Revisiting Row-Polymorphism for Set-Theoretic Types

Mickaël Laurent, Pierre Donat-Bouillud, Filip Křikava, Jan Vitek

October, 2026

Sorbonne Université, Paris

Czech Technical University, Prague

Charles University, Prague









CHARLES
UNIVERSITY

Summary

Motivations

Formalism and properties

Implementation

Motivations

Motivations

Formalism and properties

Implementation



Different flavors of polymorphism

```
let set_name r name =
    match r with
    | { kind="book" ..} -> { r with title=name }
    | { kind="author" ..} -> { r with name=name }
    end
```

Different flavors of polymorphism

```
let set_name r name =
    match r with
    | { kind="book" ..} -> { r with title=name }
    | { kind="author" ..} -> { r with name=name }
    end
```

$$\{\texttt{kind} : \text{"book"}\ \texttt{..}\} \vee \{\texttt{kind} : \text{"author"}\ \texttt{..}\} \ \to\ \texttt{string}$$

$$\to \{\texttt{kind} : \text{"book"}, \texttt{title} : \texttt{string}\ \texttt{..}\} \vee \{\texttt{kind} : \text{"author"}, \texttt{name} : \texttt{string}\ \texttt{..}\}$$

Traditionally, two record type constructors [Frisch, 2004]:

- 🧀 $\{\vec{B}\ \texttt{..}\}$ is an open record type: it must contain the fields $\vec{B}$ + any others
- 🧀 $\{\vec{B}\}$ is a closed record type: it must contain exactly the fields $\vec{B}$

Not precise!

Different flavors of polymorphism

```
let set_name r name =
    match r with
    | { kind="book" ..} -> { r with title=name }
    | { kind="author" ..} -> { r with name=name }
    end
```

$$(\{\texttt{kind} : \text{"book"}\ \texttt{..}\} \to \texttt{string} \to \{\texttt{kind} : \text{"book"}, \texttt{title} : \texttt{string}\ \texttt{..}\}) \ \wedge$$
$$(\{\texttt{kind} : \text{"author"}\ \texttt{..}\} \to \texttt{string} \to \{\texttt{kind} : \text{"author"}, \texttt{name} : \texttt{string}\ \texttt{..}\})$$

🧀 Ad hoc polymorphism (intersection types): capture overloaded behaviors

Different flavors of polymorphism

```
let set_name r name =
    match r with
    | { kind="book" ..} -> { r with title=name }
    | { kind="author" ..} -> { r with name=name }
    end
```

$$(\{\texttt{kind} : \text{"book"}\ \texttt{..}\} \to \alpha \to \{\texttt{kind} : \text{"book"}, \texttt{title} : \alpha\ \texttt{..}\}) \ \wedge$$
$$(\{\texttt{kind} : \text{"author"}\ \texttt{..}\} \to \alpha \to \{\texttt{kind} : \text{"author"}, \texttt{name} : \alpha\ \texttt{..}\})$$

- 🧀 Ad hoc polymorphism (intersection types): capture overloaded behaviors
- 🧀 Parametric polymorphism (type variables): genericity over a single type

Different flavors of polymorphism

```
let set_name r name =
    match r with
    | { kind="book" ..} -> { r with title=name }
    | { kind="author" ..} -> { r with name=name }
    end
```

$$(\{\texttt{kind} : \text{"book"} \mid \rho\} \to \alpha \to \{\texttt{kind} : \text{"book"}, \texttt{title} : \alpha \mid \rho\}) \ \wedge$$
$$(\{\texttt{kind} : \text{"author"} \mid \rho\} \to \alpha \to \{\texttt{kind} : \text{"author"}, \texttt{name} : \alpha \mid \rho\})$$

- Ad hoc polymorphism (intersection types): capture overloaded behaviors
- Parametric polymorphism (type variables): genericity over a single type
- Row polymorphism (row variables): genericity over a sequence of fields (a "row")

Goals

- Capture generic behaviors over records (e.g. adding a field while preserving the others)
  ⇒ Introduction of row-variables ranging over “rows” (i.e. sequences of fields)

Goals

- Capture generic behaviors over records (e.g. adding a field while preserving the others)
  ⇒ Introduction of row-variables ranging over "rows" (i.e. sequences of fields)

- Manipulation of heterogeneous data structures
  ⇒ Compatibility with the set-theoretic connectives $\wedge$, $\vee$, $\neg$

