Consequence of Lemma C.9, termination of merging, Lemma C.14, finiteness of the constraint-sets and termination of Unify (Proposition C.21). □

Proposition 4.8. *Let C be a constraint-set and $\mathrm{Sol}_\Delta(C) \rightsquigarrow \Theta$. Then (1) Θ is finite and (2) for all $\sigma \in \Theta$ and for all $X \in \mathrm{dom}(\sigma)$, the types in $\sigma(X)$ are contractive.*

Proof. The first item is a consequence of Lemma C.11, Lemma C.15 and finiteness of the constraints after merging. The second one is a consequence of Lemma C.12, well-orderedness of merging, Lemma C.17, Lemma C.20 and well-orderedness of Unify (Proposition C.21). □