Goals

- Capture generic behaviors over records (e.g. adding a field while preserving the others)
  ⇒ Introduction of row-variables ranging over "rows" (i.e. sequences of fields)
- Manipulation of heterogeneous data structures
  ⇒ Compatibility with the set-theoretic connectives $\wedge$, $\vee$, $\neg$
- Easily implementable within a set-theoretic type system
  ⇒ Decidable subtyping and unification ("tallying")

Formalism and properties

Motivations

Formalism and properties

Implementation



Set-Theoretic Types [Frisch, 2004]

Basic Types	b	::=	$\texttt{bool} \mid \texttt{int} \mid \ldots \mid \texttt{false} \mid \texttt{true} \mid 0 \mid 1 \mid \ldots$
Set-Theoretic Types	t	::=	$b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{empty} \mid \texttt{any}$

Set-Theoretic Types [Frisch, 2004]

Basic Types	b	::=	$\texttt{bool} \mid \texttt{int} \mid \ldots \mid \texttt{false} \mid \texttt{true} \mid 0 \mid 1 \mid \ldots$
Set-Theoretic Types	t	::=	$b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{empty} \mid \texttt{any}$

Types are interpreted as sets of values:

$[\![\texttt{false}]\!] = \{\texttt{false}\}$
$[\![\texttt{bool}]\!] = \{\texttt{false}, \texttt{true}\}$
$[\![\texttt{any}]\!] = \mathcal{V}$ (with $\mathcal{V}$ the set of all values)
$[\![\texttt{empty}]\!] = \varnothing$

Set-Theoretic Types [Frisch, 2004]

$$
\begin{array}{lrcl}
\textbf{Basic Types} & b & ::= & \texttt{bool} \mid \texttt{int} \mid \ldots \mid \texttt{false} \mid \texttt{true} \mid 0 \mid 1 \mid \ldots \\
\textbf{Set-Theoretic Types} & t & ::= & b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{empty} \mid \texttt{any}
\end{array}
$$

Types are interpreted as sets of values:

$$
\begin{array}{ll}
[\![\texttt{false}]\!] = \{\texttt{false}\} & [\![\texttt{any}]\!] = \mathcal{V} \quad \text{(with } \mathcal{V} \text{ the set of all values)} \\
[\![\texttt{bool}]\!] = \{\texttt{false}, \texttt{true}\} & [\![\texttt{empty}]\!] = \varnothing
\end{array}
$$

$$
\begin{array}{lll}
[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!] & [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!] & [\![\neg t]\!] = \mathcal{V} \setminus [\![t]\!] \\
[\![t_1 \to t_2]\!] = \text{“}[\![t_2]\!]^{[\![t_1]\!]}\text{”} * & [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!] &
\end{array}
$$

* Not the actual interpretation (for cardinality reasons).

Set-Theoretic Types [Frisch, 2004]

Basic Types $b ::= \texttt{bool} \mid \texttt{int} \mid \ldots \mid \texttt{false} \mid \texttt{true} \mid 0 \mid 1 \mid \ldots$

Set-Theoretic Types $t ::= b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{empty} \mid \texttt{any}$

Types are interpreted as sets of values:

$$[\![\texttt{false}]\!] = \{\texttt{false}\} \qquad [\![\texttt{any}]\!] = \mathcal{V} \qquad \text{(with } \mathcal{V} \text{ the set of all values)}$$
$$[\![\texttt{bool}]\!] = \{\texttt{false}, \texttt{true}\} \qquad [\![\texttt{empty}]\!] = \varnothing$$

$$[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!] \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!] \qquad [\![\neg t]\!] = \mathcal{V} \setminus [\![t]\!]$$
$$[\![t_1 \to t_2]\!] = \text{“}[\![t_2]\!]^{[\![t_1]\!]}\text{”} * \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

Semantic subtyping: $t_1 \leq t_2 \overset{def}{\Leftrightarrow} [\![t_1]\!] \subseteq [\![t_2]\!]$

* Not the actual interpretation (for cardinality reasons).

Set-Theoretic Types [Frisch, 2004]

$$\begin{array}{lrcl} \textbf{Basic Types} & b & ::= & \texttt{bool} \mid \texttt{int} \mid \ldots \mid \texttt{false} \mid \texttt{true} \mid 0 \mid 1 \mid \ldots \\ \textbf{Set-Theoretic Types} & t & ::= & b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{empty} \mid \texttt{any} \end{array}$$

Types are interpreted as sets of values:

$$\begin{array}{ll} [\![\texttt{false}]\!] = \{\texttt{false}\} & [\![\texttt{any}]\!] = \mathcal{V} \quad \text{(with } \mathcal{V} \text{ the set of all values)} \\ [\![\texttt{bool}]\!] = \{\texttt{false}, \texttt{true}\} & [\![\texttt{empty}]\!] = \varnothing \end{array}$$

$$[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!] \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!] \qquad [\![\neg t]\!] = \mathcal{V} \setminus [\![t]\!]$$

$$[\![t_1 \to t_2]\!] = \text{“}[\![t_2]\!]^{[\![t_1]\!]}\text{”} * \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

Semantic subtyping: $t_1 \leq t_2 \overset{def}{\Leftrightarrow} [\![t_1]\!] \subseteq [\![t_2]\!]$

⇒ Decidable

⇒ Inherits set properties (commutativity, idempotence, distributivity, etc.)

* Not the actual interpretation (for cardinality reasons).

Polymorphic records extension

Field Types	f	::=	$t \mid t? \mid f \wedge f \mid f \vee f \mid \neg f$
Explicit Bindings	$\vec{B}$	::=	$\ell : f, \ldots, \ell : f$ (with all ℓ distinct)
Set-Theoretic Types	t	::=	$\ldots \mid \{\vec{B} \mid f\}$

where $\ell \in$ Labels (the infinite set of all labels)

Polymorphic records extension

$$
\begin{array}{lrcl}
\textbf{Field Types} & f & ::= & t \mid t? \mid f \wedge f \mid f \vee f \mid \neg f \\
\textbf{Explicit Bindings} & \vec{B} & ::= & \ell : f, \ldots, \ell : f \quad \text{(with all } \ell \text{ distinct)} \\
\textbf{Set-Theoretic Types} & t & ::= & \ldots \mid \{\vec{B} \mid f\}
\end{array}
$$

where $\ell \in$ Labels (the infinite set of all labels)

Key idea: in record type $\{\vec{B} \mid f\}$, tail f is a field type, just like any other binding!

⇒ Not just the traditional "open" or "closed" records

⇒ The tail characterizes all remaining fields (i.e. for the infinite set Labels $\setminus$ dom($\vec{B}$)).

Polymorphic records extension

$$
\begin{array}{lrcl}
\textbf{Field Types} & f & ::= & t \mid t? \mid f \wedge f \mid f \vee f \mid \neg f \\
\textbf{Explicit Bindings} & \vec{B} & ::= & \ell : f, \ldots, \ell : f \quad \text{(with all } \ell \text{ distinct)} \\
\textbf{Set-Theoretic Types} & t & ::= & \ldots \mid \{\vec{B} \mid f\}
\end{array}
$$

where $\ell \in$ Labels (the infinite set of all labels)

Key idea: in record type $\{\vec{B} \mid f\}$, tail f is a field type, just like any other binding!

⇒ Not just the traditional "open" or "closed" records

⇒ The tail characterizes all remaining fields (i.e. for the infinite set Labels $\setminus$ dom($\vec{B}$)).

🧀 $\{\ell : \texttt{bool} \mid \texttt{int?}\}$ captures records with a field $\ell : \texttt{bool}$, and that may contain any extra field(s) of type `int`.

Polymorphic records extension

$$
\begin{array}{lrcl}
\textbf{Field Types} & f & ::= & t \mid t? \mid f \wedge f \mid f \vee f \mid \neg f \\
\textbf{Explicit Bindings} & \vec{B} & ::= & \ell : f, \ldots, \ell : f \quad \text{(with all } \ell \text{ distinct)} \\
\textbf{Set-Theoretic Types} & t & ::= & \ldots \mid \{\vec{B} \mid f\}
\end{array}
$$

where $\ell \in$ Labels (the infinite set of all labels)

Key idea: in record type $\{\vec{B} \mid f\}$, tail f is a field type, just like any other binding!

⇒ Not just the traditional "open" or "closed" records

⇒ The tail characterizes all remaining fields (i.e. for the infinite set Labels $\setminus$ dom($\vec{B}$)).

- 🧀 $\{\ell : \texttt{bool} \mid \texttt{int?}\}$ captures records with a field $\ell : \texttt{bool}$, and that may contain any extra field(s) of type `int`.
- 🧀 $\{\ell : \texttt{bool}\ \texttt{..}\}$ is a shorthand for $\{\ell : \texttt{bool} \mid \texttt{any?}\}$.

Polymorphic records extension

$$
\begin{array}{lrcl}
\textbf{Field Types} & f & ::= & t \mid t? \mid f \wedge f \mid f \vee f \mid \neg f \\
\textbf{Explicit Bindings} & \vec{B} & ::= & \ell : f, \ldots, \ell : f \quad \text{(with all } \ell \text{ distinct)} \\
\textbf{Set-Theoretic Types} & t & ::= & \ldots \mid \{\vec{B} \mid f\}
\end{array}
$$

where $\ell \in$ Labels (the infinite set of all labels)

Key idea: in record type $\{\vec{B} \mid f\}$, tail f is a field type, just like any other binding!

⇒ Not just the traditional "open" or "closed" records

⇒ The tail characterizes all remaining fields (i.e. for the infinite set Labels $\setminus$ dom($\vec{B}$)).

- 🧀 $\{\ell : \texttt{bool} \mid \texttt{int?}\}$ captures records with a field $\ell : \texttt{bool}$, and that may contain any extra field(s) of type `int`.
- 🧀 $\{\ell : \texttt{bool}\ \texttt{..}\}$ is a shorthand for $\{\ell : \texttt{bool} \mid \texttt{any?}\}$.
- 🧀 $\{\ell : \texttt{bool}\}$ is a shorthand for $\{\ell : \texttt{bool} \mid \texttt{empty?}\}$.

Polymorphic records extension

$$
\begin{array}{lrcl}
\textbf{Field Types} & f & ::= & t \mid t? \mid f \wedge f \mid f \vee f \mid \neg f \\
\textbf{Explicit Bindings} & \vec{B} & ::= & \ell : f, \ldots, \ell : f \quad \text{(with all } \ell \text{ distinct)} \\
\textbf{Set-Theoretic Types} & t & ::= & \ldots \mid \{\vec{B} \mid f\}
\end{array}
$$

where $\ell \in$ Labels (the infinite set of all labels)

Key idea: in record type $\{\vec{B} \mid f\}$, tail f is a field type, just like any other binding!

$\Rightarrow$ Not just the traditional "open" or "closed" records

$\Rightarrow$ The tail characterizes all remaining fields (i.e. for the infinite set Labels $\setminus$ dom$(\vec{B})$).

$$
[\![\{\ell_1 : f_1, \ldots, \ell_n : f_n \mid f\}]\!] = \text{"quasi-constant functions } r \in \text{Labels} \to \mathcal{V} \cup \{\Omega\} \text{ such that: } \forall i.\ r(\ell_i) \in [\![f_i]\!] \text{ and } \forall \ell \in \text{Labels} \setminus \{\ell_1, \ldots, \ell_n\}.\ r(\ell) \in [\![f]\!]\text{"}
$$

where Ω represents absent fields.

Rows and type variables

Field Types	f	$::=$	$t \mid t? \mid \rho \mid f \wedge f \mid f \vee f \mid \neg f$
Explicit Bindings	$\vec{B}$	$::=$	$\ell : f, \ldots, \ell : f$ (with all ℓ distinct)
Set-Theoretic Types	t	$::=$	$\ldots \mid \alpha \mid \{\vec{B} \mid f\}$
Rows	R	$::=$	$\langle \vec{B} \mid f \rangle$ (assignment from Labels to Field Types)

Rows and type variables

Field Types	f	$::=$	$t \mid t? \mid \rho \mid f \wedge f \mid f \vee f \mid \neg f$
Explicit Bindings	$\vec{B}$	$::=$	$\ell : f, \ldots, \ell : f$ (with all ℓ distinct)
Set-Theoretic Types	t	$::=$	$\ldots \mid \alpha \mid \{\vec{B} \mid f\}$
Rows	R	$::=$	$\langle \vec{B} \mid f \rangle$ (assignment from Labels to Field Types)

- 🧀 Type variable α: ranges over a single type t
- 🧀 Row variable ρ: ranges over a row R

Rows and type variables

Field Types	f	$::=$	$t \mid t? \mid \rho \mid f \wedge f \mid f \vee f \mid \neg f$
Explicit Bindings	$\vec{B}$	$::=$	$\ell : f, \ldots, \ell : f$ (with all ℓ distinct)
Set-Theoretic Types	t	$::=$	$\ldots \mid \alpha \mid \{\vec{B} \mid f\}$
Rows	R	$::=$	$\langle \vec{B} \mid f \rangle$ (assignment from Labels to Field Types)

- 🧀 Type variable α: ranges over a single type t
- 🧀 Row variable ρ: ranges over a row R
  - 🍷 When used in an explicit binding, e.g. $\ell : \rho$, it means $\ell : \rho.\ell$
    (it only captures the field ℓ of ρ)

Rows and type variables

Field Types	f	$::=$	$t \mid t? \mid \rho \mid f \wedge f \mid f \vee f \mid \neg f$
Explicit Bindings	$\vec{B}$	$::=$	$\ell : f, \ldots, \ell : f$ (with all ℓ distinct)
Set-Theoretic Types	t	$::=$	$\ldots \mid \alpha \mid \{\vec{B} \mid f\}$
Rows	R	$::=$	$\langle \vec{B} \mid f \rangle$ (assignment from Labels to Field Types)

- 🧀 Type variable α: ranges over a single type t
- 🧀 Row variable ρ: ranges over a row R
  - 🍷 When used in an explicit binding, e.g. $\ell : \rho$, it means $\ell : \rho.\ell$
    (it only captures the field ℓ of ρ)
  - 🍷 When in tail position, e.g. $\{\vec{B} \mid \rho\}$, it captures the row $\rho \setminus \mathsf{dom}(\vec{B})$

Rows and type variables

Field Types	f	$::=$	$t \mid t? \mid \rho \mid f \wedge f \mid f \vee f \mid \neg f$
Explicit Bindings	$\vec{B}$	$::=$	$\ell : f, \ldots, \ell : f$ (with all ℓ distinct)
Set-Theoretic Types	t	$::=$	$\ldots \mid \alpha \mid \{\vec{B} \mid f\}$
Rows	R	$::=$	$\langle \vec{B} \mid f \rangle$ (assignment from Labels to Field Types)

Example: $\{\ell_1 : \rho, \ell_2 : \rho' \mid \rho''\}$ means

- 🧀 ℓ_1 has field type captured by $\rho.\ell_1$
- 🧀 ℓ_2 has field type captured by $\rho'.\ell_2$
- 🧀 any other label $\ell \in \text{Labels} \setminus \{\ell_1, \ell_2\}$ has field type captured by $\rho''.\ell$

Rows and type variables

Field Types	f	::=	$t \mid t? \mid \rho \mid f \wedge f \mid f \vee f \mid \neg f$
Explicit Bindings	$\vec{B}$	::=	$\ell : f, \ldots, \ell : f$ (with all ℓ distinct)
Set-Theoretic Types	t	::=	$\ldots \mid \alpha \mid \{\vec{B} \mid f\}$
Rows	R	::=	$\langle \vec{B} \mid f \rangle$ (assignment from Labels to Field Types)

Compatible with set-theoretic connectives: $\{\ell_1 : \rho \vee \texttt{int},\ \ell_2 : \rho \wedge \rho' \mid \rho \vee \rho'\}$ means

- ℓ_1 has field type captured by $\rho.\ell_1 \vee \texttt{int}$
- ℓ_2 has field type captured by $\rho.\ell_2 \wedge \rho'.\ell_2$
- any other label $\ell \in \text{Labels} \setminus \{\ell_1, \ell_2\}$ has field type captured by $\rho.\ell \vee \rho'.\ell$

Substitutions

Something very cool: it is straightforward to apply a substitution, because bindings and tails are homogeneous (both are field types)!

Substitutions

Something very cool: it is straightforward to apply a substitution, because bindings and tails are homogeneous (both are field types)!

$$\phi \stackrel{\text{def}}{=} \{\rho_1 \rightsquigarrow \langle \ell_1 : t_1,\ \ell_2 : t_2 \mid \rho_1 \rangle\}$$

$$\mathbf{\{}\ell_1 : \texttt{int} \mid \rho_1 \wedge \rho_2\mathbf{\}}\phi$$

Substitutions

Something very cool: it is straightforward to apply a substitution, because bindings and tails are homogeneous (both are field types)!

$$\phi \stackrel{\text{def}}{=} \{\rho_1 \rightsquigarrow \langle \ell_1 : t_1,\ \ell_2 : t_2 \mid \rho_1 \rangle\}$$

$$\begin{aligned} & \{\ell_1 : \texttt{int} \mid \rho_1 \wedge \rho_2\}\phi \\ = \ & \{\ell_1 : \texttt{int},\ \ell_2 : \rho_1 \wedge \rho_2 \mid \rho_1 \wedge \rho_2\}\phi \qquad \text{(unfolding field } \ell_2\text{)} \end{aligned}$$

Substitutions

Something very cool: it is straightforward to apply a substitution, because bindings and tails are homogeneous (both are field types)!

$$\phi \stackrel{\text{def}}{=} \{\rho_1 \rightsquigarrow \langle \ell_1 : t_1,\ \ell_2 : t_2 \mid \rho_1 \rangle\}$$

$$
\begin{aligned}
& \{\ell_1 : \texttt{int} \mid \rho_1 \wedge \rho_2\}\phi \\
= \ & \{\ell_1 : \texttt{int},\ \ell_2 : \rho_1 \wedge \rho_2 \mid \rho_1 \wedge \rho_2\}\phi && \text{(unfolding field } \ell_2\text{)} \\
= \ & \{\ell_1 : \texttt{int},\ \ell_2 : (\rho_1.\ell_2) \wedge (\rho_2.\ell_2) \mid \rho_1 \wedge \rho_2\}\phi && \text{(unfolding projections)}
\end{aligned}
$$

Substitutions

Something very cool: it is straightforward to apply a substitution, because bindings and tails are homogeneous (both are field types)!

$$\phi \stackrel{\text{def}}{=} \{\rho_1 \rightsquigarrow \langle \ell_1 : t_1,\ \ell_2 : t_2 \mid \rho_1 \rangle\}$$

$$
\begin{aligned}
& \{\ell_1 : \texttt{int} \mid \rho_1 \wedge \rho_2\}\phi \\
= \ & \{\ell_1 : \texttt{int},\ \ell_2 : \rho_1 \wedge \rho_2 \mid \rho_1 \wedge \rho_2\}\phi && \text{(unfolding field } \ell_2\text{)} \\
= \ & \{\ell_1 : \texttt{int},\ \ell_2 : (\rho_1.\ell_2) \wedge (\rho_2.\ell_2) \mid \rho_1 \wedge \rho_2\}\phi && \text{(unfolding projections)} \\
= \ & \{\ell_1 : \texttt{int},\ \ell_2 : t_2 \wedge (\rho_2.\ell_2) \mid \rho_1 \wedge \rho_2\} && \text{(applying substitution)}
\end{aligned}
$$

Substitutions

Something very cool: it is straightforward to apply a substitution, because bindings and tails are homogeneous (both are field types)!

$$\phi \stackrel{\text{def}}{=} \{\rho_1 \rightsquigarrow \langle \ell_1 : t_1,\ \ell_2 : t_2 \mid \rho_1 \rangle\}$$

$$
\begin{aligned}
& \{\ell_1 : \texttt{int} \mid \rho_1 \wedge \rho_2\}\phi \\
= \ & \{\ell_1 : \texttt{int},\ \ell_2 : \rho_1 \wedge \rho_2 \mid \rho_1 \wedge \rho_2\}\phi && \text{(unfolding field } \ell_2\text{)} \\
= \ & \{\ell_1 : \texttt{int},\ \ell_2 : (\rho_1.\ell_2) \wedge (\rho_2.\ell_2) \mid \rho_1 \wedge \rho_2\}\phi && \text{(unfolding projections)} \\
= \ & \{\ell_1 : \texttt{int},\ \ell_2 : t_2 \wedge (\rho_2.\ell_2) \mid \rho_1 \wedge \rho_2\} && \text{(applying substitution)} \\
= \ & \{\ell_1 : \texttt{int},\ \ell_2 : t_2 \wedge \rho_2 \mid \rho_1 \wedge \rho_2\}
\end{aligned}
$$

Tallying

Tallying [Castagna et al., 2015] ("unification, but with subtyping constraints"):

$$\texttt{tally}(t_1, t_2) = \{\sigma \mid t_1\sigma \leq t_2\sigma\}$$

Tallying

Tallying [Castagna et al., 2015] ("unification, but with subtyping constraints"):

$$\texttt{tally}(t_1, t_2) = \{\sigma \mid t_1\sigma \leq t_2\sigma\}$$

🧀 Tallying (without polymorphic records) is computable [Castagna et al., 2015].

Tallying

Tallying [Castagna et al., 2015] ("unification, but with subtyping constraints"):

$$\texttt{tally}(t_1, t_2) = \{\sigma \mid t_1\sigma \leq t_2\sigma\}$$

- 🧀 Tallying (without polymorphic records) is computable [Castagna et al., 2015].
- 🧀 Solutions are characterized by a principal finite set of substitutions (compared to at most one principal substitution for HM unification).

Tallying

Tallying [Castagna et al., 2015] ("unification, but with subtyping constraints"):

$$\texttt{tally}(t_1, t_2) = \{\sigma \mid t_1\sigma \leq t_2\sigma\}$$

- Tallying (without polymorphic records) is computable [Castagna et al., 2015].
- Solutions are characterized by a principal finite set of substitutions (compared to at most one principal substitution for HM unification).

Example:

$$\alpha \times \beta \leq \texttt{int} \times \texttt{bool}$$

Tallying

Tallying [Castagna et al., 2015] ("unification, but with subtyping constraints"):

$$\texttt{tally}(t_1, t_2) = \{\sigma \mid t_1\sigma \leq t_2\sigma\}$$

- Tallying (without polymorphic records) is computable [Castagna et al., 2015].
- Solutions are characterized by a principal finite set of substitutions (compared to at most one principal substitution for HM unification).

Example:

$$\alpha \times \beta \leq \texttt{int} \times \texttt{bool}$$

Solutions: $\{\sigma_1, \sigma_2, \sigma_3\}$ where

$$\sigma_1 = \{\alpha \rightsquigarrow \texttt{int} \wedge \alpha, \quad \beta \rightsquigarrow \texttt{bool} \wedge \beta\}$$

$$\sigma_2 = \{\alpha \rightsquigarrow \texttt{empty}\}$$

$$\sigma_3 = \{\beta \rightsquigarrow \texttt{empty}\}$$

Tallying (with polymorphic records)

🧀 The traditional tallying algorithm can be extended to handle polymorphic records.

Example: $\{\ell_1 : \texttt{int},\ \ell_2 : \texttt{bool} \mid \texttt{any?}\} \leq \{\ell_1 : \alpha \mid \rho\}$

Solution: $\{\alpha \rightsquigarrow \texttt{int} \vee \alpha,\quad \rho \rightsquigarrow \langle \ell_2 : \texttt{bool} \vee \rho \mid \texttt{any?}\rangle\}$

Tallying (with polymorphic records)

🧀 The traditional tallying algorithm can be extended to handle polymorphic records.

Example: $\{\ell_1 : \texttt{int},\ \ell_2 : \texttt{bool} \mid \texttt{any?}\} \leq \{\ell_1 : \alpha \mid \rho\}$

Solution: $\{\alpha \rightsquigarrow \texttt{int} \vee \alpha,\quad \rho \rightsquigarrow \langle \ell_2 : \texttt{bool} \vee \rho \mid \texttt{any?}\rangle\}$

🧀 Sometimes, solutions cannot be all characterized by a finite set of substitutions...

Example: $\{\mid \rho\} \leq \texttt{empty}$

Solutions: for any $\ell \in$ Labels, $\quad \phi_\ell = \{\rho \rightsquigarrow \langle \ell : \texttt{empty} \mid \rho\rangle\}$ is solution

Tallying (with polymorphic records)

- The traditional tallying algorithm can be extended to handle polymorphic records.

Example: $\{\ell_1 : \texttt{int},\ \ell_2 : \texttt{bool} \mid \texttt{any?}\} \leq \{\ell_1 : \alpha \mid \rho\}$
Solution: $\{\alpha \rightsquigarrow \texttt{int} \vee \alpha,\quad \rho \rightsquigarrow \langle \ell_2 : \texttt{bool} \vee \rho \mid \texttt{any?}\rangle\}$

- Sometimes, solutions cannot be all characterized by a finite set of substitutions...

Example: $\{\mid \rho\} \leq \texttt{empty}$
Solutions: for any $\ell \in$ Labels, $\quad \phi_\ell = \{\rho \rightsquigarrow \langle \ell : \texttt{empty} \mid \rho\rangle\}$ is solution

- **Not an issue in practice.**
  We can retrieve completeness of the tallying algorithm by restricting locally the set of labels that can appear in the explicit bindings $\vec{B}$ of our rows $\langle \vec{B} \mid f \rangle$.
  Example: *"I only want solutions involving labels that appear in this application."*

Implementation

Motivations

Formalism and properties

Implementation



Implementation in MLsem (https://e-sh4rk.github.io/MLsem/)

```
5   val (+) : ((int, int) -> int) & ((string, string) -> string)
6
    { kind : "author" ; name : string ..} | { kind : "book" ; title : string ..} -> string
7   let to_string r =
8     match r with
9     | { kind="book"   ; title ..} -> "Book: " + title
10    | { kind="author" ; name ..} -> "Author: " + name
11    end
12
    ∀'a,'b,`a,`b. ({ kind : `b & "book" ; title : any? ;; `b } -> 'a -> { title : 'a ;; `b }) & ({ kind : `a & "author" ; name : any? ;; `a } -> 'b -> { name : 'b ;; `a })
13  let set_name r name =
14      match r with
15      | { kind="book" ..} -> { r with title=name }
16      | { kind="author" ..} -> { r with name=name }
17      end
18
    { kind : "author" ..} | { kind : "book" ..} -> string -> string
19  let foo data name =
20      to_string (set_name data name)
21
```

Encodings

Our polymorphic records can be used to encode many language-specific constructs (cf. my other talk "Type inference for functional and imperative dynamic languages"):

Encodings

Our polymorphic records can be used to encode many language-specific constructs (cf. my other talk "Type inference for functional and imperative dynamic languages"):

- Arguments in dynamic languages (positional/named, optional, ellipsis, etc.), e.g. `def f(a, b, *, c=42, d=True, **kwargs)` in Python, `function(a, b, c=42, d=TRUE, ...)` in R
  - Positional arguments can be associated with labels `__1`, `__2`, `__3`, etc.
  - Extra arguments passed to a function (`...` in R, `**kwargs` in Python) can be captured by a row variable

Encodings

Our polymorphic records can be used to encode many language-specific constructs (cf. my other talk "Type inference for functional and imperative dynamic languages"):

- Arguments in dynamic languages (positional/named, optional, ellipsis, etc.), e.g. `def f(a, b, *, c=42, d=True, **kwargs)` in Python, `function(a, b, c=42, d=TRUE, ...)` in R
  - Positional arguments can be associated with labels `__1`, `__2`, `__3`, etc.
  - Extra arguments passed to a function (`...` in R, `**kwargs` in Python) can be captured by a row variable
- Dictionaries/maps (cf. "Typing records, maps, and structs" [Castagna, 2023])

Encodings

Our polymorphic records can be used to encode many language-specific constructs (cf. my other talk "Type inference for functional and imperative dynamic languages"):

- Arguments in dynamic languages (positional/named, optional, ellipsis, etc.), e.g. `def f(a, b, *, c=42, d=True, **kwargs)` in Python, `function(a, b, c=42, d=TRUE, ...)` in R
  - Positional arguments can be associated with labels `__1`, `__2`, `__3`, etc.
  - Extra arguments passed to a function (`...` in R, `**kwargs` in Python) can be captured by a row variable
- Dictionaries/maps (cf. "Typing records, maps, and structs" [Castagna, 2023])
- Or even vectors, lists, classes and attributes in R

Conclusion

A general formalism for polymorphic records:

- Compatible with set-theoretic connectives ($\wedge$, $\vee$, $\neg$)
- Parametric polymorphism using type variables and row variables
- Easy to implement type inference thanks to the tallying algorithm

Related work:

- Polymorphic Records for Dynamic Languages [Castagna and Peyrot, 2025]
  - Different expressivity (both formalisms have examples that cannot be typed by the other)
  - More complex to implement (the tallying algorithm is not complete)
- Typing Records, Maps, and Structs [Castagna, 2023]
  - Different problematic: how to have fields indexed by types (not labels)?
  - Could both works be combined?

References i


Castagna, G. (2023).
Typing records, maps, and structs.
Proc. ACM Program. Lang., 7(ICFP).


Castagna, G., Nguyen, K., Xu, Z., and Abate, P. (2015).
Polymorphic functions with set-theoretic types. Part 2: local type inference and type reconstruction.
In *Proceedings of the 42nd Annual ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '15, pages 289–302.

Castagna, G. and Peyrot, L. (2025).
Polymorphic records for dynamic languages.
Proc. ACM Program. Lang., 9(OOPSLA1).

Frisch, A. (2004).
Théorie, conception et réalisation d'un langage de programmation adapté à XML.
PhD thesis, Université Paris Diderot.